# ParaView Users Guide Documentation

***Release 5.12.0***

**ParaView Developers**

**Apr 19, 2024**

# PARAVIEW USER'S GUIDE

This guide is split into several volumes:

- **User's Guide**'s Section 1.1 to Section 1.8 cover various aspects of data analysis and visualization with **ParaView**.
- **Reference Manual**'s Section 2.1 to Section 2.12 provide details on various components in the UI and the scripting API.
- **Catalyst**: Instructions on how to use ParaView's implementation of the Catalyst API.
- **Tutorials** are split into **Self-directed Tutorial** and **Classroom Tutorials**:
  - **Self-directed Tutorial**'s Section 4.1.1 to Section 4.1.5 provide an introduction to the **ParaView** software and its history, and exercises on how to use **ParaView** that cover basic usage, batch Python scripting and visualizing large models.
  - **Classroom Tutorials**'s Section 4.2.1 to Section 4.2.18 provide beginning, advanced, Python and batch, and targeted tutorial lessons on how to use **ParaView** that are presented as a 3-hour class internally within Sandia National Laboratories.

CHAPTER

# ONE

# PARAVIEW USER'S GUIDE

## 1.1 Introduction to ParaView

### 1.1.1 Introduction

**ParaView** is an open-source, multi-platform scientific data analysis and visualization tool that enables analysis and visualization of extremely large datasets. **ParaView** is both a general purpose, end-user application with a distributed architecture that can be seamlessly leveraged by your desktop or other remote parallel computing resources and an extensible framework with a collection of tools and libraries for various applications including scripting (using Python), web visualization (through trame and ParaViewWeb), or in-situ analysis (with *Catalyst*).

**ParaView** leverages parallel data processing and rendering to enable interactive visualization for extremely large datasets. It also includes support for large displays including tiled displays and immersive 3D displays with head tracking and wand control capabilities.

**ParaView** also supports scripting and batch processing using Python. Using included Python modules, you can write scripts that can perform almost all the functionality exposed by the interactive application and much more.

**ParaView** is open-source (BSD licensed, commercial software friendly). As with any successful open-source project, **ParaView** is supported by an active user and developer community. Contributions to both the code and this user's manual that help make the tool and the documentation better are always welcome.

**Did you know?**

The **ParaView** project started in 2000 as a collaborative effort between Kitware Inc. and LANL (Los Alamos National Labs). The initial funding was provided by a three year contract with the US Department of Energy ASCI Views program. The first public release, **ParaView** 0.6, was announced in October 2002.

Independent of **ParaView**, Kitware started developing a web-based visualization system in December 2001. This project was funded by Phase I and II SBIRs from the US ARL (Arm Research Laboratory) and eventually became the PVEE (ParaView Enterprise Edition). PVEE significantly contributed to the development of **ParaView**'s client/server architecture. PVEE was the precursor to ParaViewWeb, a modern web visualization solution based on **ParaView**.

Since the project began, Kitware has successfully collaborated with Sandia, LANL, the ARL, and various other academic and government institutions to continue development. Today, the project is still going strong!

In September 2005, Kitware, Sandia National Labs and CSimSoft started the development of **ParaView** 3.0. This was a major effort focused on rewriting the user interface to be more user friendly and on developing a quantitative analysis framework. **ParaView** 3.0 was released in May 2007.

### In this guide

This user's manual is designed as a guide for using the **ParaView** application. It is geared toward users who have a general understanding of common data visualization techniques. For scripting, a working knowledge of the Python language is assumed. If you are new to Python, there are several tutorials and guides for getting started that are available on the Internet.

**Did You Know?**

In this guide, we will periodically use these **Did you know?** boxes to provide additional information related to the topic at hand.

**Common Errors**

**Common Errors** blocks are used to highlight some of the common problems or complications you may run into when dealing with the topic of discussion.

This guide is split into several volumes:

- **User's Guide**'s Section 1.1 to Section 1.8 cover various aspects of data analysis and visualization with **ParaView**.
- **Reference Manual**'s Section 2.1 to Section 2.12 provide details on various components in the UI and the scripting API.
- **Catalyst**: Instructions on how to use ParaView's implementation of the Catalyst API.
- **Tutorials** are split into **Self-directed Tutorial** and **Classroom Tutorials**:
  - **Self-directed Tutorial**'s Section 4.1.1 to Section 4.1.5 provide an introduction to the **ParaView** software and its history, and exercises on how to use **ParaView** that cover basic usage, batch python scripting and visualizing large models.
  - **Classroom Tutorials**'s Section 4.2.1 to Section 4.2.18 provide beginning, advanced, python and batch, and targeted tutorial lessons on how to use **ParaView** that are presented as a 3-hour class internally within Sandia National Laboratories.

### Getting help

This guide tries to cover most of the commonly used functionality in **ParaView**. **ParaView**'s flexible, pipeline-based architecture opens up numerous possibilities. If you find yourself looking for some feature not covered in this guide, refer to the Wiki pages or to the ParaView Discourse forum, specially the FAQ and the Tips and Tricks categories. Also feel free to ask about it under the relevant Support category.

### Getting the software

**ParaView** is open source. The complete source code for all the functionality discussed in The **ParaView** Guide can be downloaded from the **ParaView** website http://www.paraview.org. We also provide binaries for the major platforms: Linux, macOS, and Windows. You can get the source files and binaries for the official releases, and follow ParaView's active development by downloading the nightly builds.

Providing details of how to build **ParaView** using the source files is beyond the scope of this guide. Refer to the **ParaView** gitlab (https://gitlab.kitware.com/paraview/paraview/-/blob/master/Documentation/dev/build.md) for more information.

### 1.1.2 Basics of visualization in ParaView

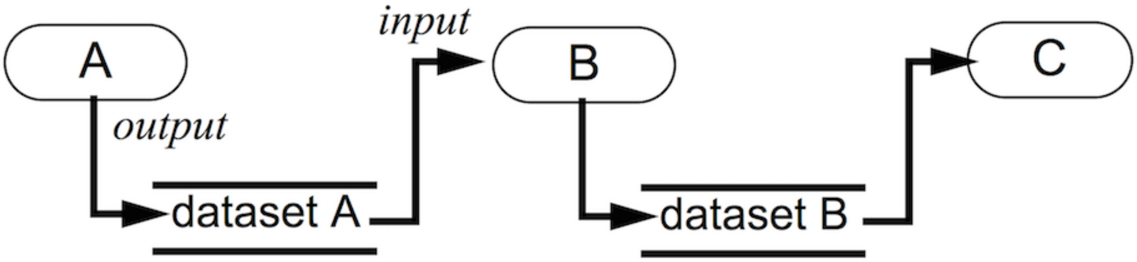


Fig. 1.1: Visualization model: Process objects A, B, and C input and/or output one or more data objects. Data objects represent and provide access to data; process objects operate on the data. Objects A, B, and C are source, filter, and mapper objects, respectively. [SML96]

Visualization is the process of converting raw data into images and renderings to gain a better cognitive understanding of the data. **ParaView** uses VTK, the Visualization Toolkit, to provide the backbone for visualization and data processing.

The VTK model is based on the data-flow paradigm. In this paradigm, data flows through the system being transformed at each step by modules known as algorithms. Algorithms could be common operations such as clipping, slicing, or generating contours from the data, or they could be computing derived quantities, etc. Algorithms have input ports through which they take in data and output ports through which they produce output. You need producers that ingest data into the system. These are simply algorithms that do not have an input port but have one or more output ports. They are called *sources* . Readers that read data from files are examples of such sources. Additionally, there are algorithms that transform the data into graphics primitives so that they can be rendered on a computer screen or saved to disk in another file. These algorithms, which have one or more input ports but do not have output ports, are called *sinks* . Intermediate algorithms with input ports and output ports are called *filters* . Together, sources, filters, and sinks provide a flexible infrastructure wherein you can create complex processing pipelines by simply connecting algorithms to perform arbitrarily complex tasks.

For more information on VTK's programming model, refer to [SML96].

This way of looking at the visualization pipeline is at the core of **ParaView**'s work flow: You bring your data into the system by creating a reader – the source. You then apply filters to either extract information (e.g., iso-contours) and render the results in a view or to save the data to disk using writers – the sinks.

**ParaView** includes readers for a multitude of file formats typically used in the computational science world. To efficiently represent data from various fields with varying characteristics, VTK provides a rich data model that **ParaView** uses. The data model can be thought of simply as ways of representing data in memory. We will cover the different data types in more detail in Section 1.3.1. Readers produce a data type suitable for representing the information the files contain. Based on the data type, **ParaView** allows you to create and apply filters to transform the data. You can also show the data in a view to produce images or renderings. Just as there are several types of filters, each perfoming different operations and types of processing, there are several kinds of views for generating various types of renderings including 3D surface views, 2D bar and line views, parallel coordinate views, etc.

**Did You Know?**

The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D computer graphics, modeling, image processing, volume rendering, scientific visualization, and information visualization. VTK also includes ancillary support for 3D interaction widgets, two and three-dimensional annotation, and parallel computing.

At its core, VTK is implemented as a C++ toolkit, requiring users to build applications by combining various objects into an application. The system also supports automated wrapping of the C++ core into Python and Java so that VTK applications may also be written using these programming languages. VTK is used world-wide in commercial applications, research and development, and as the basis of many advanced visualization applications such as **ParaView**, VisIt, VisTrails, Slicer, MayaVi, and OsiriX.

### 1.1.3 ParaView executables

**ParaView** comes with several executables that serve different purposes.

#### `paraview`

This is the main **ParaView** graphical user interface (GUI). In most cases, when we refer to **ParaView**, we are indeed talking about this application. It is a Qt-based, cross-platform UI that provides access to **ParaView** computing capabilities. Major parts of this guide are dedicated to understanding and using this application.

#### `pvpython`

`pvpython` is the Python interpreter that runs **ParaView**'s Python scripts. You can think of this as the equivalent of the `paraview` for scripting.

#### `pvbatch`

Similar to `pvpython`, `pvbatch` is also a Python interpreter that runs Python scripts for **ParaView**. The one difference is that, while `pvpython` is meant to run interactive scripts, `pvbatch` is designed for batch processing. Additionally, when running on computing resources with MPI capabilities, `pvbatch` can be run in parallel. We will cover this in more detail in Section 2.7.10.

#### `pvserver`

For remote visualization, this executable represents the server that does all of the data processing and, potentially, the rendering. You can make `paraview` connect to `pvserver` running remotely on an HPC resource. This allows you to build and control visualization and analysis on the HPC resource from your desktop as if you were simply processing it locally on your desktop!

#### `pvdataserver` and `pvrenderserver`

These can be thought of as the `pvserver` split into two separate executables: one for the data processing part, `pvdataserver`, and one for the rendering part, `pvrenderserver`. Splitting these into separate processes makes it possible to perform data processing and rendering on separate sets of nodes with appropriate computing capabilities suitable for the two tasks. Just as with `pvserver`, `paraview` can connect to a `pvdataserver`- `pvrenderserver` pair for remote visualization. Unless otherwise noted, all discussion of remote visualization or client-server visualization in this guide is applicable to both `pvserver` and `pvdataserver` - `pvrenderserver` configurations.

## 1.1.4 Getting started with `paraview`

### `paraview` graphical user interface

`paraview` is the graphical front-end to the **ParaView** application. The UI is designed to allow you to easily create pipelines for data processing with arbitrary complexity. The UI provides panels for you to inspect and modify the pipelines, to change parameters that in turn affect the processing pipelines, to perform various data selection and inspection actions to introspect the data, and to generate renderings. We will cover various aspects of the UI for the better part of this guide.

Let's start by looking at the various components of the UI. If you run `paraview` for the first time, you will see something similar to the Fig. 1.2. The UI consists of of menus, dockable panels, toolbars, and the viewport – the central portion of the application window.

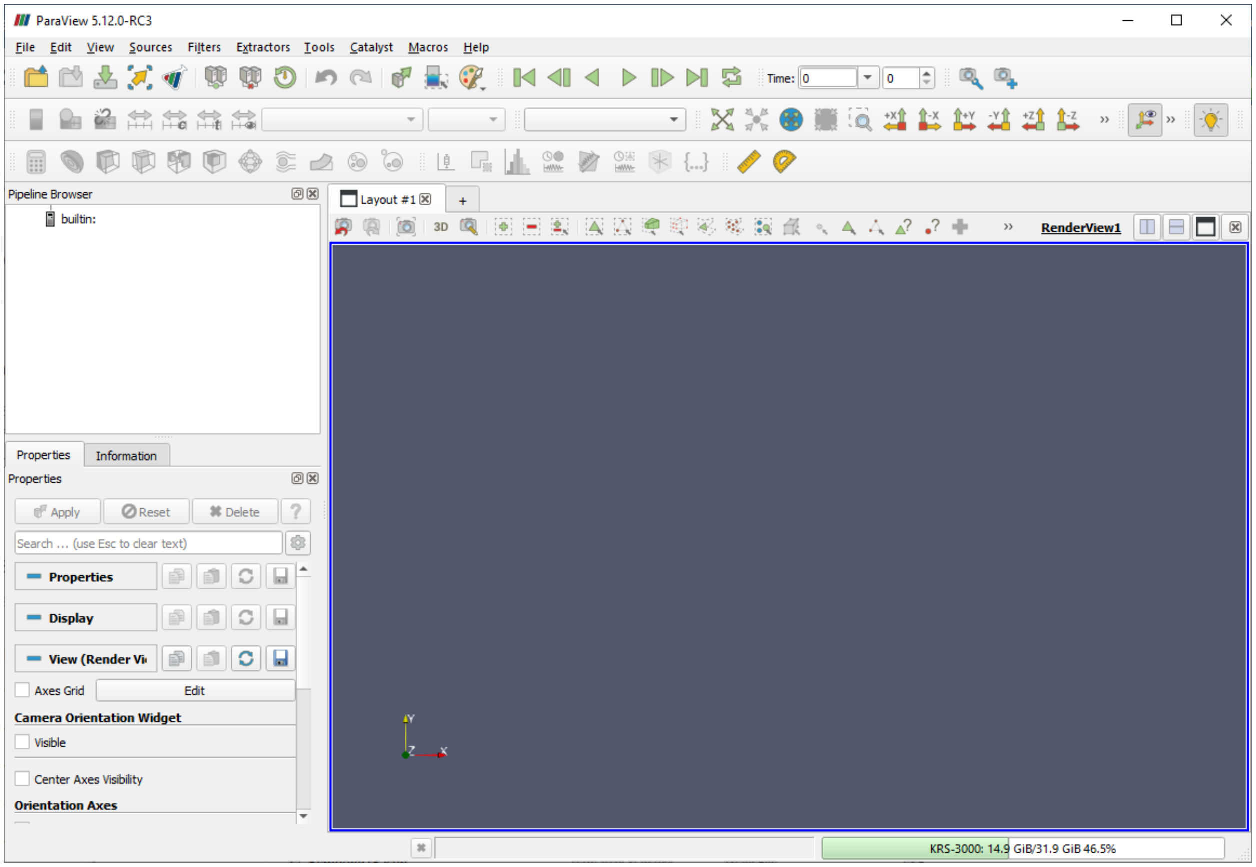


Fig. 1.2: `paraview` application window.

Menus provide the standard set of options typical with a desktop application including options for opening/saving files (*File* menu), for undo/redo (*Edit* menu), for the toggle panel, and for toolbar visibilities (*View* menu). Additionally, the menus provide ways to create sources that generate test datasets of various types (*Sources* menu), as well new filters for processing data (*Filters* menu). The *Tools* menu provides access to some of the advanced features in `paraview` such as managing plugins and favorites.

Panels provide you with the ability to peek into the application's state. For example, you can inspect the visualization pipeline that has been set up ( `Pipeline Browser` ), as well as the memory that is being used ( `Memory Inspector` ) and the parameters or properties for a processing module ( `Properties` panel). Several of the panels also allow you to change the values that are displayed, e.g., the `Properties` panel not only shows the processing module parameters, but it also allows you to change them. Several of the panels are context sensitive. For example, the `Properties` panel changes to show the parameters from the selected module as you change the active module in the `Pipeline Browser` .

Toolbars are designed to provide quick access to common functionality. Several of the actions in the toolbar are accessible from other locations, including menus or panels. Similar to panels, some of the toolbar buttons are context sensitive and will become enabled or disabled based on the selected module or view.

The viewport or the central portion of the `paraview` window is the area where **ParaView** renders results generated from the data. The containers in which data can be rendered or shown are called *views*. You can create several different types of views, all of which are laid out in this viewport area. By default, a 3D view is created, which is one of the most commonly used views in **ParaView**.

### Understanding the visualization process

To gain a better understanding of how to use the application interface, let's consider a simple example: creating a data source and applying a filter to it.

### Creating a source

The visualization process in **ParaView** begins by bringing your data into the application. Section 1.2 explains how to read data from various file formats. Besides reading files to bring in data into the application, **ParaView** also provides a collection of data sources that can produce sample datasets. These are available under the *Sources* menu. To create a source, simply click on any item in the *Source* menu.

**Did You Know?**

As you move your cursor over the items in any menu, on most platforms (except macOS), you'll see a brief description of the item in the status bar on the lower-left corner in the application window.

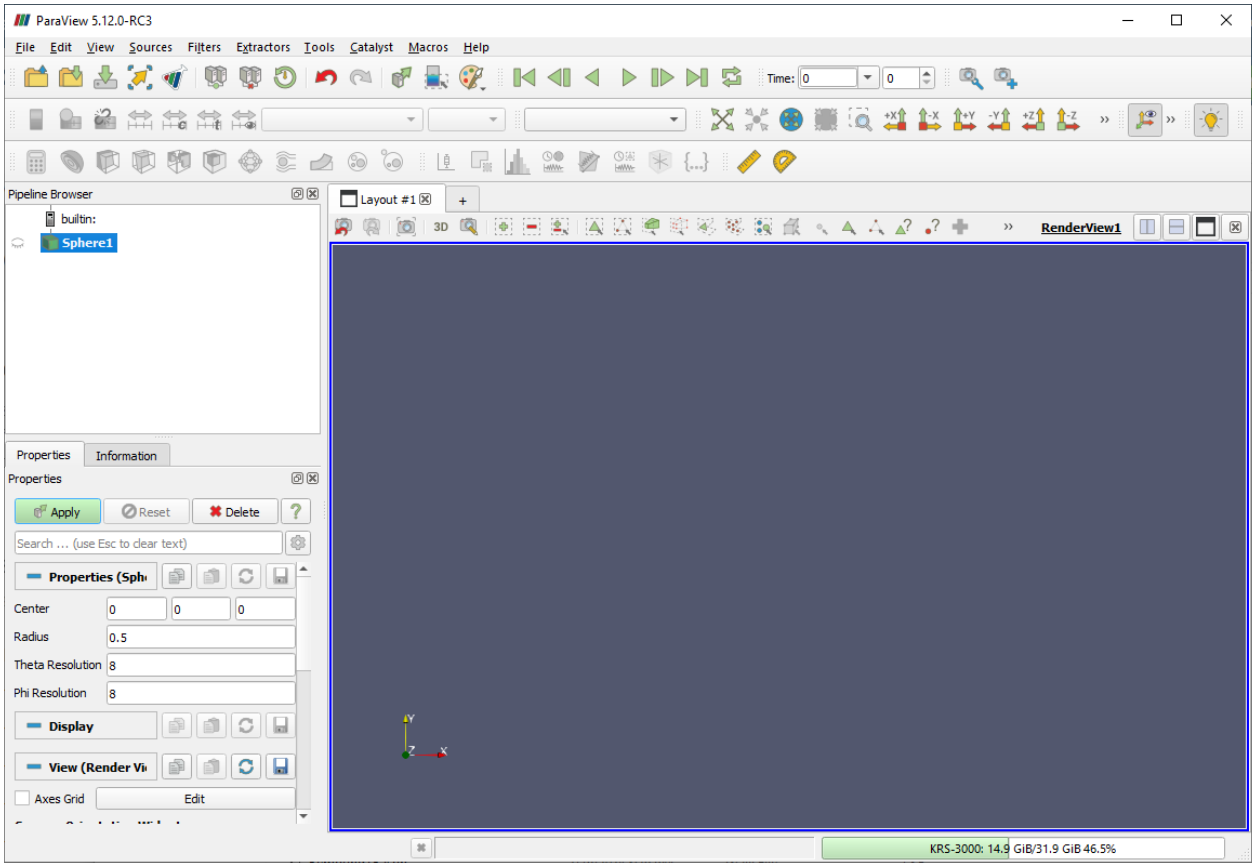
If you click on *Sources* > *Alphabetical* > *Sphere*, for example, you'll create a producer algorithm that generates a spherical surface, as shown in Fig. 1.3.

A few things to note:

1. A pipeline module is added in the `Pipeline Browser` panel with a name derived from the menu item, as is highlighted.
2. The `Properties` panel fills up with text to indicate that it's showing properties for the highlighted item (which, in this case, is `Sphere1` ), as well as to display some widgets for parameters such as `Center` , `Radius` , etc.
3. On the `Properties` panel, the `Apply` button becomes enabled and highlighted.
4. The 3D view remains unaffected, as nothing new is shown or rendered in this view as of yet.

Let's take a closer look at what has happened. When we clicked on *Sources* > *Sphere*, referring to Section 1.1.2, we created an instance of a source that can produce a spherical surface mesh – that's what is reflected in the `Pipeline Browser` . This instance receives a name, which is used by the `Sphere1` and the `Pipeline Browser` , as well as other components of the UI, to refer to this instance of the source. Pipeline modules such as sources and filters have parameters on them that you can change that affect that module's behavior. We call them *properties*. The `Properties` panel shows these properties and allows you to change them. Since the ingestion of data into the system can be a time-consuming process, `paraview` allows you to change the properties before the module executes or performs the actual processing to ingest the data. Hence, the `Apply` button is highlighted to indicate that you need to accept the properties before the application will proceed. Since no data has entered the system yet, there's nothing to show. Therefore, the 3D view remains unaffected.

Let's assume we are okay with the default values for all of the properties on the `Sphere1` . Next, click on the `Apply` button.



Time: 0 0

Center 0 0 0

Radius 0.5

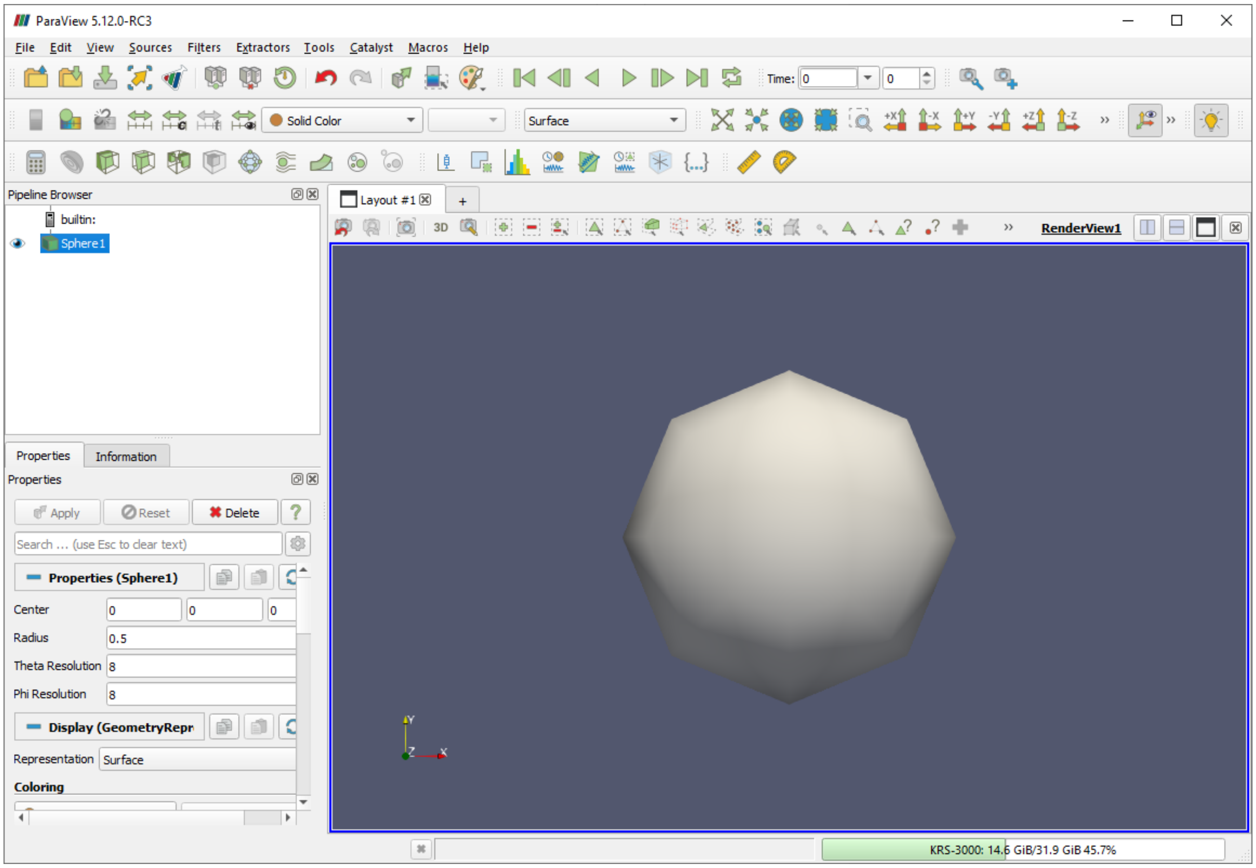
Theta Resolution 8

Phi Resolution 8

KRS-3000: 14.9 GiB/31.9 GiB 46.5%

Fig. 1.3: Visualization in `paraview`: Step 1.

Let's assume we are okay with the default values for all of the properties on the `Sphere1`. Next, click on the `Apply` button.



Center 0 0 0

Radius 0.5

Theta Resolution 8

Phi Resolution 8

Representation Surface

Fig. 1.4: Visualization in `paraview`: Step 2.

The following will ensue ( Fig. 1.4 ):

1. The `Apply` button goes back to its old disabled/un-highlighted state.
2. A spherical surface is rendered in the 3D view.
3. The `Display` section on the `Properties` panel now shows new parameters or properties.
4. Certain toolbars update, and you can see that toolbars with text, such as `Solid Color` and `Surface` , now become enabled.

By clicking `Apply` , we told `paraview` to apply the properties shown on the `Properties` panel. When a new source (or filter) is applied for the first time, `paraview` will automatically show the data that the pipeline module produces in the current view, if possible. In this case, the sphere source produces a surface mesh, which is then shown or displayed in the 3D view.

The properties that allow you to control how the data is displayed in the view are now shown on the `Properties` panel in the `Display` section. Things such as the surface color, rendering type or representation, shading parameters, etc., are shown under this newly updated section. We will look at display properties in more detail in Section 1.4.

Some of the properties that are commonly used are also duplicated in the toolbar. These properties include the data array with which the surface is colored and the representation type. These are the changes in the toolbar that allow you to quickly change some display properties.

### Changing properties

If you change any of the properties on the sphere source, such as the properties under the `Properties` section on the `Properties` panel, including the `Radius` for the spherical mesh or its `Center` , the `Apply` button will be highlighted again. Once you are finished with all of the property changes, you can hit `Apply` to apply the changes. Once the changes are applied, `paraview` will re-execute the sphere source to produce a new mesh, as requested. It will then automatically update the view, and you will see the new result rendered.

If you change any of the display properties for the sphere source, such as the properties under the `Display` section of the `Properties` panel (including `Representation` or `Opacity` ), the `Apply` button is not affected, the changes are immediately applied, and the view is updated.

The rationale behind this is that, typically, the execution of the source (or filter) is more computationally intensive than the rendering. Changing source (or filter) properties causes that algorithm to re-execute, while changing display properties, in most cases, only triggers a fresh render with an updated graphics state.

**Did You Know?**

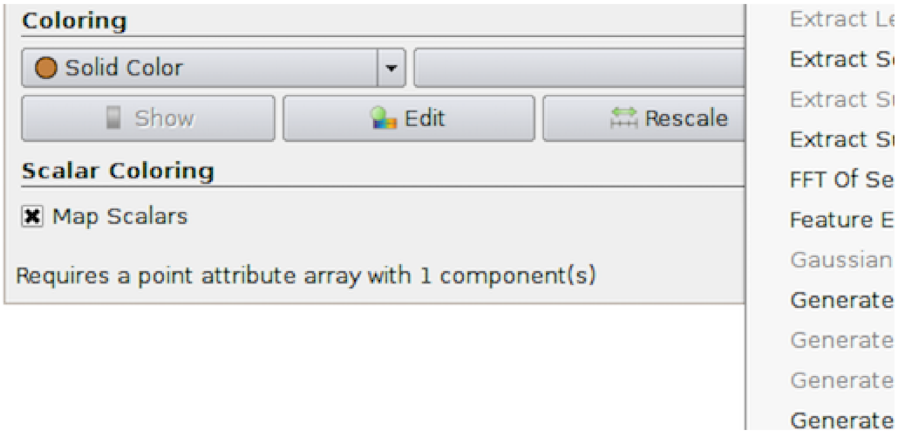
For some workflows with smaller data sizes, it may be more convenient if the `Apply` button was automatically applied even after changes are made to the pipeline module properties. You can change this from the application settings dialog, which is accessible from the *Edit* > *Settings* menu (*ParaView* > *Preferences…* on macOS). The setting is called `Auto Apply` . You can also change the `Auto Apply` state using the button from the toolbar.

### Applying filters

As per the data-flow paradigm, one creates pipelines with filters to transform data. Similar to the *Sources* menu, which allows us to create new data sources, there's a *Filters* menu that provides access to the large set of filters that are available in **ParaView**. If you peruse the items in this menu, some of them will be enabled, and some of them will be disabled. Filters that can work with the data type being produced by the sphere source are enabled, while others are disabled. You can click on any of the enabled filters to create a new instance of that filter type.

**Did You Know?**

To figure out why a particular filter doesn't work with the current source, simply move your mouse over the disabled item in the *Filters* menu. On Linux and Windows (not OS X, however), the status bar will provide a brief explanation of why that filter is not available.



Requires a point attribute array with 1 component(s)

For example, if you click on *Filters* > *Shrink*, it will create a filter that shrinks each of the mesh cells by a fixed factor. Exactly as before, when we created the sphere source, we see that the newly-created filter is given a new name, `Shrink1` , and is highlighted in the `Pipeline Browser` . The `Properties` panel is also updated to show the properties for this new filter, and the `Apply` button is highlighted to request that we accept the properties for the filter so that it can be executed and the result can be rendered. If you click back and forth between the `Sphere1` and `Shrink1` in the `Pipeline Browser` , you'll see the `Properties` panel and toolbars update, reflecting the state of the selected pipeline module. This is an important concept in **ParaView**. There's a notion of active pipeline module, called the *active source* . Several panels, toolbars, and menus will update based on the active source.

If you click `Apply` , as was the case before, the shrink filter will be executed and the resulting dataset will be generated and shown in the 3D view. `paraview` will also automatically hide the result from the `Sphere1` so that it is not shown in the view. Otherwise, the two datasets will overlap. This is reflected by the change of state for the *eyeball* icons in the `Pipeline Browser` next to each of the pipeline modules. You can show or hide results from any pipeline module by clicking on the eyeballs.

This simple workflow forms the basis of all the data analysis and visualization in **ParaView**. The process involves creating sources and filters, changing their parameters, and showing the generated result in one or more views. In the rest of this guide, we will cover various types of filters and data processing that you can do. We will also cover different types of views that can help you produce a wide array of 2D and 3D visualizations, as well as inspect your data and drill down into it.

**Common Errors**

Beginners often forget to hit the `Apply` button after creating sources or filters or after changing properties. This is one of the most common pitfalls for users new to the **ParaView** workflow.

### 1.1.5 Getting started with `pvpython`

While this section refers to `pvpython`, everything that we discuss here is applicable to `pvbatch` as well. Until we start looking into parallel processing, the only difference between the two executables is that `pvpython` provides an interactive shell wherein you can type your commands, while `pvbatch` expects the Python script to be specified on the command line argument.

#### `pvpython` scripting interface

**ParaView** provides a scripting interface to write scripts for performing the tasks that you could do using the GUI. The scripting interface can be accessed through Python, which is an interpreted programming language popular among the scientific community for its simplicity and its capabilities. While a working knowledge of Python will be useful for writing scripts with advanced capabilities, you should be able to follow most of the discussion in this book about **ParaView** scripting even without much Python exposure.

**ParaView** provides a `paraview` package with several Python modules that expose various functionalities. The primary scripting interface is provided by the `simple` module.

When you start `pvpython`, you should see a prompt in a terminal window as follows (with some platform specific differences).

```
Python 3.10.13 (main, Feb  9 2024, 16:19:38)
[Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can now type commands at this prompt, and **ParaView** will execute them. To bring in the **ParaView** scripting API, you first need to import the `simple` module from the `paraview` package as follows:

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

**Common Errors**

Remember to hit the Enter or Return key after every command to execute it. Any Python interpreter will not execute the command until Enter is hit.

If the module is loaded correctly, pvpython will present a prompt for the next command.

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

You can consider this as in the same state as when paraview was started (with some differences that we can ignore for now). The application is ready to ingest data and start processing.

### Understanding the visualization process

Let's try to understand the workflow by looking at the same use-case as we did in Section 1.1.4.

### Creating a source

In paraview, we created the data source by using the *Sources* menu. In the scripting environment, this maps to simply typing the name of the source to create.

```
>>> Sphere()
```

This will create the sphere source with a default set of properties. Just like with paraview, as soon as a new pipeline module is created, it becomes the *active source*.

Now, to show the active source in a view, try:

```
>>> Show()
>>> Render()
```

The Show call will prepare the display, while the Render call will cause the rendering to occur. In addition, a new window will popup, showing the result (Fig. 1.5). This is similar to the state after hitting Apply in the UI.

### Changing properties

To change the properties on the sphere source, you can use the SetProperties function.

```
# Set a single property on the active source.
>>> SetProperties(Radius=1.0)

# You can also set multiple properties.
>>> SetProperties(Center=[1, 0, 0], StartTheta=100)
```

Similar to the Properties panel, SetProperties affects the active source. To query the current value of any property on the active source, use GetProperty .

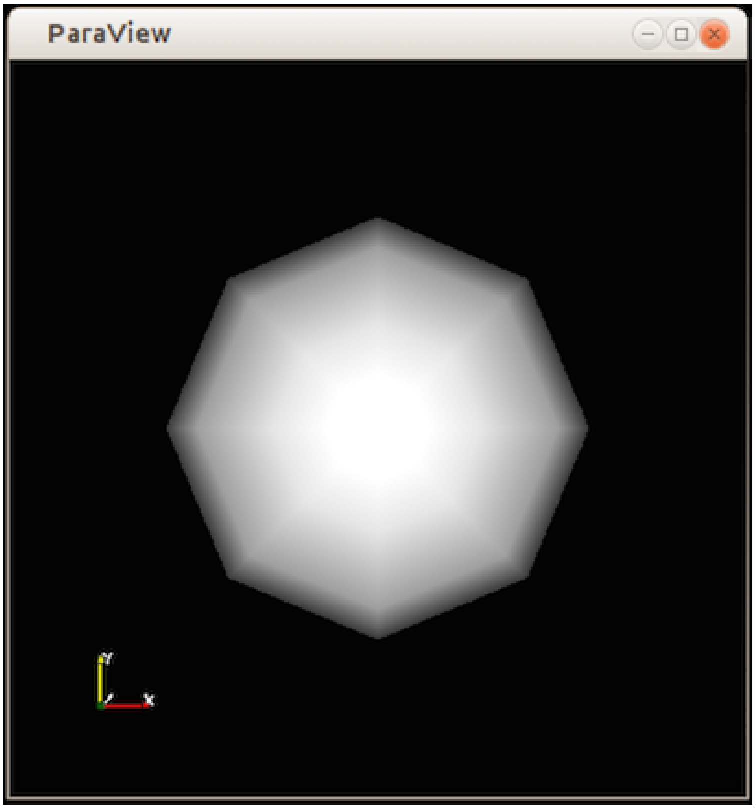


Fig. 1.5: Window showing result from the Python code.

```
>>> radius = GetProperty("Radius")
>>> print(radius)
1.0
>>> center = GetProperty("Center")
>>> print(center)
[1.0, 0.0, 0.0]
```

SetProperties and GetProperty functions serve the same function as the Properties section of the Properties panel – they allow you to set and introspect the pipeline module properties for the active source. Likewise, for the Display section of the panel, or the display properties, we have the SetDisplayProperties and GetDisplayProperty functions.

```
>>> SetDisplayProperties(Opacity=0.5)
>>> GetDisplayProperty("Opacity")
0.5
```

**Common Errors**

Note how the property names for the SetProperties and SetDisplayProperties functions are not enclosed in double-quotes, while those for the GetProperty and GetDisplayProperty methods are.

In paraview, every time you hit Apply or change a display property, the UI automatically re-renders the view. In the scripting environment, you have to do this manually by calling the Render function every time you want to re-render and look at the updated result.

fixme{we're missing blurb about reset camera}.

## Applying filters

Similar to creating a source, to apply a filter, you can simply create the filter by name.

```
# Create the `Shrink' filter and connect it to the active source
# which is the `Sphere' instance.
>>> Shrink()
# As soon as the Shrink filter is created, it will now become the new active
# source. All methods acting on active source now act on this filter instance
# and not the Sphere instance created earlier.

# Show the resulting data and render it.
>>> Show()
>>> Render()
```

If you tried the above script, you'll notice the result isn't exactly what we expected. For some reason, the shrank cells are not visible. This is because we missed one stage: In paraview, the UI was smart enough to automatically hide the input dataset for the newly created filter after we hit apply. In the scripting interface, such operations are the user's responsibility. We should have hidden the sphere source from the view. We can use the Hide method, the counterpart of Show , to hide the active source. But, now we have a problem – when we created the shrink filter, we changed the active source to be the shrink instance. Luckily, all the functions we discussed so far can take an optional first argument, which is the source or filter instance on which to operate. If provided, that instance is used instead of the active source. The solution is as follows:

```
# Get the input property for the active source, i.e. the input for the shrink.
>>> shrinksInput = GetProperty("Input")

# This is indeed the sphere instance we created earlier.
>>> print(shrinksInput)
<paraview.servermanager.Sphere object at 0x11d731e90>

# Hide the sphere instance explicitly.
>>> Hide(shrinksInput)

# Re-render the result.
>>> Render()
```

Alternatively, you could also get/set the active source using the `GetActiveSource` and `SetActiveSource` functions.

```
>>> shrinkInstance = GetActiveSource()
>>> print(shrinkInstance)
<paraview.servermanager.Shrink object at 0x11d731ed0>

# Get the input property for the active source, i.e. the input
# for the shrink.
>>> sphereInstance = GetProperty("Input")

# This is indeed the sphere instance we created earlier.
>>> print(sphereInstance)
<paraview.servermanager.Sphere object at 0x11d731e90>

# Change active source to sphere and hide it.
>>> SetActiveSource(sphereInstance)
>>> Hide()

# Now restore the active source back to the shrink instance.
>>> SetActiveSource(shrinkInstance)

# Re-render the result
>>> Render()
```

The result is shown in Fig. 1.6 .

`SetActiveSource` has same effect as changing the pipeline module, highlighted in the `Pipeline Browser` , by clicking on a different module.

### Alternative approach

Here's another way of doing something similar to what we did in the previous section for those familiar with Python and/or object-oriented programming. It's totally okay to stick with the previous approach.

```
>>> from paraview.simple import *
>>> sphereInstance = Sphere()
>>> sphereInstance.Radius = 1.0
>>> sphereInstance.Center[1] = 1.0
>>> print(sphereInstance.Center)
[0.0, 1.0, 0.0]
```

(continues on next page)

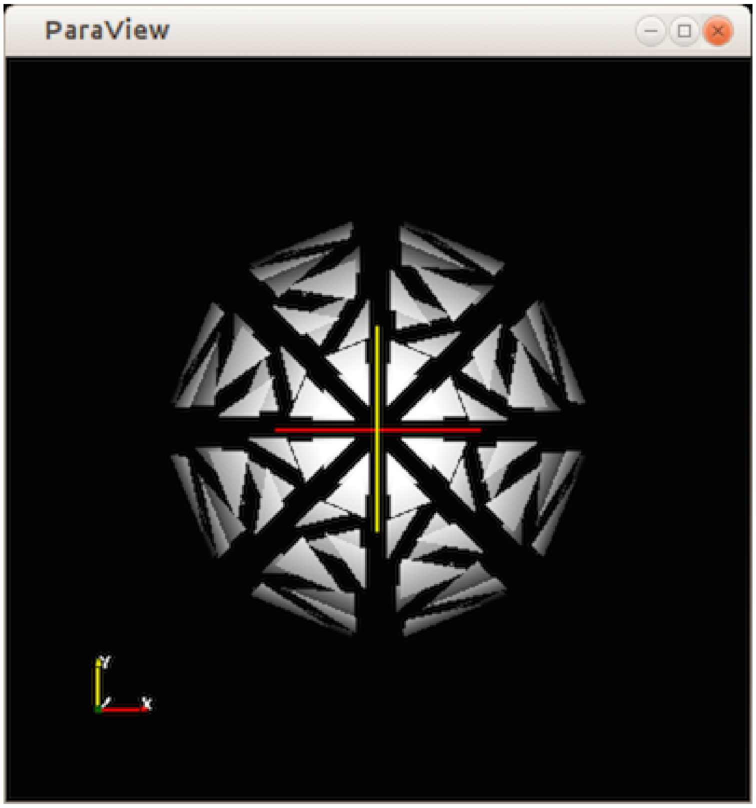


Fig. 1.6: Window showing result from the Python code after applying the shrink filter.

(continued from previous page)

```
>>> sphereDisplay = Show(sphereInstance)
>>> view = Render()
>>> sphereDisplay.Opacity = 0.5

# Render function can take in an optional view argument, otherwise it
# will simply use the active view.
>>> Render(view)

>>> shrinkInstance = Shrink(Input=sphereInstance,
                            ShrinkFactor=1.0)
>>> print(shrinkInstance.ShrinkFactor)
1.0
>>> Hide(sphereInstance)
>>> shrinkDisplay = Show(shrinkInstance)
>>> Render()
```

### Updating the pipeline

When changing properties on the Properties panel in paraview, we noticed that the algorithm doesn't re-execute until you hit Apply . In reality, Apply isn't what's actually triggering the execution or the updating of the processing pipeline. What happens is that Apply updates the parameters on the pipeline module and causes the view to render. If the output of the pipeline module is visible in the view, or if the output of any filter connected to it downstream is visible in the view, **ParaView** will determine that the data rendered is obsolete and request the pipeline to re-execute. It implies that if that pipeline module (or any of the filters downstream from it) is not visible in the view, **ParaView** will have no reason to re-execute the pipeline, and the pipeline module will not be be updated. If, later on, you do make this module visible in the view, **ParaView** will automatically update and execute the pipeline. This is often referred to as *demand-driven pipeline execution* . It makes it possible to avoid unnecessary module executions.

In paraview, you can get by without ever noticing this since the application manages pipeline updates automatically. In pvpython too, if your scripts are producing renderings in views, you'd never notice this as long as you remember to call Render . However, you may want to write scripts to produce transformed datasets or to determine data characteristics. In such cases, since you may never create a view, you'll never be seeing the pipeline update, no matter how many times you change the properties.

Accordingly, you must use the UpdatePipeline function. UpdatePipeline updates the pipeline connected to the active source (or only until the active source, i.e., anything downstream from it, won't be updated).

```
>>> from paraview.simple import *
>>> sphere = Sphere()

# Print the bounds for the data produced by sphere.
>>> print(sphere.GetDataInformation().GetBounds())
(1e+299, -1e+299, 1e+299, -1e+299, 1e+299, -1e+299)
# The bounds are invalid -- no data has been produced yet.

# Update the pipeline explicitly on the active source.
>>> UpdatePipeline()

# Alternative way of doing the same but specifying the source
# to update explicitly.
>>> UpdatePipeline(proxy=sphere)

# Let's check the bounds again.
>>> sphere.GetDataInformation().GetBounds()
(-0.48746395111083984, 0.48746395111083984, -0.48746395111083984, 0.48746395111083984, -
↪0.5, 0.5)

# If we call UpdatePipeline() again, this will have no effect since
# the pipeline hasn't been modified, so there's no need to re-execute.
>>> UpdatePipeline()
>>> sphere.GetDataInformation().GetBounds()
(-0.48746395111083984, 0.48746395111083984, -0.48746395111083984, 0.48746395111083984, -
↪0.5, 0.5)

# Now let's change a property.
>>> sphere.Radius = 10

# The bounds won't change since the pipeline hasn't re-executed.
>>> sphere.GetDataInformation().GetBounds()
(-0.48746395111083984, 0.48746395111083984, -0.48746395111083984, 0.48746395111083984, -
```

(continues on next page)

(continued from previous page)

```
↪0.5, 0.5)

# Let's update and see:
>>> UpdatePipeline()
>>> sphere.GetDataInformation().GetBounds()
(-9.749279022216797, 9.749279022216797, -9.749279022216797, 9.749279022216797, -10.0, 10.
↪0)
```

We will look at the `sphere.GetDataInformation` API in Section 1.3.3 in more detail.

For temporal datasets, `UpdatePipeline` takes in a time argument, which is the time for which the pipeline must be updated.

```
# To update to time 10.0:
>>> UpdatePipeline(10.0)

# Alternative way of doing the same:
>>> UpdatePipeline(time=10.0)

# If not using the active source:
>>> UpdatePipeline(10.0, source)
>>> UpdatePipeline(time=10.0, proxy=source)
```

## 1.1.6 Scripting in `paraview`

### The `Python Shell`

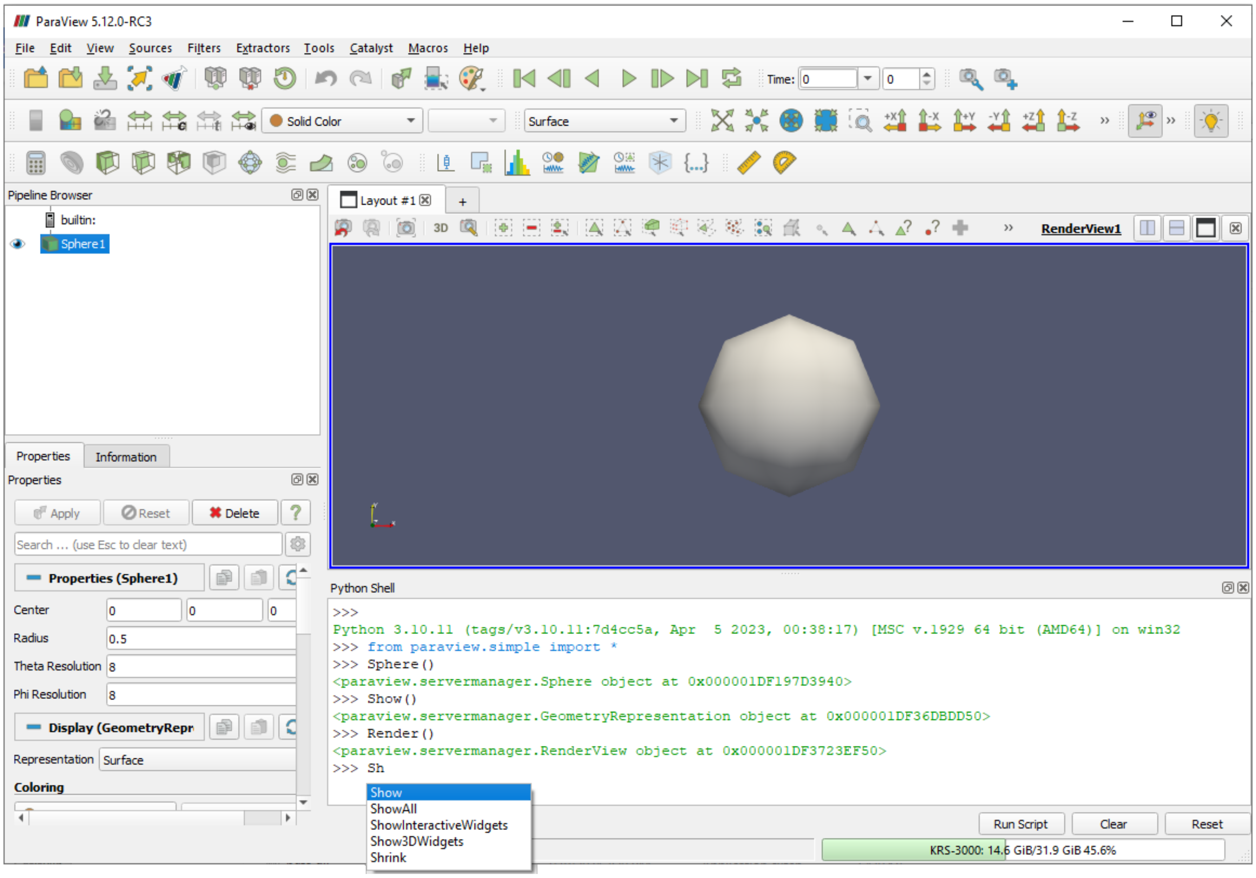
The `paraview` application also provides access to an internal shell, in which you can enter Python commands and scripts exactly as with `pvpython`. To access the Python shell in the GUI, use the *View* > *Python Shell* menu option. A dialog will pop up with a prompt exactly like `pvpython`. You can try inputting commands from the earlier section into this shell. As you type each of the commands, you will see the user interface update after each command, e.g., when you create the sphere source instance, it will be shown in the `Pipeline Browser` . If you change the active source, the `Pipeline Browser` and other UI components will update to reflect the change. If you change any properties or display properties, the `Properties` panel will update to reflect the change as well!

**Did You Know?**

The `Python Shell` in `paraview` supports auto-completion for functions and instance methods. Try hitting the `Tab` key after partially typing any command (as shown in Fig. 1.7).

### Tracing actions for scripting

This guide provides a fair overview of **ParaView**'s Python API. However, there will be cases when you just want to know how to complete a particular action or sequence of actions that you can do with the GUI using a Python script instead. To accomplish this, `paraview` supports tracing your actions in the UI as a Python script. Simply start tracing by clicking on *Tools* > *Start Trace*. `paraview` now enters a mode where all your actions (or at least those relevant for scripting) are monitored. Any time you create a source or filter, open data files, change properties and hit `Apply` , interact with the 3D scene, or save screenshots, etc., your actions will be monitored. Once you are done with the series of actions that you want to script, click *Tools* > *Stop Trace*. `paraview` will then pop up an editor window with the



Center 0 0 0

Radius 0.5

Theta Resolution 8

Phi Resolution 8

Representation Surface

```
>>>
Python 3.10.11 (tags/v3.10.11:7d4cc5a, Apr  5 2023, 00:38:17) [MSC v.1929 64 bit (AMD64)] on win32
>>> from paraview.simple import *
>>> Sphere()
<paraview.servermanager.Sphere object at 0x000001DF197D3940>
>>> Show()
<paraview.servermanager.GeometryRepresentation object at 0x000001DF36DBDD50>
>>> Render()
<paraview.servermanager.RenderView object at 0x000001DF3723EF50>
>>> Sh
```

Fig. 1.7: `Python Shell` in `paraview` provides access to the scripting.

generated trace. This will be the Python script equivalent for the actions you performed. You can now save this as a script to use for batch processing.

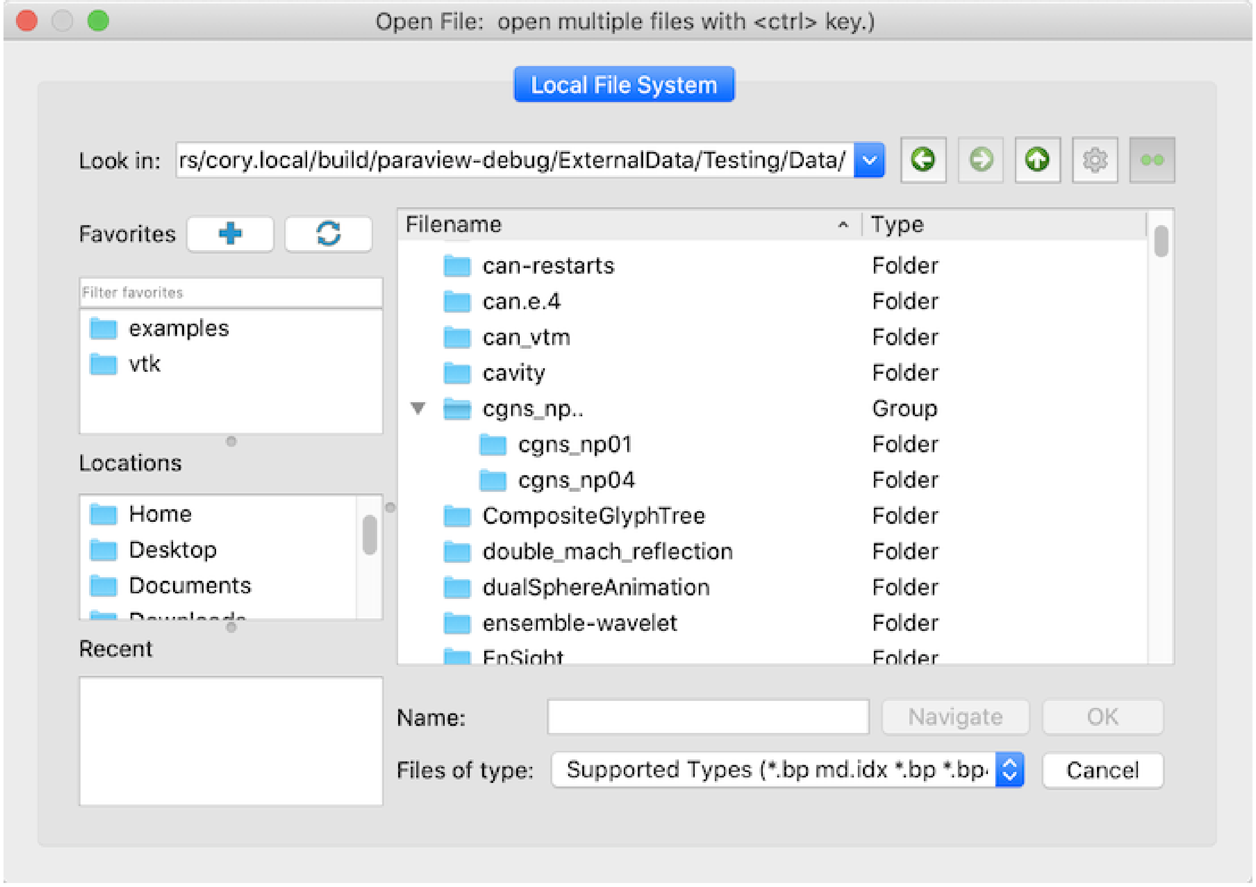
## 1.2 Loading Data

In a visualization pipeline, data sources bring data into the system for processing and visualization. Sources, such as the `Sphere` source (accessible from the `Sources` menu in `paraview`), programmatically create datasets for processing. Another type of data sources are readers. Readers can read data written out in disk files or other databases and bring it into **ParaView** for processing. **ParaView** includes readers that can read several of the commonly used scientific data formats. It's also possible to write plugins that add support for new or proprietary file formats.

**ParaView** provides several sample datasets for you to get started. You can download an archive with several types of data files from the download page at https://www.paraview.org/download under the Data section.

### 1.2.1 Opening data files in `paraview`

To open a data file in `paraview`, you use the `Open File` dialog. This dialog can be accessed from the *File* > *Open* menu or by using the button in the `Main Controls` toolbar. You can also use the keyboard shortcut CTRL + O (or + O) to open this dialog.



Look in: rs/cory.local/build/paraview-debug/ExternalData/Testing/Data/

| Filename | Type |
|---|---|
| can-restarts | Folder |
| can.e.4 | Folder |
| can_vtm | Folder |
| cavity | Folder |
| cgns_np.. | Group |
| cgns_np01 | Folder |
| cgns_np04 | Folder |
| CompositeGlyphTree | Folder |
| double_mach_reflection | Folder |
| dualSphereAnimation | Folder |
| ensemble-wavelet | Folder |
| EnSight | Folder |

Name:

Files of type: Supported Types (*.bp md.idx *.bp *.bp

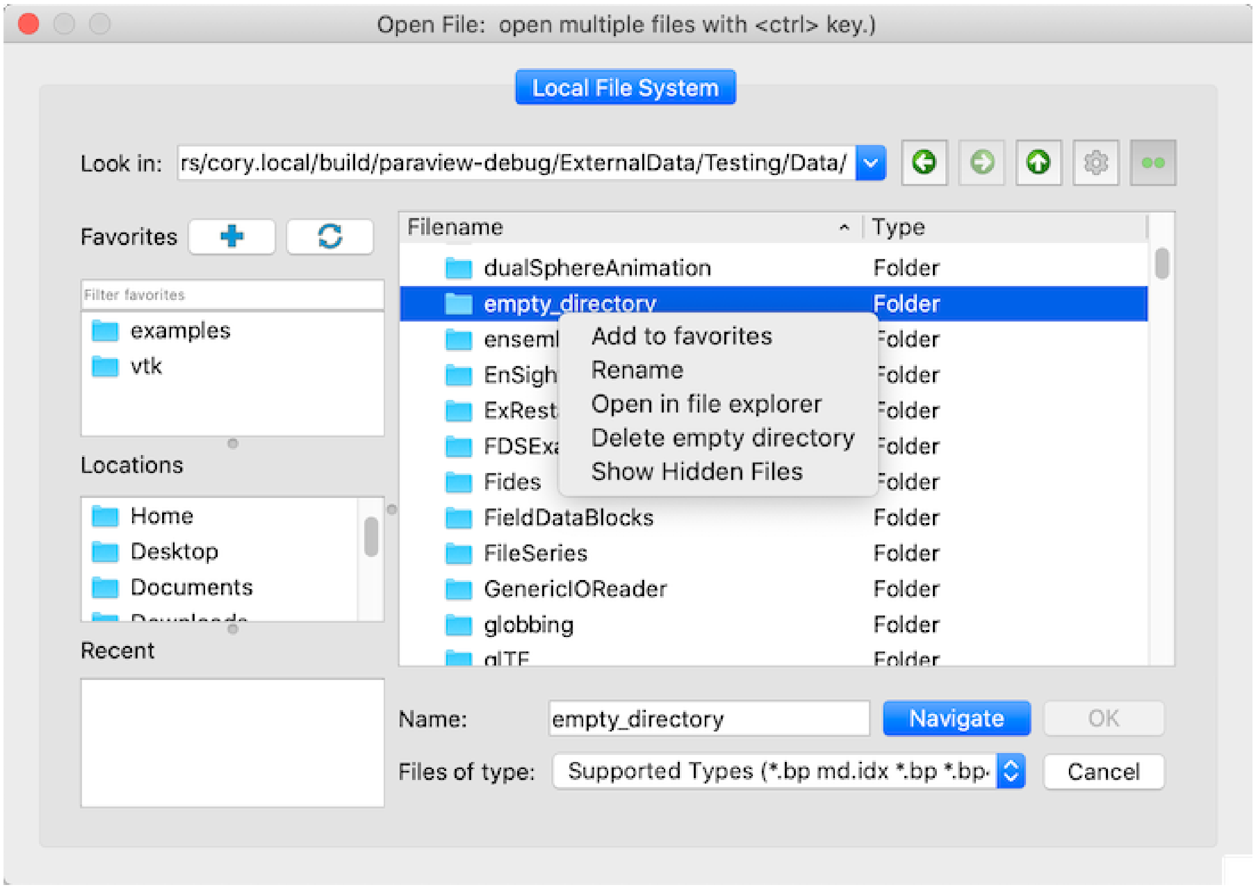
Fig. 1.8: `Open File` dialog in `paraview` for opening data (and other) files.

The `Open File` dialog allows you to browse the file system on the data processing nodes. This will become clear when we look at using **ParaView** for remote visualization. While several of the UI elements in this dialog are obvious such as navigating up the current directory, creating a new directory, and navigating back and forth between directories, which can all be done with the standard system shortcuts like CTRL + N (or  + N) to create a directory or Alt + ↑ to go to the parent directory, there are a few things to note.

- The Favorites pane contains a custom list of favorite directories that can be customized using the buttons above it, which respectively adds the current directory to the favorites or removes all the favorites. Another way to change which directories are in the favorites is to use the right-click context menus in the list of files. Right-clicking any directory in the main list will display a menu with the option to add it to the favorites. To remove edit an exising favorite, right-click or ctrl-click the favorite. A context menu appears which lets you remove the favorite or rename it.
- The Locations pane shows some platform-specific common locations such as the home directory and desktop. External drives will also appear in this list.
- The Recent Directories pane shows a few of the most recently used directories.

You can browse to the directory containing your datasets and either select the file and hit Ok or simply double click on the file to open it. You can also select multiple files using the CTRL (or ) key. This will open each of the selected files separately.

Right-clicking the files list will display a few options, depending on what was right-clicked.



Look in: rs/cory.local/build/paraview-debug/ExternalData/Testing/Data/

| Filename | Type |
|---|---|
| dualSphereAnimation | Folder |
| empty_directory | Folder |
| enseml | older |
| EnSigh | older |
| ExRest | older |
| FDSExa | older |
| Fides | older |
| FieldDataBlocks | Folder |
| FileSeries | Folder |
| GenericIOReader | Folder |
| globbing | Folder |
| glTF | Folder |

Name: empty_directory

Files of type: Supported Types (*.bp md.idx *.bp *.bp·

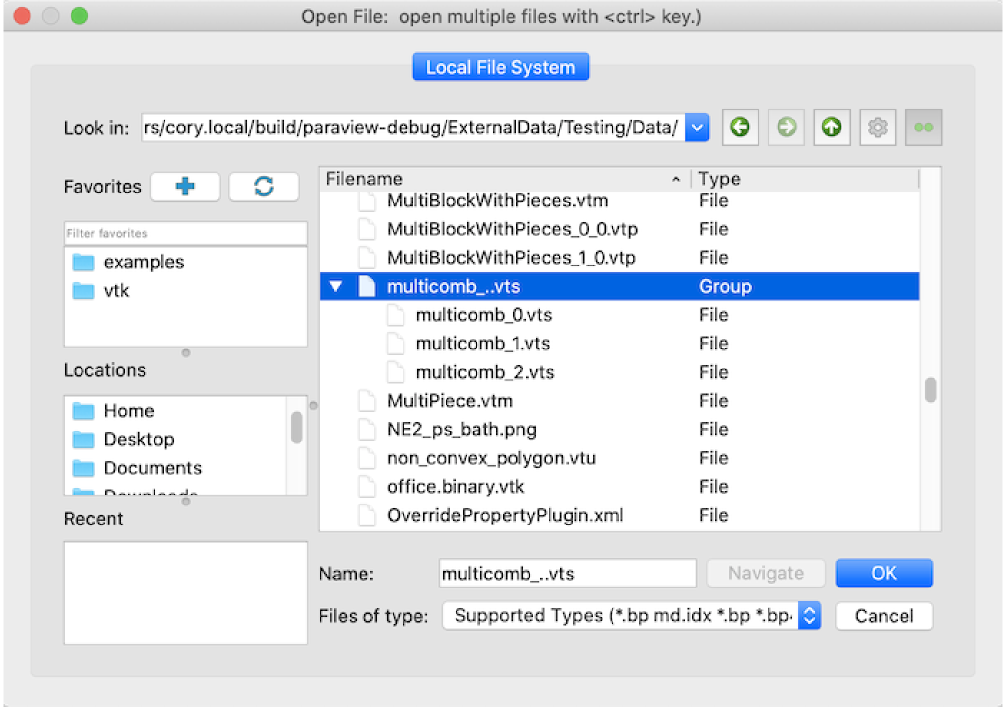
- Selecting a directory adds the Add to favorites option which adds the selected directory to the Favorites pane on the left.
- Selecting a file or a directory adds the Rename option.
- The Open in file explorer is also always present and either opens in the system file explorer the selected directory if one was right-clicked, or opens the current directory if a file or nothing was selected.

- Selecting an empty directory provides the `Delete empty directory` option.
- The `Show Hidden Files` is always visible, and can be checked to display the hidden files and directories.

The `Look in:` combobox shows the current path in the Open File dialog. Clicking on the combobox down arrow will show the parent directories of the current directory while clicking in the path line edit itself allows you to edit the path. Navigation buttons showing arrows are to the right of the combo box. These enable navigating through previously visited directories as well as moving up to the parent of the current directory. The  button to the right of the arrows toggles on or off additional columns in the dialog that show file size and last modified date.

When a file is opened, `paraview` will create a reader instance of the type suitable for the selected file based on its extension. The reader will simply be another pipeline module, similar to the source we created in Section 1.1. From this point forward, the workflow will be the same as we discussed in Section 1.1.4 : You adjust the reader properties, if needed, and hit `Apply` . `paraview` will then read the data from the file and render it in the view.

If you selected multiple files using the `CTRL` (or ) key, `paraview` will create multiple reader modules. When you hit `Apply` , all of the readers will be executed, and their data will be shown in the view.



Look in: rs/cory.local/build/paraview-debug/ExternalData/Testing/Data/

| Filename | Type |
|---|---|
| MultiBlockWithPieces.vtm | File |
| MultiBlockWithPieces_0_0.vtp | File |
| MultiBlockWithPieces_1_0.vtp | File |
| ▼ multicomb_..vts | Group |
| multicomb_0.vts | File |
| multicomb_1.vts | File |
| multicomb_2.vts | File |
| MultiPiece.vtm | File |
| NE2_ps_bath.png | File |
| non_convex_polygon.vtu | File |
| office.binary.vtk | File |
| OverridePropertyPlugin.xml | File |

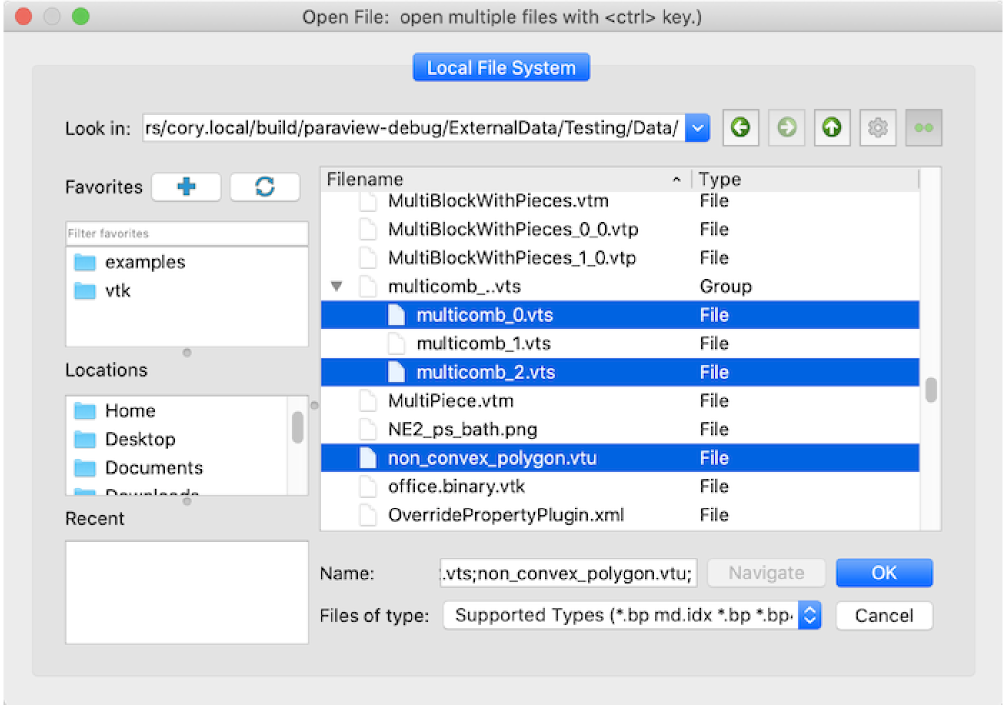
Name: multicomb_..vts

Files of type: Supported Types (*.bp md.idx *.bp *.bp·

By default, `paraview` groups files that appear to define a time series. These file series have names that contain a number sequence where the number defines the order of the files in time. `paraview` can be told not to do this by clicking on the  button at the top right of the `Open File` dialog.

**Did you know?**

This ability to hit the `Apply` button once to accept changes on multiple readers applies to other pipeline modules, including sources and filters. In general, you can change properties for multiple modules in a pipeline, and hit `Apply` to accept all of the changes at once. It is possible to override this behavior from **ParaView**'s' `Settings` dialog using the `Auto Apply Active Only` setting under the `General` tab. When this setting is enabled, only the selected source in the `Pipeline Browser` will be updated.



Look in: rs/cory.local/build/paraview-debug/ExternalData/Testing/Data/

| Filename | Type |
|---|---|
| MultiBlockWithPieces.vtm | File |
| MultiBlockWithPieces_0_0.vtp | File |
| MultiBlockWithPieces_1_0.vtp | File |
| multicomb_..vts | Group |
| multicomb_0.vts | File |
| multicomb_1.vts | File |
| multicomb_2.vts | File |
| MultiPiece.vtm | File |
| NE2_ps_bath.png | File |
| non_convex_polygon.vtu | File |
| office.binary.vtk | File |
| OverridePropertyPlugin.xml | File |

Name: .vts;non_convex_polygon.vtu;

Files of type: Supported Types (*.bp md.idx *.bp *.bp

Fig. 1.9: The `Open File` dialog can be used to select a temporal file series (top) or select multiple files to open separately (bottom).

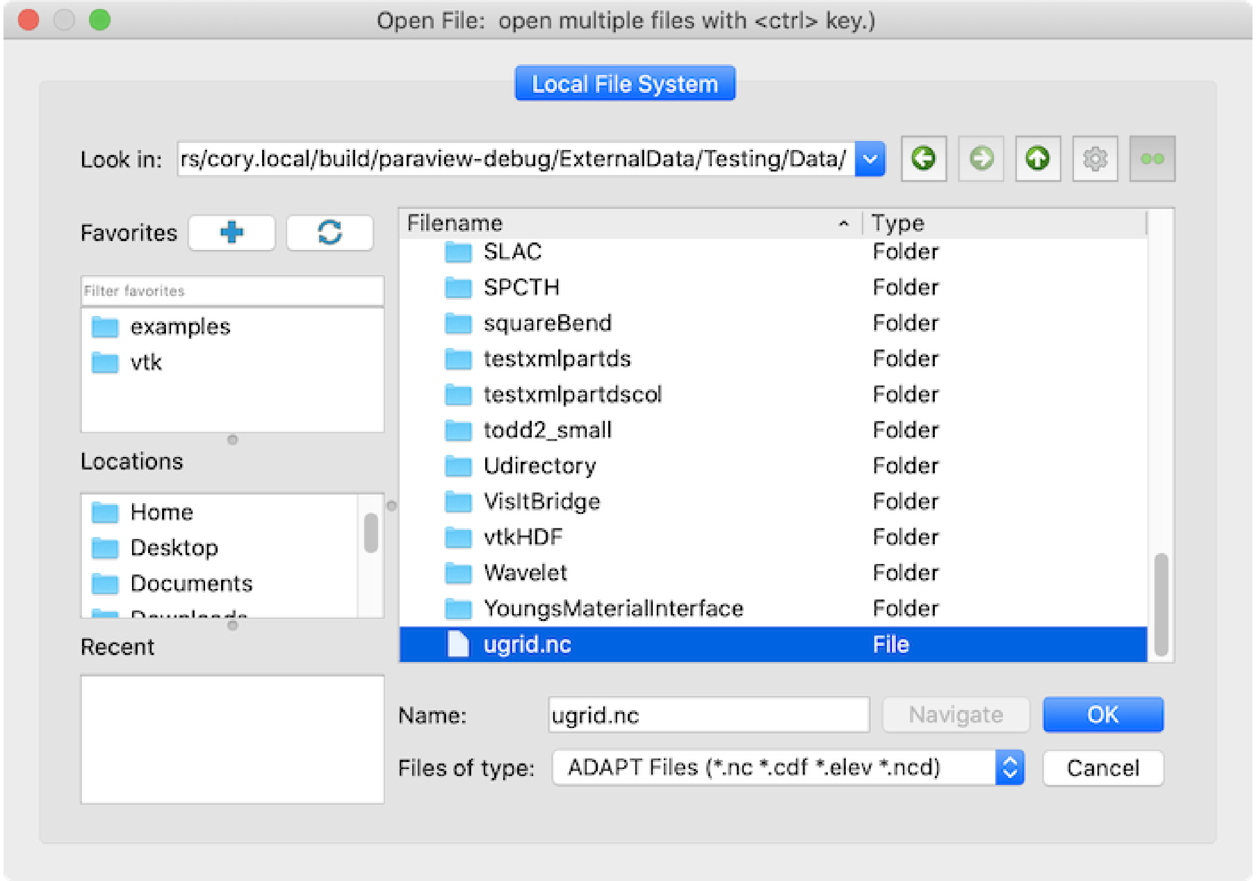
### Reader selection depending on the selected file types

When selecting files, the current category for the "Files of type" field (Fig. 1.10) changes the way the reader will be chosen.

- If the "Supported Files" value is selected, if only one reader is available for the type of file, it will be selected automatically. If multiple readers can be used, the "Open Data With…" dialog (Fig. 1.11) will appear to choose which reader to use. Clicking the "Set reader as default" button tells ParaView to automatically use this reader when one of the file name patterns it says it can support matches a file that could otherwise be opened by multiple readers. When a reader is set as default, it will be used automatically for files that match its patterns.
- If the "All Files" value is selected, the same dialog will be displayed with all the existing readers. If you picked an incorrect reader, however, you'll get error messages either when the reader module is instantiated or after you hit `Apply`. In either case, you can simply `Delete` the reader module and try opening the file again, this time choosing a different reader. If you can click the `Set reader as default` button, a small window with a line edit will be displayed where you can set the custom pattern to use for this reader (Fig. 1.12). This is a standard wildcard pattern, and multiple patterns can be used by separating them with spaces.
- If a specific reader is selected, for example "PNG Image Files", this reader will be always be used automatically, even if other readers would be available or if the file is matching a pattern present in the "Default reader details" settings.

Error messages in `paraview` are shown in the `Output Messages` window (Fig. 1.13). It is accessible from the *View > Output Messages* menu. Whenever there's a new error message, `paraview` will automatically pop open this window and raise it to the top (as long as the `Always open for new messages` option is set. This window can be attached, or docked, in the main window so that it is visible with the other user interface elements without covering them up.

The default readers settings can be seen and modified in the *Edit > Settings* menu, and `IO` tab (Fig. 1.14).


Open File:  open multiple files with <ctrl> key.)

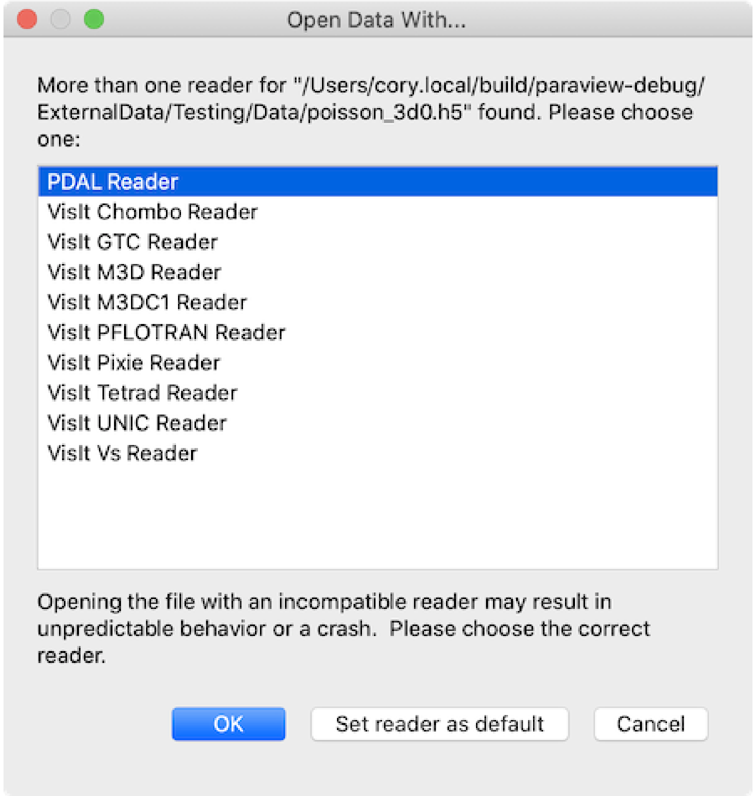
Look in: rs/cory.local/build/paraview-debug/ExternalData/Testing/Data/

| Filename | Type |
|---|---|
| SLAC | Folder |
| SPCTH | Folder |
| squareBend | Folder |
| testxmlpartds | Folder |
| testxmlpartdscol | Folder |
| todd2_small | Folder |
| Udirectory | Folder |
| VisItBridge | Folder |
| vtkHDF | Folder |
| Wavelet | Folder |
| YoungsMaterialInterface | Folder |
| ugrid.nc | File |

Name: ugrid.nc

Files of type: ADAPT Files (*.nc *.cdf *.elev *.ncd)

Fig. 1.10: File dialog “Files of type” to filter which files to show and to change how the reader will be chosen.

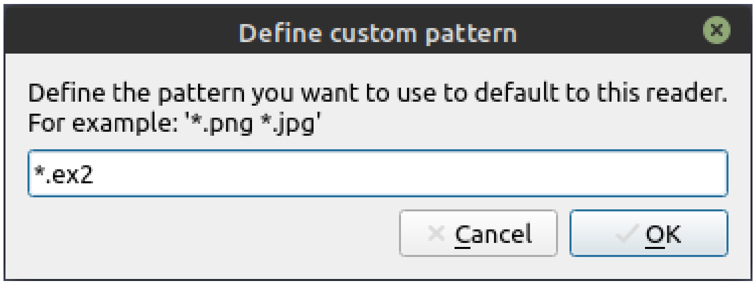


More than one reader for "/Users/cory.local/build/paraview-debug/ExternalData/Testing/Data/poisson_3d0.h5" found. Please choose one:

- PDAL Reader
- VisIt Chombo Reader
- VisIt GTC Reader
- VisIt M3D Reader
- VisIt M3DC1 Reader
- VisIt PFLOTRAN Reader
- VisIt Pixie Reader
- VisIt Tetrad Reader
- VisIt UNIC Reader
- VisIt Vs Reader

Opening the file with an incompatible reader may result in unpredictable behavior or a crash. Please choose the correct reader.

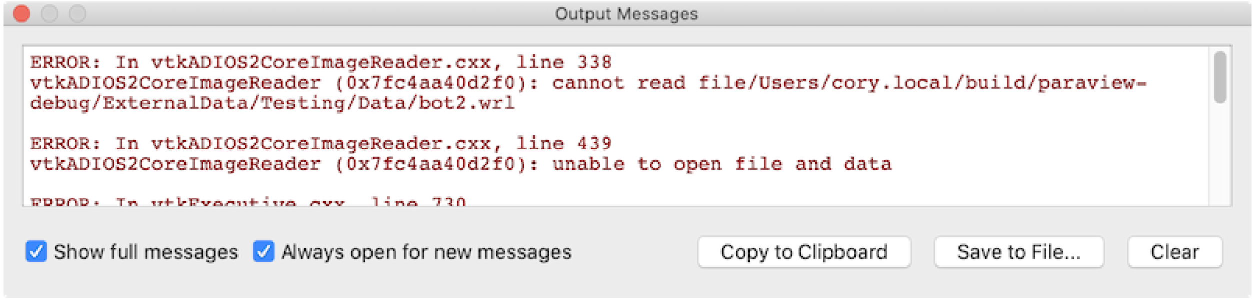
Fig. 1.11: `Open Data With...` dialog shown to manually choose the reader to use for a file with multiple available readers.



Define the pattern you want to use to default to this reader.
For example: '*.png *.jpg'

*.ex2

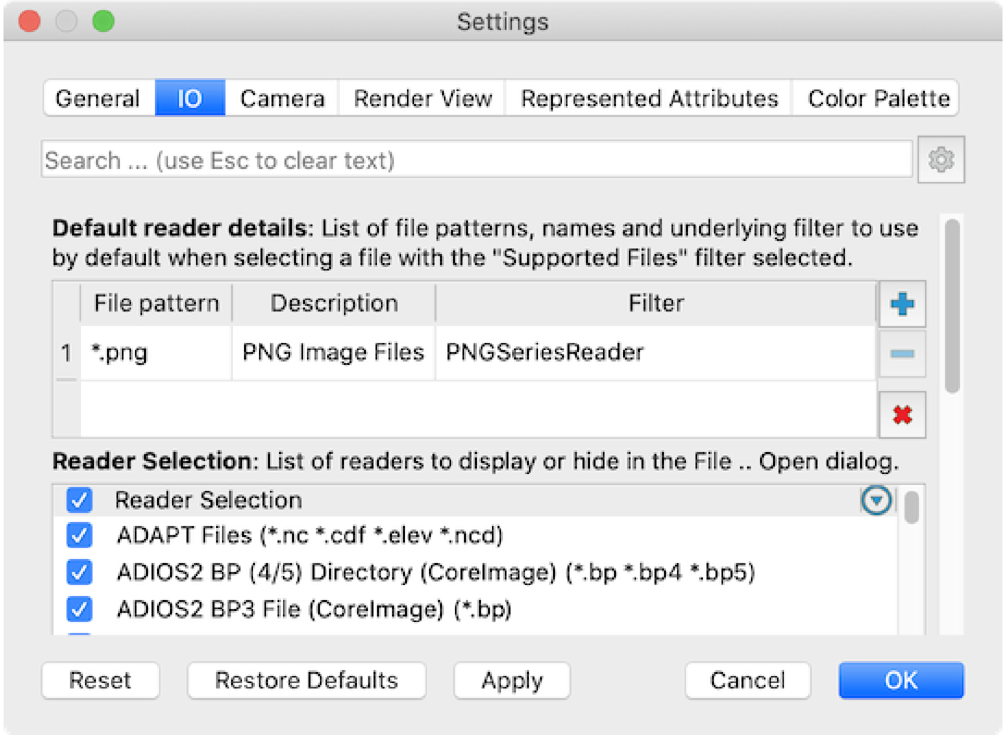
Fig. 1.12: The `Default reader details` setting is used to define file name patterns and which reader should read files that match those patterns.



```
ERROR: In vtkADIOS2CoreImageReader.cxx, line 338
vtkADIOS2CoreImageReader (0x7fc4aa40d2f0): cannot read file/Users/cory.local/build/paraview-
debug/ExternalData/Testing/Data/bot2.wrl

ERROR: In vtkADIOS2CoreImageReader.cxx, line 439
vtkADIOS2CoreImageReader (0x7fc4aa40d2f0): unable to open file and data

ERROR: In vtkExecutive.cxx, line 730
```

☑ Show full messages ☑ Always open for new messages

Fig. 1.13: The `Output Messages` window is used to show errors, warnings, and other messages raised by the application.



**Default reader details:** List of file patterns, names and underlying filter to use by default when selecting a file with the "Supported Files" filter selected.

| | File pattern | Description | Filter |
|---|---|---|---|
| 1 | *.png | PNG Image Files | PNGSeriesReader |

**Reader Selection:** List of readers to display or hide in the File .. Open dialog.

- ☑ Reader Selection
- ☑ ADAPT Files (*.nc *.cdf *.elev *.ncd)
- ☑ ADIOS2 BP (4/5) Directory (CoreImage) (*.bp *.bp4 *.bp5)
- ☑ ADIOS2 BP3 File (CoreImage) (*.bp)

Fig. 1.14: Default reader settings.

### Handling temporal file series

Most datasets produced by scientific simulation runs are temporal in nature. File formats differ in how this information is saved in the file. While several file formats support saving multiple timesteps in the same file, others save them out as a sequence of files, known as a file series

The `Open File` dialog automatically detects file series and shows them as a grouped element, as shown in Fig. 1.9. To load the file series, simply select the group, and hit `Ok` . You can also open a single file in the series as a regular file. To do so, open the file group and select the file you want to open.

`paraview` automatically detects several of the commonly-known file naming patterns used for indicating a file series. These include:

| | | | |
|---|---|---|---|
| `fooN.vtk` | `fooN.vtk` | `Nfoo.vtk` | `foo.vtk.N` |
| `foo_N.vtk` | `foo.N.vtk` | `N.foo.vtk` | `foo.vtksN` |

where *foo* could be any filename, *N* is a numeral sequence (with any number of leading zeros), and *vtk* could be any extension.

Sometimes this grouping of files with names that follow a numeral sequence into a time series is not desired. The `Open File` dialog has a toggle button that controls whether these file sequences are interpreted as time series. It defaults to on, but if toggled off, then `paraview` will not group file series until the button is toggled back on.

ParaView also supports a meta file format based on JSON. This format has support for specifying time values. The format (currently version 1.0) looks like the following

```
{
  "file-series-version" : "1.0",
```

(continues on next page)

(continued from previous page)

```
  "files" : [
    { "name" : "foo1.vtk", "time" : 0 },
    { "name" : "foo2.vtk", "time" : 5.5 },
    { "name" : "foo3.vtk", "time" : 11.2 }
  ]
}
```

Usually, a reader supporting `extension` will also supports `extension.series` as the file series meta file.

### Dealing with time

When you open a dataset with time, either as a file series or in a file format that natively supports time, `paraview` will automatically set up an animation for you so that you can play through each of the time steps in the dataset by using the  button on the `VCR Controls` toolbar (Fig. 1.15). You can change or modify this animation and further customize it, as discussed in Chapter Section 1.7.

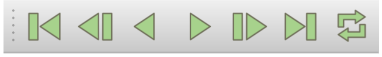


Fig. 1.15: `VCR Controls` toolbar for interacting with an animation.

### Reopening previously opened files

`paraview` remembers most recently opened files (or file series). Simply use the *File* > *Recent Files* menu. `paraview` also remembers the reader type selected for files with unknown extensions or for occasions when multiple reader choices were available. Lastly, ParaView also records the server connection used to load a particular file. If you are connected to one server and access a file through the *Recent Files* menu that was on another server, `paraview` will ask if you want to disconnect from the current server and connect to the server on which the recent file resides.

### Opening files using command line options

`paraview` provides a command line option that can be used to open datasets on startup.

```
> paraview --data=.../ParaViewData/Data/can.ex2
```

This is equivalent to opening a can.ex2 data file from the `Open File` dialog. The same set of follow-up actions happen. For example, `paraview` will try to locate a reader for the file, create that reader, and wait for you to hit `Apply` .

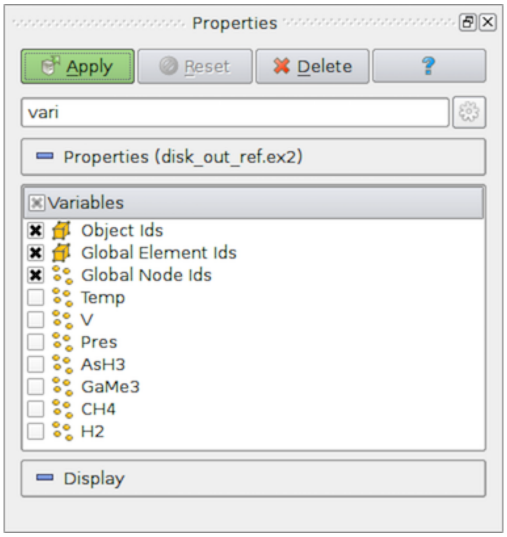
To open a file series, simply replace the numbers in the file name sequence by a `.` For example, to open a file series named `my0.vtk`, `my1.vtk` … `myN.vtk`, use `my..vtk`.

```
> paraview --data=.../ParaViewData/Data/my..vtk
```

### Common properties for readers

**ParaView** uses different reader implementations for different file formats. Each of these have different properties available to you to customize how the data is read and can vary greatly depending on the capabilities of the file format itself or the particular reader implementation. Let's look at some of the properties commonly available in readers.

### Selecting data arrays



vari

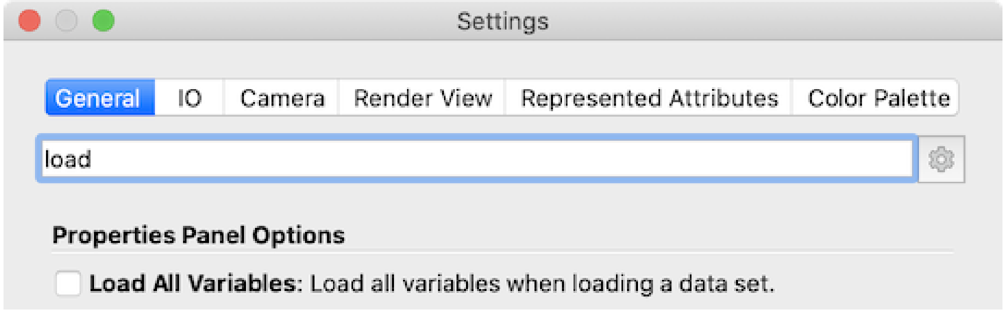
☒ Variables
☒ Object Ids
☒ Global Element Ids
☒ Global Node Ids
☐ Temp
☐ V
☐ Pres
☐ AsH3
☐ GaMe3
☐ CH4
☐ H2

Fig. 1.16: Array selection widget for selecting array to load from a data file.

One of the most common properties on readers is one that allows you to select the data arrays to be loaded, be they cell-centered, point-centered, or otherwise. Often times, loading only the data arrays you know you are going to use in the visualization will save memory, as well as processing time, since the reader does not have to read in those data arrays, and the filters in the pipeline do not have to process them.

**Did you know?**

You can change `paraview`'s default behavior to load all available data arrays by selecting the `Load All Variables` checkbox under `Settings/Properties Panel Options/Advanced`.



load

Properties Panel Options

☐ **Load All Variables:** Load all variables when loading a data set.

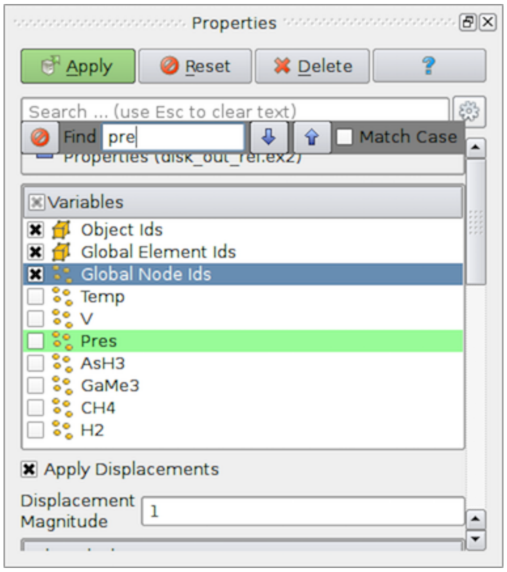
The user interface for selecting the arrays to load is simply a list with the names of the arrays and a checkbox indicating whether that array is to be loaded or not (Fig. 1.16). Icons, such as  and  are often used in this widget to give you an indication of whether the array is cell-centered or point-centered, respectively.

If you initially de-select an array, but then as you're setting up your visualization pipeline realize that you need that data array, you can always go back to the `Properties` page for the reader by making the reader active in the `Pipeline Browser` and then changing the array selection. **ParaView** will automatically re-execute any processing pipeline set up on the reader with this new data array.

**Common Errors**

Remember to hit `Apply` (or use `Auto Apply` ) after changing the array selection for the change to take effect.

Sometimes the list of data arrays can get quite large, and it can become cumbersome to find the array for which you are looking. To help with such situations, `paraview` provides a mechanism to search lists. Click inside the widget to make it get the *focus*. Then type `CTRL` + `F` (or  + `F`) to get a search widget. Now you can type in the text to search. Matching rows will be highlighted (Fig. 1.17).



Find: pre

☒ Object Ids
☒ Global Element Ids
☒ Global Node Ids
☐ Temp
☐ V
☐ Pres
☐ AsH3
☐ GaMe3
☐ CH4
☐ H2

☒ Apply Displacements

Displacement Magnitude: 1

Fig. 1.17: To search through large lists in `paraview`, you can use `CTRL` + `F`.

**Did you know?**

The ability to search for items in an array selection widget also applies to other list and tree widgets in the `paraview` UI. Whenever you see a widget with a large number of entries in a list, table, or tree fashion, try using `CTRL` + `F` (or  + `F`).

## 1.2.2 Opening data files in `pvpython`

To open data files using the scripting interface, **ParaView** provides the `OpenDataFile` function.

```
>>> reader = OpenDataFile(".../ParaViewData/Data/can.ex2")
>>> if reader:
...   print("Success")
... else:
...   print("Failed")
...
```

`OpenDataFile` will try to determine an appropriate reader based on the file extension, just like `paraview`. If no reader is determined, `None` is returned. If multiple readers can open the file, however, `OpenDataFile` simply picks the first reader. If you explicitly want to create a specific reader, you can always create the reader by its name, similar to other sources and filters.

```
>>> reader = ExodusIIReader(FileName=".../ParaViewData/Data/can.ex2")
```

To find out information about the reader created and the properties available on it, you can use the `help` function.

```
>>> reader = ExodusIIReader(FileName=".../ParaViewData/Data/can.ex2")
>>> help(reader)
Help on ExodusIIReader in module paraview.servermanager object:

class ExodusIIReader(ExodusIIReaderProxy)
 |  The Exodus reader loads
 |  Exodus II files and produces an unstructured grid output.
 |  The default file extensions are .g, .e, .ex2, .ex2v2,
 |  .exo, .gen, .exoII, .exii, .0, .00, .000, and .0000. The
 |  file format is described fully at:
 |  http://endo.sandia.gov/SEACAS/Documentation/exodusII.pdf.
 |  ...
 |
 |  -----------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  AnimateVibrations
 |      If this flag is on and HasModeShapes is also on, then
 |      this reader will report a continuous time range [0,1] and
 |      animate the displacements in a periodic sinusoid. If this
 |      flag is off and HasModeShapes is on, this reader ignores
 |      time. This flag has no effect if HasModeShapes is off.
 |
 |  ApplyDisplacements
 |      Geometric locations can include displacements. When this
 |      option is on, the nodal positions are 'displaced' by the
 |      standard exodus displacement vector. If displacements are
 |      turned 'off', the user can explicitly add them by applying
 |      a warp filter.
 |  ...
```

**Did you know?**

The `help` function can be used to get information about properties available on any source or filter instance. It not only lists the properties, but also provides information about how they affect the pipeline module. `help` can also be used on functions. For example:

```
>>> help(OpenDataFile)

Help on function OpenDataFile in module paraview.simple:

OpenDataFile(filename, \*\*extraArgs)
    Creates a reader to read the given file, if possible.
    This uses extension matching to determine the best reader
```

(continues on next page)

(continued from previous page)

```
    possible. If a reader cannot be identified, then this
    returns None.
```

### Handling temporal file series

Unlike `paraview`, `pvpython` does not automatically detect and load file series. There are two ways you can load a file series:

- You can explicitly list the filenames in the series and pass that to the `OpenDataFile` call.

```
# Create a list with the names of all the files in the file series in
# correct order.
>>> files = [".../Data/multicomb_0.vts",
             ".../Data/multicomb_1.vts",
             ".../Data/multicomb_2.vts"]
>>> reader = OpenDataFile(files)
```

- You can use globbing by inserting the wildcard * in the file name using the utility **paraview.util.Glob**. This utility runs *fnmatch* python package on the server's file system, so any pattern supported by fnmatch is supported and interpreted by this utility.

```
>>> import paraview.util
# Create a list of names of all the files in the file series.
>>> files = paraview.util.Glob(path = "multicomb_*.vts", rootDir = ".../Data")
>>> reader = OpenDataFile(files)
```

### Dealing with time

Similar to `paraview`, if you open a time series or a file with multiple timesteps, `pvpython` will automatically set up an animation for you to play through the timesteps.

```
>>> files = [".../Data/multicomb_0.vts",
             ".../Data/multicomb_1.vts",
             ".../Data/multicomb_2.vts"]
>>> reader = OpenDataFile(files)
>>> Show()
>>> Render()

# Get access to the animation scene.
>>> scene = GetAnimationScene()
# Now you use the API on the scene when doing things such as playing
# the animation, stepping through it, etc.

# This will simply play through the animation once and stop. Watch
# the rendered view after you hit `Enter.'
>>> scene.Play()
```

### Common properties on readers

### Selecting data arrays

For those properties on readers that allow you to control what to read in from the file such as point data arrays, cell data arrays, or data blocks, `paraview` uses a selection widget, as seen in Section 1.2.1. Likewise, `pvpython` provides an API that allows you to determine the available options and then select/deselect them.

The name of the property that allows you to make such selections depends on the reader itself. When in doubt, use the tracing capabilities in `paraview` (Section 1.1.6) to figure it out. You can also use `help` (Section 1.2.2).

`ExodusIIReader` has a `PointVariables` property that can be used to select the point data arrays to load. Let's use this as an example.

```
# Open an ExodusII data file.
>>> reader = OpenDataFile(".../Data/can.ex2")

# Alternatively, you can explicitly create the reader instance as:
>>> reader = ExodusIIReader(FileName = ".../Data/can.ex2")

# To query/print the current status for `PointVariables' property,
# we do what we would have done for any other property:
>>> print(GetProperty("PointVariables"))
['DISPL', 'VEL', 'ACCL']

# An alternative way of doing the same is as follows:
>>> print(reader.PointVariables)
['DISPL', 'VEL', 'ACCL']

# To set the property, simply set it to list containing the names to
# enable, e.g., if we want to read only the 'DISPL' array, we do
# the following:
>>> SetProperties(PointVariables=['DISPL'])

# Or using the alternative way for doing the same:
>>> reader.PointVariables = ['DISPL']

# Now, the new value for PointVariables is:
>>> print(reader.PointVariables)
['DISPL']

# To determine the array available, use:
>>> print(reader.PointVariables.Available)
['DISPL', 'VEL', 'ACCL']
# These are the arrays available in the file.
```

Changing `PointVariables` only changes the value on the property. The reader does not re-execute until a re-execution is requested either by calling `Render` or by explicitly updating the pipeline using `UpdatePipeline` .

```
>>> reader.PointVariables = ['DISPL', 'VEL', 'ACCL']

# Assuming that the reader is indeed the active source, let's update
# the pipeline:
>>> UpdatePipeline()
```

(continues on next page)

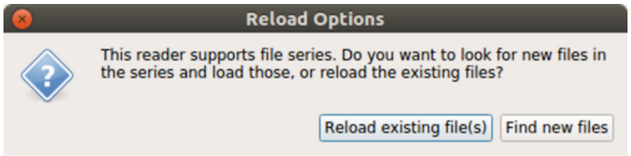
(continued from previous page)

```
# Or you can use the following form if you're unsure of the active
# source or just do not want to worry about it.
>>> UpdatePipeline(proxy=reader)

# Print the list of point arrays read in.
>>> print(reader.PointData[:])
[Array: ACCL, Array: DISPL, Array: GlobalNodeId, Array: PedigreeNodeId, Array: VEL]

# Change the selection.
>>> reader.PointVariables = ['DISPL']

# Print the list of point arrays read in, nothing changes!
>>> print(reader.PointData[:])
[Array: ACCL, Array: DISPL, Array: GlobalNodeId, Array: PedigreeNodeId, Array: VEL]

# Update the pipeline.
>>> UpdatePipeline()

# Now the arrays read in has indeed changed as we expected.
>>> print(reader.PointData[:])
[Array: DISPL, Array: GlobalNodeId, Array: PedigreeNodeId]
```

We will cover the `reader.PointData` API in more details in Section 1.3.3.

### 1.2.3 Reloading files

While ParaView is often used after the simulation has generated all the data, it is not uncommon to use ParaView to inspect data files as they are being written out by the simulation. In such cases, the simulation may either be modifying existing file(s) with new timesteps or creating new files for each timestep. In such cases, you may want to *refresh* ParaView to make it aware of the changes. In `paraview`, this can be done using `Reload Files`. When the reader is active, you can use the *File > Reload Files* menu to request the reader to refresh. `paraview` will prompt you to choose whether to reload the existing file(s) or look for new files in the file series, as shown in Fig. 1.18. Click on `Reload existing file(s)`, to force the reader to re-read the files already opened. This is useful in cases where the simulation may have modified existing file(s). Use `Find new files` to make the reader aware of any new files in the file series.


Reload Options

This reader supports file series. Do you want to look for new files in the series and load those, or reload the existing files?

Fig. 1.18: The `Reload Options` dialog allows you to choose how to refresh the reader.

Similar to `paraview`, in `pvpython`, you use `ReloadFiles` to reload existing files, and `ExtendFilesSeries` to look for new files in a file series.

```
# For file being modified in place per timestep
>>> reader = OpenDataFile(file)
...
>>> ReloadFiles(reader)
```

(continues on next page)

(continued from previous page)

```
# For files being generated per timestep
>>> reader = OpenDataFile(file)
...
>>> ExtendFilesSeries(reader)
```

## 1.3 Understanding Data

### 1.3.1 VTK data model

To use **ParaView** effectively, you need to understand the **ParaView** data model. **ParaView** uses VTK, the Visualization Toolkit, to provide the visualization and data processing model. This chapter briefly introduces the VTK data model used by **ParaView**. For more details, refer to one of the VTK books [SML06] [KInc10].

The most fundamental data structure in VTK is a data object. Data objects can either be scientific datasets, such as rectilinear grids or finite elements meshes (see below), or more abstract data structures, such as graphs or trees. These datasets are formed from smaller building blocks: mesh (topology and geometry) and attributes.

#### Mesh

Even though the actual data structure used to store the mesh in memory depends on the type of the dataset, some abstractions are common to all types. In general, a mesh consists of vertices (points ) and cells (elements, zones). Cells are used to discretize a region and can have various types such as tetrahedra, hexahedra, etc. Each cell contains a set of vertices. The mapping from cells to vertices is called the connectivity. Note that even though it is possible to define data elements such as faces and edges, VTK does not represent these explicitly. Rather, they are implied by a cell's type and by its connectivity. One exception to this rule is the arbitrary polyhedron, which explicitly stores its faces. Fig. 1.19 is an example mesh that consists of two cells. The first cell is defined by vertices $(0, 1, 3, 4)$, and the second cell is defined by vertices $(1, 2, 4, 5)$. These cells are neighbors because they share the edge defined by the points $(1, 4)$.

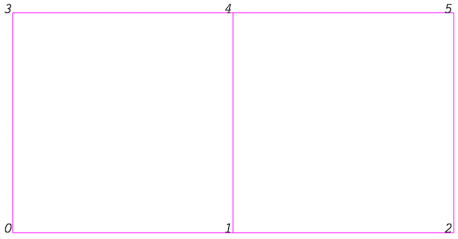


Fig. 1.19: Example of a mesh.

A mesh is fully defined by its topology and the spatial coordinates of its vertices. In VTK, the point coordinates may be implicit, or they may be explicitly defined by a data array of dimensions $(number\_of\_points \times 3)$.

### Attributes (fields, arrays)

An attribute (or a data array or field) defines the discrete values of a field over the mesh. Examples of attributes include pressure, temperature, velocity, and stress tensor. Note that VTK does not specifically define different types of attributes. All attributes are stored as data arrays, which can have an arbitrary number of components. **ParaView** makes some assumptions in regards to the number of components. For example, a 3-component array is assumed to be an array of vectors. Attributes can be associated with points or cells. It is also possible to have attributes that are not associated with either. Fig. 1.20 demonstrates the use of a point-centered attribute. Note that the attribute is only defined on the vertices. Interpolation is used to obtain the values everywhere else. The interpolation functions used depend on the cell type. See the VTK documentation for details.

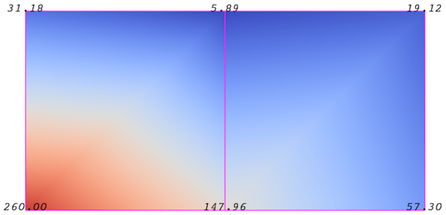


Fig. 1.20: Point-centered attribute in a data array or field.

Fig. 1.21 demonstrates the use of a cell-centered attribute. Note that cell-centered attributes are assumed to be constant over each cell. Due to this property, many filters in VTK cannot be directly applied to cell-centered attributes. It is normally required to apply a `Cell Data to Point Data` filter. In **ParaView**, this filter is applied automatically, when necessary.

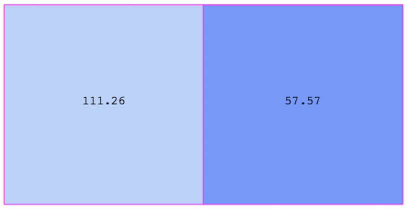


Fig. 1.21: Cell-centered attribute.

### Uniform rectilinear grid (image data)

A uniform rectilinear grid, or image data, defines its topology and point coordinates implicitly (Fig. 1.22). To fully define the mesh for an image data, VTK uses the following:

- *Extents* - These define the minimum and maximum indices in each direction. For example, an image data of extents $(0, 9)$, $(0, 19)$, $(0, 29)$ has 10 points in the x-direction, 20 points in the y-direction, and 30 points in the z-direction. The total number of points is $10 \times 20 \times 30$.
- *Origin* - This is the position of a point defined with indices $(0, 0, 0)$.
- *Spacing* - This is the distance between each point. Spacing for each direction can defined independently.

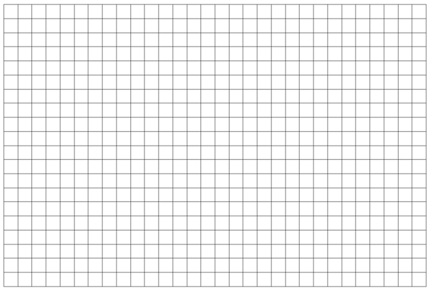


Fig. 1.22: Example uniform rectilinear grid.

The coordinate of each point is defined as follows: $coordinate = origin + index \times spacing$ where $coordinate$, $origin$, $index$, and $spacing$ are vectors of length 3.

Note that the generic VTK interface for all datasets uses a flat index. The $(i, j, k)$ index can be converted to this flat index as follows: $idx\_flat = k \times (npts_x \times npts_y) + j \times nptr_x + i$.

A uniform rectilinear grid consists of cells of the same type. This type is determined by the dimensionality of the dataset (based on the extents) and can either be vertex (0D), line (1D), pixel (2D), or voxel (3D).

Due to its regular nature, image data requires less storage than other datasets. Furthermore, many algorithms in VTK have been optimized to take advantage of this property and are more efficient for image data.

### Rectilinear grid

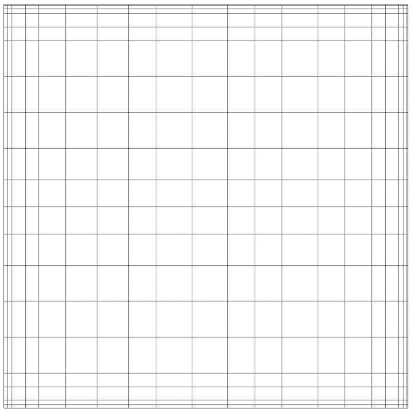


Fig. 1.23: Rectilinear grid.

A rectilinear grid, such as Fig. 1.23, defines its topology implicitly and point coordinates semi-implicitly. To fully define the mesh for a rectilinear grid, VTK uses the following:

- *Extents* - These define the minimum and maximum indices in each direction. For example, a rectilinear grid of extents $(0, 9)$, $(0, 19)$, $(0, 29)$ has 10 points in the x-direction, 20 points in the y-direction, and 30 points in the z-direction. The total number of points is $10 \times 20 \times 30$.

- *Three arrays defining coordinates in the x-, y- and z-directions* - These arrays are of length $npts_x$, $npts_y$, and $npts_z$. This is a significant savings in memory, as the total memory used by these arrays is $npts_x + npts_y + npts_z$ rather than $npts_x \times npts_y \times npts_z$.

The coordinate of each point is defined as follows:

$coordinate = (coordinate\_array_x(i), coordinate\_array_y(j), coordinate\_array_z(k))$.

Note that the generic VTK interface for all datasets uses a flat index. The $(i, j, k)$ index can be converted to this flat index as follows: $idx\_flat = k \times (npts_x \times npts_y) + j \times nptr_x + i$.

A rectilinear grid consists of cells of the same type. This type is determined by the dimensionality of the dataset (based on the extents) and can either be vertex (0D), line (1D), pixel (2D), or voxel (3D).

### Curvilinear grid (structured grid)

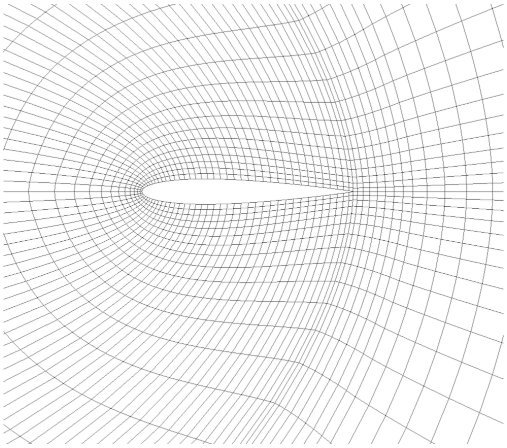


Fig. 1.24: Curvilinear or structured grid.

A curvilinear grid, such as Fig. 1.24, defines its topology implicitly and point coordinates explicitly. To fully define the mesh for a curvilinear grid, VTK uses the following:

- *Extents* - These define the minimum and maximum indices in each direction. For example, a curvilinear grid of extents $(0, 9)$, $(0, 19)$, $(0, 29)$ has $10 \times 20 \times 30$ points regularly defined over a curvilinear mesh.
- *An array of point coordinates* - This array stores the position of each vertex explicitly.

The coordinate of each point is defined as follows: $coordinate = coordinate\_array(idx\_flat)$. The $(i, j, k)$ index can be converted to this flat index as follows: $idx\_flat = k \times (npts_x \times npts_y) + j \times npts_x + i$.

A curvilinear grid consists of cells of the same type. This type is determined by the dimensionality of the dataset (based on the extents) and can either be vertex (0D), line (1D), quad (2D), or hexahedron (3D).

### AMR dataset

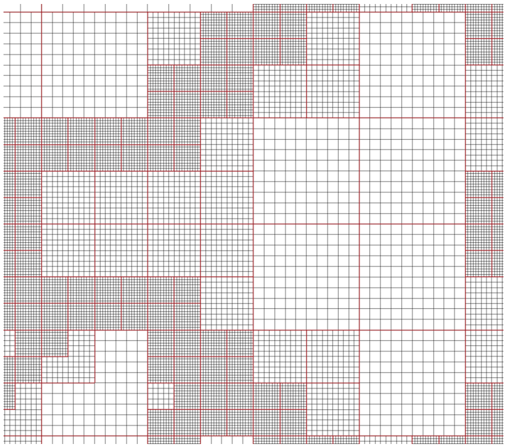


Fig. 1.25: AMR dataset.

VTK natively supports Berger-Oliger type AMR (Adaptive Mesh Refinement) datasets, as shown in Fig. 1.25. An AMR dataset is essentially a collection of uniform rectilinear grids grouped under increasing refinement ratios (decreasing spacing). VTK's AMR dataset does not force any constraint on whether and how these grids should overlap. However, it provides support for masking (blanking) sub-regions of the rectilinear grids using an array of bytes. This allows VTK to process overlapping grids with minimal artifacts. VTK can automatically generate the masking arrays for Berger-Oliger compliant meshes.

### Unstructured grid

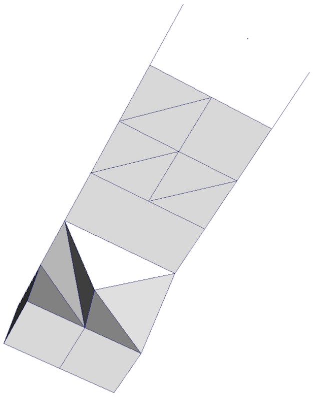


Fig. 1.26: Unstructured grid.

An unstructured grid, such as Fig. 1.26, is the most general primitive dataset type. It stores topology and point coordinates explicitly. Even though VTK uses a memory-efficient data structure to store the topology, an unstructured grid uses significantly more memory to represent its mesh. Therefore, use an unstructured grid only when you cannot represent your dataset as one of the above datasets. VTK supports a large number of cell types, all of which can exist within one heterogeneous unstructured grid. The full list of all cell types supported by VTK can be found in the file `vtkCellType.h` in the VTK source code. Here is the list of cell types as of when this document was written:

| | |
|---|---|
| VTK_EMPTY_CELL | VTK_BIQUADRATIC_TRIANGLE |
| VTK_VERTEX | VTK_CUBIC_LINE |
| VTK_POLY_VERTEX | VTK_CONVEX_POINT_SET |
| VTK_LINE | VTK_POLYHEDRON |
| VTK_POLY_LINE | VTK_PARAMETRIC_CURVE |
| VTK_TRIANGLE | VTK_PARAMETRIC_SURFACE |
| VTK_TRIANGLE_STRIP | VTK_PARAMETRIC_TRI_SURFACE |
| VTK_POLYGON | VTK_PARAMETRIC_QUAD_SURFACE |
| VTK_PIXEL | VTK_PARAMETRIC_TETRA_REGION |
| VTK_QUAD | VTK_PARAMETRIC_HEX_REGION |
| VTK_TETRA | VTK_HIGHER_ORDER_EDGE |
| VTK_VOXEL | VTK_HIGHER_ORDER_TRIANGLE |
| VTK_HEXAHEDRON | VTK_HIGHER_ORDER_QUAD |
| VTK_WEDGE | VTK_HIGHER_ORDER_POLYGON |
| VTK_PYRAMID | VTK_HIGHER_ORDER_TETRAHEDRON |
| VTK_PENTAGONAL_PRISM | VTK_HIGHER_ORDER_WEDGE |
| VTK_HEXAGONAL_PRISM | VTK_HIGHER_ORDER_PYRAMID |
| VTK_QUADRATIC_EDGE | VTK_HIGHER_ORDER_HEXAHEDRON |
| VTK_QUADRATIC_TRIANGLE | VTK_LAGRANGE_CURVE |
| VTK_QUADRATIC_QUAD | VTK_LAGRANGE_TRIANGLE |
| VTK_QUADRATIC_POLYGON | VTK_LAGRANGE_QUADRILATERAL |
| VTK_QUADRATIC_TETRA | VTK_LAGRANGE_TETRAHEDRON |
| VTK_QUADRATIC_HEXAHEDRON | VTK_LAGRANGE_HEXAHEDRON |
| VTK_QUADRATIC_WEDGE | VTK_LAGRANGE_WEDGE |
| VTK_QUADRATIC_PYRAMID | VTK_LAGRANGE_PYRAMID |
| VTK_BIQUADRATIC_QUAD | VTK_BEZIER_CURVE |
| VTK_TRIQUADRATIC_HEXAHEDRON | VTK_BEZIER_TRIANGLE |
| VTK_TRIQUADRATIC_PYRAMID | VTK_BEZIER_QUADRILATERAL |
| VTK_QUADRATIC_LINEAR_QUAD | VTK_BEZIER_TETRAHEDRON |
| VTK_QUADRATIC_LINEAR_WEDGE | VTK_BEZIER_HEXAHEDRON |
| VTK_BIQUADRATIC_QUADRATIC_WEDGE | VTK_BEZIER_WEDGE |
| VTK_BIQUADRATIC_QUADRATIC_HEXAHEDRON | VTK_BEZIER_PYRAMID |

Many of these cell types are straightforward. For details, see the VTK documentation.

### Polygonal grid (polydata)

A polydata, such as Fig. 1.27, is a specialized version of an unstructured grid designed for efficient rendering. It consists of 0D cells (vertices and polyvertices), 1D cells (lines and polylines), and 2D cells (polygons and triangle strips). Certain filters that generate only these cell types will generate a polydata. Examples include the Contour and Slice filters. An unstructured grid, as long as it has only 2D cells supported by polydata, can be converted to a polydata using the `Extract Surface Filter`. A polydata can be converted to an unstructured grid using `Clean to Grid`.

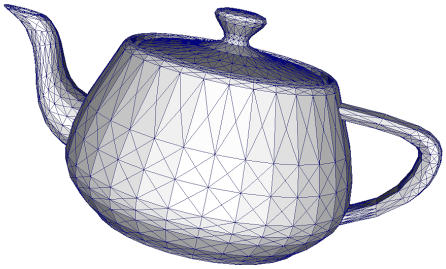
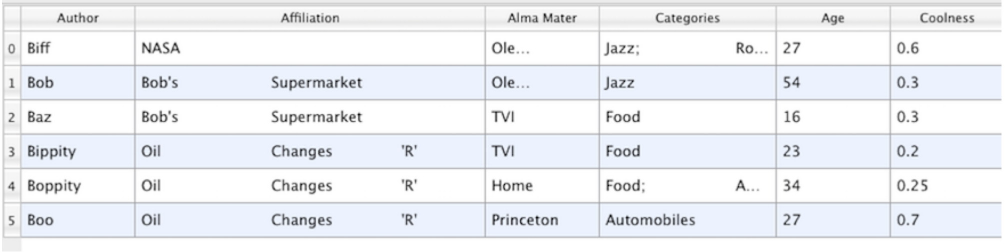


Fig. 1.27: Polygonal grid.

### Table



| | Author | Affiliation | Alma Mater | Categories | Age | Coolness |
|---|---|---|---|---|---|---|
| 0 | Biff | NASA | Ole... | Jazz; Ro... | 27 | 0.6 |
| 1 | Bob | Bob's Supermarket | Ole... | Jazz | 54 | 0.3 |
| 2 | Baz | Bob's Supermarket | TVI | Food | 16 | 0.3 |
| 3 | Bippity | Oil Changes 'R' | TVI | Food | 23 | 0.2 |
| 4 | Boppity | Oil Changes 'R' | Home | Food; A... | 34 | 0.25 |
| 5 | Boo | Oil Changes 'R' | Princeton | Automobiles | 27 | 0.7 |

Fig. 1.28: Table

A table, such as Fig. 1.28, is a tabular dataset that consists of rows and columns. All chart views have been designed to work with tables. Therefore, all filters that can be shown within the chart views generate tables. Also, tables can be directly loaded using various file formats such as the comma-separated values format. Tables can be converted to other datasets as long as they are of the right format. Filters that convert tables include `Table to Points` and `Table to Structured Grid`.

### Multiblock dataset

You can think of a multiblock dataset (Fig. 1.29) as a tree of datasets where the leaf nodes are *simple* datasets. All of the data types described above, except AMR, are *simple* datasets. Multiblock datasets are used to group together datasets that are related. The relation between these datasets is not necessarily defined by **ParaView**. A multiblock dataset can represent an assembly of parts or a collection of meshes of different types from a coupled simulation. Multiblock datasets can be loaded or created within **ParaView** using the `Group` filter. Note that the leaf nodes of a multiblock dataset do not all have to have the same attributes. We call attributes that are not present on all blocks of a multiblock dataset *partial attributes* or *partial arrays* If you apply a filter that requires an attribute, it will be applied only to blocks that have that attribute.

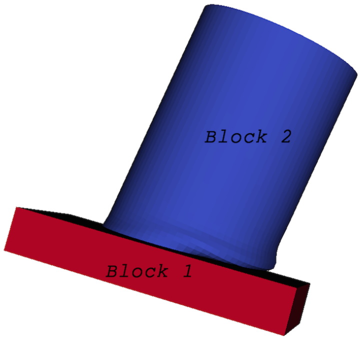


Fig. 1.29: Multiblock dataset.

### Multipiece dataset

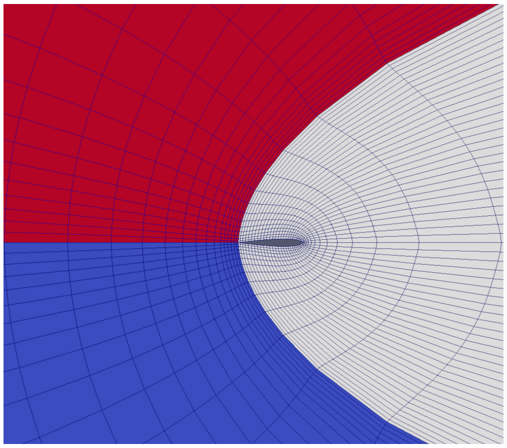


Fig. 1.30: Multipiece dataset.

Multipiece datasets, such as Fig. 1.30, are similar to multiblock datasets in that they group together simple datasets. There is one key difference. Multipiece datasets group together datasets that are part of a whole mesh - datasets of the same type and with the same attributes. This data structure is used to collect datasets produced by a parallel simulation without having to append the meshes together. Note that there is no way to create a multipiece dataset within **ParaView**. It can only be created by using certain readers. Furthermore, multipiece datasets act, for the most part, as simple datasets. For example, it is not possible to extract individual pieces or to obtain information about them.

### 1.3.2 Getting data information in paraview

In the visualization pipeline (Section 1.1.2), sources, readers, and filters are all producing data. In a VTK-based pipeline, this data is one of the types discussed. Thus, when you create a source or open a data file in paraview and hit Apply , data is being produced. The Information panel and the Statistics Inspector panel can be used to inspect the characteristics of the data produced by any pipeline module.

#### The Information panel

The Information panel provides summary information about the data produced by the active source. By default, this panel is tucked under a tab below the Properties panel. You can toggle its visibility using *View > Information*.

The Information panel shows the data information for the active source. Thus, similar to the Properties panel, it changes when the active source is changed (e.g., by changing the selection in the Pipeline Browser ). One way to think of this panel is as a panel showing a summary for the data *currently* produced by the active source. Remember that a newly-created pipeline module does not produce any data until you hit Apply . Thus, valid information for a newly-created source will be shown on this panel only after that Apply . Similarly, if you change properties on the source and hit Apply , this panel will reflect any changes in data characteristics. Additionally, for temporal pipelines, this panel shows information for the current timestep alone (except as noted). Thus, as you step through timesteps in a temporal dataset, the information displayed will potentially change, and the panel will reflect those changes.

**Did you know?**

Any text on this panel is *copy*-able. For example, if want to copy the number of points value to use it as a property value on the Properties panel, simply double-click on the number or click-and-drag to select the number and use the common keyboard shortcut CTRL + C (or + C) to copy that value to the clipboard. Now, you can paste it in an input widget in paraview or any other application, such as an editor, by using CTRL + V (or + V) or the application-specific shortcut for pasting text from the clipboard. The same is true for numbers shown in lists, such as the Data Ranges .

The several groups of information comprise the panel itself. Groups may be hidden based on the type of pipeline module or the type of data being produced.

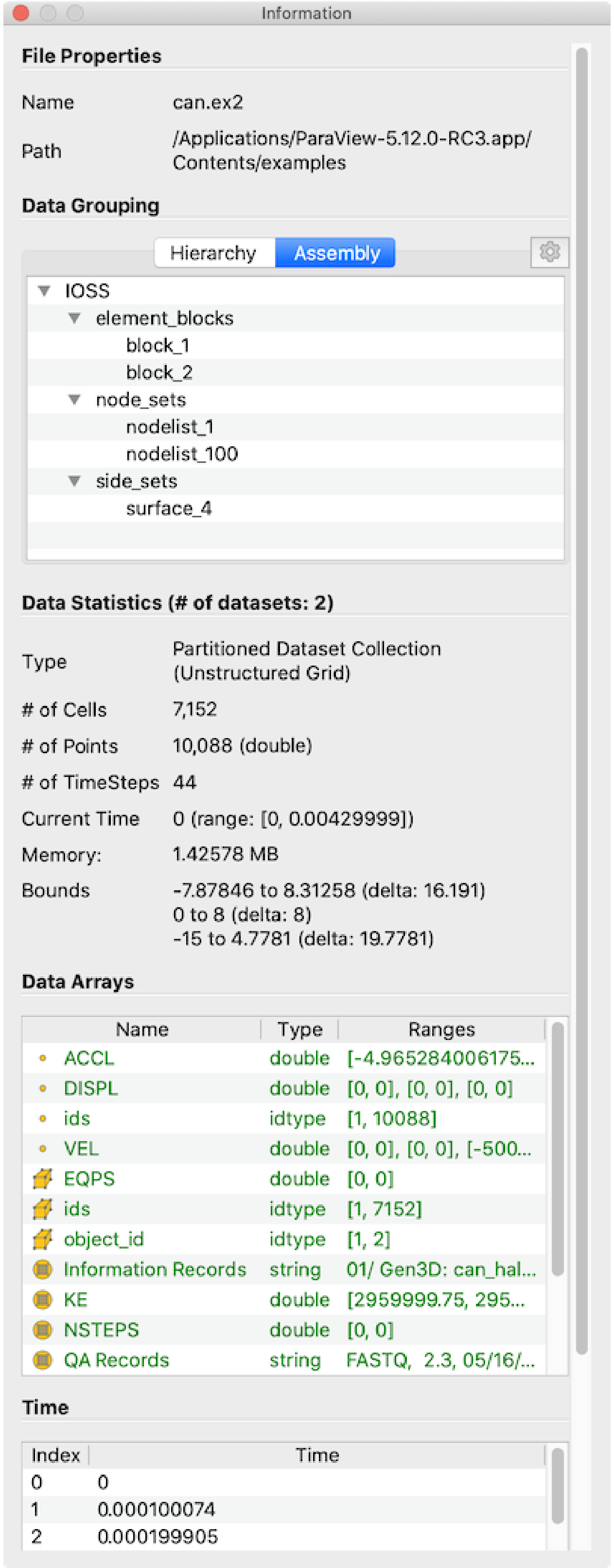
The file File Properties group is shown for readers with information about the file that is opened. For a temporal file series, as you step through each time step, the file name is updated to point to the name of the file in the series that corresponds to the current time step.

The Data Grouping section shows information about composite datasets consisting of more than one datasets in some kind of hierarchical arrangement. Two different hierarchies are available. The Hierarchy shows the relationships of datasets inherent in the composite dataset structure, while the Assembly shows relationships among datasets that are explicity defined by the data source.

The Data Statistics group provides a summary of the dataset produced including its type, its number of cells and points (or rows and columns in cases of Tabular datasets), the number of timesteps, the current time, and an estimate of the memory used by the dataset. This number only includes the memory space needed to save the data arrays for the dataset. It does not include the memory space used by the data structures themselves and, hence, must only be treated as an estimate. Lastly, the group shows the spatial bounds of the datasets in 3D Cartesian space. This will be unavailable for non-geometric datasets such as tables.

The Data Arrays group lists all of the available point, cells, and field arrays, as well as their types and ranges for the current time step. The Current Time field shows the time value for the current timestep as a reference. As with other places in paraview, the icons , , and are used to indicate cell, point, and field data arrays. Since data arrays can have multiple components, the range for each component of the data array is shown.

For reader modules, the Time group shows the available time steps and corresponding time values provided by the file.



**File Properties**

| | |
|---|---|
| Name | can.ex2 |
| Path | /Applications/ParaView-5.12.0-RC3.app/<br>Contents/examples |

**Data Grouping**

```
▼ IOSS
    ▼ element_blocks
          block_1
          block_2
    ▼ node_sets
          nodelist_1
          nodelist_100
    ▼ side_sets
          surface_4
```

**Data Statistics (# of datasets: 2)**

| | |
|---|---|
| Type | Partitioned Dataset Collection<br>(Unstructured Grid) |
| # of Cells | 7,152 |
| # of Points | 10,088 (double) |
| # of TimeSteps | 44 |
| Current Time | 0 (range: [0, 0.00429999]) |
| Memory: | 1.42578 MB |
| Bounds | -7.87846 to 8.31258 (delta: 16.191)<br>0 to 8 (delta: 8)<br>-15 to 4.7781 (delta: 19.7781) |

**Data Arrays**

| | Name | Type | Ranges |
|---|---|---|---|
| ● | ACCL | double | [-4.965284006175... |
| ● | DISPL | double | [0, 0], [0, 0], [0, 0] |
| ● | ids | idtype | [1, 10088] |
| ● | VEL | double | [0, 0], [0, 0], [-500... |
| ▨ | EQPS | double | [0, 0] |
| ▨ | ids | idtype | [1, 7152] |
| ▨ | object_id | idtype | [1, 2] |
| ◉ | Information Records | string | 01/ Gen3D: can_hal... |
| ◉ | KE | double | [2959999.75, 295... |
| ◉ | NSTEPS | double | [0, 0] |
| ◉ | QA Records | string | FASTQ, 2.3, 05/16/... |

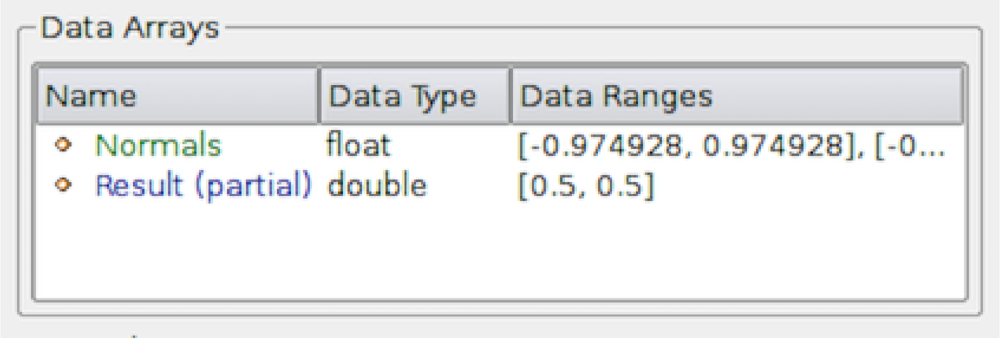
**Time**

| Index | Time |
|---|---|
| 0 | 0 |
| 1 | 0.000100074 |
| 2 | 0.000199905 |

Fig. 1.31: The Information panel in paraview showing data summaries for the active source.

For structured datasets such as uniform rectilinear grids or curvilinear grids, the `Extents` group is shown that displays the structured extents and dimensions of the datasets.

All of the summary information discussed so far provides a synopsis of the entire dataset produced by the pipeline module, including across all ranks (which will become clearer once we look at using **ParaView** for parallel data processing). In cases of composite datasets, such as mutliblock datasets or AMR datasets, recall that these are datasets that are made up of other datasets. In such cases, these are summaries over all the blocks in the composite dataset. Every so often, you will notice that the `Data Arrays` table lists an array with the suffix `(partial)` (Figure Fig. 1.32). Such arrays are referred to as *partial arrays*. Partial arrays is a term used to refer to arrays that are present on some non-composite blocks or leaf nodes in a composite dataset, but not all. The `(partial)` suffix to indicate partial arrays is also used by `paraview` in other places in the UI.



Data Arrays

| Name | Data Type | Data Ranges |
|---|---|---|
| ○ Normals | float | [-0.974928, 0.974928], [-0... |
| ○ Result (partial) | double | [0.5, 0.5] |

Fig. 1.32: The `Data Arrays` section on `Information` panel showing *partial* arrays. Partial arrays are arrays that present on certain blocks in a composite dataset, but not all.

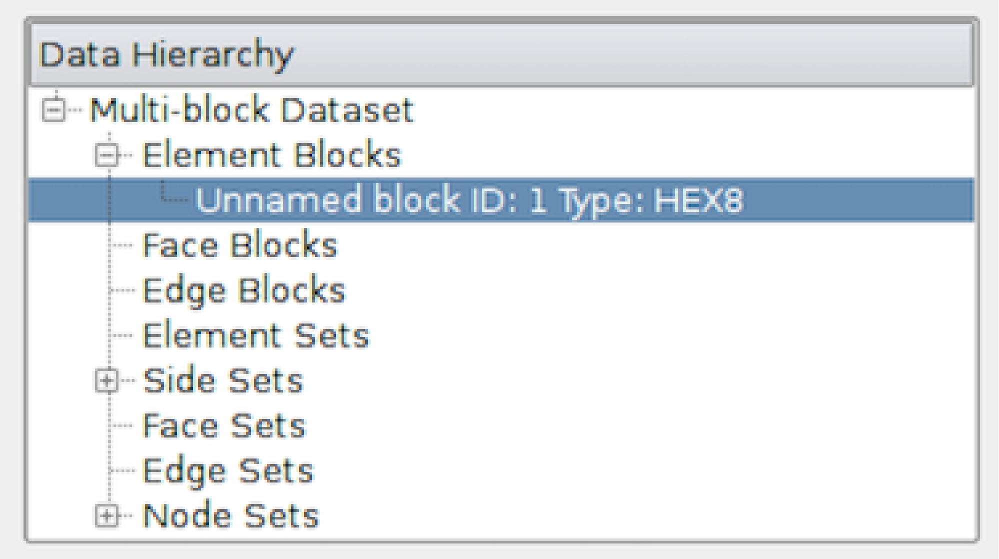
While summaries over all of the datasets in the composite dataset are useful, you may also want to look at the data information for individual blocks. To do so, you can use the `Data Hierarchy` group, which appears when summarizing composite datasets. The `Data Hierarchy` widget shows the structure or hierarchy of the composite dataset (Figure Fig. 1.33). The `Information` panel switches to showing the summaries for the selected sub-tree. By default, the root element will be selected. You can now select any block in the hierarchy to view the summary limited to just that sub-tree.

**Did you know?**

Memory information shown on the `Information` panel and the `Statistics Inspector` should only be used as an approximate reference and does not translate to how much memory the data produced by a particular pipeline module takes. This is due to the following factors:

- The size does not include the amount of memory needed to build the data structures to store the data arrays. While, in most cases, this is negligible compared to that of the data arrays, it can be nontrivial, especially when dealing with deeply-nested composite datasets.
- Several filters such as `Calculator` and `Shrink` simply pass input data arrays through, so there's no extra space needed for those data arrays that are shared with the input. The memory size numbers shown, however, do not take this into consideration.

If you need an overview of how much physical memory is being used by **ParaView** in its current state, you can use the `Memory Inspector` (Section 2.8).



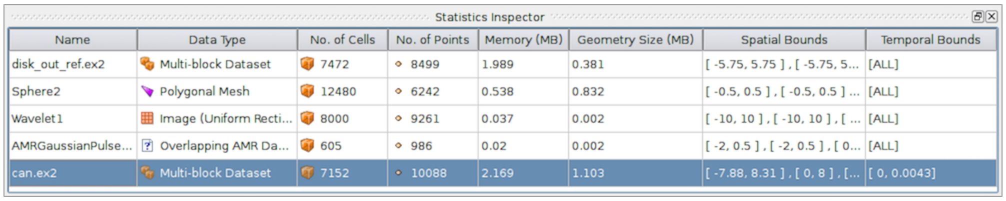
Data Hierarchy

- Multi-block Dataset
  - Element Blocks
    - Unnamed block ID: 1 Type: HEX8
  - Face Blocks
  - Edge Blocks
  - Element Sets
  - Side Sets
  - Face Sets
  - Edge Sets
  - Node Sets

Fig. 1.33: The Data Hierarchy section on the Information panel showing the composite data hierarchy. Selecting a particular block or subtree in this widget will result in the reset of the Information panel showing summaries for that block or subtree alone.

### The Statistics Inspector panel

The Information panel shows data information for the active source. If you need a quick summary of the data produced by all the pipeline modules, you can use the Statistics Inspector panel. It's accessible from *Views > Statistics Inspector*.



Statistics Inspector

| Name | Data Type | No. of Cells | No. of Points | Memory (MB) | Geometry Size (MB) | Spatial Bounds | Temporal Bounds |
|---|---|---|---|---|---|---|---|
| disk_out_ref.ex2 | Multi-block Dataset | 7472 | 8499 | 1.989 | 0.381 | [ -5.75, 5.75 ] , [ -5.75, 5... | [ALL] |
| Sphere2 | Polygonal Mesh | 12480 | 6242 | 0.538 | 0.832 | [ -0.5, 0.5 ] , [ -0.5, 0.5 ] ... | [ALL] |
| Wavelet1 | Image (Uniform Recti... | 8000 | 9261 | 0.037 | 0.002 | [ -10, 10 ] , [ -10, 10 ] , [ ... | [ALL] |
| AMRGaussianPulse... | Overlapping AMR Da... | 605 | 986 | 0.02 | 0.002 | [ -2, 0.5 ] , [ -2, 0.5 ] , [ 0... | [ALL] |
| can.ex2 | Multi-block Dataset | 7152 | 10088 | 2.169 | 1.103 | [ -7.88, 8.31 ] , [ 0, 8 ] , [... | [ 0, 0.0043] |

Fig. 1.34: The Statistics Inspector panel in paraview showing summaries for all pipeline modules.

All of the information on this panel is also presented on the Information panel, except Geometry Size . This corresponds to how much memory is needed for the transformed dataset used for rendering in the active view. For example, to render a 3D dataset as a surface in the 3D view, **ParaView** must extract the surface mesh as a polydata. Geometry Size represents the memory needed for this polydata with the same memory-size-related caveats as with the Information panel.

### 1.3.3 Getting data information in pvpython

When scripting with **ParaView**, you will often find yourself needing information about the data. While paraview sets up filter properties and color tables automatically using the information from the data, you must do that explicitly when scripting.

In pvpython, for any pipeline module (sources, readers, or filters), you can use the following ways to get information about the data produced.

```
>>> from paraview.simple import *
>>> reader = OpenDataFile(".../ParaViewData/Data/can.ex2")

# We need to update the pipeline. Otherwise, all of the data
# information we get will be from before the file is actually
# read and, hence, will be empty.
>>> UpdatePipeline()

>>> dataInfo = reader.GetDataInformation()

# To get the number of cells or points in the dataset:
>>> dataInfo.GetNumberOfPoints()
10088
>>> dataInfo.GetNumberOfCells()
7152

# You can always nest the call, e.g.:
>>> reader.GetDataInformation().GetNumberOfPoints()
10088
>>> reader.GetDataInformation().GetNumberOfCells()
7152

# Use source.PointData or source.CellData to get information about
# point data arrays and cell data arrays, respectively.

# Let's print the available point data arrays.
>>> reader.PointData[:]
[Array: ACCL, Array: DISPL, Array: GlobalNodeId, Array: PedigreeNodeId, Array: VEL]

# Similarly, for cell data arrays:
>>> reader.CellData[:]
[Array: EQPS, Array: GlobalElementId, Array: ObjectId, Array: PedigreeElementId]
```

PointData (and CellData ) is a map or dictionary where the keys are the names of the arrays, and the values are objects that provide more information about each of the arrays. In the rest of this section, anything we demonstrate on PointData is also applicable to CellData .

```
# Let's get the number of available point arrays.
>>> len(reader.PointData)
5

# Print the names for all available point arrays.
>>> reader.PointData.keys()
['ACCL', 'DISPL', 'GlobalNodeId', 'PedigreeNodeId', 'VEL']
```

(continues on next page)

(continued from previous page)

```
>>> reader.PointData.values()
[Array: ACCL, Array: DISPL, Array: GlobalNodeId, Array: PedigreeNodeId, Array: VEL]

# To test if a particular array is present:
>>> reader.PointData.has_key("ACCL")
True

>>> reader.PointData.has_key("--non-existent-array--")
False
```

From PointData (or CellData ), you can get access to an object that provides information for each of the arrays. This object gives us methods to get data ranges, component counts, tuple counts, etc.

```
# Let's get information about 'ACCL' array.
>>> arrayInfo = reader.PointData["ACCL"]
>>> arrayInfo.GetName()
'ACCL'

# To get the number of components in each tuple and the number
# of tuples in the data array:
>>> arrayInfo.GetNumberOfTuples()
10088
>>> arrayInfo.GetNumberOfComponents()
3

# Alternative way for doing the same:
>>> reader.PointData["ACCL"].GetNumberOfTuples()
10088
>>> reader.PointData["ACCL"].GetNumberOfComponents()
3

# To get the range for a particular component, e.g. component 0:
>>> reader.PointData["ACCL"].GetRange(0)
(-4.965284006175352e-07, 3.212448973499704e-07)

# To get the range for the magnitude in cases of multi-component arrays
# use -1 as the component number.
>>> reader.PointData["ACCL"].GetRange(-1)
(0.0, 1.3329898584157294e-05)

# To determine the data data type for this array:
>>> from paraview import vtk
>>> reader.PointData["ACCL"].GetDataType() == vtk.VTK_DOUBLE
True
# The paraview.vtk module provides access to these constants such as
# VTK_DOUBLE, VTK_FLOAT, VTK_INT, etc.

# Likewise, to test the dataset type, itself:
>>> reader.GetDataInformation().GetDataSetType() == \
                               vtk.VTK_MULTIBLOCK_DATA_SET
True
```

Here's a sample script to iterate over all point data arrays and print their magnitude ranges:

```
>>> def print_point_data_ranges(source):
...     """Prints array ranges for all point arrays"""
...     for arrayInfo in source.PointData:
...         # get the array's name
...         name = arrayInfo.GetName()
...         # get magnitude range
...         range = arrayInfo.GetRange(-1)
...         print "%s = [%.3f, %.3f]" % (name, range[0], range[1])

# Let's call this function on our reader.
>>> print_point_data_ranges(reader)
ACCL = [0.000, 0.000]
DISPL = [0.000, 0.000]
GlobalNodeId = [1.000, 10088.000]
PedigreeNodeId = [1.000, 10088.000]
VEL = [0.000, 5000.000]
```

**Did you know?**

The example scripts in this section all demonstrated how to obtain information about the data such as the number of points and cells, data bounds, and array ranges. However, what they do not show is how to access the raw data itself. To see how to obtain the full data, please see Section 2.7.11.

## 1.4 Displaying data

The goal of any visualization process is to produce visual representations of the data. The visual representations are shown in modules called views. *Views* provide the canvas on which to display such visual representations, as well as to dictate how these representations are generated from the raw data. The role of the visualization pipeline is often to transform the data so that relevant information can be represented in these views.

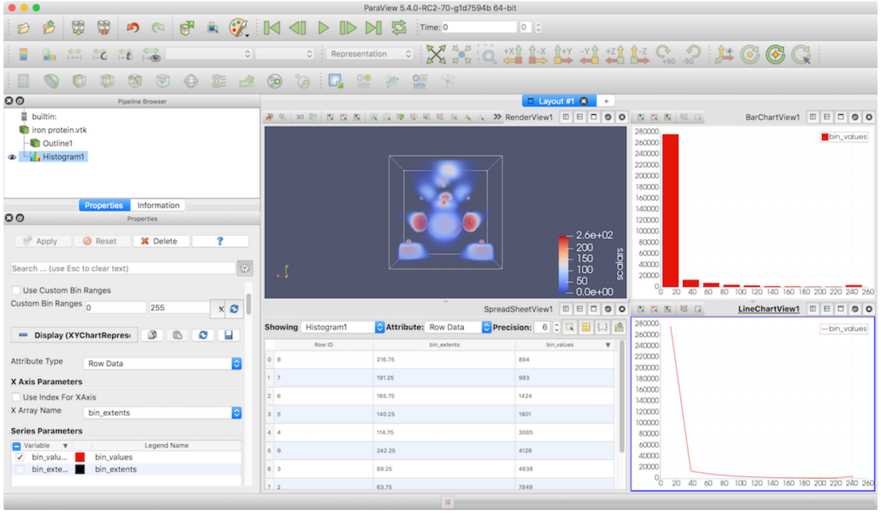
Referring back to the visualization pipeline from Section 1.1.2, views are sinks that take in input data but do not produce any data output (i.e., one cannot connect other pipeline modules such as filters to process the results in a view). However, views often provide mechanisms to save the results as images or in other formats including PDF, VRML, and X3D.

Different types of views provide different ways of visualizing data. These can be broadly grouped as follows:

- *Rendering Views* are views that render geometries or volumes in a graphical context. The `Render View` is one such view. Other Render View-based views, such as `Slice View` and `Quad View` , extend the basic render view to add the ability to add mechanisms to easily inspect slices or generate orthogonal views.
- *Chart Views* cover a wide array of graphs and plots used for visualizing non-geometric data. These include views such as line charts ( `Line Chart View` ), bar charts ( `Bar Chart View` ), bag charts ( `Bag Chart View` ), parallel coordinates ( `Parallel Coordinates View` ), etc.
- *Comparative Views* are used to quickly generate side-by-side views for parameter study, i.e., to visualize the effects of parameter changes. Comparative variants of `Render View` and several types of the `Chart Views` are available in **ParaView**.

In this chapter, we take a close look at the various views available in **ParaView** and how to use these views for displaying data.

### 1.4.1 Multiple views



Search ... (use Esc to clear text)

□ Use Custom Bin Ranges

Custom Bin Ranges 0 255

Display (XYChartRepres

Attribute Type: Row Data

**X Axis Parameters**

□ Use Index For XAxis

X Array Name: bin_extents

**Series Parameters**

| | Variable | | Legend Name |
|---|---|---|---|
| ☑ | bin_valu... | 🟥 | bin_values |
| □ | bin_exte... | ⬛ | bin_extents |

Showing: Histogram1 Attribute: Row Data Precision: 6

| | Row ID | bin_extents | bin_values |
|---|---|---|---|
| 0 | 8 | 216.75 | 894 |
| 1 | 7 | 191.25 | 993 |
| 2 | 6 | 165.75 | 1424 |
| 3 | 5 | 140.25 | 1901 |
| 4 | 4 | 114.75 | 3085 |
| 5 | 9 | 242.25 | 4126 |
| 6 | 3 | 89.25 | 4638 |
| 7 | 2 | 63.75 | 7849 |

Fig. 1.35: Using multiple views in `paraview` to generate different types of visualizations from a dataset.

With multiple types of views comes the need for creating and viewing multiple views at the same time. In this section, we look at how you can create multiple views and lay them out.

**Did you know?**

Multiple views were first supported in **ParaView** 3.0. Before that, all data was shown in a single 3D render view, including line plots!

#### Multiple views in `paraview`

`paraview` shows all views in the central part of the application window. When `paraview` starts up, the `Render View` is created and shown in the application window by default.

New views can be created by splitting the view frame using the `Split View` controls at the top-right corner of the view frame. Splitting a view divides the view into two equal parts, either vertically or horizontally, based on the button used for the split. On splitting a view, an empty frame with buttons for all known types of views is shown. Simply click on one of those buttons to create a new view of a chosen type.

You can move views by clicking and dragging the title bar for the view (or empty view frame) and dropping it on the title bar on another view (or empty view frame). This will swap the positions of the two views.

Similar to the notion of active source, there is a notion of *active view* . Several panels, toolbars, and menus will update based on the active view. The `Display` properties section on the `Properties` panel, for example, reflects the display properties of the active source in the active view. Similarly, the eyeball icons in the `Pipeline Browser` show the visibility status of the pipeline module in the active view. Active view is marked in the UI by a blue border around the view frame. Only one view can be active at any time in the application.

Besides being able to create multiple views and laying them out in a pane, `paraview` also supports placing views in multiple layouts under separate tabs. To create new tabs, use the + button in the tab bar. You can close a tab, which will destroy all views laid out in that tab, by clicking on the ☒ button. To pop out an entire tab as a separate window, use the □ button on the tab bar.

The active view is always present in the active tab. Thus, if you change the active tab, the active view will also be changed to be a view in the active tab layout. Conversely, if the active view is changed (by using the Python Shell , for example), the active tab will automatically be updated to be the tab that contains the active view.

**Did you know?**

You can make the views of the active layout fullscreen by using *View* > *Fullscreen (layout)* (or using the F11 key). You can also make the active view alone fullscreen by using *View* > *Fullscreen (active view)* (or using CTRL + F11 keys). To return back to the normal mode, use the Esc key.

### Multiple views in pvpython

In pvpython, one can create new views using the CreateView function or its variants, e.g., CreateRenderView .

```
>>> from paraview.simple import *
>>> view = CreateRenderView()
# Alternatively, use CreateView.
>>> view = CreateView("RenderView")
```

When a new view is created, it is automatically made active. You can manually make a view active by using the SetActiveView function. Several of the functions available in pvpython will use the active view when no view is passed as an argument to the function.

```
# Create a view
>>> view1 = CreateRenderView()
# Create a second view
>>> view2 = CreateRenderView()

# Check if view2 is the active view
>>> view2 == GetActiveView()
True

# Make view1 active
>>> SetActiveView(view1)
>>> view1 == GetActiveView()
True
```

When using Python Shell in paraview, if you create a new view, it will automatically be placed in the active tab by splitting the active view. You can manually control the layout and placement of views from Python too, using the layout API.

In Python, each tab corresponds to a layout.

```
# To get exisiting tabs/layouts
>>> layouts = GetLayouts()
>>> print(layouts)
{('ViewLayout1', '264'): <paraview.servermanager.ViewLayout object at 0x2e5b7d0>}

# To get layout corresponding to a particular view
>>> print(GetLayout(view))
<paraview.servermanager.ViewLayout object at 0x2e5b7d0>

# If view is not specified, active view is used
```

(continues on next page)

(continued from previous page)

```
>>> print(GetLayout())
<paraview.servermanager.ViewLayout object at 0x2e5b7d0>

# To create a new tab
>>> new_layout = servermanager.misc.ViewLayout(registrationGroup="layouts")

# To split the cell containing the view, either horizontally or vertically
>>> view = GetActiveView()
>>> layout = GetLayout(view)
>>> locationId = layout.SplitViewVertical(view=view,
                                          fraction=0.5)
# fraction is optional, if not specified the frame is split evenly.

# To assign a view to a particular cell.
>>> view2 = CreateRenderView()
>>> layout.AssignView(locationId, view2)
```

## 1.4.2 View properties

Just like parameters on pipeline modules, such as readers and filters, views provide parameters that can be used for customizing the visualization such as changing the background color for rendering views and adding title texts for chart views. These parameters are referred to as `View Properties` and are accessible from the `Properties` panel in `paraview`.

### View properties in `paraview`

Similar to properties on pipeline modules like sources and readers, view properties are accessible from the `Properties` panel. These are grouped under the `View` section. When the active view is changed, the `Properties` panel updates to show the view properties for the active view. Unlike pipeline modules, however, when you change the view properties, they affect the visualization immediately, without use of the `Apply` button.

**Did you know?**

It may seem odd that `View` and `Display` properties on the `Properties` panel don't need to be `Apply` -ed to take effect, while properties on pipeline modules like sources, readers and filter require you to hit the `Apply` button.

To understand the reasoning behind that, we need to understand why the `Apply` action is needed in the first place. Generally, executing a data processing filter or reader is time consuming on large datasets. If the pipeline module keeps on executing as you are changing the parameter, the user experience will quickly deteriorate, since the pipeline will keep on executing with intermediate (and potentially invalid) property values. To avoid this, we have the `Apply` action. This way, you can set up the pipeline properties to your liking and then trigger the potentially time consuming execution.

Since the visualization process in general focuses on *reducing* data to generate visual representations, the rendering (broadly speaking) is less time-intensive than the actual data processing. Thus, changing properties that affect rendering are not as compute-intensive as transforming the data itself. For example, changing the color on a surface mesh is not as expensive as generating the mesh in the first place. Hence, the need to `Apply` such properties becomes less relevant. At the same time, when changing display properties such as opacity, you may want to see the result as you change the property to decide on the final value. Hence, it is desirable to see the updates immediately.

Of course, you can always enable `Auto Apply` to have the same immediate update behavior for all properties on the `Properties` panel.

### View properties in `pvpython`

In `pvpython`, once you have access to the view, you can directly change view properties on the view object. There are several ways to get access to the view object.

```
# 1. Save reference when a view is created
>>> view = CreateView("RenderView")

# 2. Get reference to the active view.
>>> view = GetActiveView()
```

The properties available on the view will change based on the type of the view. You can use the `help` function to discover available properties.

```
>>> view = CreateRenderView()
>>> help(view)

  Help on RenderView in module paraview.servermanager object:

class RenderView(Proxy)
 |  View proxy for a 3D interactive render
 |  view.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  CenterAxesVisibility
 |      Toggle the visibility of the axes showing the center of
 |      rotation in the scene.
 |
 |  CenterOfRotation
 |      Center of rotation for the interactor.
 |
 ...

# Once you have a reference to the view, you can then get/set the properties.

# Get the current value
>>> print(view.CenterAxesVisibility)
1

# Change the value
>>> view.CenterAxesVisibility = 0
```

### 1.4.3 Display properties

Display properties refers to available parameters that control how data from a pipeline module is displayed in a view, e.g., choosing to view the output mesh as a wireframe, coloring the mesh using a data attribute, and selecting which attributes to plot in chart view. A set of display properties is associated with a particular pipeline module and view. Thus, if the data output from a source is shown in two views, there will be two sets of display properties used to control the appearance of the data in each of the two views.

#### Display properties in `paraview`

Display properties are accessible from the `Display` section on the `Properties` panel. When the active source or active view changes, this section updates to show the display properties for the active source in the active view, if available. If the active source produces data that cannot be shown (or has never been shown) in the view, then the `Display` properties section may be empty.

Similar to view properties, display property changes are immediately applied, without requiring the use of the `Apply` button.

#### Display properties in `pvpython`

To access display properties in `pvpython`, you can use `SetDisplayProperties` and `GetDisplayProperty` methods.

```
# Using SetDisplayProperties/GetDisplayProperties to access the display
# properties for the active source in the active view.

>>> print(GetDisplayProperties("Opacity"))
1.0

>>> SetDisplayProperties(Opacity=0.5)
```

Alternatively, you can get access to the display properties object using `GetDisplayProperties` and then changing properties directly on the object.

```
# Get display properties object for the active source in the active view.
>>> disp = GetDisplayProperties()

# You can also save the object returned by Show.
>>> disp = Show()

# Now, you can directly access the properties.
>>> print(disp.Opacity)
0.5

>>> disp.Opacity = 0.75
```

As always, you can use the `help` method to discover available properties on a display object.

```
>>> disp = Show()

>>> help(disp)
>>> help(a)
Help on GeometryRepresentation in module paraview.servermanager object:

```

(continues on next page)

(continued from previous page)

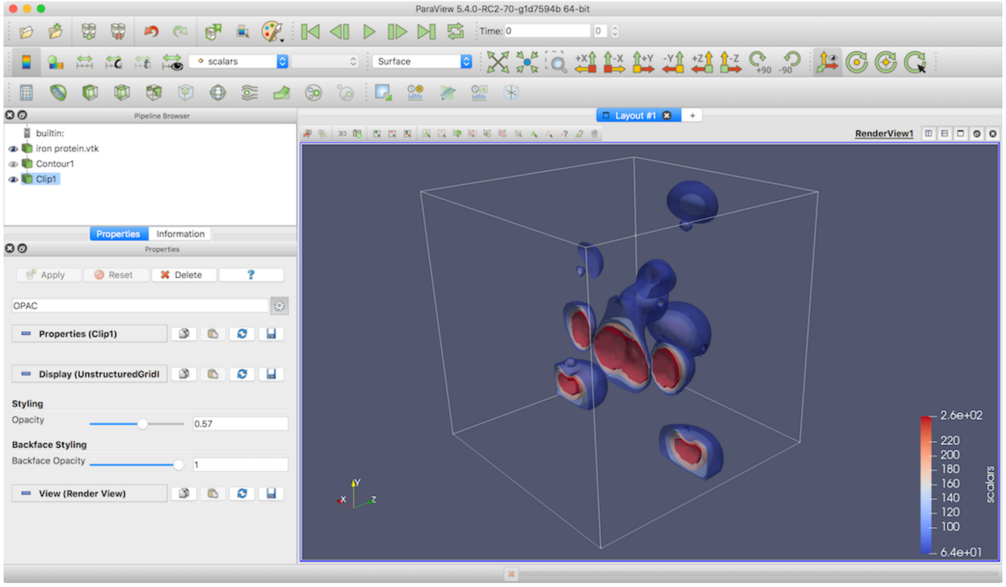
```
class GeometryRepresentation(SourceProxy)
 |  ParaView`s default representation for showing any type of
 |  dataset in the render view.
 |
 |  Method resolution order:
 |      GeometryRepresentation
 |      SourceProxy
 |      Proxy
 |      __builtin__.object
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  ...
 |
 |  CenterStickyAxes
 |      Keep the sticky axes centered in the view window.
 |
 |  ColorArrayName
 |      Set the array name to color by. Set it to empty string
 |      to use solid color.
 |
 |  ColorAttributeType
 |  ...
```

### 1.4.4 Render View

The `Render View` is the most commonly used view in **ParaView**. It is used to render geometries and volumes in a 3D scene. This is the view that you typically think of when referring to 3D visualization. The view relies on techniques to map data to graphics primitives such as triangles, polygons, and voxels, and it renders them in a scene.

Most of the scientific datasets discussed in Section 1.3.1 are composed of meshes. These meshes can be mapped to graphics primitives using several of the established visualization techniques. That is, you can compute the outer surface of these meshes and then render that surface as filled polygons, you can just render the edges, or you can render the data as a nebulous blob to get a better understanding of the internal structure in the dataset. Plugins, like `Digital Rock Physics`, can provide additional ways of rendering data using advanced techniques that provide more insight into the data.

If the dataset doesn't represent a mesh, e.g., a table (Section 1.3.1), you cannot directly show that data in this view. However, in such cases, it may be possible to construct a mesh by mapping columns in the table to positions to construct a point cloud, for example.



OPAC

Properties (Clip1)

Display (UnstructuredGridl

Styling

Opacity 0.57

Backface Styling

Backface Opacity 1

View (Render View)

Fig. 1.36: `paraview` using `Render View` to generate 3D visualizations from a dataset.

### Understanding the rendering process

`Render View` uses data processing techniques to map raw data to graphics primitives, which can then be rendered in a 3D scene. These mapping techniques can be classified as follows:

- *Surface rendering* methods provide general rendering by rendering a surface mesh for the dataset. For polygonal datasets (Section 1.3.1), this is simply the raw data. In cases of other datasets including structured (Section 1.3.1, Section 1.3.1, Section 1.3.1) and unstructured (Section 1.3.1) grids, this implies extracting a surface mesh for all external faces in the dataset and then rendering that mesh. The surface mesh itself can then be rendered as a filled surface or as a wireframe simply showing the edges, etc.
- *Slice* rendering is available for uniform rectilinear grid datasets (Section 1.3.1) where the visualization is generated by simply rendering an orthogonal slice through the dataset. The slice position and alignment can be selected using the display properties.
- *Volume* rendering generates rendering by tracing a ray through the dataset and accumulating intensities based on the color and opacity transfer functions set.

Each of these techniques are referred to as *representations*. When available, you can change the representation type from the display properties on the `Properties` panel or using the `Representation Toolbar` .

### Render View in `paraview`

### Creating a Render View

Unless you changed the default setting, a new `Render View` will be created when `paraview` starts up or connects to a new server. To create a `Render View` in `paraview`, split or close a view, and select the `Render View` button. You can also convert a view to a `Render View` (or any other type) by right-clicking on the view's title bar and picking from the `Convert To` sub-menu. It simply closes the chosen view and creates a selected view type in its place.

You can use the `Pipeline Browser` to control the visibility of datasets produced by pipeline modules in this view. The eyeball icons reflect the visibility state. Clicking on the eyeball icon will toggle the visibility state. If no eyeball icon is shown, it implies that the pipeline module doesn't produce a data type that can be directly shown in the active view, e.g., if the module produced a table, then when `Render View` is active, there will not be any eyeball icon next to that module.

### Interactions

You can interact with the Render View to move the camera in the scene for exploring the visualization and setting up optimal viewing angles. Each of the three mouse buttons, combined with keyboard modifier keys (`CTRL` or , and ), move the camera differently. The interaction mode can be changed from the `Camera` tab in the `Settings` dialog, which is accessible from *Tools > Settings* (or *ParaView > Preferences* on macOS). There are six interaction modes available in **ParaView**:

- *Pan* for translating the camera in the view plane.
- *Zoom* for zooming in or out of the center of the view.
- *Roll* for rolling the camera.
- *Rotate* for rotating the camera around the center of rotation.
- *Zoom To Mouse* for zooming in or out of the projected point under the mouse position.
- *Multi Rotate* for allowing azimuth and elevation rotations by dragging from the middle of the view and rolls by dragging from the edges.
- *Rotate Skybox* for rotating the environment skybox. Useful when using Environment Lighting and PBR shader.

The default interactions options are as follows:

| Modifier | Left Button | Middle Button | Right Button |
|---|---|---|---|
| | Rotate | Pan | Zoom |
| | Roll | Rotate | Pan |
| `CTRL` or | Rotate Skybox | Rotate | Zoom To Mouse |

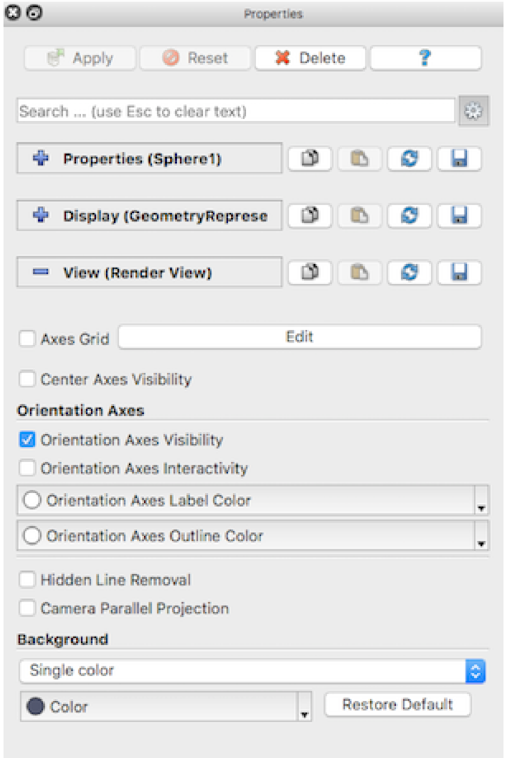
Usually in **ParaView**, you are interacting with a 3D scene. However, there are cases when you are working with a 2D dataset such as a slice plane or a 2D image. In such cases, `paraview` provides a separate set of interaction options suitable for 2D interactions. You can toggle between the default 3D interaction options and 2D interaction options by clicking the `2D` or `3D` button in the view toolbar. The default interaction options for 2D interactions are as follows:

| Modifier | Left Button | Middle Button | Right Button |
|---|---|---|---|
| | Pan | Roll | Zoom |
| | Zoom | Zoom | Zoom To Mouse |
| `CTRL` or | Roll | Pan | Rotate |

By default, **ParaView** will determine whether your data is 2D or 3D when loading the data and will set the interaction mode accordingly. This behavior can be changed in the `Settings` dialog by changing the `Default Interaction Mode` setting under the `Render View` tab. The default setting is “Automatic, based on the first time step”, but the setting can be changed to “Always 2D” or “Always 3D” in case you wish to force the interaction mode.

### View properties

Several of the view properties in `Render View` control the annotations shown in the view (Fig. 1.37).



□ Axes Grid

□ Center Axes Visibility

Orientation Axes

☑ Orientation Axes Visibility

□ Orientation Axes Interactivity

○ Orientation Axes Label Color

○ Orientation Axes Outline Color

□ Hidden Line Removal

□ Camera Parallel Projection

Background

Single color

● Color

Fig. 1.37: The `Properties` panel showing view properties for `Render View` .

`Axes Grid` refers to an annotation axis rendered around all datasets in the view (Fig. 1.38). You use the checkbox next to the `Edit Axis Grid` button to show or hide this annotation. To control the annotation formatting, labels, etc., click on the `Edit Axes Grid...` button. The `Axes Grid` is described in Chapter Section 2.11.

The *Center axes* refers to axes rendered in the scene positioned as the center of rotation, i.e., the location is space around which the camera revolves during `Rotate` camera interaction. `Center Axes Visibility` controls the visibility of the center axes.

The *Orientation axes* is the widget shown at the lower-left corner by default, which is used to get a sense for the orientation of the scene. The properties grouped under the `Orientation Axes` group allow you to toggle the visibility and the interactivity of this widget. When interactivity is enabled, you can click and drag this widget to the location of your choosing in the scene.

You can also change the `Background` used for this view. You can either set it as a `Single` color or as a `Gradient` changing between two colors, or you can select an `Image` (or texture) to use as the background.

There are two advanced properties you may wish to set: hidden line removal and camera parallel projection. The `Hidden Line Removal` option can be enabled to hide lines that would be occluded by a solid object when drawing objects in a `Wireframe` representation. If you want to render your data using parallel projection instead of the default perspective projection you can check the `Camera Parallel Projection` checkbox.

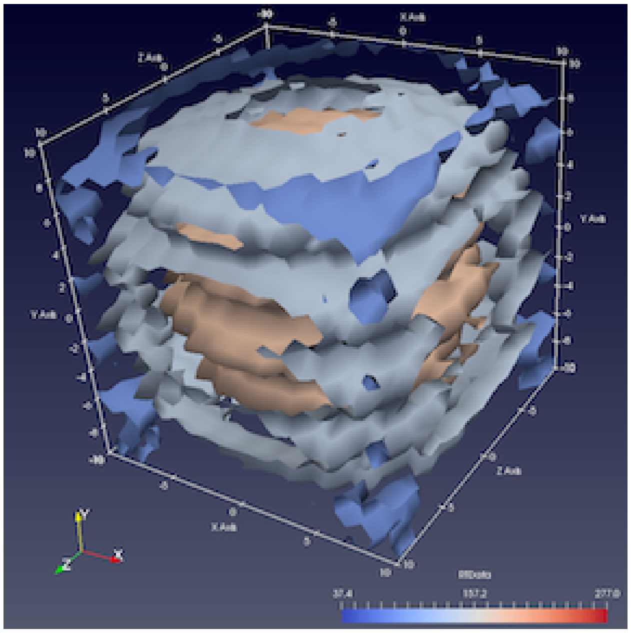
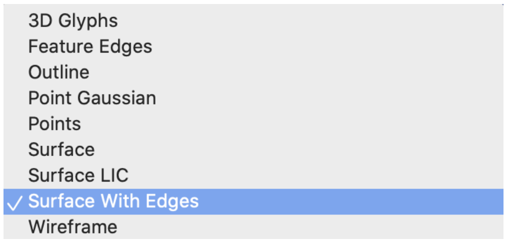


Fig. 1.38: `Axes Grid` is used to annotate data bounds in `Render View` .

### Display properties



One of the first (and probably the most often used) display properties is `Representation` . `Representation` allows you to pick one of the *mapping* modes. The options available depend on the data type, as well as the plugins loaded. While all display properties are accessible from the advanced view for the `Properties` panel, certain properties may be shown/hidden from the default view based on the chosen representation type.

The `Outline` representation can be used to render an outline for the dataset. This is arguably the fastest way of rendering the dataset since only the bounding box is rendered. Scalar coloring options, i.e., selecting an array with which to color, has no effect on this representation type. You can still, however, change the `Solid Color` to use as well as the `Opacity` . To change the color, select `Solid Color` in the combo-box under `Coloring` , and then click `Edit` to pick the color to use. To change the opacity, simply change the `Opacity` slider. 0 implies total transparency and, hence, invisiblity, while 1 implies totally opacity.

**Did you know?**

Rendering translucent data generally adds computational costs to the rendering process. Thus, when rendering large datasets, you may want to leave changing opacities to anything less than 1 to the very end, after having set up the visualization. In doing so, you avoid translucent geometries during exploration, but use them for generating images or screenshots for presentations and publications.

`Points` , `Surface` , `Surface With Edges` , and `Wireframe` rely on extracting the surface mesh from the dataset and

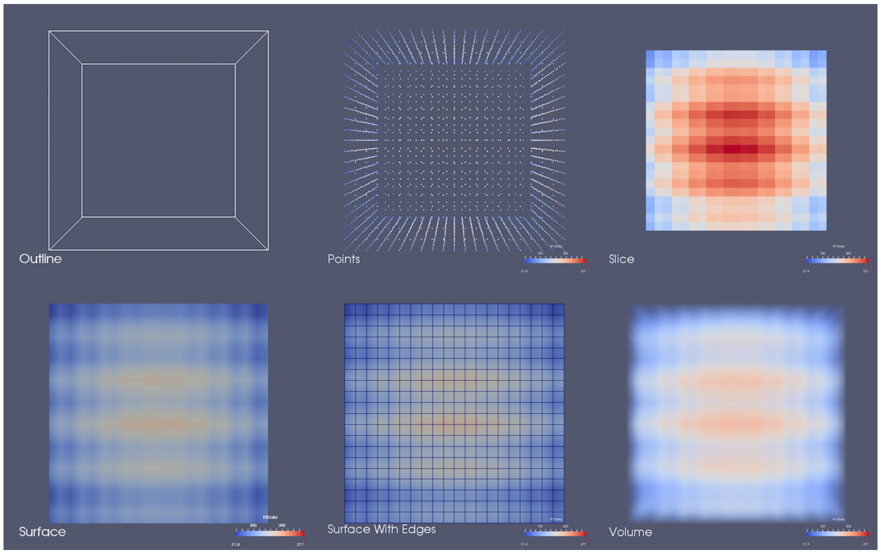


Fig. 1.39: Different renderings generated by rendering data produced by the `Wavelet` source as outline, points, slice, surface, surface with edges, and volume.

then rendering that either as a collection of points, as solid surface, as solid surface with cell boundaries highlighted, or as a wireframe of cell boundaries only. `Feature Edges` is a subset of `Wireframe` consisting of prominent edges on the surface such as edges between cells that form a sharp angle or edges with only one adjacent cell. For these representations, you can either set a single solid color to use, as with `Outline` , or select a data array to use for scalar coloring (also known as pseudocoloring).

Two other representations are available for most datasets. `3D Glyphs` draws a copy of a 3D geometry (e.g., arrow, cone, or sphere, etc.), or glyph, at a subset of points in the dataset. These glyphs can be set to a single color or pseudocolored by a data array. The `Point Gaussian` representation is similar, but instead of drawing 3D geometry at every point, it draws a 2D image sprite that may have transparency. The image drawn can be one of several predefined image sprites such as `Gaussian Blur` , `Sphere` , `Black-edged circle` , `Plain cirlce` , `Triangle` , or `Square outline` , or a custom sprite can be defined with custom GLSL shader code.

Lastly, the `SurfaceLIC` representation is available for surface datasets with vector point data arrays. LIC stands for line integral convolution, which is a visualization technique that shows the direction of flow as a noise pattern smeared in the direction of flow.

**Did you know?**

For visualizations that feature 3D glyphs, it is typically much faster to use the `3D Glyph` representation rather than the `Glyph` filter. This is because the glyph representation draws the same geometry at many different locations (a graphics technique called geometry instancing) while the `Glyph` filter makes many copies of the same source geometry and renders the resulting mesh in its entirety. Generating all the glyphs and rendering them takes potentially a lot of memory and is typically slower to render, so you should use the `3D Glyph` representation when possible.

Next, we will cover each of the property groups available under `Display` properties. Several of these are marked as advanced. Accordingly, you may need to either toggle the panel to show advanced properties using the  button or search for it by name using the search box.

Display properties under `Coloring` allow you to set how the dataset is colored. To select a single solid color to use to fill the surface or color the wireframe or points, select `Solid Color` in the combo-box, and then click `Edit` . That will pop up the standard color chooser dialog from which you can pick a color to use.

If instead you want to pseudocolor using an attribute array available on the dataset, select that array name from the

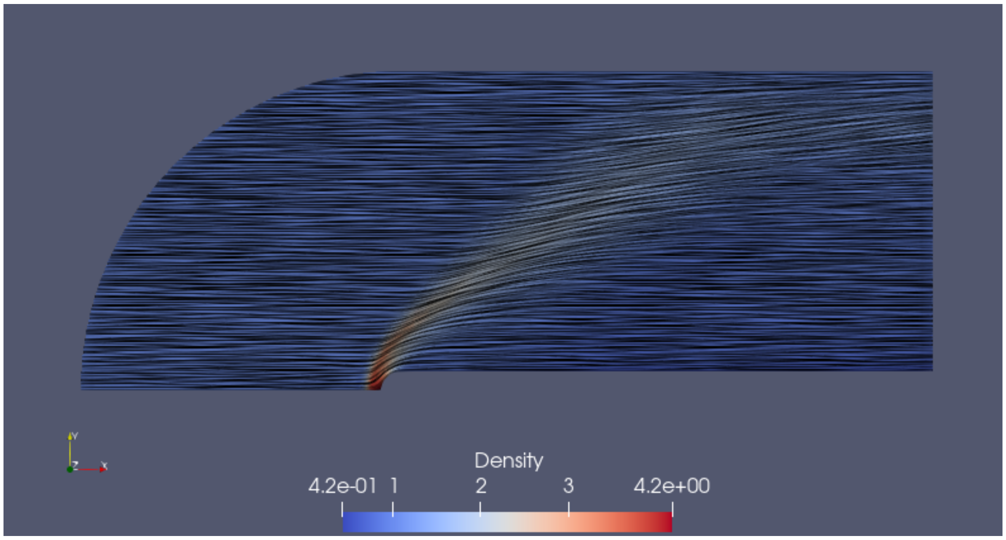
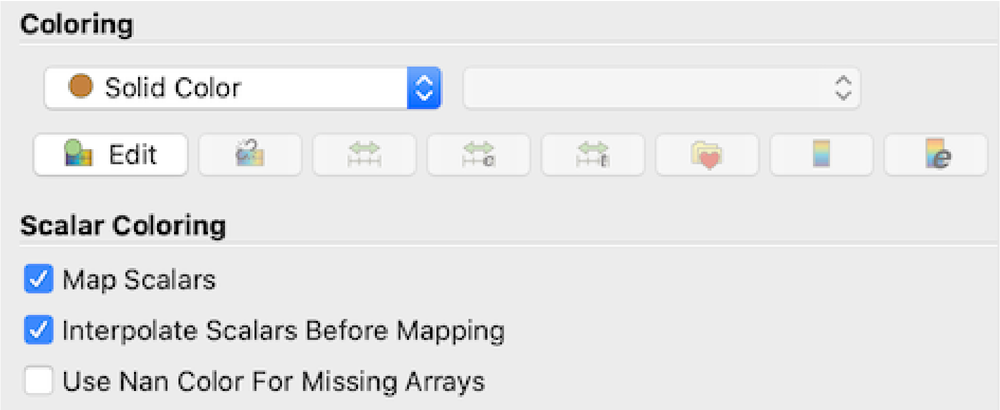


Fig. 1.40: An example of the `SurfaceLIC` representation showing the direction of a vector data array and colored by a different scalar array showing (Density).



Coloring

Solid Color

Scalar Coloring

☑ Map Scalars

☑ Interpolate Scalars Before Mapping

□ Use Nan Color For Missing Arrays

combo-box. For multi-component arrays, you can pick a particular component or `Magnitude` to use for scalar coloring. **ParaView** will automatically set up a color transfer function it will use to map the data array to colors. The default range for the transfer function is set up based on the `Transfer Function Reset Mode` general setting in the `Settings` dialog when the transfer function is first created. If another dataset is later colored by a data array with the same name, the range of the transfer function will be updated according to the `Automatic Rescale Range Mode` property in the `Color Map Editor` . To reset the transfer function range to the range of the data array in the selected dataset, you can use the `Rescale` button. Remember that, despite the fact that you can set the scalar array with which to color when rendering as `Outline` , the outline itself continues to use the specified solid color.

`Scalar Coloring` properties are only relevant when you have selected a data array with which to pseudocolor. The `Map Scalars` checkbox affects whether a color transfer function should be used (Fig. 1.41). If unchecked, and the data array can directly be interpreted as colors, then those colors are used directly. If not, the color transfer function will be used. A data array can be interpreted as colors if, and only if, it is an unsigned char, float, or double array with two, three, or four components. If the data array is unsigned char, the color values are defined between 0 and 255 while if the data array is float or double, the color values are expected to be between 0 and 1. `Interpolate Scalars Before Mapping` controls how color interpolation happens across rendered polygons. If on, scalars will be interpolated within polygons, and color mapping will occur on a per-pixel basis. If off, color mapping occurs at polygon points, and colors are interpolated, which is generally less accurate. Refer to the Kitware blog [PatMarion] for a detailed explanation of this option. `Use Nan Color For Missing Arrays` is a property that, if enabled, will use the special color designated for NaN values in a dataset to also be used as the color for parts of a composite dataset that are missing the scalars array used for color mapping.

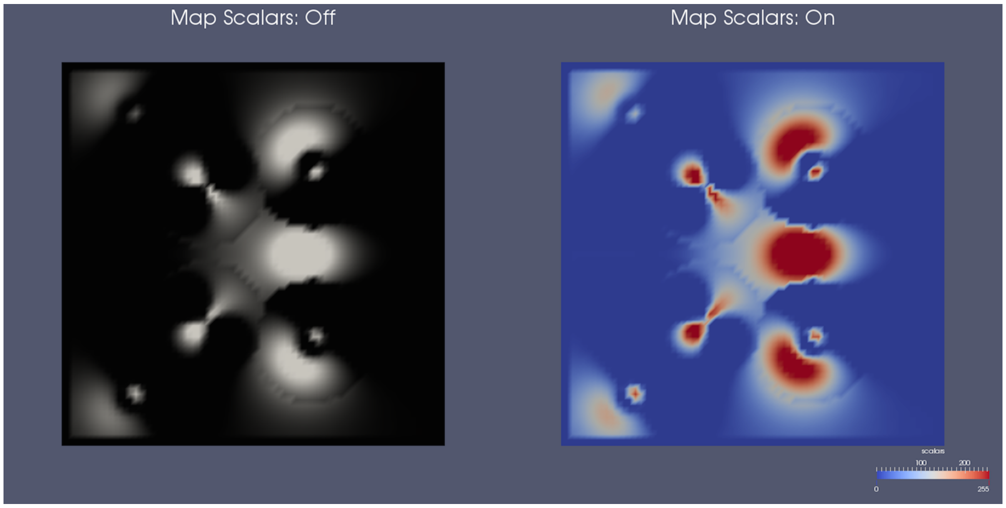


Fig. 1.41: The `Map Scalars` property can be used to avoid using a transfer function and directly interpreting the array values as colors, if possible.

The `Polar Axes` checkbox toggles polar axes shown around the data. Many parameters can be accessed via an `Edit button` alongside it. The parameters include angles, tick range, labels, logarithmic mode, ellipse ratio and more.

`Styling` properties include `Opacity` (useful when rendering translucent geometries), `Point Size` (used to control size of points rendered with using `Points` representation), and `Line Width` (used to control the thickness of lines when rendering as `Wireframe` or that of the edges when rendering as `Surface With Edges` .

`Lighting` properties affect the shading for rendered surfaces. `Interpolation` allows you to pick between `Flat` and `Gouraud` shading. `Specular` , together with `Specular Color` and `Specular Power` , affects the shininess of the surface. Set this to a non-zero value to render shiny, metallic surfaces.

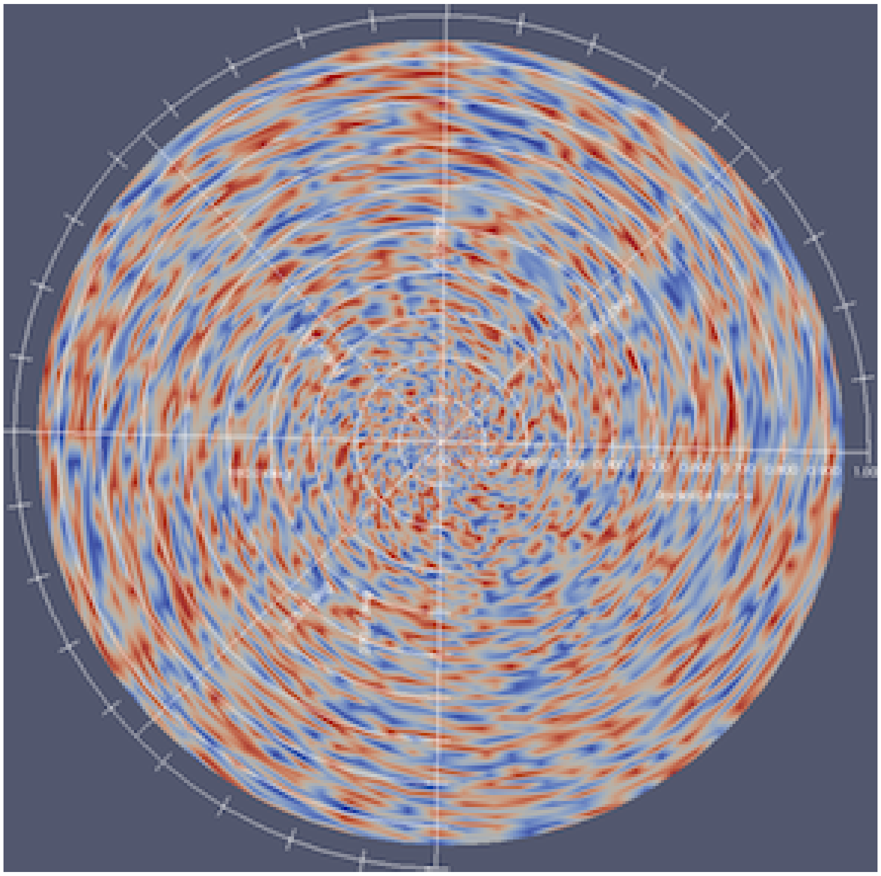
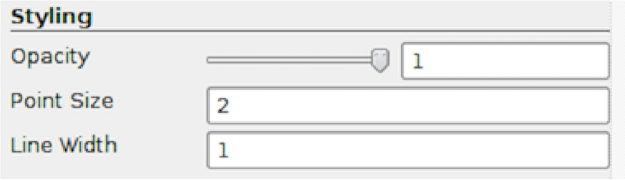


Fig. 1.42: A `Polar Axes` usage example.

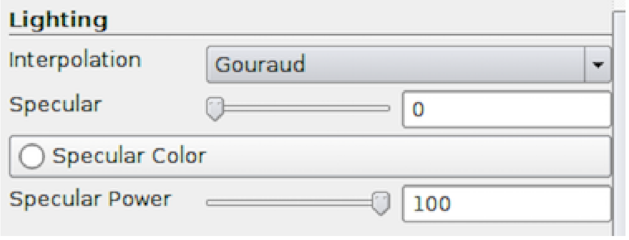


**Styling**

Opacity 1

Point Size 2

Line Width 1



**Lighting**

Interpolation Gouraud

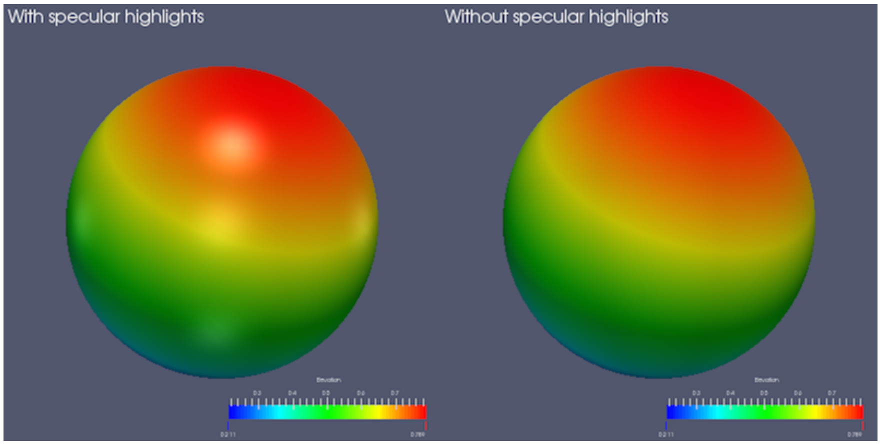
Specular 0

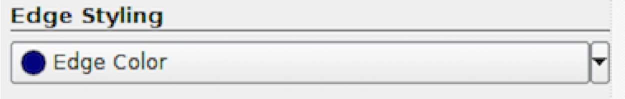
○ Specular Color

Specular Power 100

### Common Errors

Specular highlights can lead to misinterpretation of scalar values when using scalar coloring, since the color shown on the shiny part of the surface will not correspond to any color on the color transfer function. Hence, it is generally advisable to use specular highlights on surfaces colored with a single solid color and not on those using scalar coloring (or pseudocoloring).

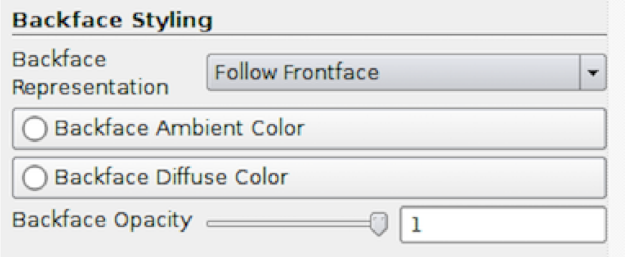




Edge Styling

● Edge Color

Edge Styling allows you to set the Edge Color with which to color the edges when using Surface With Edges representation.



Backface Styling

Backface Representation: Follow Frontface

○ Backface Ambient Color

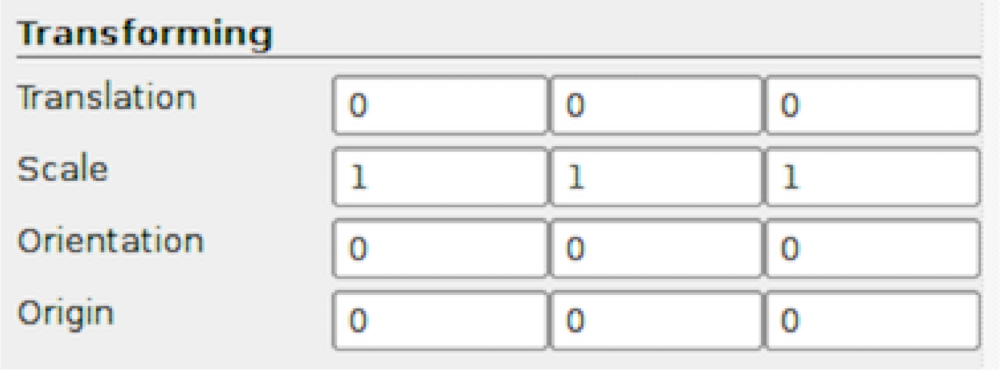
○ Backface Diffuse Color

Backface Opacity: 1

Backface Styling provides advanced controls to fine-tune the rendering by controlling front and back faces. A front face is any mesh face facing the camera, while a back face is the one facing away from the camera. By choosing to Cull Frontface or Cull Backface , or by selecting a specific representation type to use for the backface, you can customize your visualizations.

Transforming properties can be used to transform the rendered data in the scene without affecting the raw data itself. Thus, if you apply filters on the data source, it will indeed be working with the untransformed data. To transform the data itself, you should use the Transform filter.

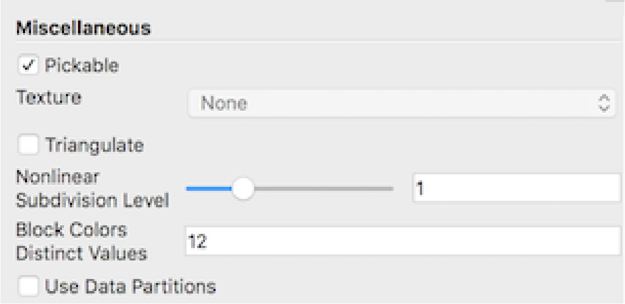
The Coordinate Shift Scale Method is used to choose how to normalize point coordinates to improve rendering quality. Mesh points are sent to the GPU as single-precision float data which can result in resolution issues due to limited precision. VTK includes a variety of methods to normalize the point coordinates to a better range for single-precision floats prior to sending them to the GPU. Auto Shift Scale is a good setting that should work for most datasets - it recomputes a shift and scale factor according to a heuristic involving dataset size and position relative to the origin. Always Auto Shift Scale recomputes the shift and scale every time. Auto Shift Only only shifts



Transforming

| | | | |
|---|---|---|---|
| Translation | 0 | 0 | 0 |
| Scale | 1 | 1 | 1 |
| Orientation | 0 | 0 | 0 |
| Origin | 0 | 0 | 0 |

the data - this is useful when data is far away from the origin. `Near Focal Plane Shift Scale` and `Focal Point Shift Scale` works based on the current camera near clipping point and viewpoint, respectively. This makes it the most robust setting, especially for very large datasets, but it will renormalize the points occasionally as the camera's settings change. Renormalizing points requires reuploading the data to the GPU, so there may be a performance cost with these last methods.



Miscellaneous

☑ Pickable

Texture: None

□ Triangulate

Nonlinear Subdivision Level: 1

Block Colors Distinct Values: 12

□ Use Data Partitions

Several properties are available under the `Miscellaneous` group. Uncheck the `Pickable` option if you want the dataset to be ignored when making selections. If the dataset has a texture coordinates array, you can apply a texture to the dataset surface using the `Texture` combo-box. Choose `Load` to load a texture or apply a previously loaded texture listed in the combo-box. If your dataset doesn't have texture coordinates, you can create them by applying one of `Texture Map to Cylinder` , `Texture Map to Sphere` , or `Texture Map To Plane` filters, or using the filters `Calculator` or `Programmable Filter` .

The `Triangulate` option is useful for rendering objects with non-convex polygons. It comes with some additional processing cost for converting polygons to triangles, so it should be used only when necessary.

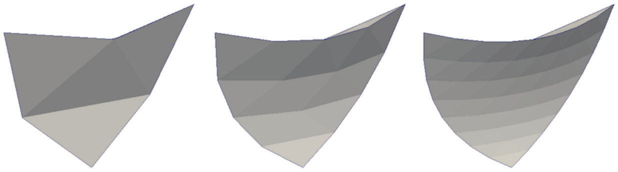


Fig. 1.43: A dataset made of quadratic tetra hedra displayed with 1, 2, and 3 levels of nonlinear subdivision.

The property `Use Shader Replacements` enables you to customize the shader code VTK uses for rendering by specifying shader replacements with a JSON string. The JSON string can be a single node or an array of nodes with the following properties:

- "type": specifies the type of shader the replacement is about. It can be either "vertex", "fragment" or "geometry".
- "original": specifies the original string to be replaced in the shader code. This string is generally a pattern defined by the mapper vtkOpenGLPolyDataMapper at specific locations of the shader GLSL source code.

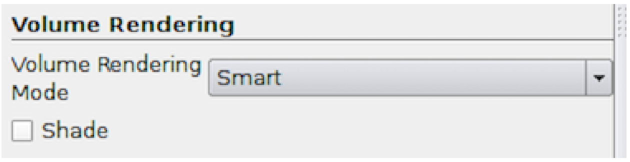
- "replacement": specifies the replacement string in GLSL source code. Note that the Json parser supports multiple lines entries.

Here's an example of a simple shader replacement (draw all the fragments in full red color without any shading consideration):

```
{
  "type": "fragment",
  "original": "//VTK::Light::Impl",
  "replacement": "gl_FragData[0]=vec4(1.0,0.0,0.0,1.0);"
}
```

The `Nonlinear Subdivision Level` property is used when rendering datasets with higher- order elements. Use this to set the subdivision level for triangulating higher order elements. The higher the value, the smoother the edges. This comes at the cost of more triangles and, hence, potentially, increased rendering time.

The `Block Colors Distinct Values` property sets the number of unique colors to use when coloring multiblock datasets by block ID. Finally, `Use Data Partitions` controls whether data is redistributed when it is rendered translucently. When off (default value), data is repartitioned by the compositing algorithm prior to rendering. This is typically an expensive operation that slows down rendering. When this option is on, the existing data partitions are used, and the cost of data restribution is avoided. However, if the partitions are not sortable in back-to-front order, rendering artifacts may occur.

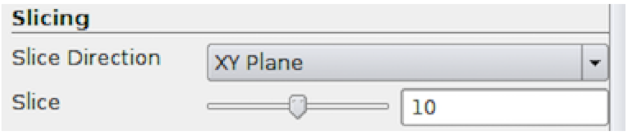


Volume Rendering

Volume Rendering Mode: Smart

□ Shade

`Volume Rendering` options are available if the data can be volume rendered. You can pick a specific type of `Volume Rendering Mode` , although the default ( `Smart` ) should work in most cases, since it attempts to pick a volume rendering mode suitable for your data and graphics setup. To enable gradient-based shading, check `Shade` , if available.



Slicing

Slice Direction: XY Plane

Slice: 10

`Slicing` properties are available when the `Slice` representation type is present. These allow you to pick the orthogonal slice plane orientation and slice offset using `Slice Direction` and the `Slice` slider.

### Render View in `pvpython`

### Creating a Render View

You use `CreateRenderView` or `CreateView` functions to create a new instance of a render view.

```
>>> from paraview.simple import *
>>> view = CreateRenderView()
# Alternatively, use CreateView.
>>> view = CreateView("RenderView")
```

noindent You use `Show` and `Hide` to show or hide data produced by a pipeline module in the view.

```
>>> source = Sphere()
>>> view = CreateRenderView()

# Show active source in active view.
>>> Show()

# Or specify source and view explicitly.
>>> Show(source, view)

# Hide source in active view.
>>> Hide(source)
```

### Interactions

Since pvpython is designed for scripting and batch processing, it has limited support for direct interaction with the view. To interact with a scene, invoke the Interact function in Python.

```
Interact()
```

More often, you will programmatically change the camera as follows:

```
# Get camera from the active view, if possible.
>>> camera = GetActiveCamera()

# or, get the camera from a specific render view.
>>> camera = view.GetActiveCamera()

# Now, you can use methods on camera to move it around the scene.

# Divide the camera's distance from the focal point by the given dolly value.
# Use a value greater than one to dolly-in toward the focal point, and use a
# value less than one to dolly-out away from the focal point.
>>> camera.Dolly(10)

# Set the roll angle of the camera about the direction of projection.
>>> camera.Roll(30)

# Rotate the camera about the view up vector centered at the focal point. Note
# that the view up vector is whatever was set via SetViewUp, and is not
# necessarily perpendicular to the direction of projection. The result is a
# horizontal rotation of the camera.
>>> camera.Azimuth(30)

# Rotate the focal point about the view up vector, using the camera's position
# as the center of rotation. Note that the view up vector is whatever was set
# via SetViewUp, and is not necessarily perpendicular to the direction of
# projection. The result is a horizontal rotation of the scene.
>>> camera.Yaw(10)

# Rotate the camera about the cross product of the negative of the direction
# of projection and the view up vector, using the focal point as the center
# of rotation. The result is a vertical rotation of the scene.
```

(continues on next page)

(continued from previous page)

```
>>> camera.Elevation(10)

# Rotate the focal point about the cross product of the view up vector and the
# direction of projection, using the camera's position as the center of
# rotation. The result is a vertical rotation of the camera.
>>> camera.Pitch(10)
```

Alternatively, you can explicitly set the camera position, focal point, view up, etc,. to explicitly place the camera in the scene.

```
>>> camera.SetFocalPoint(0, 0, 0)
>>> camera.SetPosition(0, 0, -10)
>>> camera.SetViewUp(0, 1, 0)
>>> camera.SetViewAngle(30)
>>> camera.SetParallelProjection(False)

# If ParallelProjection is set to True, then you'll need
# to specify parallel scalar as well i.e. the height of the viewport in
# world-coordinate distances. The default is 1. Note that the `scale'
# parameter works as an `inverse scale' where larger numbers produce smaller
# images. This method has no effect in perspective projection mode.
>>> camera.SetParallelScale(1)
```

### View properties

In pvpython, view properties are directly accessible on the view object returned by CreateRenderView or GetActiveView .

Once you get access to the view properties objects, you can then set properties on it similar to properties on pipeline modules such as sources, filters, and readers.

```
>>> view = GetActiveView()

# Set center axis visibility
>>> view.CenterAxesVisibility = 0

# Or you can use this variant to set the property on the active view.
>>> SetViewProperties(CenterAxesVisibility=0)

# Another way of doing the same
>>> SetViewProperties(view, CenterAxesVisibility=0)

# Similarly, you can change orientation axes related properties
>>> view.OrientationAxesVisibility = 0
>>> view.OrientationAxesLabelColor = (1, 1, 1)
```

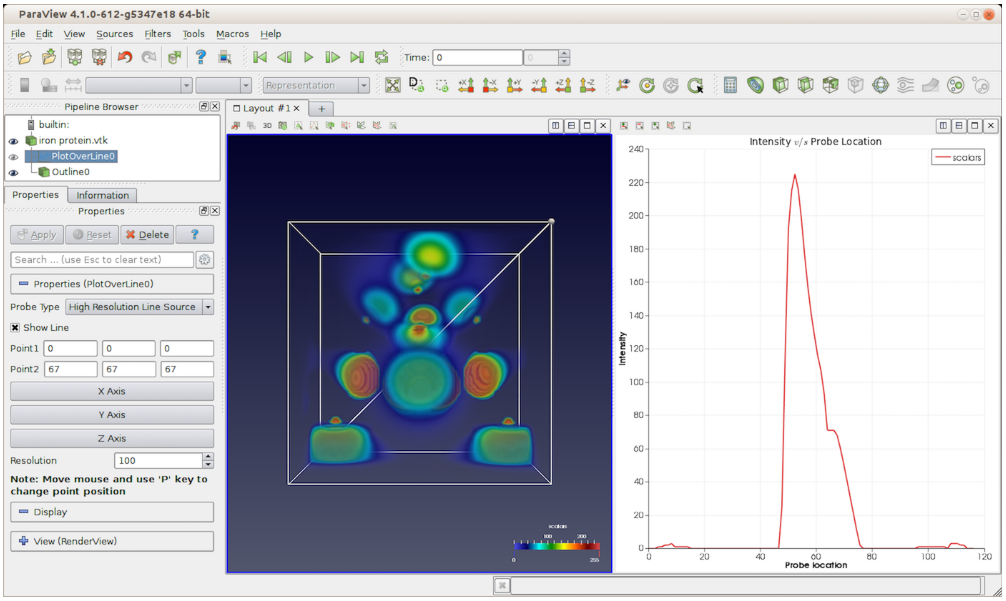
### Display properties

Similar to view properties, display properties are accessible from the display properties object or using the `SetDisplayProperties` function.

```
>>> displayProperties = GetDisplayProperties(source, view)
# Both source and view are optional. If not specified, the active source
# and active view will be used.

# Now one can change properties on this object
>>> displayProperties.Representation = "Outline"

# Or use the SetDisplayProperties API.
>>> SetDisplayProperties(source, view, Representation=Outline)

# Here too, source and view are optional and when not specified,
# active source and active view will be used.
```

You can always use the `help` function to get information about available properties on a display properties object.

## 1.4.5 Line Chart View



Time: 0

Probe Type High Resolution Line Source

☒ Show Line

Point1 0 0 0

Point2 67 67 67

Resolution 100

Note: Move mouse and use 'P' key to change point position

Fig. 1.44: `paraview` using `Line Chart View` to plot data values probed along a line through the dataset using `Plot Over Line` filter.

`Line Chart View` can be used to plot data as a line plot representing changes in dependent variables against an independent variable. Using display properties, you can also show scatter plots in this view. This view and other charting views in **ParaView** follow a similar design, where you pick attribute arrays to plot using display properties, and they are plotted in the view. How those values get plotted depends on the type of the view: `Line Chart View` draws a line connecting sample points, `Bar Chart View` renders bars at each sample point, etc.

One of the most common ways of showing a line plot is to apply the `Plot Over Line` filter to any dataset. This will probe the dataset along the probe line specified. You then plot the sampled values in the `Line Chart View` . Alternatively, if you have a tabular dataset (i.e. `vtkTable` ), then you can directly show the data in this view.

**Did you know?**

You can plot any arbitrary dataset, even those not producing `vtkTable` outputs, by using the `Plot Data` filter. Remember, however, that for extremely large datasets, while `Render View` may use parallel rendering strategies to improve performance and reduce memory requirements, chart views rarely, if ever, support such parallel strategies.

### Understanding plotting

`Line Chart View` plots data arrays. For any dataset being shown in the view, you first select which data array is to be treated as the independent variable and plotted along the x-axis. Then, you select which arrays to plot along the Y-axis. You can select multiple of these and setup properties for each of the series so they are rendered with different colors and line styles. Since data arrays in VTK datasets are associated with cells or points, and the two are not directly comparable to one another, you can only pick arrays associated with one type of attribute at any time.

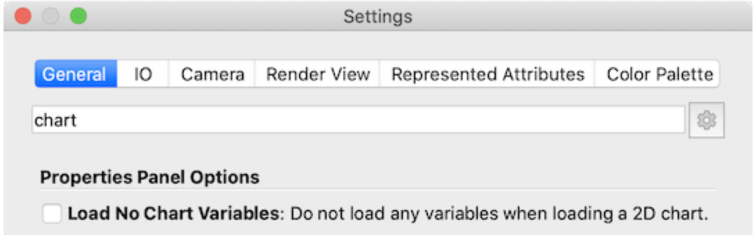
### Line Chart View in `paraview`

### Creating a Line Chart View

Similar to creating `Render View` , you can split the viewport or convert an existing view to `Line Chart View` . `Line Chart View` will also be automatically created if you apply a filter that needs this view, e.g., the `Plot Over Line` filter.

**Did you know?**

If you generate lengthy data for the `Line Chart View` , the default variables that are selected may be slow to adjust. You can change `paraview`'s default behavior to initially load no variables at all by selecting the `Load No Chart Variables` checkbox under `Settings/General/Properties Panel Options` .



chart

Properties Panel Options

□ **Load No Chart Variables:** Do not load any variables when loading a 2D chart.

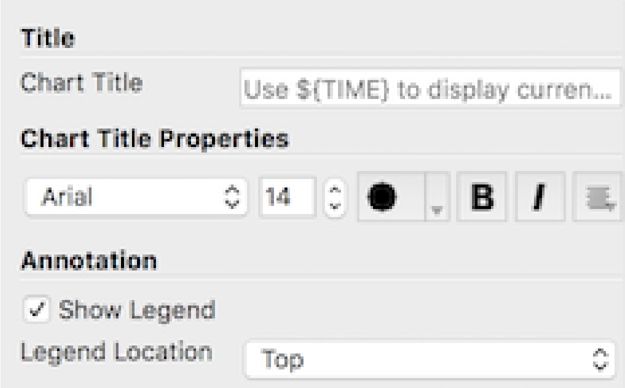
### Interactions

Interactions with the chart view result in changing the plotted axes ranges. You can left-click and drag to pan, i.e., change the origin. To change the range on either of the axes, you can right-click and drag vertically and/or horizontally to change the scale on the vertical axes and/or horizontal axes, respectively.

You can also explicitly specify the axes range using view properties.

### View properties

The view properties for `Line Chart View` are grouped as properties that affect the view and those that affect each of the potential four axes.



**Title**

Chart Title: Use ${TIME} to display curren...

**Chart Title Properties**

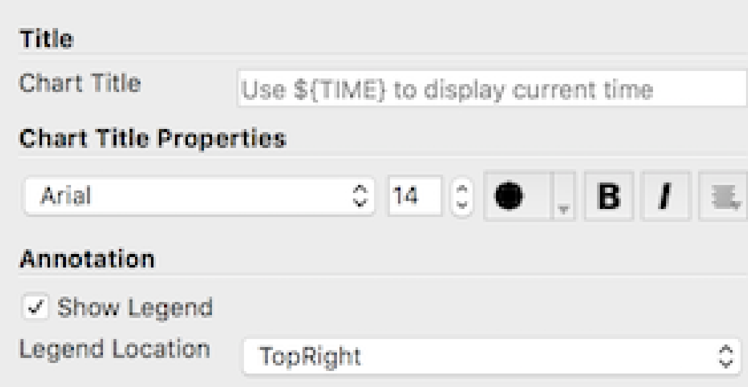
Arial | 14 | ● | B | I

**Annotation**

☑ Show Legend

Legend Location: Top

To set a title, use `Chart Title`. Title text properties such as font, size, style, and alignment with respect to the chart can be set under `Chart Title Properties`. To toggle the visibility of the legend, use `Show Legend`. While you cannot interactively place the legend in this view, you can use `Legend Location` to place it at one of the corners.



**Title**

Chart Title: Use ${TIME} to display current time

**Chart Title Properties**

Arial | 14 | ● | B | I

**Annotation**

☑ Show Legend

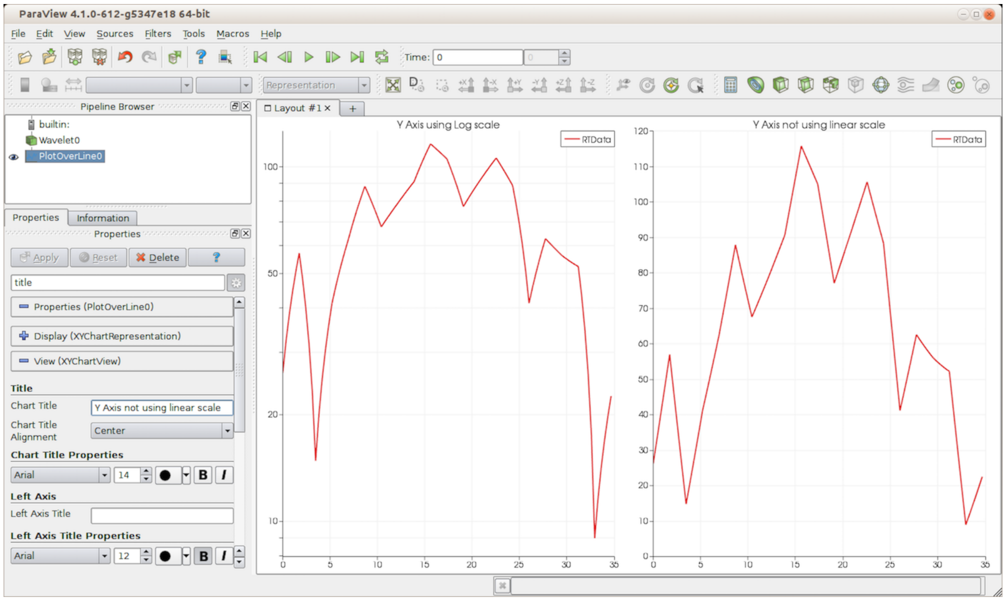
Legend Location: TopRight

There are four axes possible in this view: left, bottom, top, and right. The top and right axes are shown only when some series is set to use those. (We will cover this in the *Display properties* subsection.) For each of the axes, you can set a title (e.g., `Left Axis Title`, `Bottom Axis Title`, etc.) and adjust the title font properties. You can turn on a grid with a customizable color by checking the `Show Left Axis Grid`, for example.

Next, you can customize the axes ranges. You can always simply interact with the mouse to set the axes ranges. To precisely set the range, check the `Axis Use Custom Range` for the appropriate axis, e.g., `Bottom Axis Use Custom Range` for fixing the bottom axis range, and then specify the data values to use for the min and the max.

The labels on the axes are, by default, automatically determined to avoid visual clutter. By default, the axis labels are arranged on a linear scale, but by enabling the `Axis Log Scale` option you can use log scaling instead. In addition, you can override the default labelling strategy for any of the axes separately and, instead, specify the locations to label explicitly. This can be done by checking `Axis Use Custom Labels` for a particular axis, e.g., `Bottom Axis Use Custom Labels`. When checked, a list widget will be shown where you can manually add values at which labels will be placed.

For generating log plots, simply check the corresponding `Axis Use Log Scale`, e.g., `Left Axis Use Log Scale` to use log scale for Y-axis (Fig. 1.45). Note that log scale should only be used for an axis with a non-zero positive range, since the log of a number less than or equal to 0 is undefined.



title

Title

Chart Title: Y Axis not using linear scale

Chart Title Alignment: Center

Chart Title Properties: Arial, 14, **B** *I*

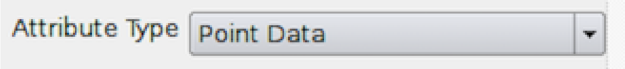
Left Axis

Left Axis Title:

Left Axis Title Properties: Arial, 12, **B** *I*

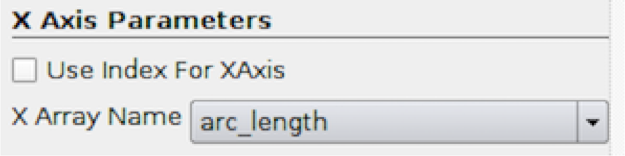
Fig. 1.45: Differences between line charts when using log scale for the Y-axis.

### Display properties



Attribute Type: Point Data

Display properties allow you to setup which series or data arrays are plotted in this view. You start by picking the `Attribute Type` . Select the attribute type that has the arrays of interest. For example, if you are plotting arrays associated with points, then you should pick `Point Data` .) Arrays with different associations cannot be plotted together. You may need to apply filters such as `Cell Data to Point Data` or `Point Data to Cell Data` to convert arrays between different associations to do that.



**X Axis Parameters**

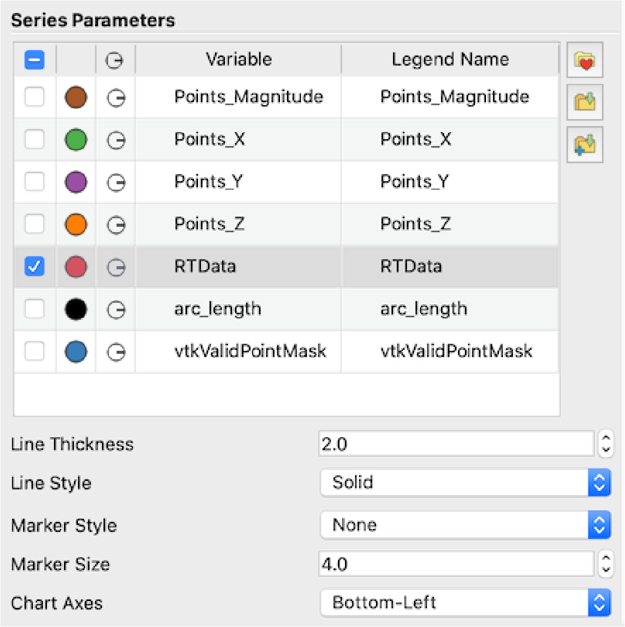
☐ Use Index For XAxis

X Array Name: arc_length

Properties under `X Axis Parameters` allow you to select the independent variable plotted on the X axis by choosing the `X Array Name` . If none of the arrays are appropriate, you can choose to use the element index in the array as the X axis by checking `Use Index for XAxis` .

`Series Parameters` control series or data arrays plotted on the Y-axis. All available data arrays are lists in the table widget that allows you to check/uncheck a series to plot in the first column. The second column in the table shows the associated color used to plot that series. You can double-click the color swatch to change the color to use. By default, **ParaView** will try to pick a palette of discrete colors. The third column lets you set the opacity of the series plot elements. The fourth column (`Variable`) shows the name of the variable to plot. The fifth column (`Legend Name`) shows the label to use for that series in the legend. By default, it is set to be the same as the array name. You can double-click to change the name to your choice, e.g., to add units.

Other series parameters include `Line Thickness`, `Line Style`, `Marker Style`, and `Marker Size`. To change any of these, highlight a row in The `Series Parameters` widget, and then change the associated parameter to affect the highlighted series. You can change properties for multiple series and can select multiple of them by using the `CTRL` (or



Series Parameters

| ☐ | | | Variable | Legend Name |
|---|---|---|---|---|
| ☐ | ● | | Points_Magnitude | Points_Magnitude |
| ☐ | ● | | Points_X | Points_X |
| ☐ | ● | | Points_Y | Points_Y |
| ☐ | ● | | Points_Z | Points_Z |
| ☑ | ● | | RTData | RTData |
| ☐ | ● | | arc_length | arc_length |
| ☐ | ● | | vtkValidPointMask | vtkValidPointMask |

| | |
|---|---|
| Line Thickness | 2.0 |
| Line Style | Solid |
| Marker Style | None |
| Marker Size | 4.0 |
| Chart Axes | Bottom-Left |

) and  keys.

Using `Chart Axes` , you can change which axes on which a series is shown. The default is `Bottom-Left` , but you can change it to be `Bottom-Right` , `Top-Left` , or `Top-Right` to accommodate series with widely different ranges in the same plot.

### Line Chart View in `pvpython`

The principles involved in accessing `Line Chart View` from `pvpython` are similar to those with `Render View` . You work with the view properties and display properties objects to change views and display properties, respectively. The thing that changes is the set of available properties.

The following script demonstrates the typical usage:

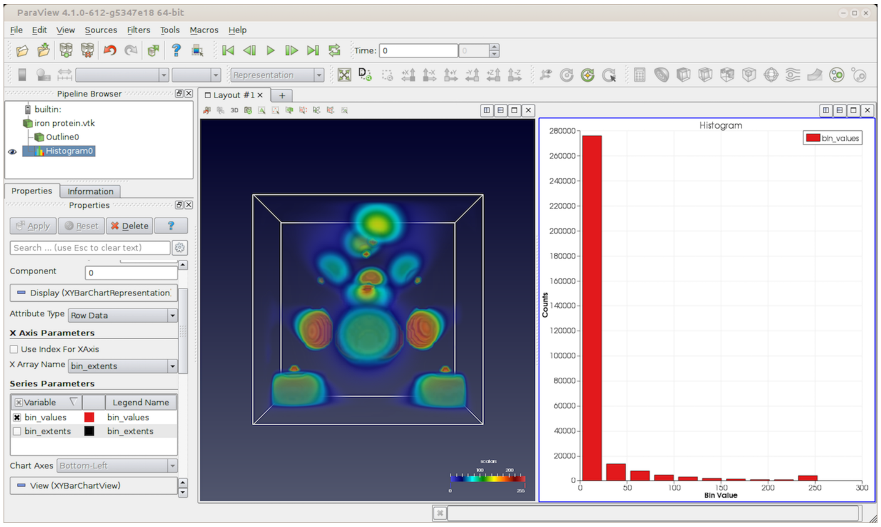
```
>>> from paraview.simple import *

# Create a data source to probe into.
>>> Wavelet()
<paraview.servermanager.Wavelet object at 0x1156fd810>

# We update the source so that when we create PlotOverLine filter
# it has input data available to determine good defaults. Otherwise,
# we will have to manually set up the defaults.
>>> UpdatePipeline()

# Now, create the PlotOverLine filter. It will be initialized using
# defaults based on the input data.
>>> PlotOverLine()
<paraview.servermanager.PlotOverLine object at 0x1156fd490>

# Show the result.
>>> Show()
```

(continues on next page)

(continued from previous page)

```
<paraview.servermanager.XYChartRepresentation object at 0x1160a6a10>

# This will automatically create a new Line Chart View if the
# the active view is no a Line Chart View since PlotOverLine
# filter indicates it as the preferred view. You can also explicitly
# create it by using CreateView() function.

# Display the result.
>>> Render()

# Access display properties object.
>>> dp = GetDisplayProperties()
>>> print(dp.SeriesVisibility)
['arc_length', '0', 'RTData', '1']

# This is  list with key-value pairs where the first item is the name
# of the series, then its visibility and so on.
# To toggle visibility, change this list e.g.
>>> dp.SeriesVisibility = ['arc_length', '1', 'RTData', '1']

# Same is true for other series parameters including series color,
# line thickness etc.

# For series color, the value consists of 3 values: red, green, and blue
# color components.
>>> print(dp.SeriesColor)
['arc_length', '0', '0', '0', 'RTData', '0.89', '0.1', '0.11']

# For series labels, value is the label to use.
>>> print(dp.SeriesLabel)
['arc_length', 'arc_length', 'RTData', 'RTData']

# e.g. to change RTData's legend label, we can do something as follows:
>>> dp.SeriesLabel[3] = 'RTData -- new label'

# Access view properties object.
>>> view = GetActiveView()
# or
>>> view = GetViewProperties()

# To change titles
>>> view.ChartTitle = "My Title"
>>> view.BottomAxisTitle = "X Axis"
>>> view.LeftAxisTitle = "Y Axis"
```

### 1.4.6 Bar Chart View



Time: 0

Component: 0

Attribute Type: Row Data

☐ Use Index For XAxis

X Array Name: bin_extents

| ☒ Variable | | Legend Name |
|---|---|---|
| ☒ bin_values | 🟥 | bin_values |
| ☐ bin_extents | ⬛ | bin_extents |

Chart Axes: Bottom-Left

Fig. 1.46: `paraview` using `Bar Chart View` to plot the histogram for a dataset using the `Histogram` filter.

`Bar Chart View` is very similar to `Line Chart View` when it comes to creating the view, view properties, and display properties. One difference is that, instead of rendering lines for each series, this view renders bars. In addition, under the display properties, the `Series Parameters` like `Line Style` and `Line Thickness` are not available, since they are not applicable to bars.

### 1.4.7 Box Chart View

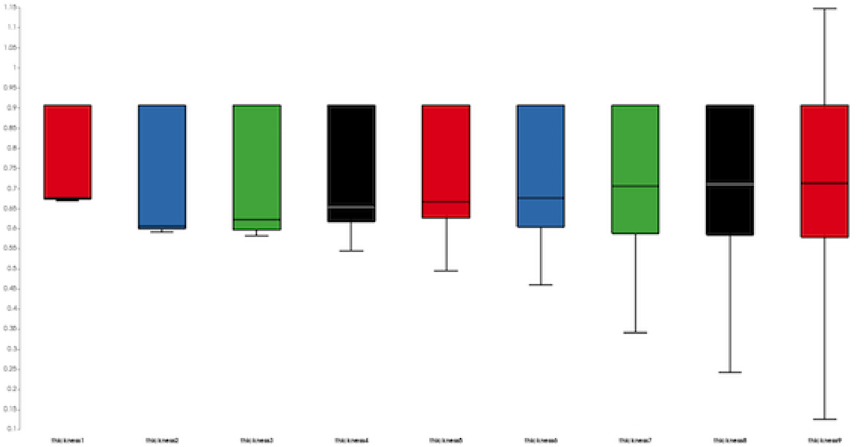
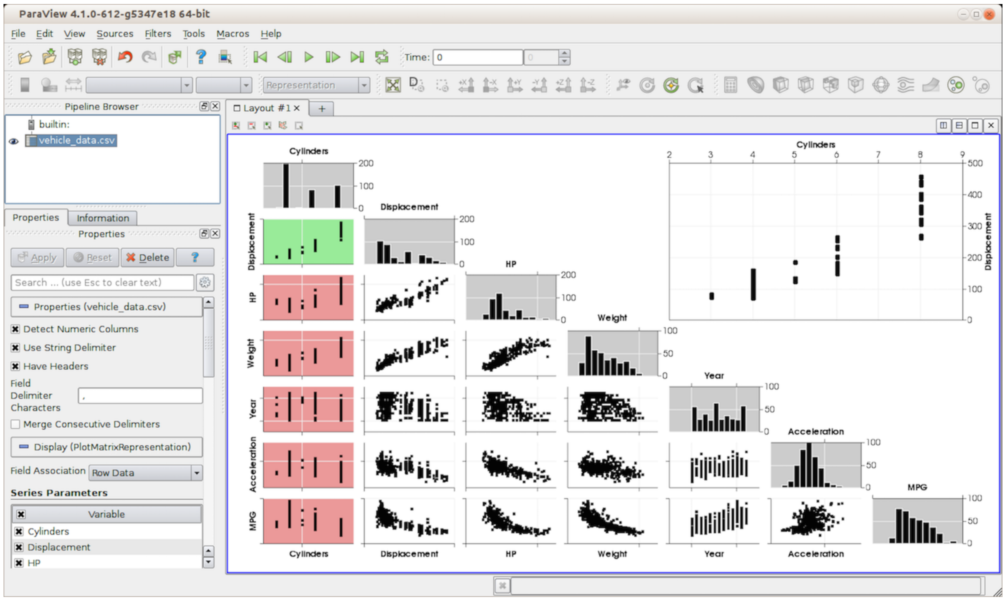


Fig. 1.47: `paraview` using `Box Chart View` to plot the box chart for a dataset using the `Compute Quartiles` filter.

Box plot is a standard method for graphically depicting groups of statistical data based on their quartiles. A box plot is represented by a box with the following properties: the bottom of the rectangle corresponds to the first quartile, a horizontal line inside the rectangle indicates the median and the top of the rectangle corresponds to the third quartile. The maximum and minimum values are depicted using vertical lines that extend from the top and the bottom of the rectangle.

In **ParaView**, the `Box Chart View` can be used to display such box plots through the `Compute Quartiles` filter which computes the statistical data needed by the view to draw the box plots.

### 1.4.8 Plot Matrix View



Properties (vehicle_data.csv)

☒ Detect Numeric Columns

☒ Use String Delimiter

☒ Have Headers

Field Delimiter Characters ,

☐ Merge Consecutive Delimiters

Display (PlotMatrixRepresentation)

Field Association Row Data

Series Parameters

| ☒ | Variable |
|---|---|
| ☒ | Cylinders |
| ☒ | Displacement |
| ☒ | HP |

Fig. 1.48: `paraview` using `Plot Matrix View` to generate a scatter plot matrix to understand correlations between pairs of variables.

`Plot Matrix View` is a chart view that renders a scatter plot matrix. It allows you to spot patterns in the small scatter plots, change focus to those plots of interest, and perform basic selection. The principle is that, for all selected arrays or series to be plotted, the view generates a scatter plot for each pair. You can activate a particular scatter plot, in which case the active plot is re-drawn at a bigger scale to make it easier to inspect. Additionally, the view shows a histogram for each plotted variable or series.

The view properties allow you to set colors to use for active plot, histograms, etc., while the display properties allow you to pick which series are plotted.

#### Interactions

You can click on any of the plots (except the histograms) in the matrix to make it active. Once actived, the active plot will show that plot. You can then interact with the active plot exactly like `Line Chart View` or `Bar Chart View` for panning and zoom.

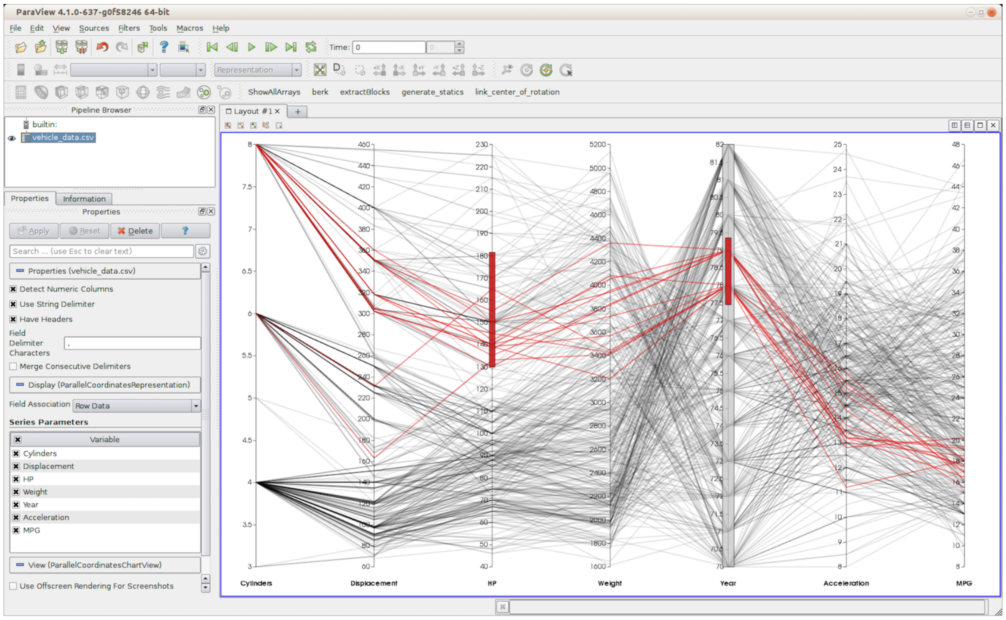
#### View properties

View properties on this view allow you to pick styling parameters for the rendering ranging from title ( `Chart Title` ) to axis colors ( `Active Plot Axis Color` , `Active Plot Grid Color` ). You can also control the visibility of the histrogram plots, the active plot, the axes labels, the grids, and so on.

### Display properties

Similar to `Line Chart View` , you select the `Attribute Type` and then the arrays to plot. Since, in a scatter plot matrix, the order in which the selected series are rendered can make it easier to understand correlations, you can change the order by clicking and dragging the rows in the `Series Parameters` table.

## 1.4.9 Parallel Coordinates View


ParaView 4.1.0-637-g0f58246 64-bit
File Edit View Sources Filters Tools Macros Help
Time: 0
Representation
ShowAllArrays berk extractBlocks generate_statics link_center_of_rotation
Pipeline Browser
builtin:
vehicle_data.csv
Properties Information
Apply Reset Delete ?
Search ... (use Esc to clear text)
Layout #1
Cylinders
Displacement
HP
Weight
Year
Acceleration
MPG

Properties (vehicle_data.csv)

☒ Detect Numeric Columns

☒ Use String Delimiter

☒ Have Headers

Field Delimiter Characters: ,

☐ Merge Consecutive Delimiters

Display (ParallelCoordinatesRepresentation)

Field Association: Row Data

**Series Parameters**

| ☒ | Variable |
|---|---|
| ☒ | Cylinders |
| ☒ | Displacement |
| ☒ | HP |
| ☒ | Weight |
| ☒ | Year |
| ☒ | Acceleration |
| ☒ | MPG |

View (ParallelCoordinatesChartView)

☐ Use Offscreen Rendering For Screenshots

Fig. 1.49: `paraview` using `Parallel Coordinates View` to plot correlations between columns in a table.

Like `Plot Matrix View` , `Parallel Coordinates View` is also used to visualize correlations between data arrays.

One of the main features of this view is the ability to select specific data in order to analyse the factors influencing the data. e.g., with a table of three variables, one being the "output" variable the other two being the potential factor influencing the first, selecting only the output will enable you to see if none, one, or both of the factors are actually influencing the output.

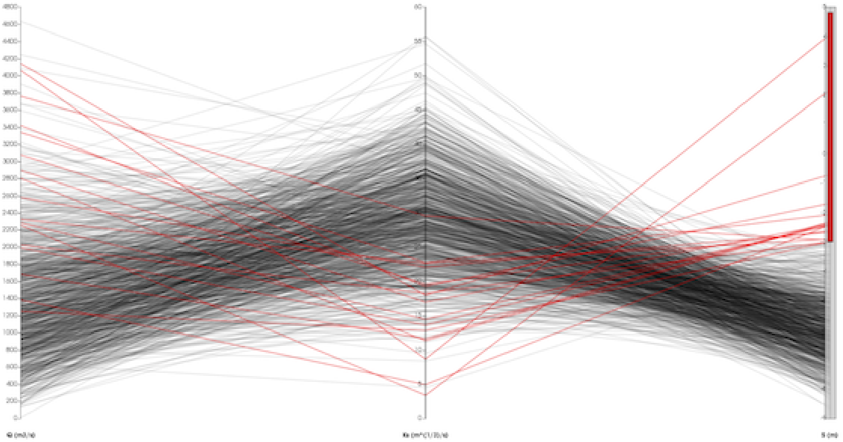
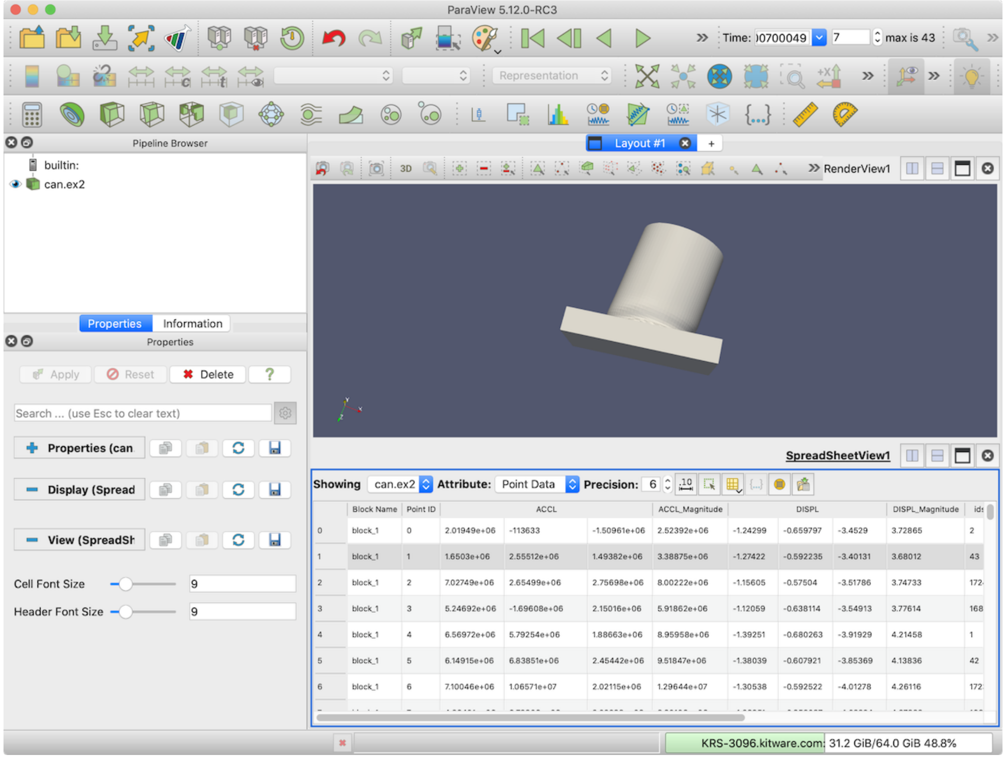


Fig. 1.50: High "S" data point are influenced more by low "Ks" than high "Q"

### 1.4.10 Spreadsheet View



Time: 0700049 7 max is 43

Cell Font Size 9

Header Font Size 9

Showing can.ex2 Attribute: Point Data Precision: 6

| | Block Name | Point ID | ACCL | | | ACCL_Magnitude | DISPL | | | DISPL_Magnitude | ids |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | block_1 | 0 | 2.01949e+06 | -113633 | -1.50961e+06 | 2.52392e+06 | -1.24299 | -0.659797 | -3.4529 | 3.72865 | 2 |
| 1 | block_1 | 1 | 1.6503e+06 | 2.55512e+06 | 1.49382e+06 | 3.38875e+06 | -1.27422 | -0.592235 | -3.40131 | 3.68012 | 43 |
| 2 | block_1 | 2 | 7.02749e+06 | 2.65499e+06 | 2.75698e+06 | 8.00222e+06 | -1.15605 | -0.57504 | -3.51786 | 3.74733 | 172 |
| 3 | block_1 | 3 | 5.24692e+06 | -1.69608e+06 | 2.15016e+06 | 5.91862e+06 | -1.12059 | -0.638114 | -3.54913 | 3.77614 | 168 |
| 4 | block_1 | 4 | 6.56972e+06 | 5.79254e+06 | 1.88663e+06 | 8.95958e+06 | -1.39251 | -0.680263 | -3.91929 | 4.21458 | 1 |
| 5 | block_1 | 5 | 6.14915e+06 | 6.83851e+06 | 2.45442e+06 | 9.51847e+06 | -1.38039 | -0.607921 | -3.85369 | 4.13836 | 42 |
| 6 | block_1 | 6 | 7.10046e+06 | 1.06571e+07 | 2.02115e+06 | 1.29644e+07 | -1.30538 | -0.592522 | -4.01278 | 4.26116 | 172 |

Fig. 1.51: `paraview` using `SpreadSheet View` to plot raw data values for the *can.ex2* dataset.

`SpreadSheet View` is used to inspect raw data values in a tabular form. Unlike most other views, this view is primarily intended to be used in the `paraview` user interface and, hence, is not available in `pvpython`.

To use this view, simply create this view and show the dataset produced by any pipeline module in it by using the `Pipeline Browser`. `SpreadSheet View` can only show one dataset at a time. Therefore, showing a new dataset will automatically hide the previously shown dataset.

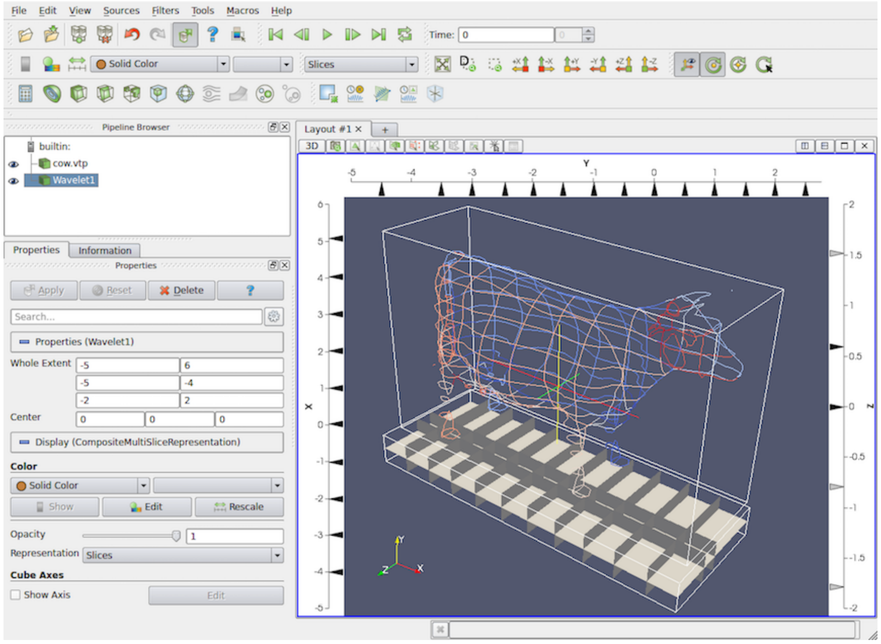
The view's toolbar provides quick access to several of this view's capabilities. Use the `Showing` widget on the view toolbar to view as well as to change the dataset being shown. The `Attribute` field allows you to pick which types of elements to show, e.g., `Cell Data`, `Point Data`, `Field Data`, etc. `Precision` can be utilized to change the precision used when displaying floating point numbers. The ▦ button enables you to select columns to show. Click on the button to get a popup menu in which you check/uncheck the columns to show/hide. If showing `Cell Data`, the button, when checked, enables you to see the point ids that form each of the cells.

Section 1.6.1 discusses how selections can be made in views to select elements of interest. Use the  button to make the view show only selected elements. Now, as you make selections in other views, this `SpreadSheet View` will update to only show the values of the selected elements (as long as the dataset selected in are indeed being shown in the view).

**Did you know?**

Some filters and readers produce `vtkTable`, and they are automatically displayed in a spreadsheet view. Thus, one can very easily read the contents of a .csv file with **ParaView**.

### 1.4.11 Slice View



Properties (Wavelet1)
Whole Extent -5 6 / -5 -4 / -2 2
Center 0 0 0
Display (CompositeMultiSliceRepresentation)
Color: Solid Color
Show / Edit / Rescale
Opacity 1
Representation Slices
Cube Axes
□ Show Axis
Edit

Fig. 1.52: `Slice View` can be used to show orthogonal slices from datasets.

`Slice View` is special type of `Render View` that can be used to view orthogonal slices from any dataset. Any dataset shown in the view will be sliced in axis-aligned slices based on the locations specified on the view. The slice locations along the three orthogonal axis planes can be specified by using the frame decoration around the view window.

#### Interactions

Since this view is a type of `Render View` , the camera interactions are same as that of `Render View` . Additionally, you can interact with the frame decoration to manipulate slice locations along the three axis planes.

- Double-click the left mouse button in the region between the axis border and the view to add a new slice.
- You can click-and-drag a marker to move the slice location.
- To remove a slice, double-click with the left mouse button on the marker corresponding to that slice.
- To toggle visibility of the slice, you can right-click on the marker.

#### Slice View in `pvpython`

```
# To create a slice view in use:
>>> view = CreateView("MultiSlice")

# Use properties on view to set/get the slice offsets.
>>> view.XSliceValues = [-10, 0, 10]
>>> print(view.XSliceValues)
[-10, 0, 10]

# Similar to XSliceValues, you have YSliceValues and ZSliceValues.
```

(continues on next page)

(continued from previous page)

```
>>> view.YSliceValues = [0]
>>> view.ZSliceValues = []
```

### 1.4.12 Python View

Some Python libraries, such as matplotlib, are widely used for making publication-quality plots of data. The `Python View` provides a way to display plots made in a Python script right within `paraview`.

The `Python View` has a single property, a Python script that generates the image to be displayed in the viewport. All the Python bindings for **ParaView** and VTK that are available in the Python scripting module are available from within this script, making it possible to plot any array from just about any dataset that can be loaded into **ParaView**. The Python script for the view is evaluated in a unique Python environment so that global variables defined in the script do not clobber global variables in other Python scripts (either in other instances of the `Python View` or in the Python interpreter). This environment is reset each time the script is evaluated, so data cannot be saved between evaluations.

The `Python View` requires that the Python script where the plotting occurs define two functions. In the first function, you request which arrays you would like to transfer to the client for rendering. At present, all rendering in this view takes place on the client, even in client-server mode. These arrays can be point data, cell data, field data, and table row data. This function runs only on data-server processes. It provides access to the underlying data object on the server so that you can query any aspect of the data using the Python-wrapped parts of VTK and **ParaView**.

The second function is where you put Python plotting or rendering commands. This function runs only on the **ParaView** client. It has access to the complete data object gathered from the data server nodes, but only has access to the arrays requested in the first function. This function will typically set up objects from a plotting library, convert data from VTK to a form that can be passed to the plotting library, plot the data, and convert the plot to an image (a `vtkImageData` object) that can be displayed in the viewport.

#### Selecting data arrays to plot

All the rendering in the `Python View` occurs in the client, so the client must get the data from the server. Because the dataset residing on the **ParaView** server may be large, transferring all the data to the client may not be possible or practical. For that reason, we have provided a mechanism to select which data arrays in a data object on the server to transfer to the client. The overall structure of the data object, however, (including cell connectivity, point positions, and hierarchical block structure) is always transferred to the client. By default, no data arrays are selected for transfer from the server.

The Python script for the view must define a function called `setup_data(view)` . The `view` argument is the VTK object for the `Python View` . The current datasets loaded into **ParaView** may be accessed through the view object.

Here's an example of this function that was used to generate the image in Fig. 1.53:

```
def setup_data(view):
  # Iterate over visible data objects
  for i in range(view.GetNumberOfVisibleDataObjects()):
    # You need to use GetVisibleDataObjectForSetup(i)
    # in setup_data to access the data object.
    dataObject = view.GetVisibleDataObjectForSetup(i)

    # The data object has the same data type and structure
    # as the data object that sits on the server. You can
    # query the size of the data, for instance, or do anything
    # else you can do through the Python wrapping.
    print('Memory size: {0} kilobytes'.format(dataObject.GetActualMemorySize()))
```

(continues on next page)

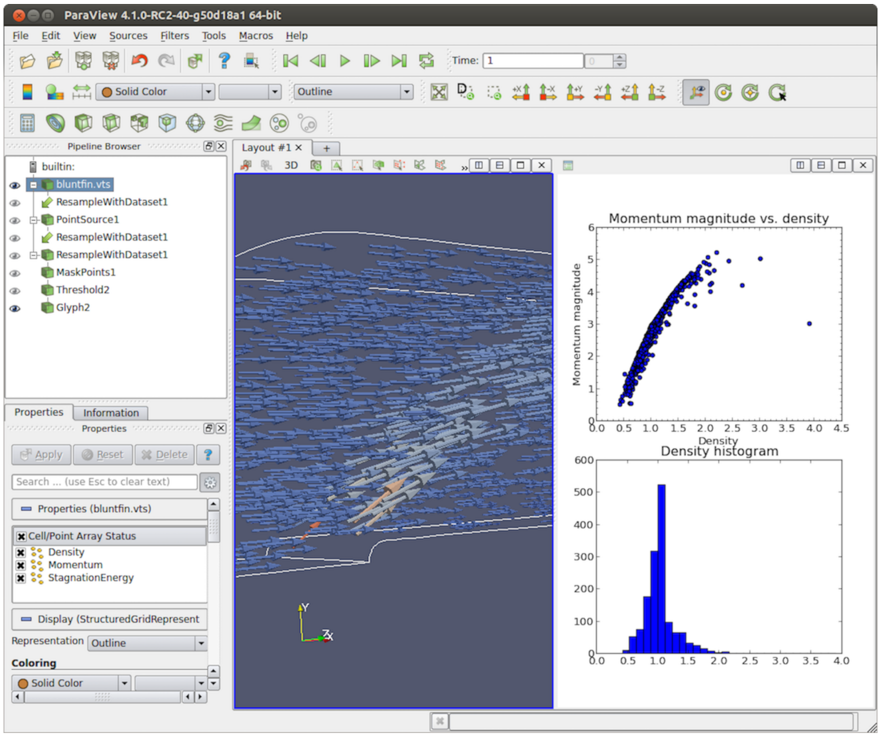


Fig. 1.53: `paraview` using `Python View` with matplotlib to display a scatterplot of momentum magnitude versus density (upper right) and a histogram of density (lower right) in the bluntfin.vts dataset.

(continued from previous page)

```
# Clean up from previous calls here. We want to unset
# any of the arrays requested in previous calls to this function.
view.DisableAllAttributeArrays()

# By default, no arrays will be passed to the client.
# You need to explicitly request the arrays you want.
# Here, we'll request the Density point data array
view.SetAttributeArrayStatus(i, vtkDataObject.POINT, "Density", 1)
view.SetAttributeArrayStatus(i, vtkDataObject.POINT, "Momentum", 1)

# Other attribute arrays can be set similarly
view.SetAttributeArrayStatus(i, vtkDataObject.FIELD, "fieldData", 1)
```

The `vtkPythonView` class passed in as the `view` argument to `setup_data(view)` defines several methods useful for specifying which data arrays to copy:

- `GetNumberOfVisibleDataObjects()` - This returns the number of visible data objects in the view. If an object is not visible, it should not show up in the rendering, so all the methods provided by the view deal only with visible objects.
- `GetVisibleDataObjectForSetup(visibleObjectIndex)` - This returns the `visibleObjectIndex'th` visible data object in the view. (The data object will have an open eye next to it in the `Pipeline Browser` .)
- `GetNumberOfAttributeArrays(visibleObjectIndex, attributeType)` - This returns the number of attribute arrays for the `visibleObjectIndex'th` visible object and the given `attributeType` (e.g.,

vtkDataObject.POINT , vtkDataObject.CELL , etc.).

- GetAttributeArrayName(visibleObjectIndex, attributeType, arrayIndex) - This returns the name of the array of the given attribute type at the given array index for the visibleObjectIndex'th object.
- SetAttributeArrayStatus(visibleObjectIndex, vtkDataObject.POINT, "Density", 1) - This sets the array status of an attribute array. The first argument is the visible object index, the second object is the attribute association of the array, the third argument is the name of the array, and the last argument specifies if the array is to be copied (1) or not (0).
- GetAttributeArrayStatus(visibleObjectIndex, vtkDataObject.POINT, "Density") - This retrieves the array status for the object with the given visible index with a given attribute association (second argument) and a name (last argument).
- EnableAllAttributeArrays() - This sets all arrays to be copied.
- DisableAllAttributeArrays() - This sets all arrays to not be copied.

The methods GetNumberOfVisibleDataObjects() , GetVisibleDataObjectForSetup(...) , GetNumberOfAttributeArrays(...) , and GetAttributeArrayName(...) are all convenient methods for obtaining information about visible data objects in the view that could otherwise be accessed with existing view and representation methods. The last four methods are valid only from within the setup_data(view) function.

### Plotting data in Python

After the setup_data(view) function has been called, **ParaView** will transfer the data object and selected arrays to the client. When that is done, it will call the render(view, width, height) function you have defined in your script.

The view argument to the render(view, width, height) function is the vtkPythonView object on the client. The width and height arguments are the width and height of the viewport, respectively. The render(view, width, height) function uses the data available through the view, along with the width and height, to generate a vtkImageData object that will be displayed in the viewport. This vtkImageData object must be returned from the render(view, width, height) function. If no vtkImageData is returned, the viewport will be black. If the size of the image does not match the size of the viewport, the image will be stretched to fit the viewport.

Putting it all together, here is a simple example that generates a solid red image to display in the viewport.

```python
def render(view, width, height):
  from paraview.vtk import vtkImageData
  image = vtkImageData()
  image.SetDimensions(width, height, 1)
  from paraview.numeric import VTK_UNSIGNED_CHAR
  image.AllocateScalars(VTK_UNSIGNED_CHAR, 4)
  pixel_array = image.GetPointData().GetArray(0)
  pixel_array.FillComponent(0, 255.0)
  pixel_array.FillComponent(1, 0.0)
  pixel_array.FillComponent(2, 0.0)
  pixel_array.FillComponent(3, 0.0)

  return image
```

This example does not produce an interesting visualization, but serves as a minimal example of how the render(view, width, height) function should be implemented. Typically, we expect that the Python plotting library you use has some utilities to expose the generated plot image pixel data. You need to copy that pixel data to the vtkImageData object returned by the render(view, width, height) function. Exactly how you do this is up to you, but **ParaView** comes with some utilities to make this task easier for matplotlib.

### Set up a matplotlib Figure

The Python View comes with a Python module, called python_view , that has some utility functions you can use. To import it, use:

```
from paraview import python_view
```

This module has a function, called matplotlib_figure(view, width, height) , that returns a matplotlib.figure.Figure given width and height arguments. This figure can be used with matplotlib plotting commands to plot data as in the following:

```
def render(view, width, height):
  figure = python_view.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
  dataObject = view.GetVisibleDataObjectForRendering(0)

  x = dataObject.GetPointData().GetArray('X')

  # Convert VTK data array to numpy array for plotting
  from paraview.numpy_support import vtk_to_numpy
  np_x = vtk_to_numpy(x)

  ax.hist(np_x, bins=10)

  return python_view.figure_to_image(figure)
```

This definition of the render(view, width, height) function creates a histogram of a point data array named X from the first visible object in the Pipeline Browser . Note the conversion function, python_view.figure_to_image(figure) , in the last line. This converts the matplotlib Figure object created with python_view.matplotlib_figure(width, height) into a vtkImageData object suitable for display in the viewport.

### 1.4.13 Comparative Views

`Comparative Views` , including `Render View (Comparative)` , `Line Chart View (Comparative)` , and `Bar Chart View (Comparative)` , are used for generating comparative visualization from parameter studies. These views are covered in Section 2.4 of the Reference Manual.

### 1.4.14 Virtual reality

**ParaView** supports immersive data visualization in virtual reality (VR) with head-mounted displays (HMDs). VR can be used to view the scene in the active *Render View* and interact with it. This environment provides enhanced perception of depth and more intuitive manipulation of 3D objects by creating the sensation of the user being physically present inside of the scene.

#### XRInterface plugin

VR support in **ParaView** is provided through the `XRInterface` plugin (previously called `OpenVR` plugin). To load this plugin, open the `Plugin Manager` via *Tools* > *Manage Plugins…*. Click on `XRInterface`, then on the button `Load Selected`. This will open the *XRInterface panel*.

#### How to install/build

By default, the `XRInterface` plugin is already enabled in **ParaView** Windows binaries. Simply launch **ParaView** with a VR runtime such as SteamVR on your machine to easily experience VR with **ParaView**.

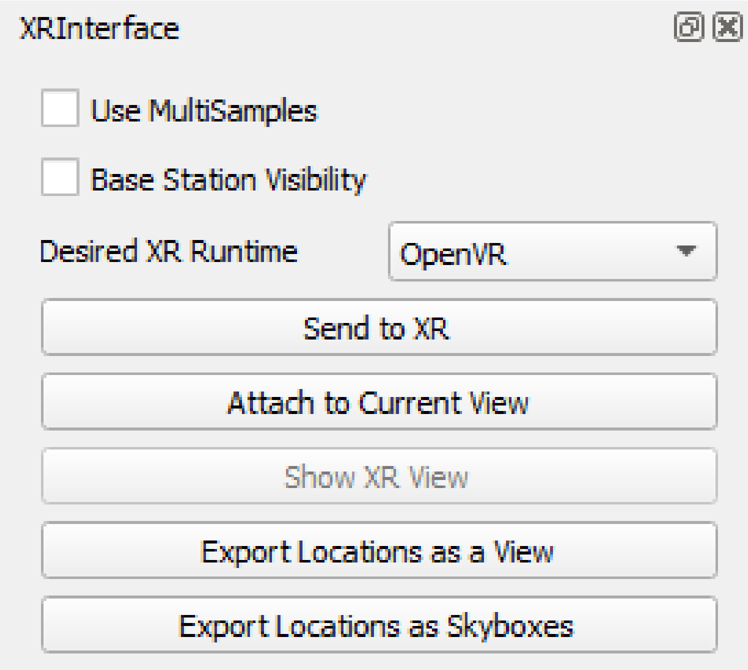
Otherwise, the plugin needs to be enabled before building **ParaView**, see README.md in the plugin sources for more information.

#### XRInterface panel

By default, the `XRInterface` panel appears on the right upon loading the plugin. To open it manually, search for the corresponding checkbox in the main menu via *View* > *XRInterface*.

Once the `XRInterface` plugin is loaded and an HMD connected, click on the button `Send to XR` to start rendering in VR. Other options are available in the plugin panel:

- `Use MultiSamples` — This checkbox indicates whether multisampled framebuffers are used. Using multisamples can reduce flickering but currently does not work with volume rendering.
- `Base Station Visibility` — This checkbox indicates whether the base stations models are displayed in VR (for devices using base stations).
- `Desired XR Runtime` — This drop down list indicates whether to use OpenVR or OpenXR when both are available.
- `Send to XR` - This button launches the VR mode using the selected runtime.
- `Attach to Current View` — This button allows VR-only elements to appear in the current Render View, such as cropping planes or avatars. This option is mainly useful in Collaboration mode for desktop users when an HMD is not available.
- `Show XR View` - This button opens a window showing a stabilized view of what the VR headset is showing, which is useful to produce screenshots or videos.



XRInterface

□ Use MultiSamples

□ Base Station Visibility

Desired XR Runtime: OpenVR

Fig. 1.54: XRInterface panel.

- `Export Locations as a View` — This button exports saved locations as `.vtp` files in a folder called `pv-view`. This option is meant to be used for Mineview.
- `Export Locations as Skyboxes` — This button exports saved locations as skyboxes in a folder called `pv-skybox`. This generates six *.jpg* images (*right*, *left*, *up*, *down*, *back* and *front*) producing a skybox that can then be used outside of VR as well.

### Interactions

On top of head tracking to reproduce the user physical movements for navigation, the controllers can be used to interact with the data through actions such as scaling, picking, etc.

The controls mapping for different controllers is detailed in Section 1.4.14.

| Action | Controls |
|---|---|
| Movement | (Left/Right) Joystick |
| Open XR Menu | Right Menu Button |
| Confirm (XR Menu) | Right Trigger |
| Trigger Action | Right Trigger |
| Move to Next Saved Location | Left Trigger |
| Grip Action | Left Grip + Right Grip |

### Trigger actions

Trigger actions are assigned to the right trigger by default and include grabbing, picking, probing, interactive clipping, teleportation, and adding points to sources (such as a polyline source). The current action can be chosen via the XR menu (see Section 1.4.14).

- **Adding points to a source** — Press the right trigger to place a point at the tip of the right controller. Only valid when the active source allows placing points, such as a polyline source.

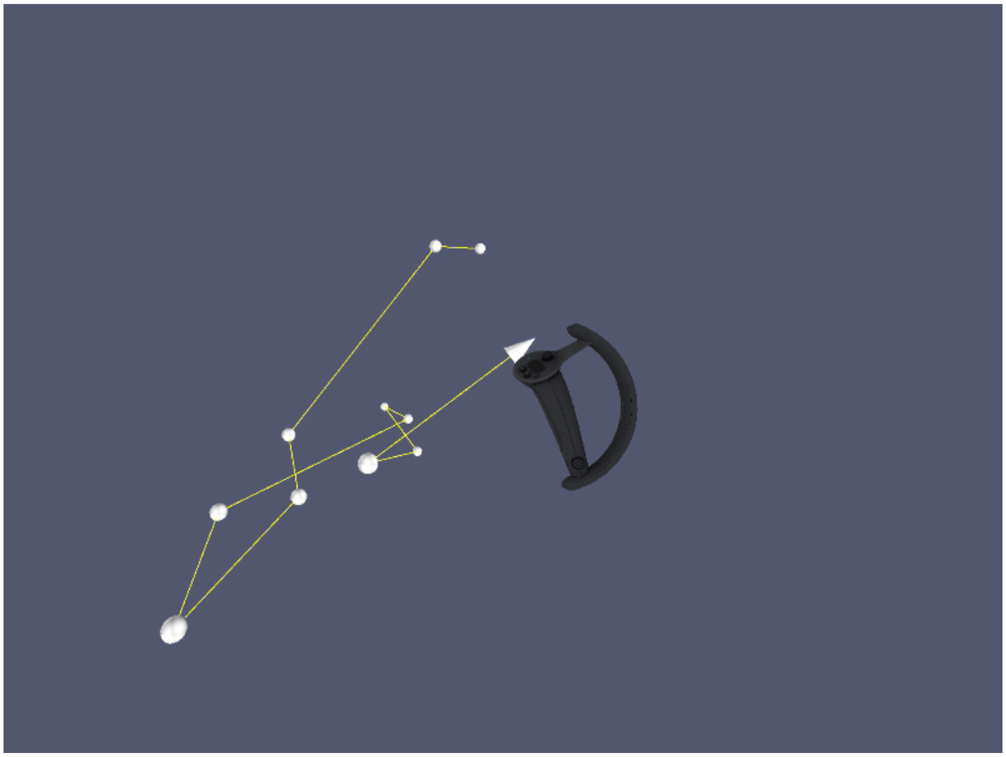


Fig. 1.55: Adding a point to a polyline source in VR.

- **Grabbing** — Press the right trigger when the right controller is in the bounding box of an object to grab it and move it.
- **Picking** — Press the right trigger while pointing at an object to pick it and move it from a distance.
- **Interactive Cropping** — Press the right trigger to crop the scene in real time with a plane placed at the tip of the right controller.
- **Probing** — Press the right trigger while pointing at a dataset cell to select it and display information on its position and data values.
- **Teleportation** — Press the right trigger while pointing at a dataset with the ray to instantly move at the pointed location.

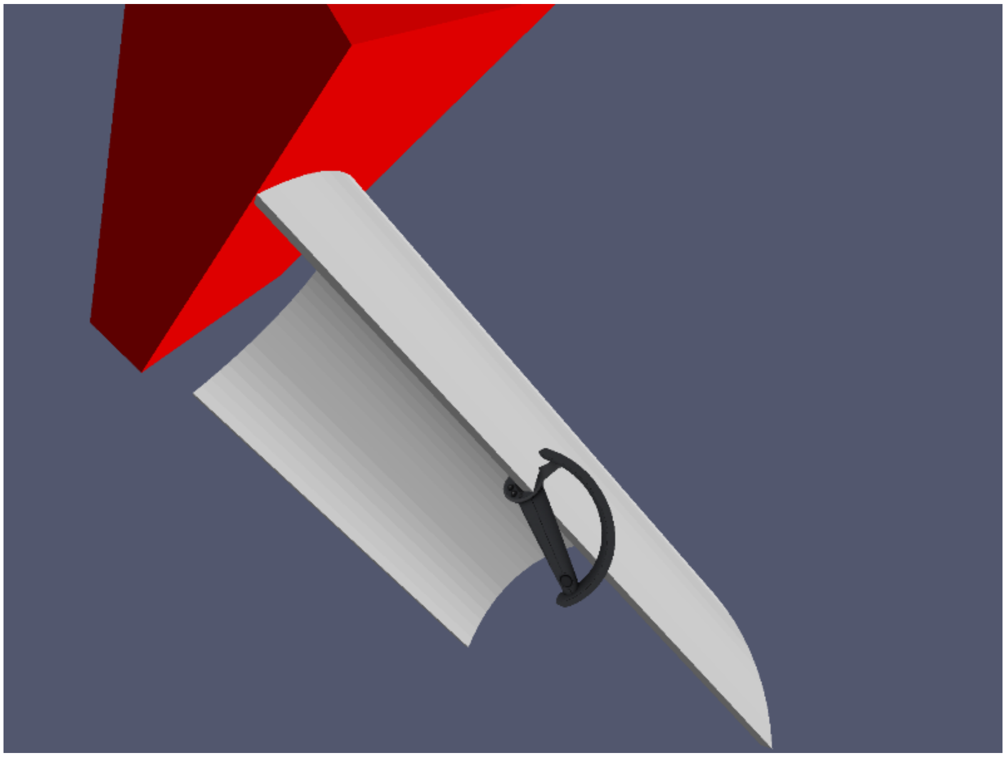


Fig. 1.56: Grab action in VR.

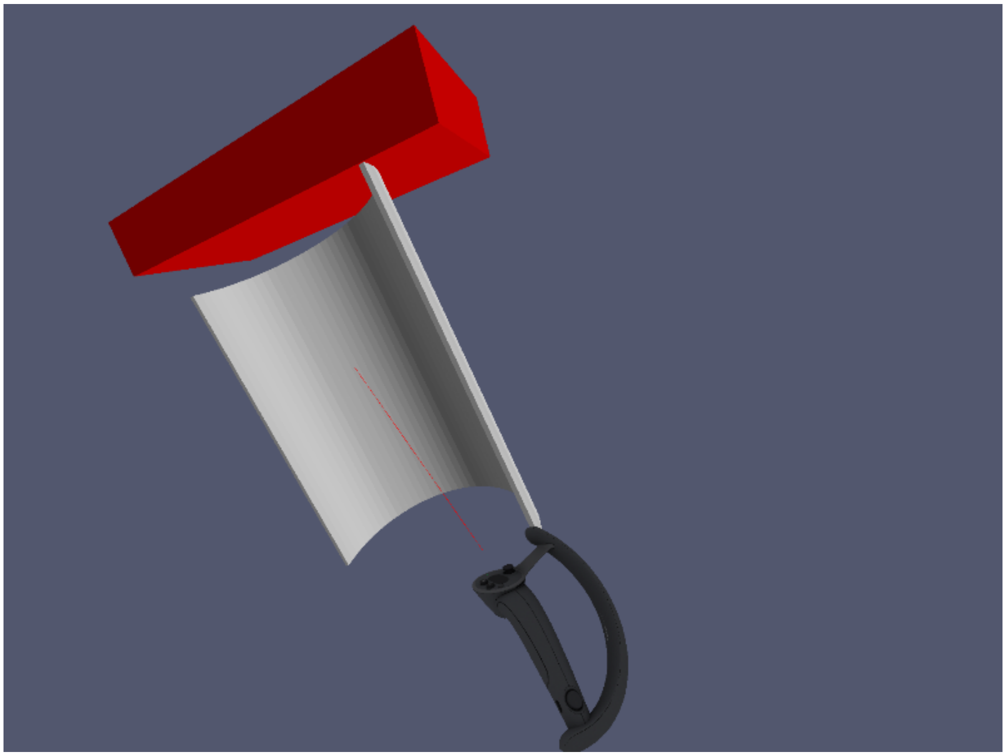


Fig. 1.57: Picking action in VR.

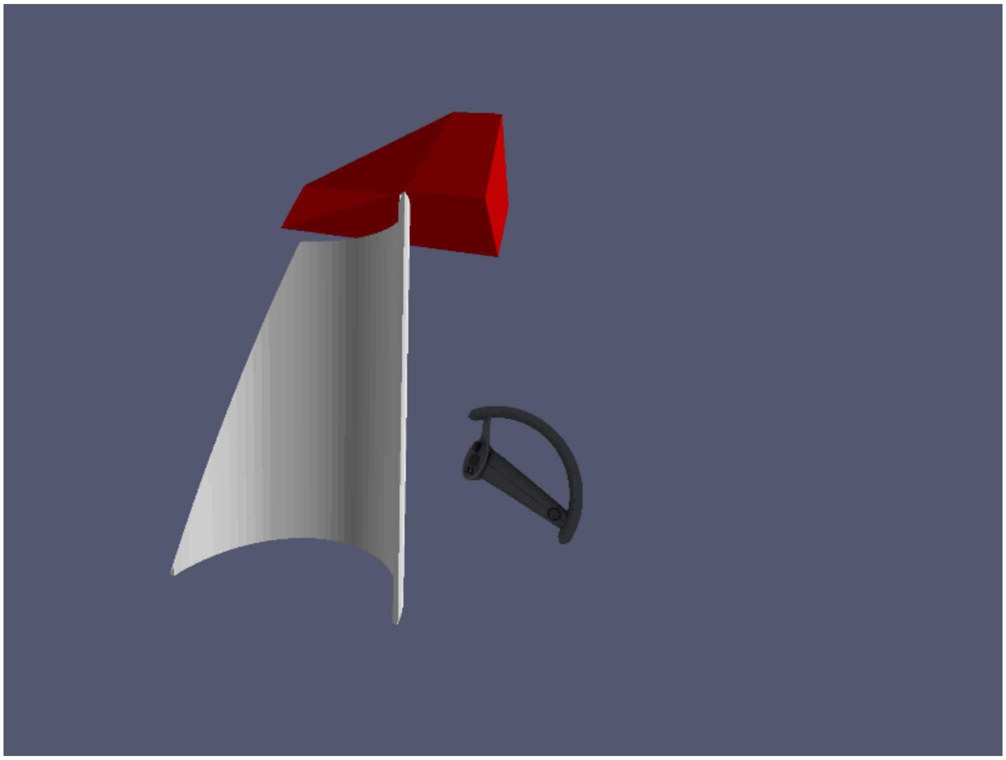
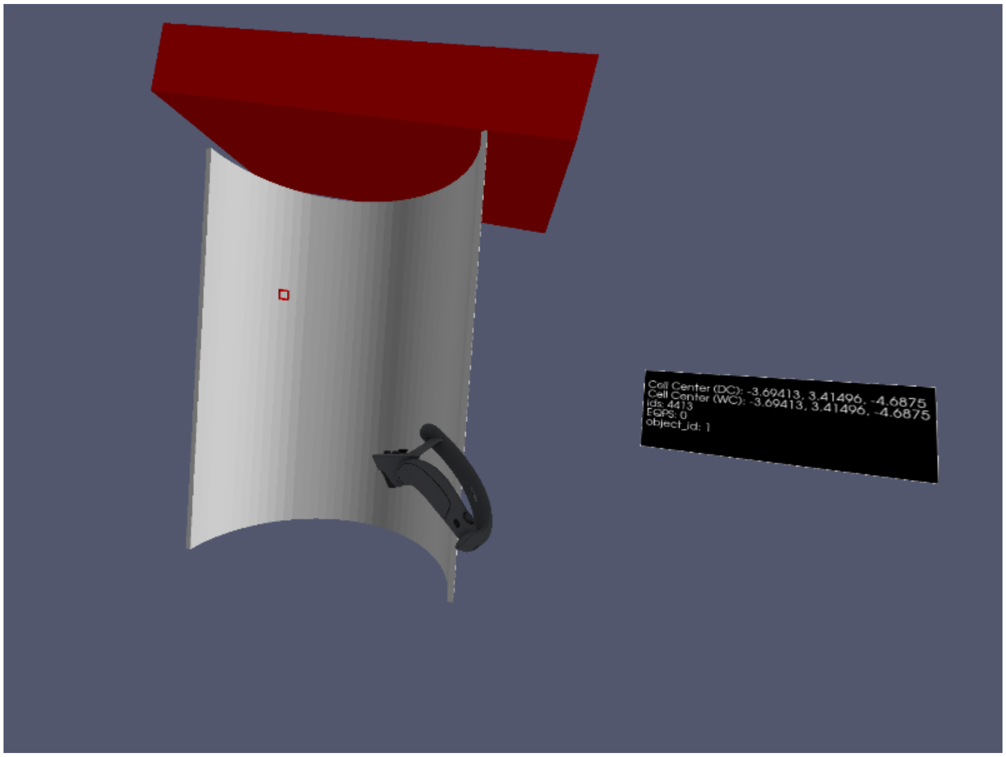


Fig. 1.58: Real time cropping in VR.



```
Cell Center (DC): -3.69413, 3.41496, -4.6875
Cell Center (WC): -3.69413, 3.41496, -4.6875
ids: 4413
EQPS: 0
object_id: 1
```

Fig. 1.59: Cell probing action in VR.

### Grip actions

Grip actions are used to transform the scene through translation, scaling and rotation.

- **Translation** — Press both grip buttons, then move the controllers together to translate the scene.

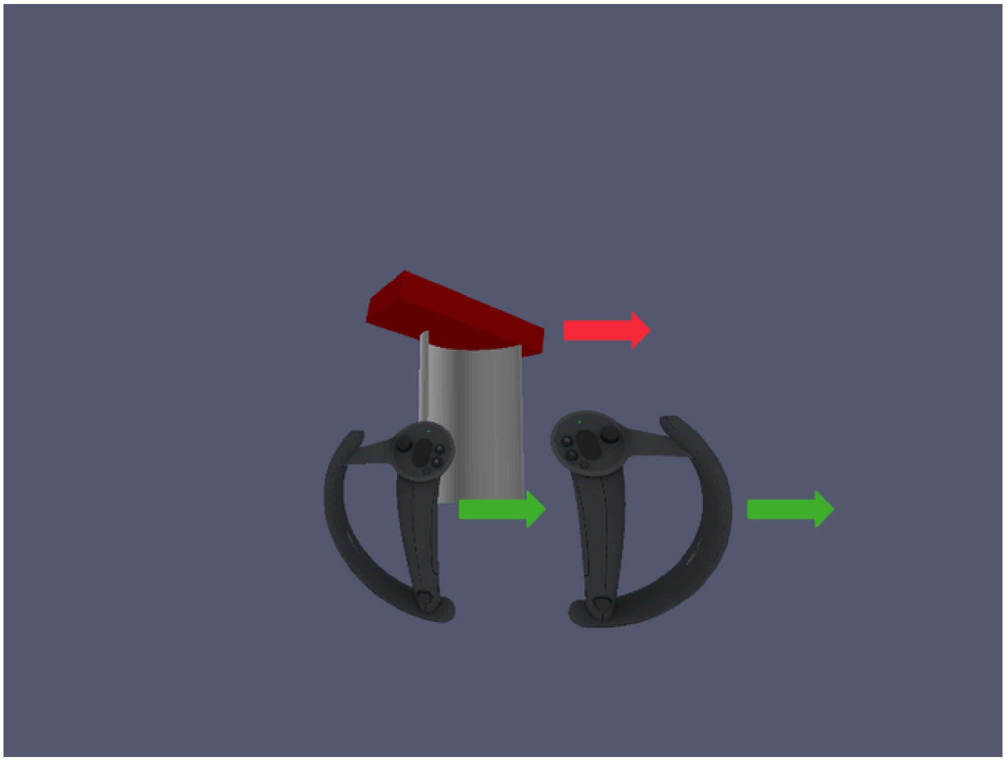


Fig. 1.60: Translation action in VR. Green and red arrows correspond to movements of the controllers and dataset, respectively.

- **Scaling/zoom** — Press both grip buttons, then move the controllers farther or closer to each other to zoom in or out of the scene.
- **Rotation** — Press both grip buttons, then move one controller around the other to rotate the scene around a vertical axis.

### XR menu

In the XR View, pressing the menu button opens a dedicated menu floating in the 3D scene in front of the user.

This menu is composed of the following elements:

1. **Pipeline Browser** — This is the same `Pipeline Browser` present in **ParaView**. The visibility for each item in the pipeline can be modified by pointing the navigation ray on the eye icon and pressing the right trigger.
2. **Panels** — VR options are distributed into 4 panels, that can be displayed by clicking on the corresponding tab:
   - `Interaction` — This panel contains options related to the interactions with the scene using the controllers (see Section 1.4.14).
   - `Movement` — This panel contains options related to the camera movement and poses (see Section 1.4.14).

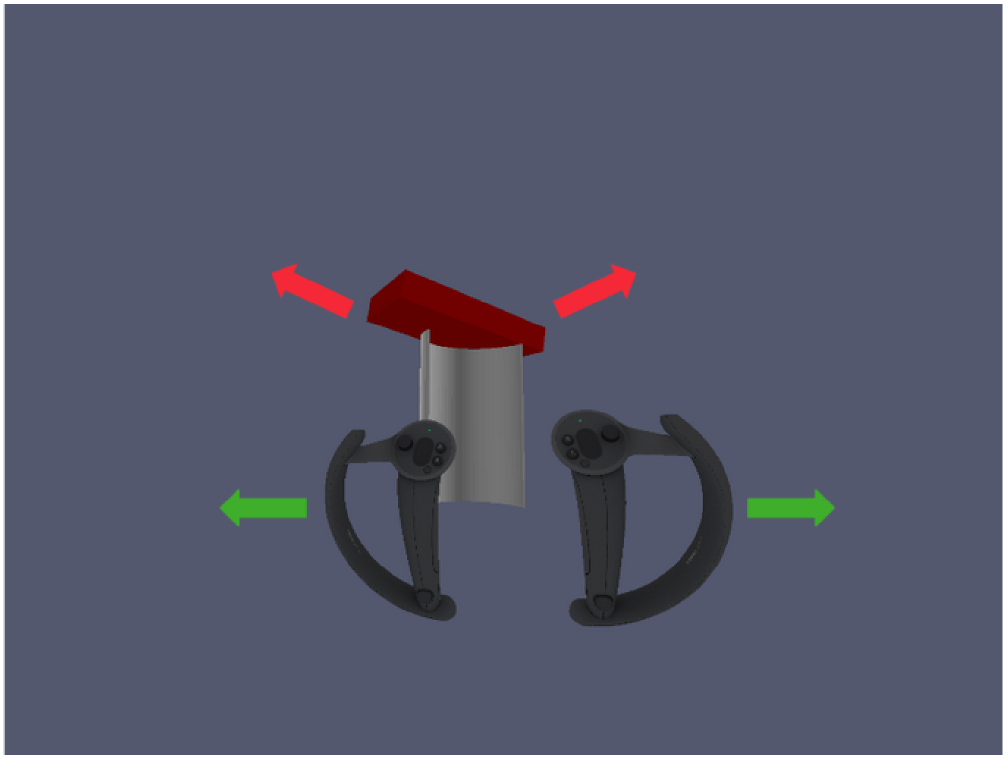


Fig. 1.61: Scaling action in VR. Green and red arrows correspond to movements of the controllers and dataset, respectively.

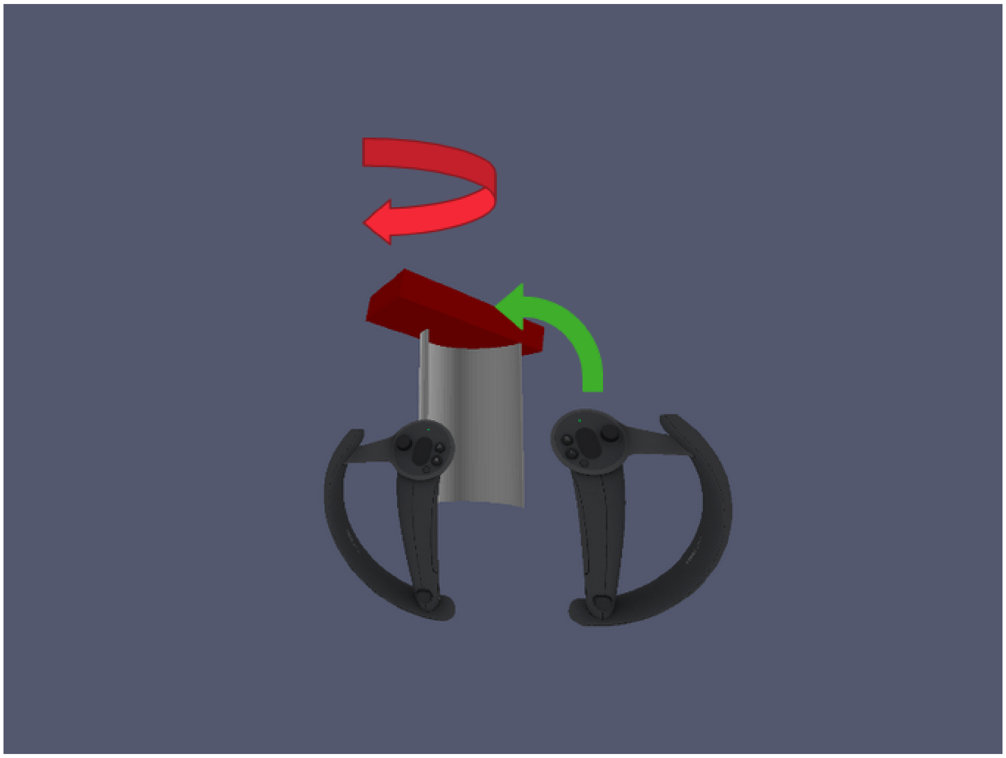


Fig. 1.62: Rotation action in VR. Green and red arrows correspond to movements of the controllers and dataset, respectively.

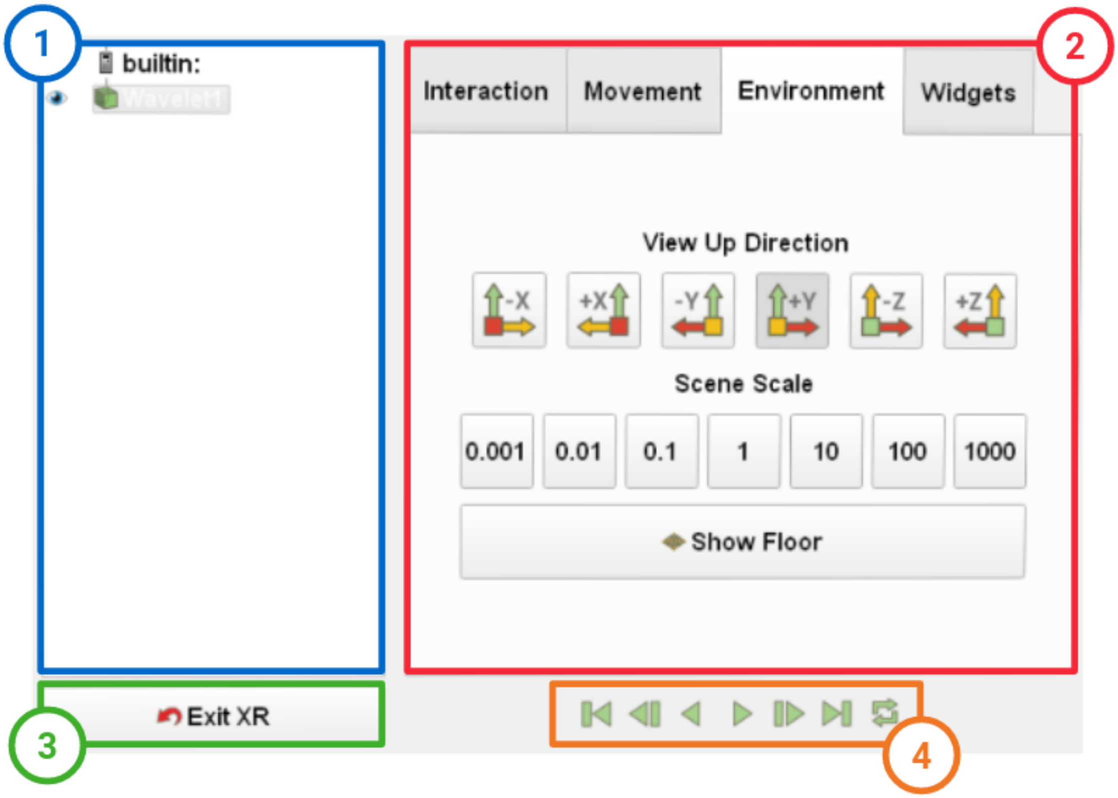


Fig. 1.63: XR integrated menu.

- `Environment` — This panel contains global options related to the scene (see Section 1.4.14).
- `Widgets` — This panel contains options related to VR-specific widgets (see Section 1.4.14).

3. **Exit XR** — This button closes the current XR View.
4. **Animation Buttons** — These buttons are used to navigate timesteps for temporal datasets.

### Interaction panel

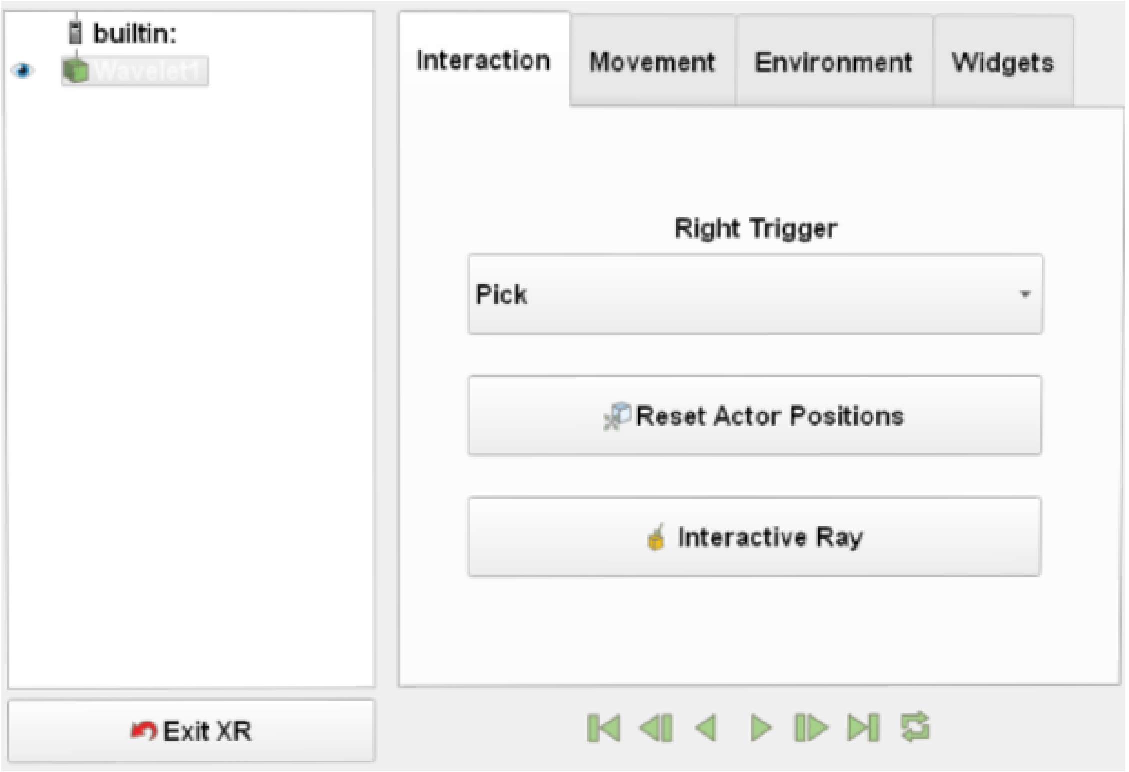
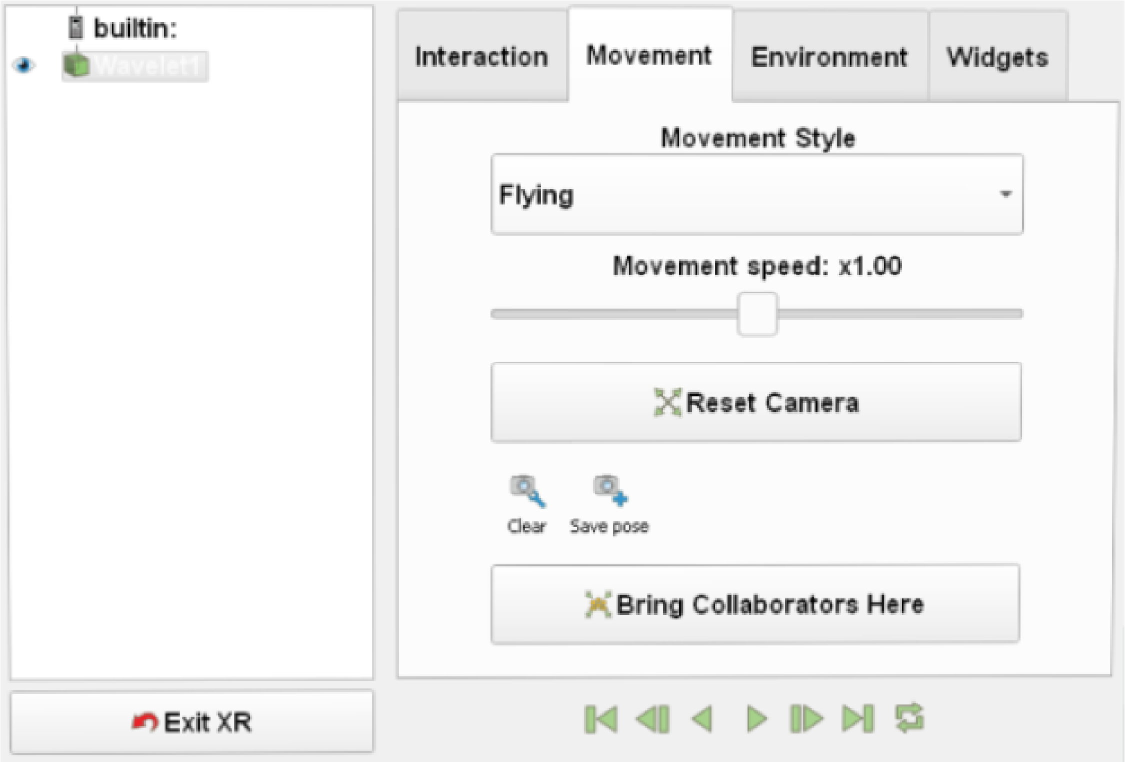


Fig. 1.64: Interaction panel of the XR integrated menu.

- `Right Trigger` — This drop down list indicates which action is mapped to the right trigger outside of the menu, such as picking, probing, etc. See Section 1.4.14 for the list of available actions.
- `Reset Actors Positions` — This button resets the position of all objects within the scene.
- `Interactive Ray` — This checkbox indicates whether the ray changes color when pointing at an object to signal a collision.

### Movement Panel



Movement Style
Flying

Movement speed: x1.00

Fig. 1.65: Movement panel of the XR integrated menu.

- `Movement Style` — This drop down list indicates which movement style is mapped to the controller joysticks.
  - **Flying** mode uses only one joystick and movement follows the orientation of the controller itself.
  - **Grounded** mode uses the left joystick to move horizontally in all directions, while the right joystick controls elevation.
- `Movement Speed` — This horizontal slider modifies the movement speed when using the joysticks. A higher multiplier corresponds to a higher speed.
- `Reset Camera` — This button scales and places the objects at the center of the physical room.
- **Camera pose buttons**:
  - `Clear` — This button clears all previously saved camera poses.
  - `Save pose` — This button saves the current pose in the list of saved camera poses. Up to 6 poses can be saved this way. For each saved pose, a dedicated button is added to the right of this button.
- `Bring Collaborators Here` — This button moves all collaborators to your current location. Only applicable in Collaboration mode.

### Environment panel

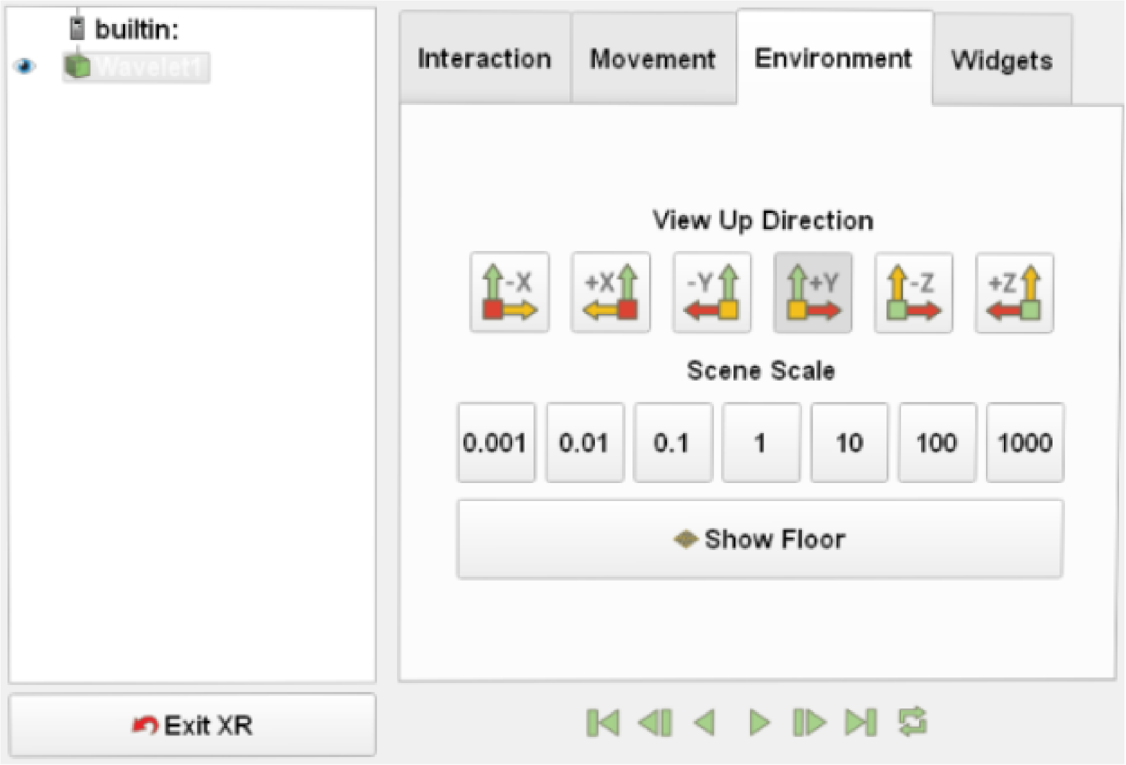


Fig. 1.66: Environment panel of the XR integrated menu.

- `View Up Direction` — These buttons set which axis points upwards from the top of the HMD. This is useful when datasets or skyboxes are oriented differently from the default.
- `Scene Scale` — These buttons change the scaling factor of the scene. A higher value results in all objects appearing larger.
- `Show Floor` — This button allows hiding or showing the floor as a white plane.

### Widgets panel

- `Distance Widget` — This button adds a measuring tool to the scene. Press the right trigger once to place the starting point where the right controller is located, then press a second time with the controller at the desired location to place the second point. Four values are displayed next to the tool: distance and X, Y, Z difference between both points. The tips of the line can be grabbed and moved individually after placing them.
- `Navigation Panel` — This button allows to display a tooltip above the left controller with information on its spatial position in world coordinates.
- **Cropping buttons** — The following buttons provide tools to crop data in real time. Cropping planes can be moved by placing the right controller on them and grabbing them with the right trigger. More than one plane can be added to the scene.
  - `Add Crop Plane` — This button adds a cropping plane to the scene.

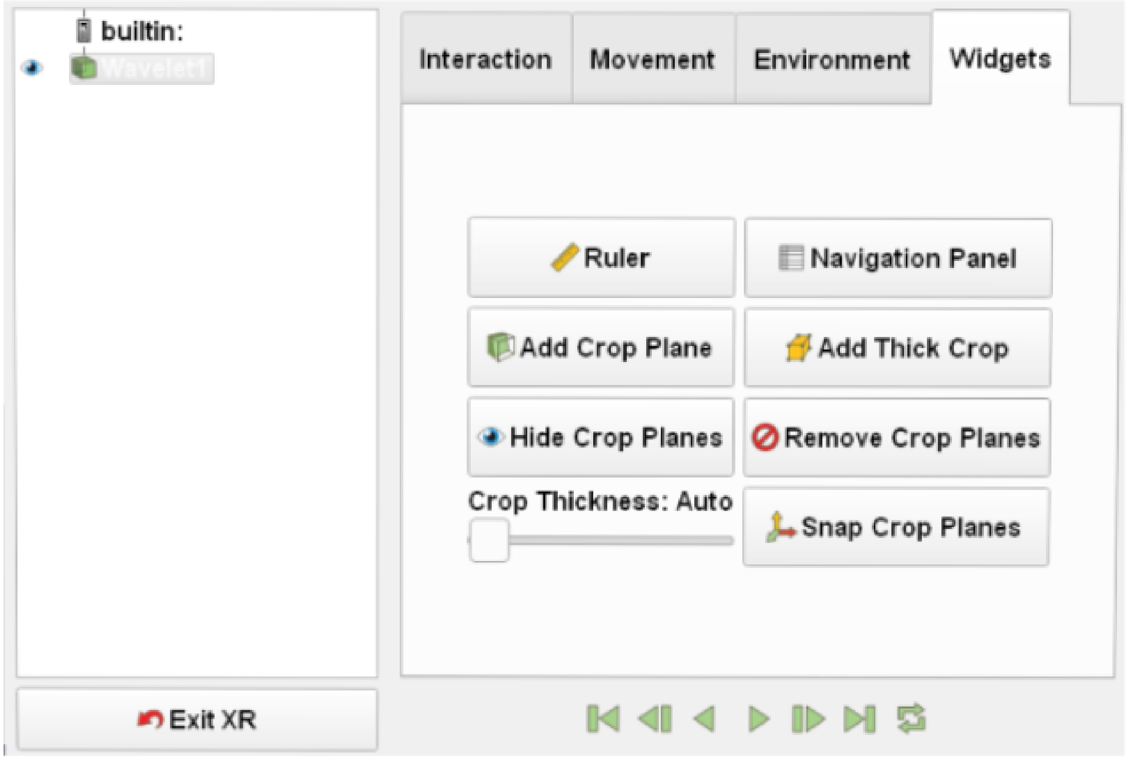


Fig. 1.67: Widgets panel of the XR integrated menu.

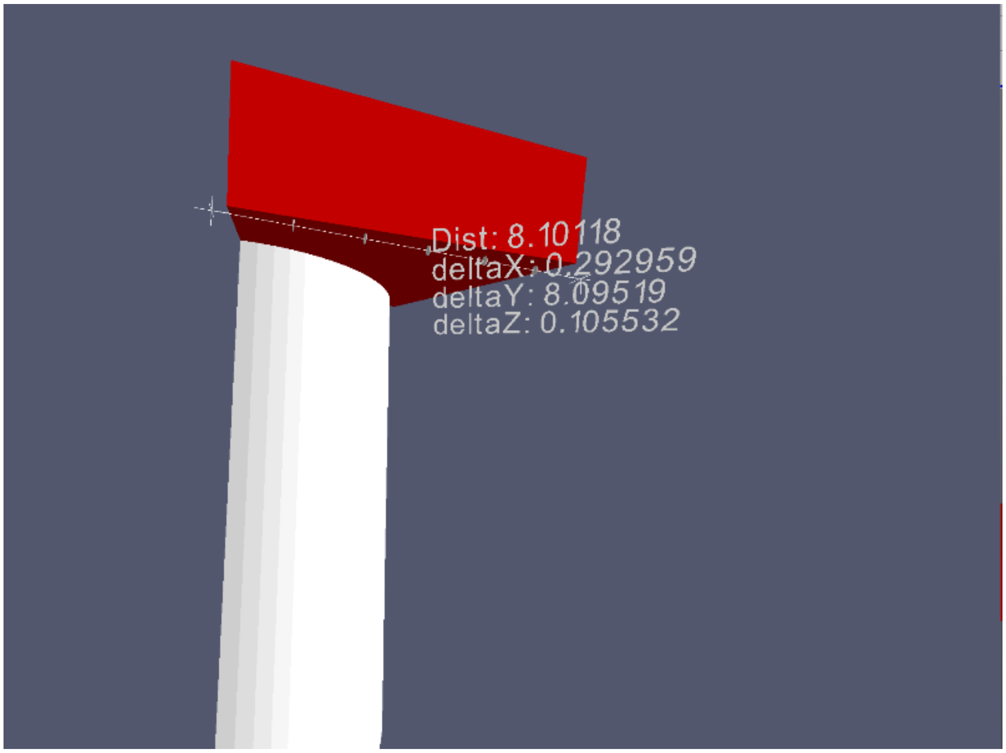


Fig. 1.68: Distance measurement widget in VR.

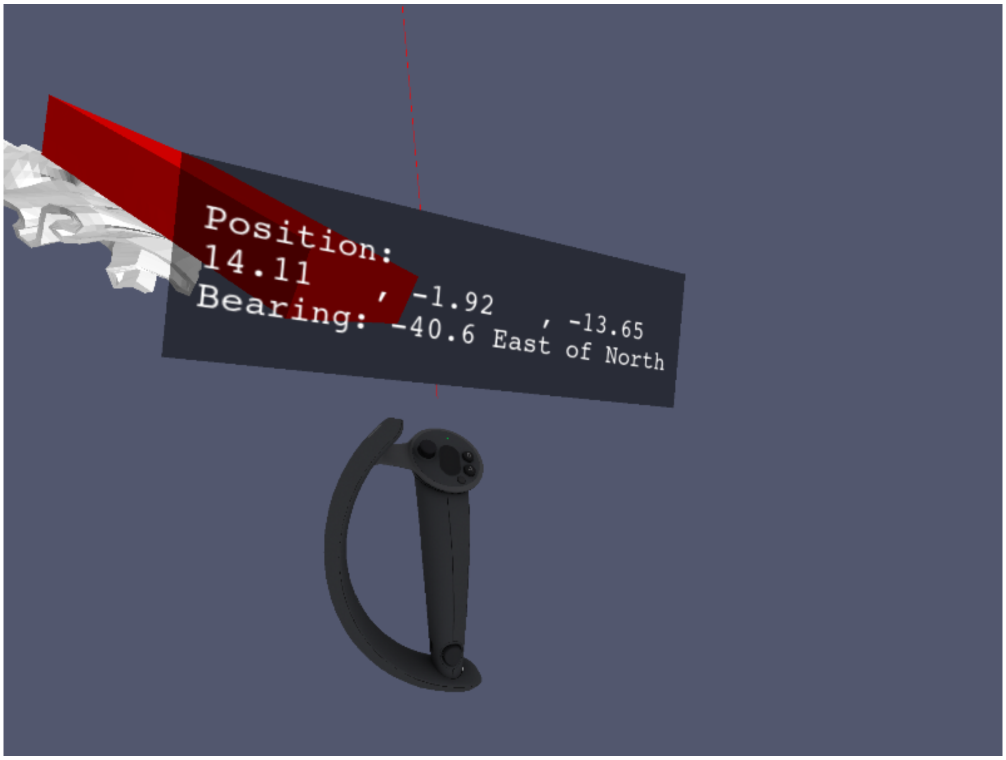


Fig. 1.69: Navigation panel widget in VR.

- `Add Thick Crop` — This button adds a thick cropping plane to the scene.
- `Hide Crop Planes` — This button hides all cropping planes in the scene.
- `Remove All Crop Planes` — This button removes all cropping planes from the scene.
- `Crop Thickness` — This horizontal slider sets the thickness of created thick cropping planes (this parameter does not affect current ones). By default, the value is set to **auto**, which adjusts the plane thickness according to the current scene scale.
- `Snap Crop Planes` — This button allows to choose whether the cropping planes should snap to the coordinates axes.

### Remoting

**Notice**

Remoting support is currently experimental.

Remoting allows streaming the active render view of ParaView to a remote device.

### Prerequisites

The remoting feature is only available on Windows for the Hololens 2 and requires an additional package named Microsoft.Holographic.Remoting.OpenXr. With this, ParaView can connect to another application in the remote device if both applications use the same version of this package.

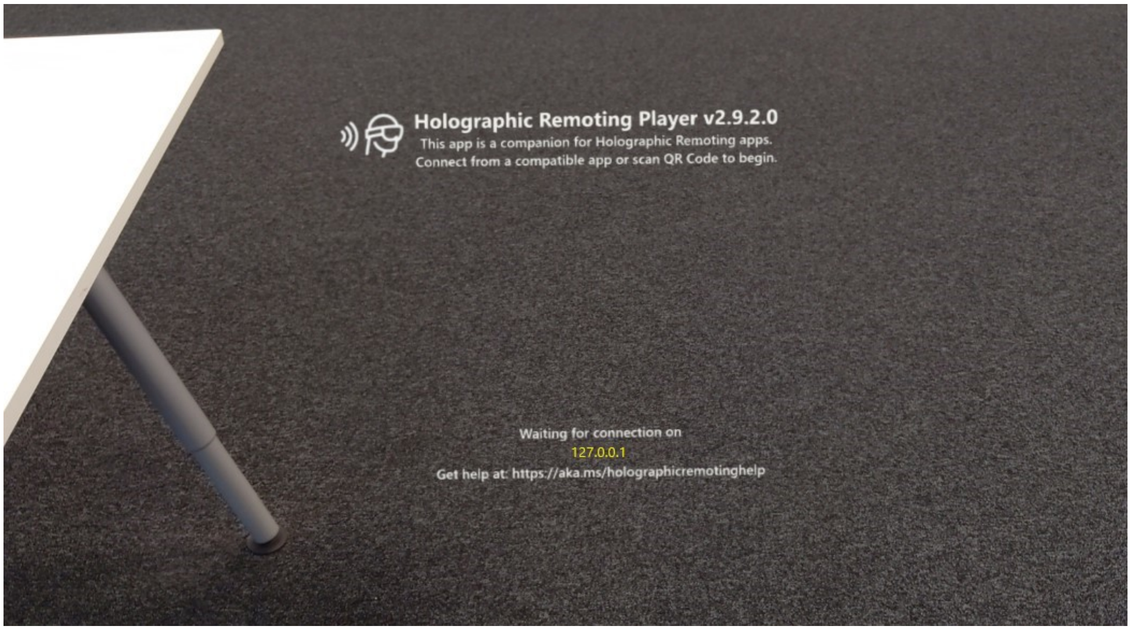
Note that the ParaView release uses the same version as the official player application developed by Microsoft, available in the Microsoft Store, which is version **2.9.2**.

If you do have not an application already deployed in the remote device, we recommend downloading the *Holographic Remoting Player* application in the Microsoft Store.

### How to launch

First, start the application on the remote device.

After launching this application, it will wait for another application to connect to it via an IP address.



Holographic Remoting Player v2.9.2.0
This app is a companion for Holographic Remoting apps.
Connect from a compatible app or scan QR Code to begin.

Waiting for connection on
127.0.0.1
Get help at: https://aka.ms/holographicremotinghelp

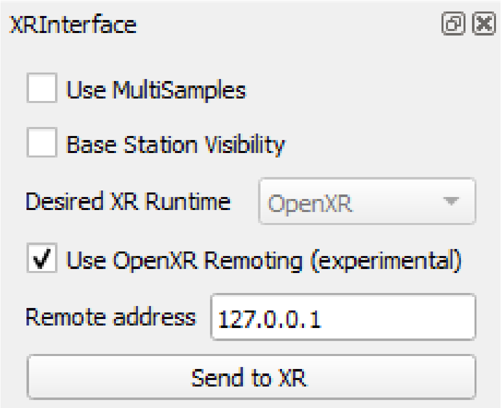
Fig. 1.70: Remote application awaiting connection in the Hololens 2.

You can now start ParaView and do any process on your data that you want. When you are ready to test it in the Hololens 2, enable the XR Interface plugin. You will need to set different options:

- `Desired XR Runtime` — set it to *OpenXR* because the *Microsoft.Holographic.Remoting.OpenXr* depends on it.
- `Use OpenXR Remoting` — enable or disable the remoting support.
- `Remote address` — set the IP address to connect ParaView and the application in the Hololens 2.

After setting these options, you can click on `Send To XR`. Once the connection is established, you will be able to see and interact with your dataset.

For now, only grabbing feature is available as interaction.



XRInterface

□ Use MultiSamples

□ Base Station Visibility

Desired XR Runtime: OpenXR

☑ Use OpenXR Remoting (experimental)

Remote address: 127.0.0.1

Fig. 1.71: XRInterface panel with OpenXR Remoting options.

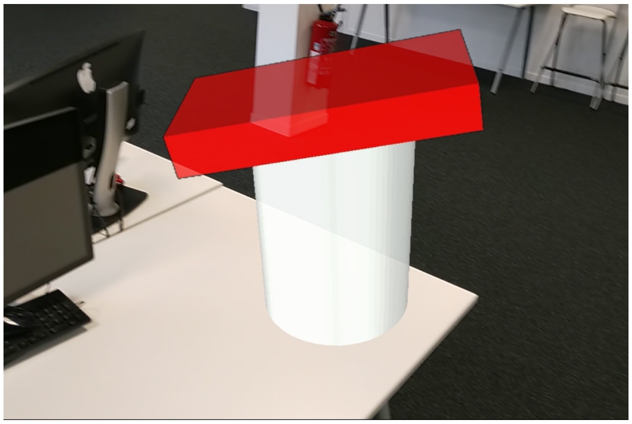


Fig. 1.72: can.ex2 in the Hololens 2.

### Controls mapping

This section details the button mappings used in **ParaView** for different types of controllers. The controls names are those used throughout the VR section.

#### HP Motion

| XR View Controls | Controller Button |
|---|---|
| Joystick | Joystick |
| Trigger | Trigger |
| Grip | Grip Button |
| Menu Button | Menu Button |

#### HTC Vive

| XR View Controls | Controller Button |
|---|---|
| Joystick | Trackpad |
| Trigger | Trigger |
| Grip | Grip Button |
| Menu Button | Menu Button |

#### Oculus Touch

| XR View Controls | Controller Button |
|---|---|
| Joystick | Joystick |
| Trigger | Trigger |
| Grip | Grip Button |
| Menu Button | Menu Button |

#### Valve Index

| XR View Controls | Controller Button |
|---|---|
| Joystick | Joystick |
| Trigger | Trigger |
| Grip | A Button |
| Menu Button | B Button |

### Microsoft Hand

| XR View Controls | Controller Gesture |
|---|---|
| Trigger | Hand squeeze |
| Grip | Hand grip |

## 1.5 Filtering Data

Visualization can be characterized as a process of transforming raw data produced from experiments or simulations until it takes a form in which it can be interpreted and analysed. The visualization pipeline introduced in Section 1.1.2 formalizes this concept as a data flow paradigm where a pipeline is set up of sources, filters, and sinks (collectively called pipeline modules or algorithms). Data *flows* through this pipeline, being transformed at each node until it is in a form where it can be consumed by the sinks. In previous chapters, we saw how to ingest data into **ParaView** (Section 1.2) and how to display it in views (Section 1.4). If the data ingested into **ParaView** already has all the relevant attribute data, and it is in the form that can be directly represented in one the existing views, then that is all you would need. The true power of the visualization process, however, comes from leveraging the various visualization techniques such as slicing, contouring, clipping, etc., which are available as filters. In this chapter, we look at constructing pipelines to transform data using such filters.

### 1.5.1 Understanding filters

In **ParaView**, filters are pipeline modules or algorithms that have inputs and outputs. They take in data on their inputs and produce transformed data or results on their outputs. A filter can have multiple input and output ports. The number of input and output ports on a filter is fixed. Each input port accepts input data for a specific purpose or role within the filter. (E.g., the `Resample With Dataset` filter has two input ports. The one called `Input` is the input port through which the dataset providing the attributes to interpolate is ingested. The other, called `Source` , is the input port through which the dataset used as the mesh on which to re-sample is accepted.)

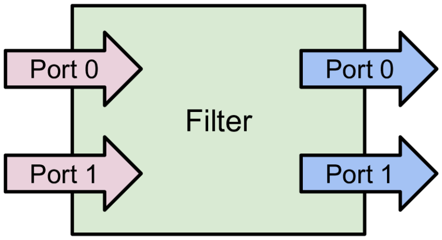


Fig. 1.73: A filter is a pipeline module with inputs and outputs. Data enters a filter through the inputs. The filter transforms the data and produces the resulting data on its outputs. A filter can have one or more input and output ports. Each input port can optionally accept multiple input connections.

An input port itself can optionally accept multiple input connections, e.g., the `Append Datasets` filter, which appends multiple datasets to create a single dataset only has one input port (named `Input` ). However, that port can accept multiple connections for each of the datasets to be appended . Filters define whether a particular input port can accept one or many input connections.

Similar to readers, the properties on the filter allow you to control the filtering algorithm. The properties available depend on the filter itself.

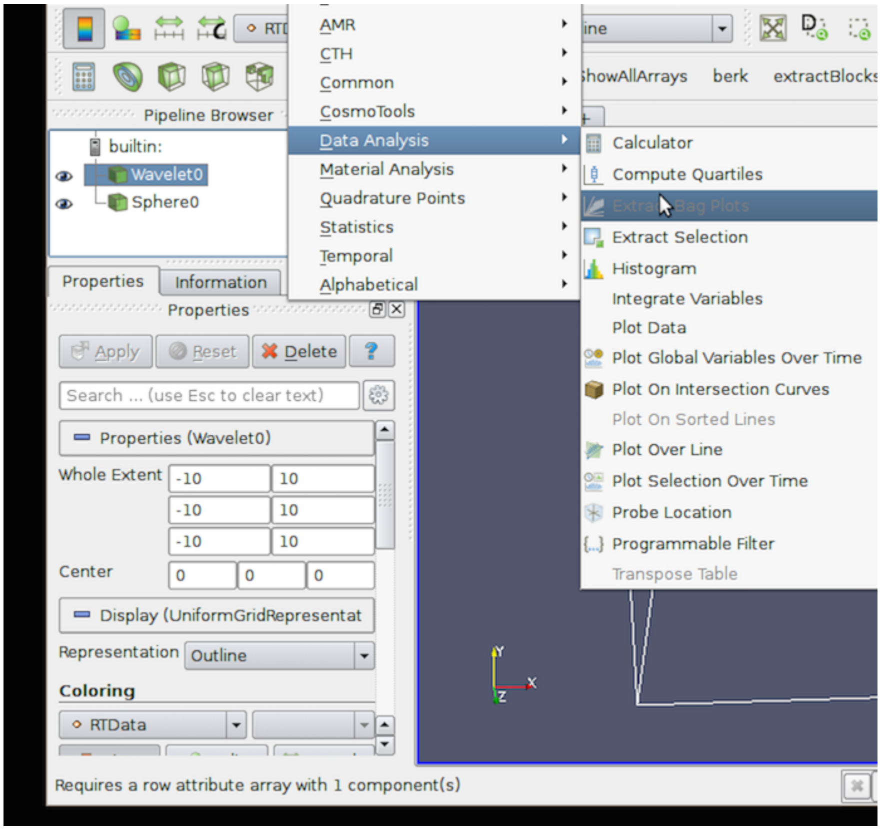
### 1.5.2 Creating filters in `paraview`

All available filters in `paraview` are listed under the *Filters* menu. These are organized in various categories. To create a filter to transform the data produced by a source or a reader, you select the source in the `Pipeline Browser` to make it active, and then click on the corresponding menu item in the *Filters* menu. If a menu item is disabled, it implies that the active source does not produce data that can be transformed by this filter.

---

**Did you know?**

> If a menu item in the *Filters* menu is disabled, it implies that the active source(s) is not producing data of the expected type or the characteristics needed by the filter. On Windows and Linux machines, if you hover over the disabled menu item, the status bar will show the reason why the filter is not available.



Whole Extent -10 10 / -10 10 / -10 10

Center 0 0 0

Representation Outline

RTData

Requires a row attribute array with 1 component(s)

---

### Multiple input connections

When you create a filter, the active source is connected to the first input port of the filter. Filters like `Append Datasets` can take multiple input connections on that input port. In such a case, to pass multiple pipeline modules as connections on a single input port of a filter, select all the relevant pipeline modules in the `Pipeline Browser` . You can select multiple items by using the `CTRL` (or ) and  key modifiers. When multiple pipeline modules are selected, only the filters that accept multiple connections on their input ports will be enabled in the *Filters* menu.

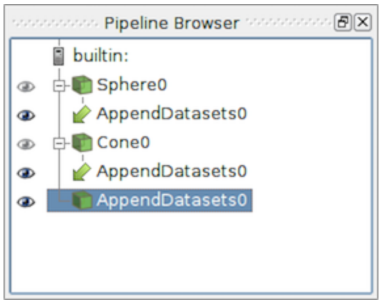


Fig. 1.74: The `Pipeline Browser` showing a pipeline with multiple input connections. The `Append Datasets` filter has two input connections on its only input port, `Sphere0` and `Cone0` .

### Multiple input ports

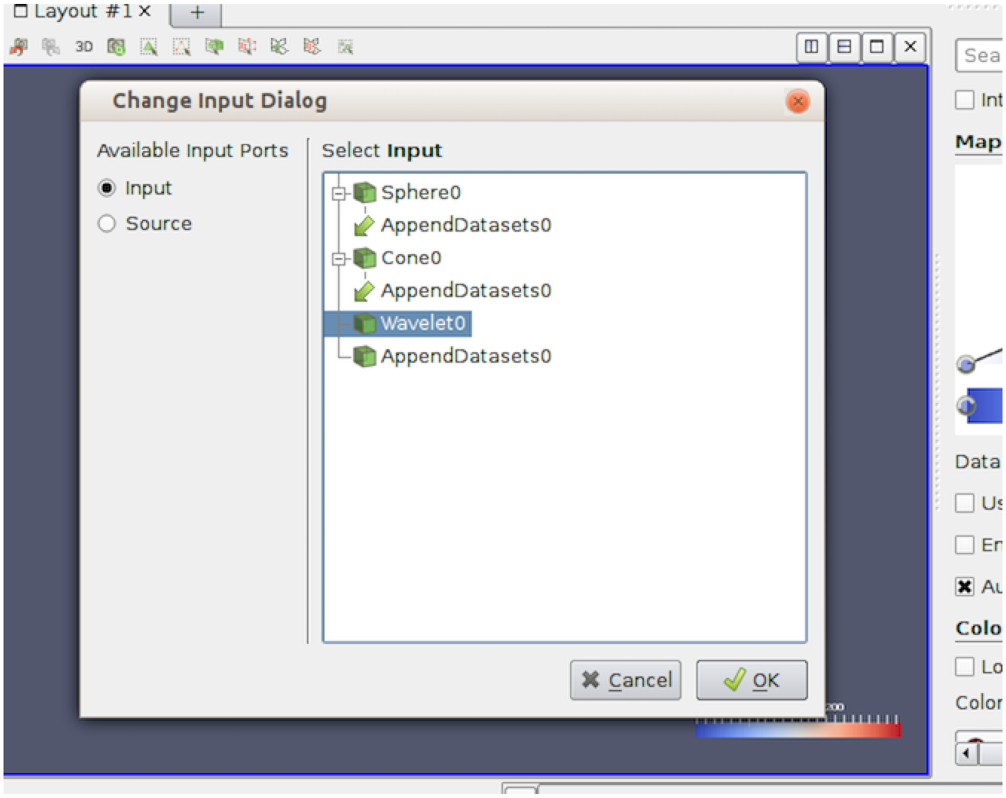
Most filters have just one input port. Hence, as soon as you click on the filter name in the *Filters* menu, it will create a new filter instance and that will show up in the `Pipeline Browser` . Certain filters, such as `Resample With Dataset` , have multiple inputs that must be set up before the filter can be created. In such a case, when you click on the filter name, the `Change Input Dialog` will pop up, as seen in Fig. 1.75. This dialog allows you to select the pipeline modules to be connected to each of the input ports. The active source(s) is connected by default to the first input port. You are free to change those as well.

### Changing input connections

`paraview` allows you to change the inputs to a filter after the filter has been created. To change inputs to a filter, right-click on the filter in the `Pipeline Browser` to get the context menu, and then select `Change Input...` . This will pop up the same `Change Input Dialog` as when creating a filter with multiple input ports. You can use this dialog to set new inputs for this filter.

**Did you know?**

While the *Filters* menu is a handy way to create new filters, with the long list of filters available in **ParaView**, manually finding a particular filter in this menu can be very challenging. To make it easier, **ParaView** incorporates a quick launch mechanism. When you want to create a new filter (or a source), simply type `CTRL` + `Space` or `Alt` + `Space`. This will pop up the quick-launch dialog. Now, start typing the name of the filter you want. As you type, the dialog will update to show the filters and sources that match the typed text. You can use the arrow keys to navigate and use the `Enter` key to create the selected filter (or source). Press  while pressing `Enter` to quickly apply the filter on creation, equivalent to creating the filter and then clicking the `Apply` button. Note that filters may be disabled, as was the case in the *Filters* menu but by default the selected item will be the first enabled filter. You can use `Esc` to clear the text you have typed so far. Hit the `Esc` a second time, and the dialog will close without creating any new filter.



Change Input Dialog

Available Input Ports

● Input

○ Source

Select **Input**

- Sphere0
  - AppendDatasets0
- Cone0
  - AppendDatasets0
- Wavelet0
  - AppendDatasets0

Fig. 1.75: The Change Input Dialog is shown to allow you to pick inputs for each of the input ports for a filter with multiple input ports. To use this dialog, first select the Input Port you want to edit on the left side, and select the pipeline module(s) that are to be connected to this input port. Repeat the step for the other input port(s). If an input port can accept multiple input connections, you can select multiple modules, just like in the Pipeline Browser .

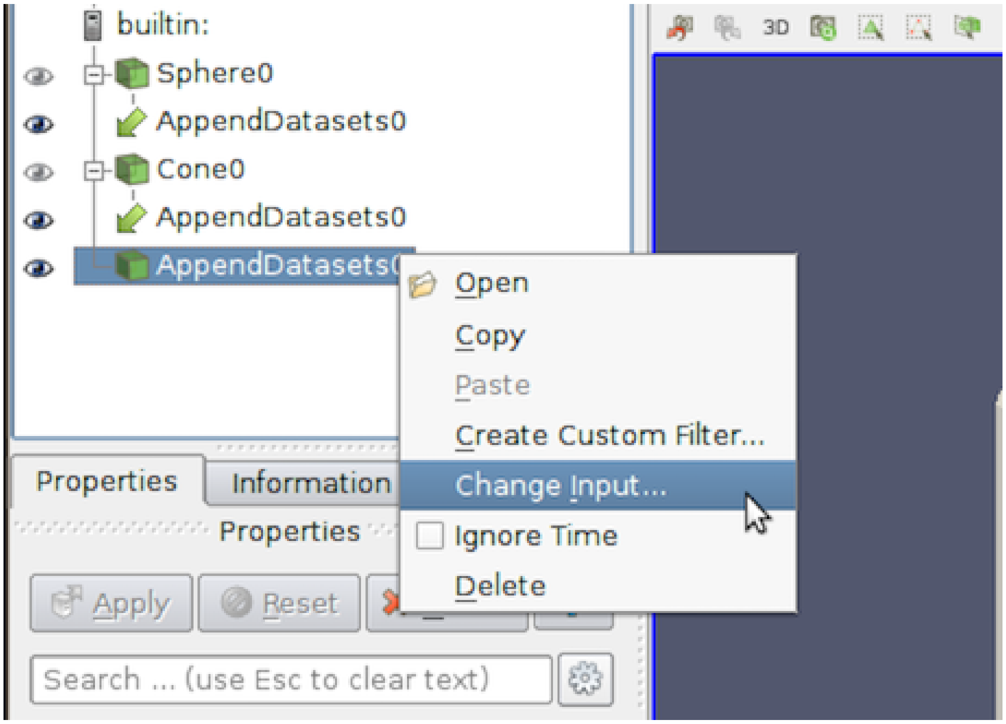
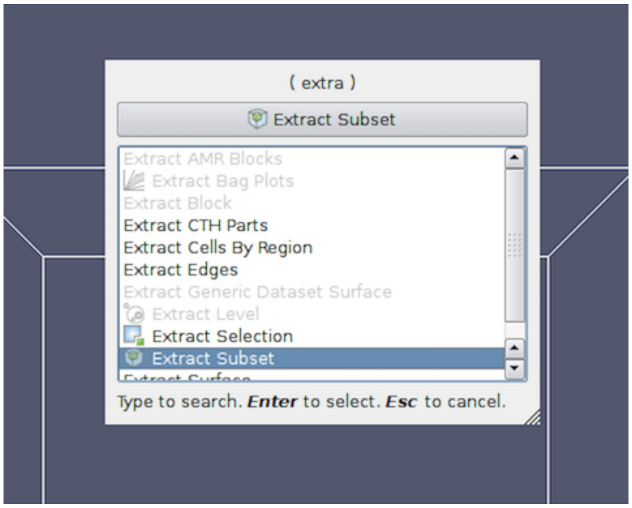


Fig. 1.76: The context menu in the `Pipeline Browser` showing the option to change inputs for a filter.



### 1.5.3 Creating filters in pvpython

To create a filter in pvpython, you simply create the object by using its name as a constructor function.

```
>>> from paraview.simple import *
    ...
>>> filter = Shrink()
```

Similar to paraview, the filter will use the active source(s) as the input. Additionally, you can explicitly specify the input in the function arguments.

```
>>> reader = OpenDataFile(...)
    ...
>>> shrink = Shift(Input=reader)
```

#### Multiple input connections

To setup multiple input connections, you can specify the connections as follows:

```
>>> sphere = Sphere()
>>> cone = Cone()

# Simply pass the sources as a list to the constructor function.
>>> appendDatasets = AppendDatasets(Input=[sphere, cone])
>>> print(appendDatasets.Input)
[<paraview.servermanager.Sphere object at 0x6d75f90>, <paraview.servermanager.Cone␣
↪object at 0x6d75c50>]
```

#### Multiple input ports

Setting up connections to multiple input ports is similar to the multiple input connections, except that you need to ensure that you name the input ports properly.

```
>>> sphere = Sphere()
>>> wavelet = Wavelet()

>>> resampleWithDataSet = ResampleWithDataset(Input=sphere, Source=wavelet)
```

#### Changing input connections

Changing inputs in Python is as simple as setting any other property on the filter.

```
# For filter with single input connection
>>> shrink.Input = cone

# for filters with multiple input connects
>>> appendDatasets.Input = [reader, cone]

# to add a new input.
>>> appendDatasets.Input.append(sphere)
```

(continues on next page)

(continued from previous page)

```
# to change multiple ports
>>> resampleWithDataSet.Input = wavelet2
>>> resampleWithDataSet.Source = cone
```

### 1.5.4 Changing filter properties in paraview

Filters provide properties that you can change to control the processing algorithm employed by the filter. Changing and viewing properties on filters is the same as with any other pipeline module, including readers and sources. You can view and change these properties, when available, using the Properties panel. Section 2.1 covers how to effectively use the Properties panel. Since this panel only shows the properties present on the *active source* , you must ensure that the filter you are interested in is active. To make the filter active, use the Pipeline Browser to click on the filter and select it.

### 1.5.5 Changing filter properties in pvpython

With pvpython, the available properties are accessible as properties on the filter object, and you can get or set their values by name (similar to changing the input connections (Section 1.5.3)).

```
# You can save the object reference when it's created.
>>> shrink = Shrink()

# Or you can get access to the active source.
>>> Shrink() # <-- this will make the Shrink the active source.
>>> shrink = GetActiveSource()

# To figure out available properties, you can always use help.
>>> help(shrink)
Help on Shrink in module paraview.servermanager object:

class Shrink(SourceProxy)
|  Shrink(**args)
|
|  The Shrink filter causes the individual cells of a dataset to break apart from
   each other by moving each cell's points toward the centroid of the cell. (The
   centroid of a cell is the average position of its points.) This filter operates
   on any type of dataset and produces unstructured grid output.
|
|  Method resolution order:
|      Shrink
|      SourceProxy
|      Proxy
|      builtins.object
|
|  Methods defined here:
|
|  Initialize = aInitialize(self, connection=None, update=True)
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
```

(continues on next page)

(continued from previous page)

```
|  Input
|      This property specifies the input to the Shrink filter.
|
|  ShrinkFactor
|      The value of this property determines how far the points will move. A value
       of 0 positions the points at the centroid of the cell; a value of 1 leaves
       them at their original positions.
 ....

# To get the current value of a property:
>>> sf = shrink.ShrinkFactor
>>> print sf
0.5

# To set the value
>>> shrink.ShrinkFactor = 0.75
```

In the rest of this chapter, we will discuss some of the commonly used filters in detail. They are grouped under categories based on the type of operation that they perform.

### 1.5.6 Filters for sub-setting data

These filters are used for extracting subsets from an input dataset. How this subset is defined and how it is extracted depends on the type of the filter.

#### Clip

`Clip` is used to clip any dataset using either an implicit function (such as a plane, sphere, or a box) or using values of a scalar data array in the input dataset. A scalar array is a point or cell attribute array with a single component. Clipping involves iterating over all cells in the input dataset and then removing those cells that are considered *outside* of the space defined by the implicit function or that have an attribute values less than the selected value. For cells that straddle the clipping surface, these are *clipped* to pass through the part of the cell that is truly inside the specified implicit function (or greater than the scalar value).

This filter converts any dataset into an unstructured grid (Section 1.3.1) or a multiblock of unstructured grids (Section 1.3.1) in the case of composite datasets.

#### Clip in `paraview`

To create the `Clip` filter, access it through the *Filters* > *Common* or the *Filters* > *Alphabetical* menus. This filter is also accessible from the `Common` filters toolbar by clicking the  button to create this filter.

On the `Properties` panel, you will see the available properties for this filter. One of the first things that you should select is the `Clip Type` . `Clip Type` is used to specify the type of implicit function to use for the clipping operations. The available options include `Plane` , `Box` , `Sphere` , and `Scalar` . Selecting any one of these options will update the panel to show properties that are used to define the implicit function, e.g., the `Origin` and the `Normal` for the `Plane` or the `Center` and the `Radius` for the `Sphere` . If you select `Scalar` , the panel will let you pick the data array and the value with which to clip. Remember, cells with the data value greater than or equal to the selected value are considered *in* and are passed through the filter.

**Did you know?**

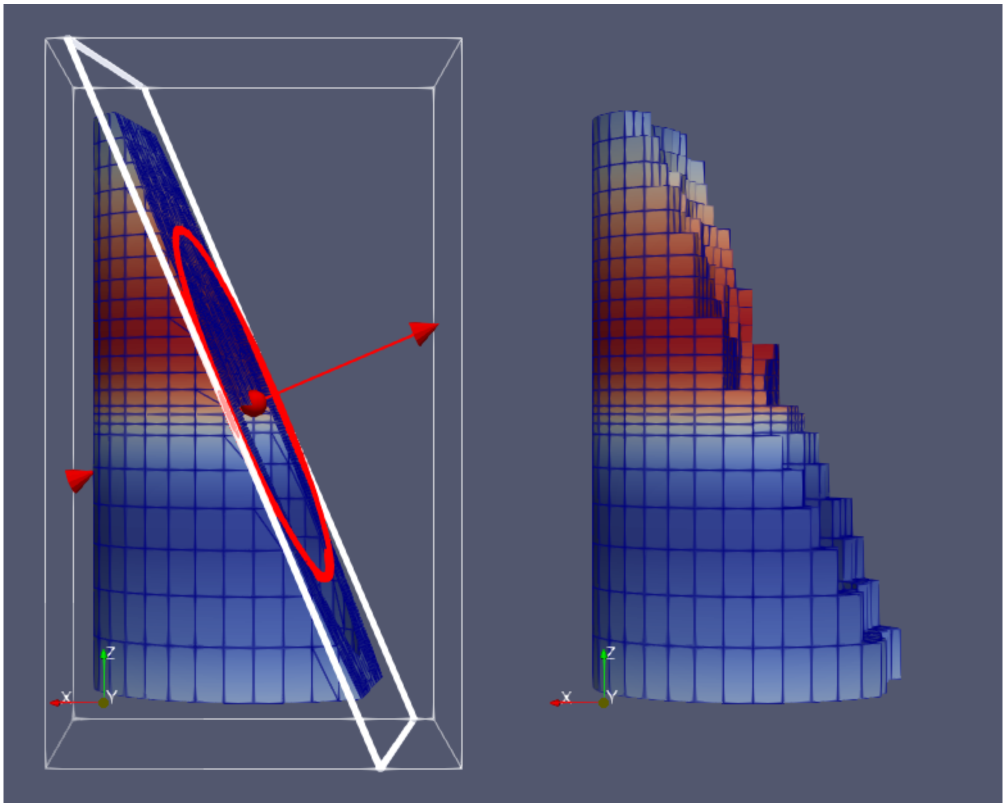


Fig. 1.77: Comparison between results produced by the `Clip` filter with `Crinkle Clip` unchecked (left) and checked (right) when clipping with an implicit plane. The image on the left also shows the 3D widget used to interactivly place the implicit plane for the clipping operation.

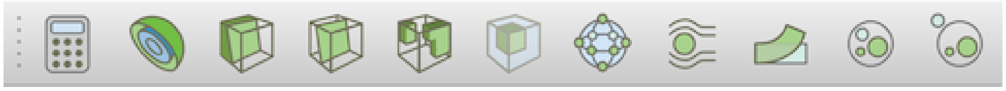


Fig. 1.78: The `Common` filters toolbar in `paraview` for quick access to the commonly used filters.

When clipping with implicit functions, **ParaView** renders widgets in the active view that you can use to interactively control the implicit function, called `3D widgets`. As you interact with the 3D widget, the panel will update to reflect the current values. The 3D widget is considered an aid and not as a part of the actual visualization scene. Thus, if you change the active source and the `Properties` panel navigates away from this filter, the 3D widget will automatically be hidden.

---

The `Inside Out` option can be used to invert the behavior of this filter. Basically, it flips the notion of what is considered inside and outside of the given clipping space.

Check `Crinkle Clip` if you don't want this filter to truly clip cells on the boundary, but want to preserve the input cell structure and to pass the entire cell on through the boundary (Fig. 1.77). This option is not available when clipping by `Scalar`.

### Clip in `pvpython`

This following script demonstrates various aspects of using the `Clip` filter in `pvpython`.

```
# Create the Clip filter.
>>> clip = Clip(Input=...)

# Specify a 'ClipType' to use.
>>> clip.ClipType = 'Plane'

# You can also use the SetProperties API instead.
>>> SetProperties(clip, ClipType='Plane')

>>> print(clip.GetProperty('ClipType').GetAvailable())
['Plane', 'Box', 'Sphere', 'Scalar']

# To set the plane origin and normal
>>> clip.ClipType.Origin = [0, 0, 0]
>>> clip.ClipType.Normal = [1, 0, 0]

# If you want to change to Sphere and set center and
# radius, you can do the following.
>>> clip.ClipType = 'Sphere'
>>> clip.ClipType.Center = [0, 0, 0]
>>> clip.ClipType.Radius = 12

# Using SetProperties API, the same looks like
>>> SetProperties(clip, ClipType='Sphere')
>>> SetProperties(clip.ClipType, Center=[0, 0, 0],
                                 Radius = 12)

# To set Crinkle clipping.
>>> clip.Crinkleclip = 1

# For clipping with scalar, you pick the scalar array
# and then the value as follows:
>>> clip.ClipType = 'Scalar'
>>> clip.Scalars = ('POINTS', 'Temp')
>>> clip.Value = 100
```

```
# As always, to get the list of available properties on
# the clip filter, use help()
>>> help(clip)
Help on Clip in module paraview.servermanager object:

class Clip(SourceProxy)
|  Clip(**args)
|
|  The Clip filter cuts away a portion of the input data set using an implicit
   function (an implicit description). This filter operates on all types of
   data sets, and it returns unstructured grid data on output.
|
|  Method resolution order:
|      Clip
|      SourceProxy
|      Proxy
|      builtins.object
|
|  Methods defined here:
|
|  Initialize = aInitialize(self, connection=None, update=True)
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  ClipType
|      This property specifies the parameters of the clip function (an implicit
       description) used to clip the dataset.
|
|  Crinkleclip
|      This parameter controls whether to extract entire cells in the given
       region or clip those cells so all of the output will stay only on that
       side of region.
|
|  Exact
|      If this property is set to 1 it will clip to the exact specifications
       for the **Box** option only, otherwise the clip will only approximate
       the box geometry. The exact clip is very expensive as it requires
       generating 6 plane clips. Additionally, **Invert** must be checked and
       **Crinkle clip** must be unchecked.
|
|  HyperTreeGridClipper
|      This property specifies the parameters of the clip function (an implicit
       description) used to clip the hyper tree grid.
|
|  Input
|      This property specifies the dataset on which the Clip filter will operate.
|
|  Invert
|      Invert which part of the geometry is clipped.
 ...

# To get help on a specific implicit function type, make it the active
```

(continues on next page)

(continued from previous page)

```
# ClipType and then use help()
>>> clip.ClipType = 'Plane'
>>> help(clip.ClipType)
Help on Plane in module paraview.servermanager object:

class Plane(Proxy)
  ...
```

**Common Errors**

It is very easy to forget that clipping a structured dataset such as image data can dramatically increase the memory requirements, since this filter will convert the structured dataset into an unstructured grid due to the nature of the clipping operation itself. For structured dataset, think about using `Slice` or `Extract Subset` filters instead, whenever appropriate. Those are not entirely identical operations, but they are often sufficient.

### Slice

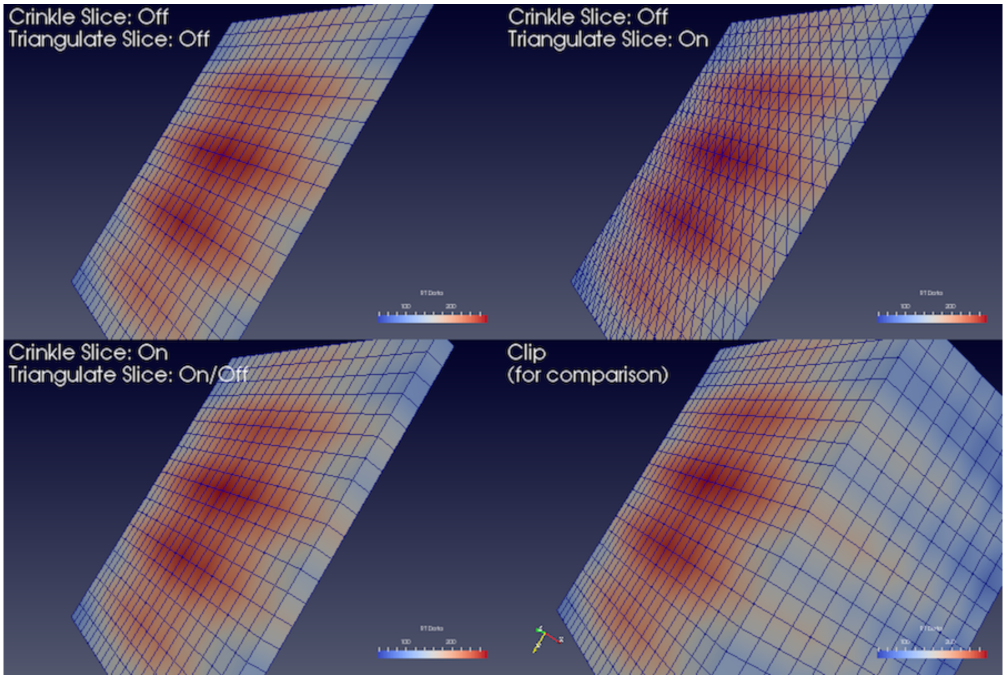


Fig. 1.79: Comparison between results produced by the `Slice` filter when slicing image data with an implicit plane with different options. The lower-left image shows the output produced by the `Clip` filter when clipping with the same implicit function, for contrast.

The `Slice` filter slices through the input dataset with an implicit function such as a plane, a sphere, or a box. Since this filter returns data elements along the implicit function boundary, this is a dimensionality reducing filter (except when crinkle slicing is enabled), i.e., if the input dataset has 3D elements like tetrahedrons or hexahedrons, the output will have 2D elements, line triangles, and quads, if any. While slicing through a dataset with 2D elements, the result will be lines.

The properties available on this filter, as well as the way of setting this filter up, is very similar to the `Clip` filter with a few notable differences. What remains similar is the set up of the implicit function – you have similar choices: `Plane`

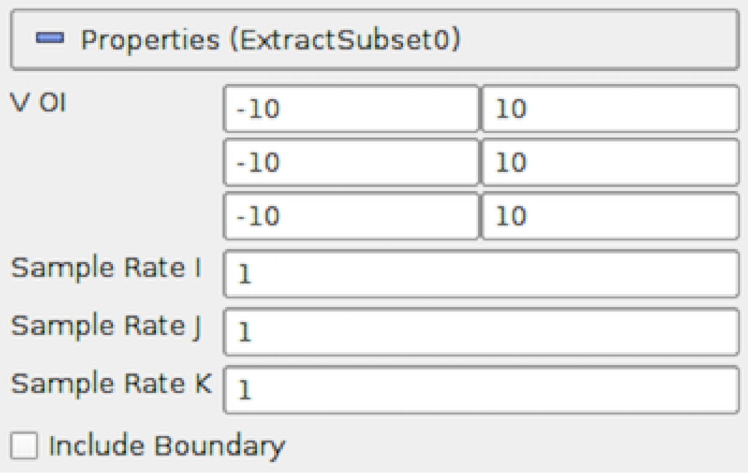
, Box , Sphere , and Cylinder , as well as the option to toggle Crinkle slice (i.e., to avoid cutting through cells, pass complete cells from the input dataset that intersects the implicit function).

What is different includes the lack of slicing by Scalar (for that, you can use the Contour filter) and a new option, Triangulate the slice . Fig. 1.79 shows the difference in the generated meshes when various slice properties are changed.

The Slice filter is more versatile than the Slice representation. First, the Slice representation is available for image datasets only, whereas the Slice filter can be used on any type of 3D dataset. Second, the representation extracts a subset of the image consisting of a 2D slice oriented in the XY, YZ, or XZ planes at the image voxel locations while the plane used by the filter can be placed arbitrarily. Third, since the Slice representation always shows a flat object and lighting may interfere with interpretation of data values on the slice, lighting is not applied to the Slice representation. Lighting is applied, however, to results from the Slice filter. Lastly, the Slice representation may be faster than the filter to update and scrub through different slices because it does not need to compute the intersection of a plane with cells in the dataset.

In paraview, this filter can be created using the  button on the Common filters toolbar, besides the *Filters* menu.

### Extract Subset



Properties (ExtractSubset0)

V OI -10 10
-10 10
-10 10

Sample Rate I 1

Sample Rate J 1

Sample Rate K 1

☐ Include Boundary

Fig. 1.80: The Properties panel for the Extract Subset filter showing all available properties (including the advanced properties).

For structured datasets such as image datasets (Section 1.3.1), rectilinear grids (Section 1.3.1), and curvilinear grids (Section 1.3.1), Extract Subset filter can be used to extract a region of interest or a subgrid. The region to extract is specified using structured coordinates, i.e., the $i$, $j$, $k$ values. Whenever possible, this filter should be preferred over Clip or Slice for structured datasets, since it preserves the input data type. Besides extracting a subset, this filter can also be used to resample the dataset to a coarser resolution by specifying the sample rate along each of the structured dimensions.

### Extract Subset in `paraview`

This is one of the filters available on the `Common` filters toolbar  To specify the region of interest, use the `VOI` property. The values are specified as min and max values for each of the structured dimensions ($i$, $j$, $k$,) in each row. `Sample Rate I` , `Sample Rate J` , and `Sample Rate K` specify the sub-sampling rate. Set it to a value greater than one to sub-sample. `Include Boundary` is used to determine if the boundary slab should be included in the extracted result, if the sub-sampling rate along that dimension is greater than 1, and the boundary slab would otherwise have been skipped.

### Threshold

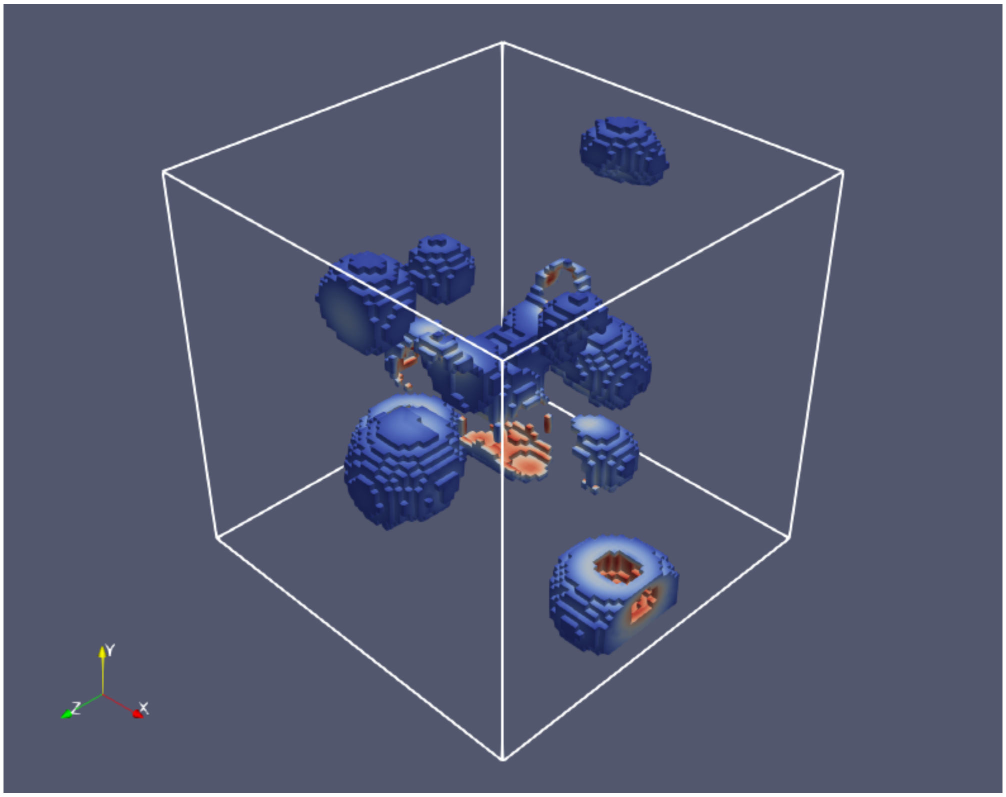
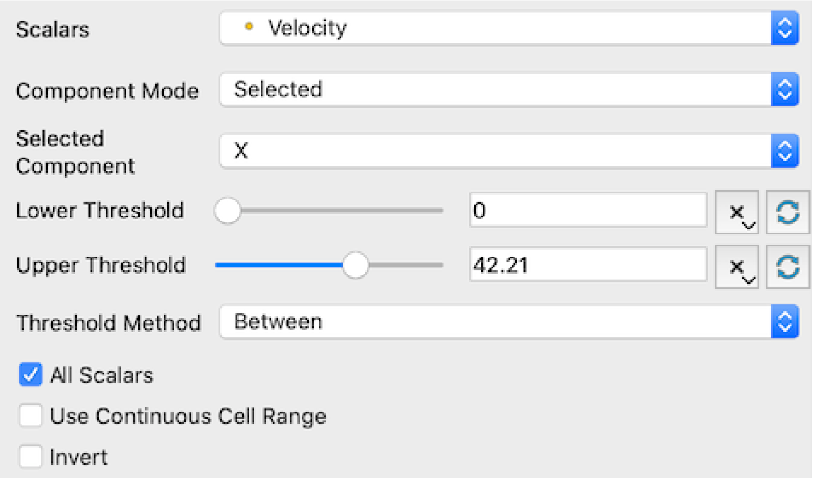


Fig. 1.81: Results from using the `Threshold` filter on the *iron_protein.vtk* dataset from **ParaView**'s testing data.

The `Threshold` filter extracts cells of the input dataset with scalar values lying within the specified range, depending on the selected threshold method. This filter operates on either point-centered or cell-centered data. Any type of dataset can be used as input. The filter produces an unstructured grid output.

When thresholding with cell data, all cells that have scalars within the specified range will be passed through the filter. When thresholding with point data, cells with *all* points with scalar values within the range are passed through if `All Scalars` is checked; otherwise, cells with *any* point that passes the thresholding criteria are passed through.

### Threshold in `paraview`



Scalars ● Velocity

Component Mode: Selected

Selected Component: X

Lower Threshold: 0

Upper Threshold: 42.21

Threshold Method: Between

☑ All Scalars

□ Use Continuous Cell Range

□ Invert

Fig. 1.82: The `Properties` panel for the `Threshold` filter.

This filter is represented as  on the `Common` filters toolbar. After selecting the `Scalars` with which to threshold from the combo-box, the `Lower Threshold` and `Upper Threshold` values can be modified to specify the range. If the range shown by the sliders is not sufficient, it is also possible to manually type the values in the input boxes. The values are deliberately not clamped to the current data range.

The threshold method can also be selected using the `Threshold Method` combo box:

- `Between`: Extracts cells with scalar values between the `Lower Threshold` and `Upper Threshold`.
- `Below Lower Threshold`: Extracts cells with scalar values smaller than the `Lower Threshold`.
- `Above Upper Threshold`: Extracts cells with scalar values larger than the `Upper Threshold`.

If the `Scalars` property is set to a vector array, the `Component Mode` property can be used to choose whether “All” components must pass the threshold test, “Any” component needs to pass the threshold test, or if a “Selected” component needs to pass the threshold test. If `Component Mode` is “Selected”, the `Selected Component` property designates which vector component needs to pass the threshold test. Other components are not tested.

### Threshold in `pvpython`

```
# Create the filter. If Input is not specified, the active source will be
# used as the input.
>>> threshold = Threshold(Input=...)

# Here's how to select a scalar array.
>>> threshold.Scalars = ("POINTS", "scalars")

# The value is a tuple where the first value is the association: either "POINTS"
# or "CELLS", and the second value is the name of the selected array.
>>> print(threshold.Scalars)
['POINTS', 'scalars']

>>> print(threshold.Scalars.GetArrayName())
'scalars'
```

(continues on next page)

(continued from previous page)

```
>>> print(threshold.Scalars.GetAssociation())
'POINTS'

# Different threshold methods are available and are set using one of the following:
>>> threshold.ThresholdMethod = "Between"                # Uses both lower and upper␣
↪values
>>> threshold.ThresholdMethod = "Below Lower Threshold"  # Uses only lower value
>>> threshold.ThresholdMethod = "Above Upper Threshold"  # Uses only upper value

# The adequate threshold values are then specified as:
>>> threshold.LowerThreshold = 63.75
>>> threshold.UpperThreshold = 252.45
```

To determine the types of arrays available in the input dataset, and their ranges, refer to the discussion on data information in Section 1.3.3.

### Iso Volume

The `Iso Volume` filter is similar to `Threshold` in that you use this to create an output dataset from an input where the cells that satisfy the specified range are scalar values. In fact, the filter is identical to `Threshold` when the cell data scalars are selected. For point data scalars, however, this filter acts similar to the `Clip` filters when clipping with scalars, in that cells are clipped along the iso-surface formed by the scalar range.

### Extract Selection

`Extract Selection` is a general-purpose filter to extract selected elements from a dataset. There are several ways of making selections in **ParaView**. Once you have made the selection, this filter allows you to extract the selected elements as a new dataset for further processing. We will cover this filter in more detail when looking at selections in **ParaView** in Section 1.6.6.

## 1.5.7 Filters for geometric manipulation

These filters are used to transform the geometry of the dataset without affecting its topology or its connectivity.

### Transform

The `Transform` can be used to arbitrarily translate, rotate, and scale a dataset. The transformation is applied by scaling the dataset, rotating it, and then translating it based on the values specified.

As this is a geometric manipulation filter, this filter does not affect connectivity in the input dataset. While it tries to preserve the input dataset type, whenever possible, there are cases when the transformed dataset can no longer be represented in the same data type as the input. For example, with image data (Section 1.3.1) and rectilinear grids (Section 1.3.1) that are transformed by rotation, the output dataset can be non-axis aligned and, hence, cannot be represented as either data types. In such cases, the dataset is converted to a structured, or curvilinear, grid (Section 1.3.1). Since curvilinear grids are not as compact as the other two, the need to store the results in a more general data type implies a considerable increase in the memory footprint.

### Transform in `paraview`

You can create a new `Transform` from the *Filters* > *Alphabetical* menu. Once created, you can set the transform as the translation, rotation, and scale to use utilizing the `Properties` panel. Similar to `Clip` , this filter also supports using a 3D widget to interactively set the transformation.

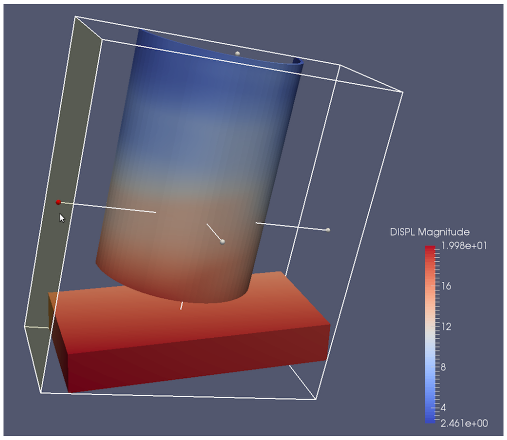


Fig. 1.83: The `Transform` filter showing the 3D widget that can be used to interactively set the transform.

### Transform in `pvpython`

```
# To create the filter(if Input is not specified, the active source will be
# used as the input).
>>> transform = Transform(Input=...)

# Set the transformation properties.
>>> transform.Translate.Scale = [1, 2, 1]
>>> transform.Transform.Translate = [100, 0, 0]
>>> transform.Transform.Rotate = [0, 0, 0]
```

### Reflect

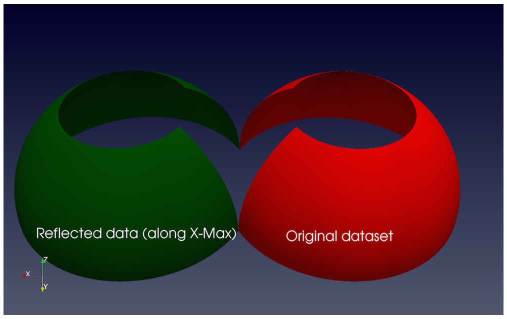


Fig. 1.84: The `Reflect` filter can be used to reflect a dataset along a specific axis plane.

`Reflect` can be used to reflect any dataset across an axis plane. You can pick the axis plane to be one of the planes formed by the bounding box of the dataset. For that, set `Plane` as `X Min` , `X Max` , `Y Min` , `Y Max` , `Z Min` , or `Z Max` .

To reflect across an arbitrary axis plane, select X , Y , or Z for the Plane property, and then set the Center to the plane offset from the origin.

This filter reflects the input dataset and produces an unstructured grid (Section 1.3.1). Thus, the same caveats for Clip and Threshold filter apply here when dealing with structured datasets.

### Warp By Vector

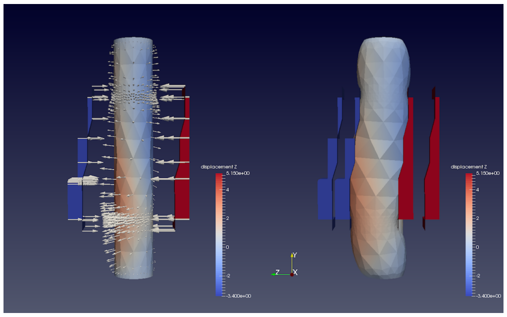


Fig. 1.85: The Warp By Vector filter can be used to displace points in original data shown on the left, using the *displacement* vectors (indicated by arrow glyphs Section 1.5.8) to produce the result shown on the right.

Warp By Vector can be used to displace point coordinates in an input mesh using vectors in the dataset itself. You select the vectors to use utilizing the Vectors property on the Properties panel. Scale Factor can be used to scale the displacement applied.

### Warp By Scalar

Warp By Scalar is similar to Warp By Vector in the sense that it warps the input mesh. However, it does so using a scalar array in the input dataset. The direction of displacement can either be explicitly specified using the Normal property, or you can check Use Normal to use normals at the point locations.

## 1.5.8 Filters for sampling

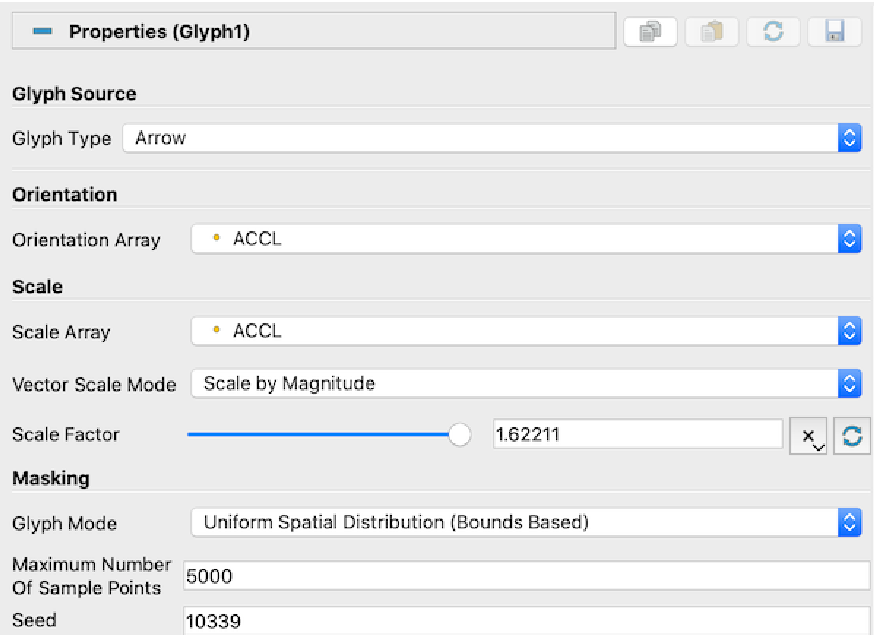
These filters compute new datasets that represent some essential features from the datasets that they take as input.

### Glyph

Glyph is used to place markers or glyphs at point locations in the input dataset. The glyphs can be oriented or scaled based on vector and scalar attributes on those points.

To create this filter in paraview, you can use the *Filters* menu, as well as the  button on the Common filters toolbar. You first select the type of glyph using one of the options in Glyph Type . The choices include Arrow , Sphere , Cylinder , etc. Next, you select the point arrays to use as the Orientation Array (selecting No orientation array will result in the glyphs not being oriented). Similarly, you select a point array to serve as the glyph Scale Array (no scaling is performed if No scale array is chosen).

If the Scale Array is set to a vector array, the Vector Scale Mode property is available to select which properties of the vector should be used to transform each glyph. If Scale by Magnitude is chosen, then the glyph at a point will be scaled by the magnitude of the vector at that point. If Scale by Components is chosen, glyphs will be scaled separately in each dimension by the vector component in that dimension.



Properties (Glyph1)

Glyph Source

Glyph Type: Arrow

Orientation

Orientation Array: ACCL

Scale

Scale Array: ACCL

Vector Scale Mode: Scale by Magnitude

Scale Factor: 1.62211

Masking

Glyph Mode: Uniform Spatial Distribution (Bounds Based)

Maximum Number Of Sample Points: 5000

Seed: 10339

Fig. 1.86: The `Properties` panel for the `Glyph` filter.

The `Scale Factor` is used to apply a constant scaling to all the glyphs, independent of the `Scale Array` and `Vector Scale Mode` properties. Choosing a good scale factor depends on several things including the bounds on the input dataset, the `Scale Array` and `Vector Scale Mode` selected, and the range for the array selected as the `Scale Array` . You can use the  button next to the `Scale Factor` widget to have `paraview` pick a usually reasonable scale factor value based on the current dataset and scaling properties.

The `Masking` properties control which points from the input dataset get glyphed. The `Glyph Mode` controls how points are selected to be glyphs (Fig. 1.87). The available options are as follows:

- `All Points` : This selects all points in the input dataset for glyphing. Use this mode with caution and only when the input dataset has relatively few points. Since all points in the input dataset are glyphed, this can not only cause visual clutter, but also clog up memory and take a long to time to generate and render the glyphs.
- `Every Nth Points` : This elects every $n^{th}$ point in the input dataset for glyphing, where $n$ can be specified using `Stride` . Setting `Stride` to 1 will have the same effect as `All Points` .
- `Uniform Spatial Distribution (Bounds Based)` : This selects a random set of points. The algorithm works by first computing up to `Maximum Number of Sample Points` in the space defined by the bounding box of the input dataset. Then, points in the input dataset that are close to the point in this set of sample points are glyphed. The `Seed` is used to seed the random number generator used to generate the sample points. This ensures that the random sample points are reproducible and consistent.
- `Uniform Spatial Distribution (Surface Sampling)` : Selects a random set of points from the outer bounding surface of the input dataset. An inverse transform sampler is used to find a 2D cell on the surface to sample and a point is uniformly sampled from that cell.
- `Uniform Spatial Distribution (Volume Sampling)` : Similar to the surface sampling mode described above, but the inverse transform sampler is used to find a 3D cell from which a random point is uniformly sampled

**Did you know?**

The `Glyph` representation can be used for many of the same visualizations where a `Glyph` filter might be used. It may offer faster rendering and consume less memory than the `Glyph` filter with similar capabilities. In circumstances where

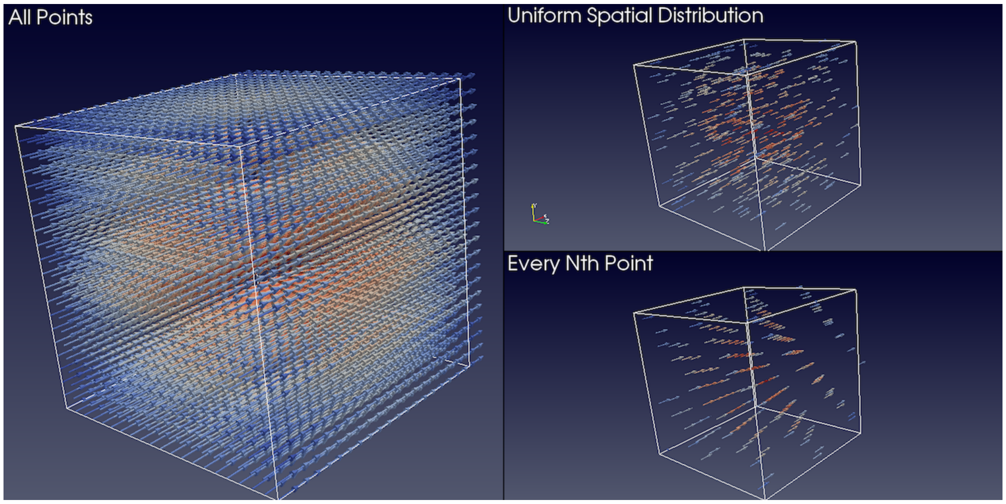
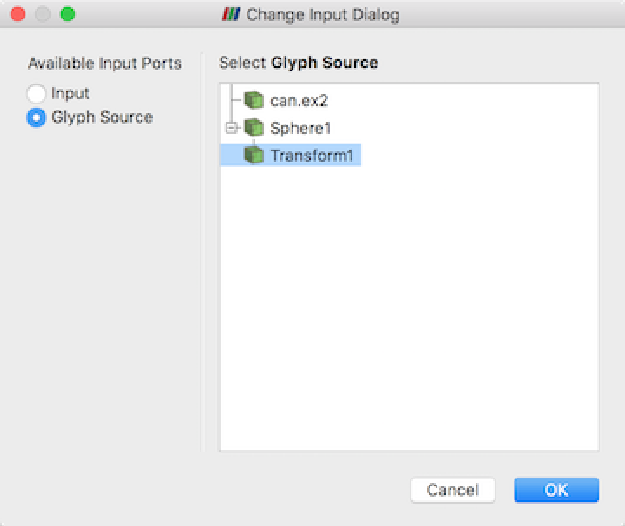


Fig. 1.87: Comparison between various `Glyph Mode` s when applied to the same dataset generated by the `Wavelet` source.

generating a 3D geometry is required, e.g., when exporting glyph geometry to a file, the `Glyph` filter is required.

### Glyph With Custom Source

`Glyph With Custom Source` is the same as `Glyph` , except that instead of a limited set of `Glyph Type` , you can select any data source producing a polygonal dataset (Section 1.3.1) available in the `Pipeline Browser` . To use this filter, select the data source you wish to glyph in the `Pipeline Browser` and attach this filter to it. You will be presented a dialog where you can set the `Input` (which defaults to the source you selected) and the `Glyph Source` .


Change Input Dialog

Available Input Ports
○ Input
● Glyph Source

Select **Glyph Source**
- can.ex2
- Sphere1
  - Transform1

Fig. 1.88: Setting the `Input` and `Glyph Source` in the `Glyph With Custom Source` filter.

### Stream Tracer

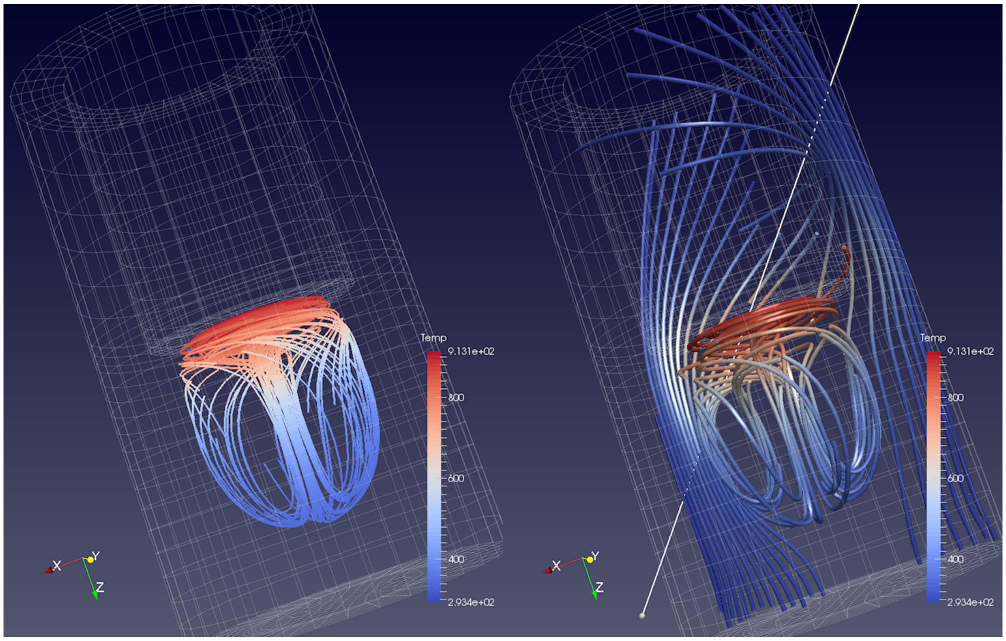


Fig. 1.89: Streamlines generated from the `disk_out_ref.ex2` dataset using the `Point Source` (left) and the `High Resolution Line Source` (right). On the left, we also added the `Tube` filter to the output of the `Stream Tracer` filter to generate 3D tubes rather than 1D polygonal lines, which can be hard to visualize due to lack of shading.

The `Stream Tracer` filter is used to generate streamlines for vector fields. In visualization, streamlines refer to curves that are instanteneously tangential to the the vector field in the dataset. They provide an indication of the direction in which the particles in the dataset would travel at that instant in time. The algorithm works by taking a set of points, known as *seed* points, in the dataset and then integrating the streamlines starting at these seed points.

In `paraview`, you can create this filter using the *Filters* menu, as well as the button on the `Common` filters toolbar. To use this filter, you first select the attribute array to use as the `Vectors` for generating the streamline. `Integration Parameters` let you fine tune the streamline integration by specifying the direction to integrate, `Integration Direction` , as well as the type of integration algorithm to use, `Integrator Type` . Advanced integration parameters are available in the advanced view of the `Properties` panel that let you further tune the integration, including specifying the step size and others. You use the `Maximum Streamline Length` to limit the maximum length for the streamline – the longer the length, the longer the generated streamlines.

`Seeds` group lets you set how the seed points for generating the streamlines are produced. You have two options: `Point Source` , which produces a point clound around the user-specified `Point` based on the parameters specified, and `High Resolution Line Source` , which produces seed points along the user-specified line. You can use the 3D widgets shown in the active `Render View` to interactively place the center for the point cloud or for defining the line.

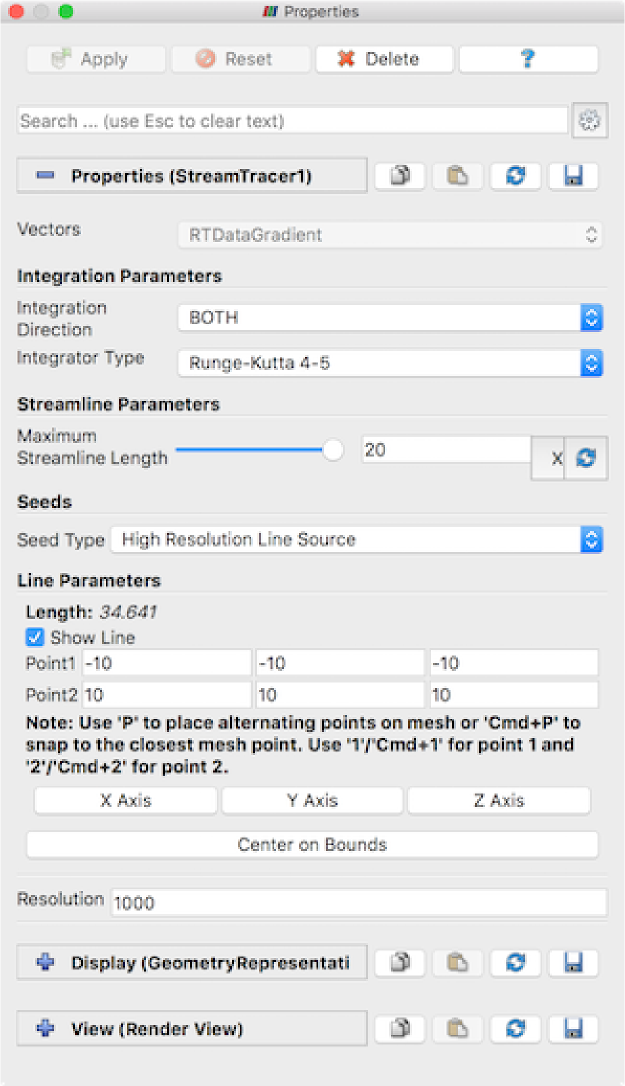
---

**Did you know?**

The `Stream Tracer` filter produces a polydata with 1D lines for each of the generated streamlines. Since 1D lines cannot be shaded like surfaces in the `Render View` , you can get visualizations where it is hard to follow the streamlines. To give the streamlines some 3D structure, you can apply the `Tube` filter to the output of the streamlines. The properties on the `Tube` filter let you control the thickness of the tubes. You can also vary the thickness of the tubes based on data array, e.g., the magnitude of the vector field at the sample points in the streamline!

---

A script using the `Stream Tracer` filter in `paraview` typically looks like this:


Properties
Apply
Reset
Delete
?
Search ... (use Esc to clear text)
Properties (StreamTracer1)

Vectors RTDataGradient

**Integration Parameters**

Integration Direction BOTH

Integrator Type Runge-Kutta 4-5

**Streamline Parameters**

Maximum Streamline Length 20

**Seeds**

Seed Type High Resolution Line Source

**Line Parameters**

**Length:** *34.641*

☑ Show Line

Point1 -10 -10 -10

Point2 10 10 10

**Note: Use 'P' to place alternating points on mesh or 'Cmd+P' to snap to the closest mesh point. Use '1'/'Cmd+1' for point 1 and '2'/'Cmd+2' for point 2.**

Resolution 1000

Fig. 1.90: The `Properties` panel showing the default properties for the `Stream Tracer` filter.

```
# find source
>>> disk_out_refex2 = FindSource('disk_out_ref.ex2')

# create a new 'Stream Tracer'
>>> streamTracer1 = StreamTracer(Input=disk_out_refex2,
                                 SeedType='Point Source')
>>> streamTracer1.Vectors = ['POINTS', 'V']

# init the 'Point Source' selected for 'SeedType'
>>> streamTracer1.SeedType.Center = [0.0, 0.0, 0.07999992370605469]
>>> streamTracer1.SeedType.Radius = 2.015999984741211

# show data in view
>>> Show()

# create a new 'Tube'
>>> tube1 = Tube(Input=streamTracer1)

# Properties modified on tube1
>>> tube1.Radius = 0.1611409378051758

# show the data from tubes in view
>>> Show()
```

### Stream Tracer With Custom Source

`Stream Tracer` allows you to specify the seed points either as a point cloud or as a line source. However, if you want to provide your own seed points from another data producer, use the `Stream Tracer With Custom Source` . Similar to `Glyph With Custom Source` , this filter allows you to pick a second input connection to use as the seed points.

### Resample With Dataset

`Resample With Dataset` samples the point and cell attributes of one dataset on to the points of another dataset. The two datasets are supplied to the filter using its two input ports: `Input` , which is the dataset that provides the attributes to resample, and `Source` , which is the dataset that provides the points to sample at. This filter is available under the *Filters* menu.

### Resample To Image

`Resample To Image` is a specialization of `Resample With Dataset` . The filter takes one input and samples its point and cell attributes onto a uniform grid of points. The bounds and extents of the uniform grid can be specified using the properties panel. By default, the bounds are set to the bounds of the input dataset. The output of the filter is an Image dataset.

Some operations can be performed more efficiently on uniform grid datasets. Volume rendering is one such operation. The `Resample to Image` filter can be used to convert any dataset to Image data before performing such operations.

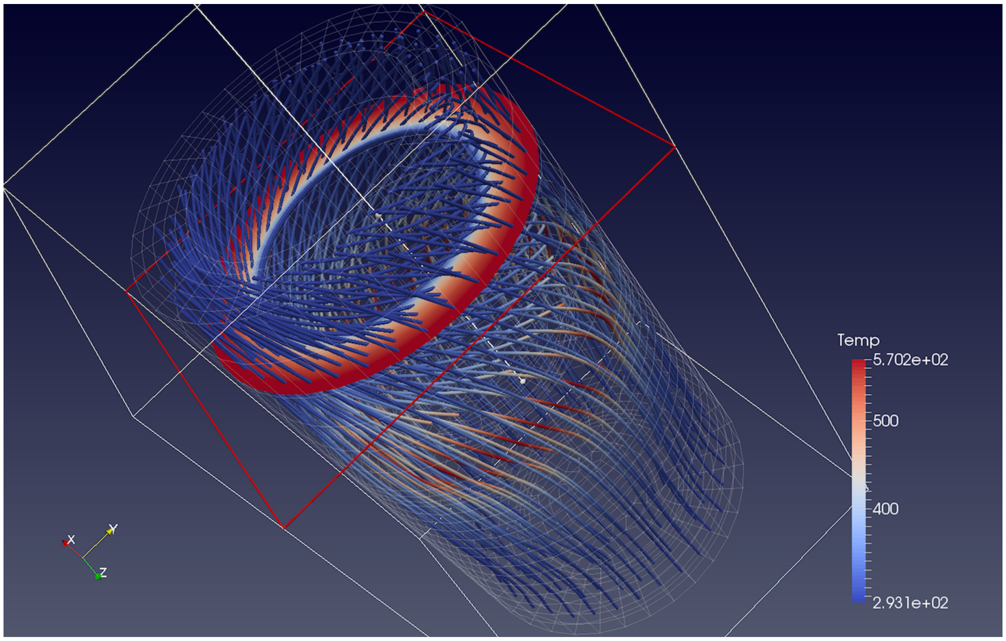


Fig. 1.91: Streamlines generated from the `disk_out_ref.ex2` dataset using the output of the `Slice` filter as the `Source` for seed points.

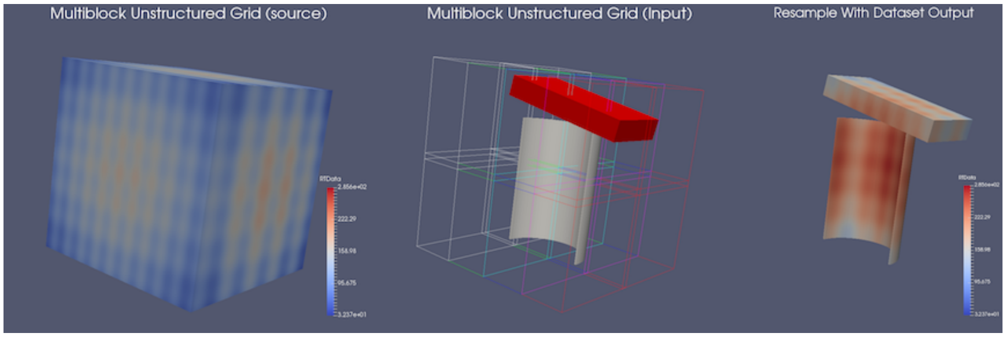
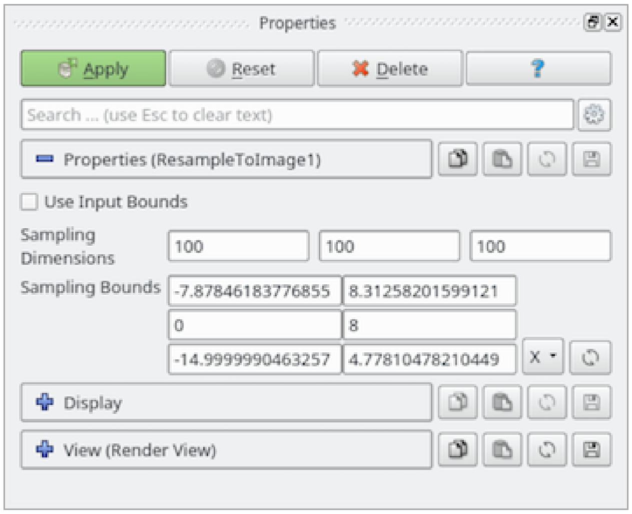

Multiblock Unstructured Grid (source)
Multiblock Unstructured Grid (Input)
Resample With Dataset Output
RTData
2.856e+02
222.29
158.98
95.675
3.237e+01

Fig. 1.92: An example of `Resample With Dataset` . On the left is a multiblock tetrahedra mesh ( `Input` ). The middle shows a multiblock unstructured grid ( `Source` ). The outline of `Input` is also shown in this view. The result of applying the filter is shown on the right


Properties
Apply
Reset
Delete
?
Search ... (use Esc to clear text)
Properties (ResampleToImage1)
Display
View (Render View)

☐ Use Input Bounds

Sampling Dimensions: 100 | 100 | 100

Sampling Bounds: -7.87846183776855 | 8.31258201599121
0 | 8
-14.9999990463257 | 4.77810478210449

Fig. 1.93: The `Properties` panel for `Resample To Image` filter.

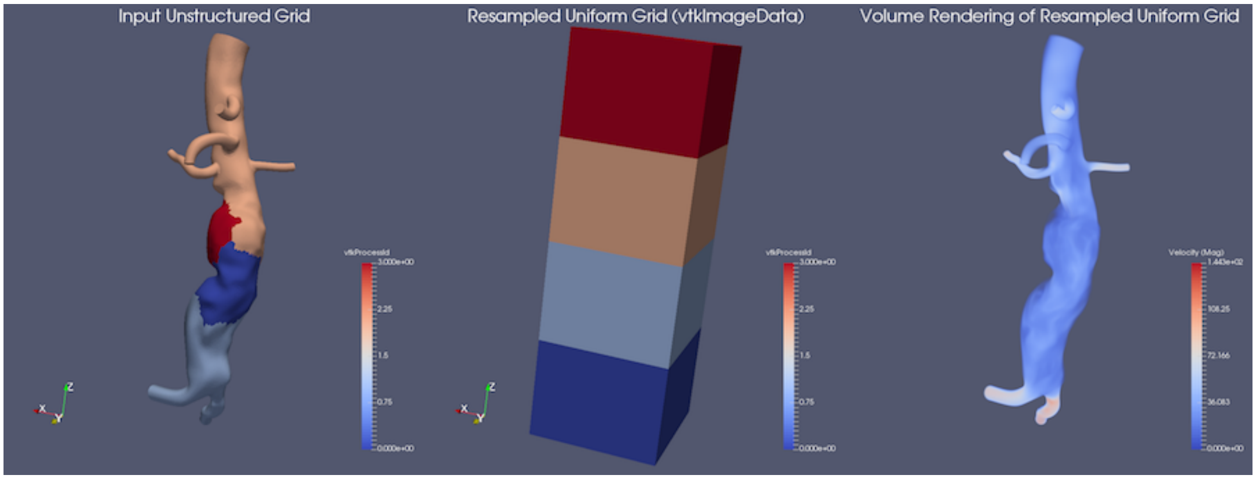


Fig. 1.94: An example of `Resample To Image` . The left portion shows the input (unstructured grid), and the middle displays the output image data. On the right is a volume rendering of the resampled data.

### Probe

Probe samples the input dataset at a specific point location to obtain the cell data attributes for the cell containing the point as well as the interpolated point data attributes. You can either use the SpreadSheet View or the Information panel to inspect the probed values. The probe location can be specified using the interactive 3D widget shown in the active Render View .

### Plot over line

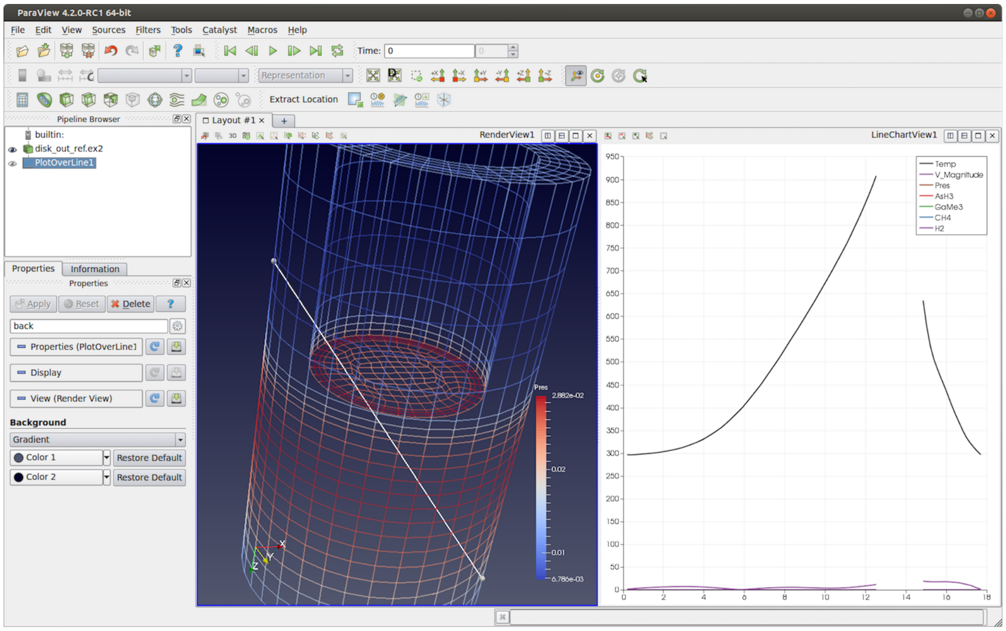


Fig. 1.95: The Plot Over Line filter applied to the disk_out_ref.ex2 dataset to plot values at sampled locations along the line. Gaps in the line correspond to the locations in the input dataset where the line falls outside the dataset.

Plot Over Line will sample the input dataset along the specified line and then plot the results in Line Chart View . Internally, this filter uses the same mechanism as the Probe filter, probing along the points in the line to get the containing cell attributes and interpolated point attributes.

Using the Resolution property on the Properties panel, you can control the number of sample points along the line.

### 1.5.9 Filters for attribute manipulation

The filters covered in this section are used to add new attribute arrays to the dataset, which are typically used to add derived quantities to use in pipelines for further processing.

#### Calculator

The `Calculator` filter computes a new data array or new point coordinates as a function of existing input arrays. If point-centered arrays are used in the computation of a new data array, the resulting array will also be point-centered. Similarly, computations using cell-centered arrays will produce a new cell-centered array. If the function is computing point coordinates (requested by checking the `Coordinate Results` property on the `Properties` panel) , the result of the function must be a three-component vector. The `Calculator` interface operates similarly to a scientific calculator. In creating the function to evaluate, the standard order of operations applies. Each of the calculator functions is described below. Unless otherwise noted, enclose the operand in parentheses using the `(` and `)` buttons.

- `Clear` : Erase the current function.
- `/`: Divide one scalar by another. The operands for this function are not required to be enclosed in parentheses.
- `*`: Multiply two scalars, or multiply a vector by a scalar (scalar multiple). The operands for this function are not required to be enclosed in parentheses.
- `-`: Negate a scalar or vector (unary minus), or subtract one scalar or vector from another. The operands for this function are not required to be enclosed in parentheses.
- `+`: Add two scalars or two vectors. The operands for this function are not required to be enclosed in parentheses.
- `iHat` , `jHat` , and `kHat` are vector constants representing unit vectors in the X, Y, and Z directions, respectively.
- `sin(x)` : Compute the sine of a scalar.
- `cos(x)` : Compute the cosine of a scalar.
- `tan(x)` : Compute the tangent of a scalar.
- `abs(x)` : Compute the absolute value of a scalar.
- `sqrt(x)` : Compute the square root of a scalar.
- `asin(x)` : Compute the arcsine of a scalar.
- `acos(x)` : Compute the arccosine of a scalar.
- `atan(x)` : Compute the arctangent of a scalar.
- `ceil(x)` : Compute the ceiling of a scalar.
- `floor(x)` : Compute the floor of a scalar.
- `sinh(x)` : Compute the hyperbolic sine of a scalar.
- `cosh(x)` : Compute the hyperbolic cosine of a scalar.
- `tanh(x)` : Compute the hyperbolic tangent of a scalar.
- `x^y` : Raise one scalar to the power of another scalar. The operands for this function are not required to be enclosed in parentheses.
- `exp(x)` Raise $e$‘ to the power of a scalar.
- `dot(x, y)` : Compute the dot product of two vectors x and y.
- `mag(x)` : Compute the magnitude of a vector.

- `norm(x)` : Normalize a vector. The operands are described below. The digits 0-9 and the decimal point are used to enter constant scalar values.
- `ln(x)` : Compute the logarithm of a scalar to the base $e$.
- `log10(x)` : Compute the logarithm of a scalar to the base 10.

Additional operations are available in the `Calculator` filter that do not have buttons in the user interface, including:

- `avg(x, y, z, ...)` : Average of all the input arguments.
- `clamp(r0, x, r1)` : Clamp x in range between r0 and r1.
- `cross(x, y)` : Compute cross product of two vectors x and y.
- `equal(x, y)` : Equality test between x and y using normalized epsilon.
- `erf(x)` : Error function of x.
- `erfc(x)` : Complimentary error function of x.
- `frac(x)` : Fractional portion of x.
- `hypot(x, y)` : Hypotenuse of x and y, equivalent of `sqrt(x*x + y*y)`.
- `iclamp(r0, x, r1)` : Inverse-clamp x outside of the range r0 and r1. If x is within the range it will snap to the closest bound.
- `inrange(r0, x, r1)` : Returns true when x is within the range r0 and r1.
- `log1p(x)` : Natural logarithm of 1 + x, where x is very small.
- `log2(x)` : Base 2 logarithm of x.
- `logn(x, n)` : Base N logarithm of x where n is a positive integer.
- `min(x, y)` : Compute minimum of two scalars.
- `max(x, y)` : Compute maximum of two scalars.
- `mul(z, y, z, ...)` : Multiply all the inputs together.
- `ncdf(x)` : Normal cumulative distribution function.
- `not_equal(x, y)` : Not-equal test between x and y using normalised epsilon.
- `pow(x, y)` : x to the power of y.
- `root(x, n)` : nth-root of x where n is a positive integer.
- `round(x)` : Round x to the nearest integer
- `roundn(x, n)` : Round x to n decimal places.
- `sgn(x)` : Compute the sign of x: -1 where x < 0, +1 where x > 0, and 0 otherwise.
- `sum(x, y, z, ...)` : Sum of all the inputs.
- `trunc(x)` : Integer portion of x.
- `acosh(x)` : Inverse hyperbolic cosine of x expressed in radians.
- `asinh(x)` : Inverse hyperbolic sine of x expressed in radians.
- `atan2(x, y)` : Arc tangent of (x / y) expressed in radians.
- `atanh(x)` : Inverse hyperbolic tangent of x expressed in radians.
- `cot(x)` : Cotangent of x.
- `csc(x)` : Cosecant of x.

- `sec(x)` : Secant of x.
- `sinc(x)` : Cardinal sine of x.
- `deg2rad(x)` : Convert x from degrees to radians.
- `deg2grad(x)` : Convert x from degrees to gradians.
- `rad2deg(x)` : Convert x from radians to degrees.
- `grad2deg(x)` : Convert x from gradians to degrees.

The following equalities and inequalities are available:

- == or = : True only if x is strictly equal to y.
- <> or `!=` : True only if x does not equal y.
- < : True only if x is less than y.
- <= : True only if x is less than or equal to y.
- > : True only if x is greater than y.
- >= : True only if x is greater than or equal to y.

The following conditionals and boolean operators are available:

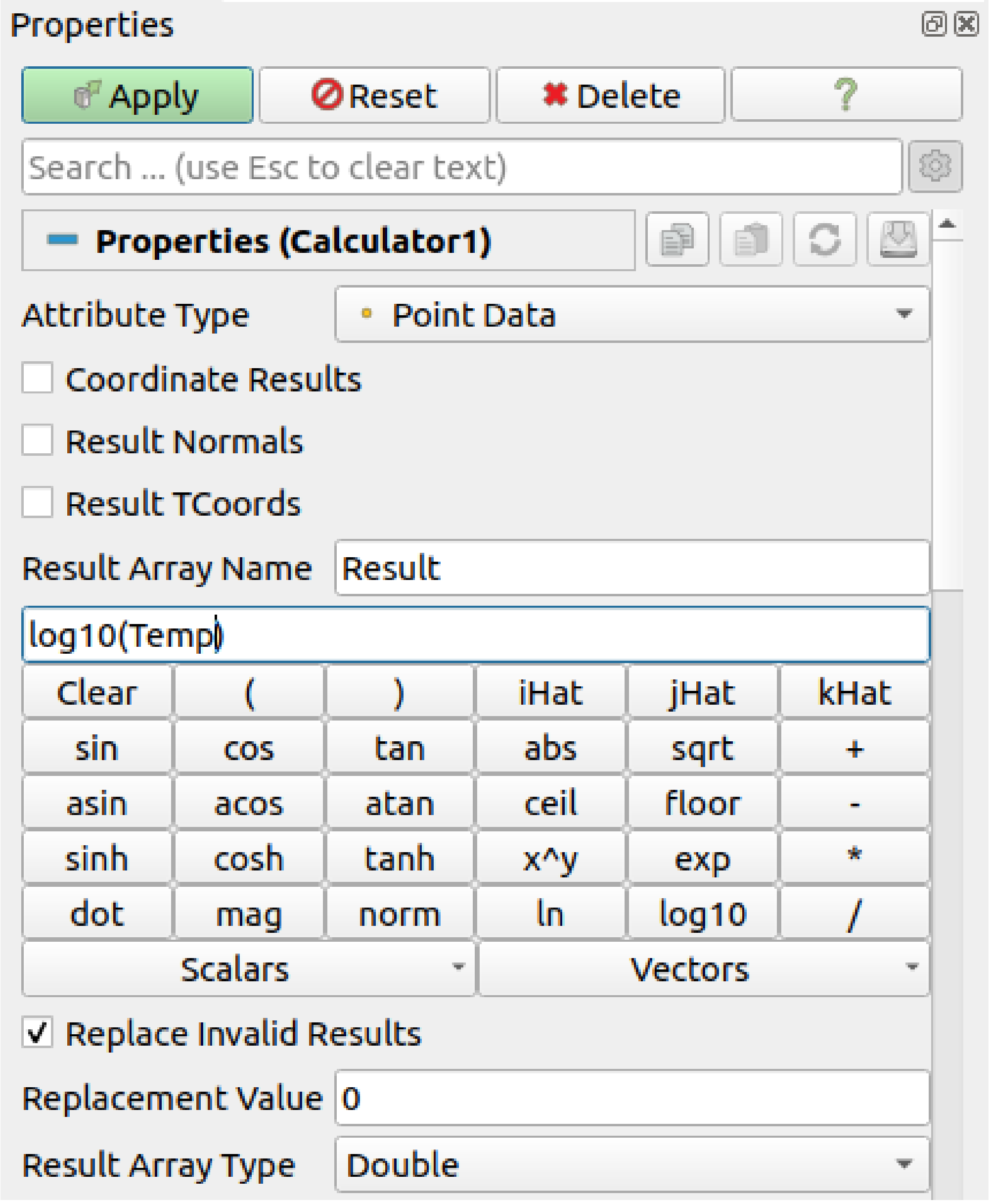
- `if(x, y, z)` : If x evaluates to true, then y, otherwise z.
- `true` : True state.
- `false` : False state.
- `x and y` : Logical and, true only if x and y are both true.
- `mand(x, y, z, ...)` : Multi-input logical and, true only if all arguments are true.
- `mor(x, y, z, ...)` : Multi-input logical or, true if any arguments are true.
- `x nand y` : Logical nand, true only if either x or y is false.
- `x nor y` : Logical nor, true only if neither x or y is false.
- `not x` : Logical not, evaluate to the opposite of the input boolean value.
- `x or y` : Logical or, true if either x or y is true.
- `x xor y` : Logical xor, true only if the logical state of x or y are different.
- `x xnor y` : True if and only if both logical inputs are the same.

The `Scalars` menu lists the names of the scalar arrays and the components of the vector arrays of either the point-centered or cell-centered data. The `Vectors` menu lists the names of the point-centered or cell-centered vector arrays. The function will be computed for each point (or cell) using the scalar or vector value of the array at that point (or cell). The filter operates on any type of dataset, but the input dataset must have at least one scalar or vector array. The arrays can be either point-centered or cell-centered. The `Calculator` filter's output is of the same dataset type as the input.

**Did you know?**

It used to be a common use-case for the `Calculator` filter to convert three input scalars into a vector array. For that, the `Function` would look something like: $scalar_x * iHat + scalar_y * jHat + scalar_z * kHat$.

Now, the **Merge Vector Components** filter provides a simpler way to do this by simply selecting the three scalars to combine into a vector array.



Attribute Type: Point Data

□ Coordinate Results

□ Result Normals

□ Result TCoords

Result Array Name: Result

log10(Temp)

☑ Replace Invalid Results

Replacement Value: 0

Result Array Type: Double

Fig. 1.96: The `Properties` panel for the `Calculator` filter showing the advanced properties.

The Properties panel provides access to several options for this filter. Checking Coordinate Results , Result Normals , or Result TCoords will set the computed array as the point coordinates, normals, or texture coordinates, respectively. Result Array Name is used to specify a name for the computed array. The default is Result.

Sometimes, the expression can yield invalid values. To replace all invalid values with a specific value, check the Replace Invalid Results checkbox and then enter the value to use to replace invalid values using the Replacement Value . The output array data type is set with the Result Array Type property.

To ease the reuse of expressions, three helper buttons are also present to load an expression, save the current one and inspect the list of already saved expressions from the Expression Manager.

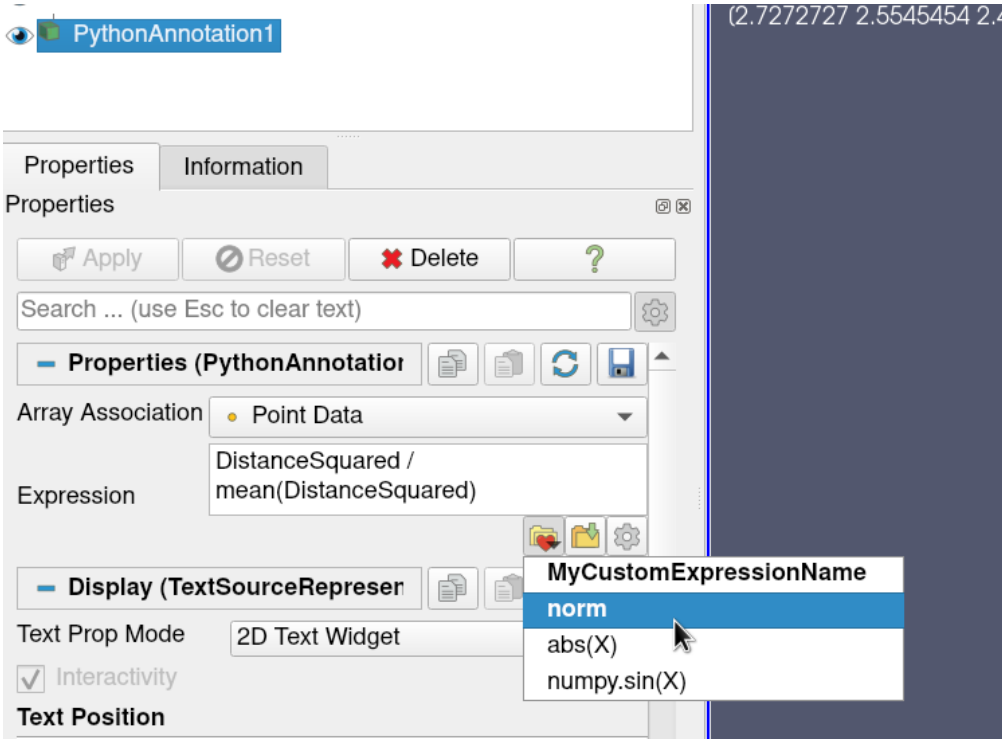
### Expression Manager

ParaView provides an Expression Manager to ease the expression property configuration by storing expressions, and giving quick access to them. Each expression can be named and has an associated group so it is easy to filter Python expressions from others.

This feature comes in two parts:

From the Property Panel, the one-line property text entry is augmented with:

- a drop down list to access existing expressions
- a Save Current Expression button
- a shortcut to the Choose Expression dialog



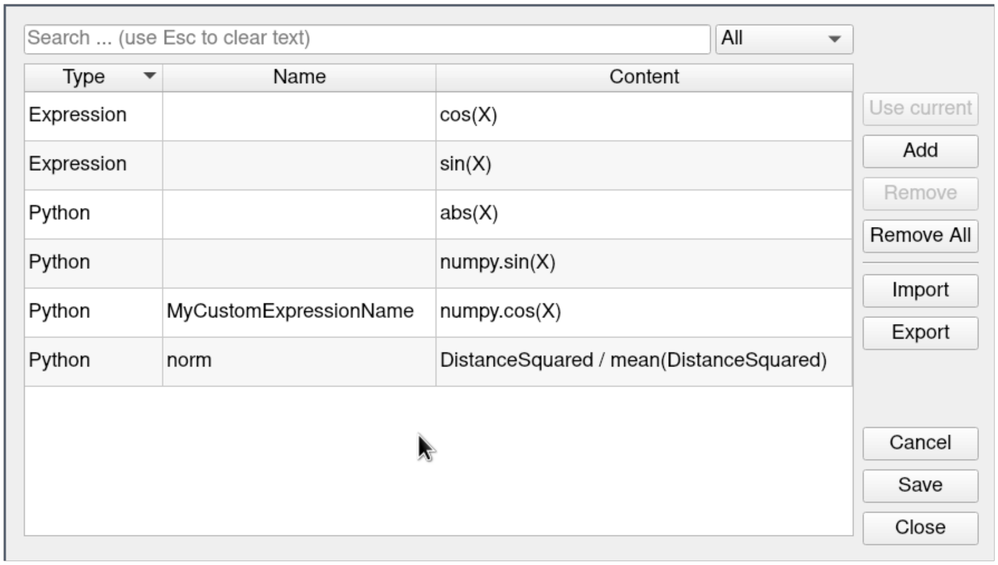
Array Association: Point Data

Expression: DistanceSquared / mean(DistanceSquared)

Text Prop Mode: 2D Text Widget

Fig. 1.97: Expression-related buttons in the Properties Panel.

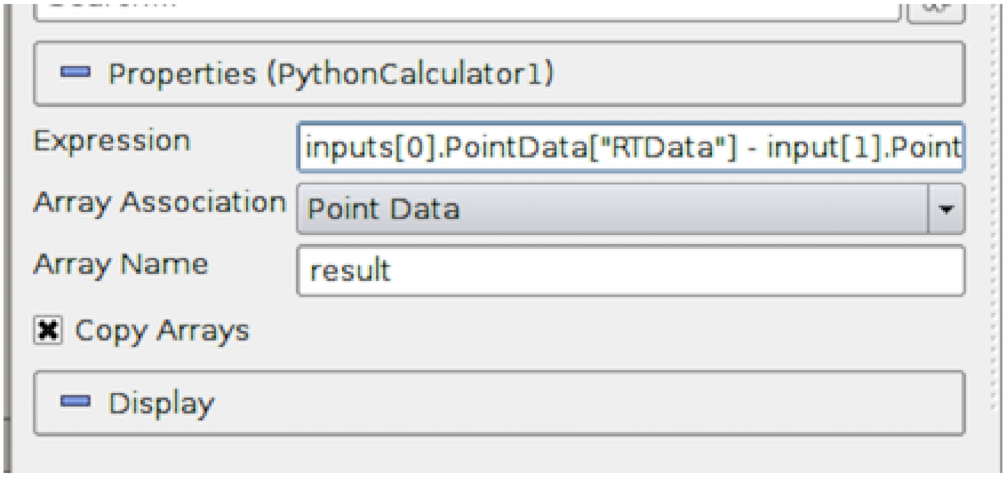
The Choose Expression dialog, also accessible from the *Tools* > *Manage Expressions* menu item, is an editable and searchable list of the stored expressions. ParaView keeps track of them through the settings, but they can also be exported to a JSON file for backup and sharing.



| Type | Name | Content |
|---|---|---|
| Expression | | cos(X) |
| Expression | | sin(X) |
| Python | | abs(X) |
| Python | | numpy.sin(X) |
| Python | MyCustomExpressionName | numpy.cos(X) |
| Python | norm | DistanceSquared / mean(DistanceSquared) |

Fig. 1.98: Choose Expression Dialog.

### Python calculator



Expression: inputs[0].PointData["RTData"] - input[1].Point

Array Association: Point Data

Array Name: result

☒ Copy Arrays

Fig. 1.99: The Properties Panel for `Python Calculator`.

The `Python Calculator` is similar to `Calculator` in that it processes one or more input arrays based on an expression provided by the user to produce a new output array. However, it uses Python (and *NumPy*) to do the computation. Therefore, it provides more expressive computational capabilities.

Specify whether to `Use Multiline Expression`, the `Expression` to use, the `Array Association` to indicate the array association (`Point Data`, `Cell Data`, or `Field Data`), the name of output array (`Array Name`), a toggle that controls whether the input arrays are copied to the output (`Copy Array`), and a `Result Array Type` to specify the type of output data array to store the calculated results in.

The `Python Calculator` also integrated the `Expression Manager` described in Section 1.5.9.

## Basic tutorial

Start by creating a Sphere source and applying the `Python Calculator` to it. As the first expression, use the following and apply:

```
5
```

This should create an array name *result* in the output point data. Note that this is an array that has a value of 5 for each point. When the expression results in a single value, the calculator will automatically make a constant array. Next, try the following:

```
Normals
```

Now, the *result* array should be the same as the input array Normals. As described in detail later, various functions are available through the calculator. For example, the following is a valid expression:

```
sin(Normals) + 5
```

It is very important to note that the `Python Calculator` has to produce one value per point or cell depending on the Array Association parameter. Most of the functions described here apply individually to all point or cell values and produce an array the same dimensions as the input. However, some of them, such as `min()` and `max()` , produce single values.

**Common Errors**

In the `Programmable Filter` , all the functions in `vtk.numpy_interface.algorithms` are imported prior to executing the script. As a result, some built-in functions, such as `min` and `max` , are clobbered by that import. To use the built-in functions, import the `import __builtin__` module and access those functions with, e.g., `__builtin__.min` and `__builtin__.max`

## Accessing data

There are several ways of accessing input arrays within expressions. The simplest way is to access it by name:

```
sin(Normals) + 5
```

This is equivalent to:

```
sin(inputs[0].PointData['Normals']) + 5
```

The example above requires some explanation. Here, `inputs[0]` refer to the first input (dataset) to the filter. `Python Calculator` can accept multiple inputs. Each input can be accessed as `inputs[0]` , `inputs[1]` , … You can access the point or cell data of an input using the `.PointData` or `.CellData` qualifiers. You can then access individual arrays within the point or cell data containers using the `[]` operator. Make sure to use quotes or double-quotes around the array name. Arrays that have names with certain characters (such as space, +, -, *, /) can only be accessed using this method.

Certain functions apply directly on the input mesh. These filters expect an input dataset as argument. For example,

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

For data types that explicitly define the point coordinates, you can access the coordinates array using the `.Points` qualifier. The following extracts the first component of the coordinates array:

```
inputs[0].Points[:,0]
```

Note that for certain data types, mainly image data (uniform rectilinear grids) and rectilinear grids, point coordinates are defined implicitly and cannot be accessed as an array.

### Comparing multiple datasets

The `Python Calculator` can be used to compare multiple datasets, as shown by the following example.

- Go to the Menu Bar, and select *Edit* > *Reset Session* to clear the Pipeline.
- Select *Source* > *Mandelbrot*, and then click Apply, which will set up a default version of the Mandelbrot Set. The data for this set are stored in a $251 \times 251$ scalar array.
- Select *Source* > *Mandelbrot* again, and then go to the `Properties` panel and set the Maximum Number of Iterations to 50. Click `Apply` , which will set up a different version of the Mandelbrot Set, represented by the same size array.
- Hold the Shift key down and select both of the Mandelbrot entries in the Pipeline Inspector, and then go to the Menu Bar, and select *Filter* > *Python Calculator*. The two Mandelbrot entries will now be shown as linked, as inputs, to the `Python Calculator` .
- In the Properties panel for the Python Calculator filter, enter the following into the Expression box:

  ```
  inputs[1].PointData['Iterations'] - inputs[0].PointData['Iterations']
  ```

  This expression specifies the difference between the second and the first Mandelbrot arrays. The result is saved in a new array called `results` . The prefixes in the names for the array variables, `inputs[1]` and `inputs[0]` , refer to the first and second Mandelbrot entries, respectively, in the Pipeline. `PointData` specifies that the inputs contain point values. The quoted label `'Iterations'` is the local name for these arrays. Click `Apply` to initiate the calculation.

Click the `Display` tab in the `Properties Panel` for the `Python Calculator` , and go to the first tab to the right of the *Color by* label. Select the item results in that tab, which will cause the display window to the right to show the results of the expression we entered in the `Python Calculator` . The scalar values representing the difference between the two Mandelbrot arrays are represented by colors that are set by the current color map (click the *Edit* button to open a detailed editor for the current color map).

There are a few things to note:

- `Python Calculator` will always copy the mesh from the first input to its output.
- All operations are applied point-by-point. In most cases, this requires that the input meshes (topology and geometry) are the same. At the least, it requires that the inputs have the same number of points and cells.
- In parallel execution mode, the inputs have to be distributed exactly the same way across processes.

### Basic Operations

The `Python Calculator` supports all of the basic arithmetic operations using the $+$, $-$, $*$ and $/$ operators. These are always applied element-by-element to point and cell data including scalars, vectors, and tensors. These operations also work with single values. For example, the following adds 5 to all components of all Normals.

```
Normals + 5
```

The following adds 1 to the first component, 2 to the second component, and 3 to the third component:

```
Normals + [1,2,3]
```

This is specially useful when mixing functions that return single values. For example, the following normalizes the Normals array:

```
(Normals - min(Normals))/(max(Normals) - min(Normals))
```

A common use case in a calculator is to work on one component of an array. This can be accomplished with the following:

```
Normals[:, 0]
```

The expression above extracts the first component of the Normals vector. Here, `:` is a placeholder for “all elements”. One element can be extracted by replacing `:` with an index. For example, the following creates a constant array from the first component of the normal of the first point:

```
Normals[0, 0]
```

Alternatively, the following assigns the normal of the first point to all points:

```
Normals[0, :]
```

It is also possible to create a vector array from two or three scalar arrays using the `make_vector()` function:

```
make_vector(velocity_x, velocity_y, velocity_z)
```

For temporal datasets, you also have access to the dataset timestep index or time value in the expression as `t_index` or `time_index` , and `t_value` or `time_value` respectively. When dealing with multiple inputs, you can specify the same variable names scoped on the appropriate input e.g. `inputs[0].t_index` .

The locations of points are available in the `Points` variable for datasets that define explicit points positions.

In some datasets, field data is used to store global data values not associated with cells or points. To use field data in a `Python Calculator` expression, access it with the `FieldData` dictionary available in the input as in the following example:

```
VolumeOfCell * inputs[0].FieldData['MaterialData'][time_index]
```

**Did you know?**

Under the cover, the `Python Calculator` uses NumPy. All arrays in the expression are compatible with NumPy arrays and can be used where NumPy arrays can be used. For more information on what you can do with these arrays, consult with the NumPy references [NumPydevelopers].

### Functions

The following is a list of functions available in the `Python Calculator`. Note that this is a partial list, since most of the NumPy and SciPy functions can be used in the `Python Calculator`. Many of these functions can take single values or arrays as argument.

- `abs(x)` : Returns the absolute value(s) of $x$.
- `add(x, y)` : Returns the sum of two values. $x$ and $y$ can be single values or arrays. This is the same as $x + y$.
- `area(dataset)` : Returns the surface area of each cell in a mesh.
- `aspect(dataset)` : Returns the aspect ratio of each cell in a mesh.
- `aspect_gamma(dataset)` : Returns the aspect ratio gamma of each cell in a mesh.
- `condition(dataset)` : Returns the condition number of each cell in a mesh.
- `cross(x, y)` : Returns the cross product for two 3D vectors from two arrays of 3D vectors.
- `curl(array)` : Returns the curl of an array of 3D vectors.
- `divergence(array)` : Returns the divergence of an array of 3D vectors.
- `divide(x, y)` : Element-by-element division. $x$ and $y$ can be single values or arrays. This is the same as math:*frac{x}{y}*.
- `det(array)` : Returns the determinant of an array of 2D square matrices.
- `determinant(array)` : Returns the determinant of an array of 2D square matrices.
- `diagonal(dataset)` : Returns the diagonal length of each cell in a dataset.
- `dot(a1, a2)` : Returns the dot product of two scalars/vectors of two array of scalars/vectors.
- `eigenvalue(array)` : Returns the eigenvalue of an array of 2D square matrices.
- `eigenvector(array)` : Returns the eigenvector of an array of 2D square matrices.
- `exp(x)` : Returns $e^x$.
- `gradient(array)` : Returns the gradient of an array of scalars or vectors.
- `inv(array)` : Returns the inverse an array of 2D square matrices.
- `inverse(array)` : Returns the inverse of an array of 2D square matrices.
- `jacobian(dataset)` : Returns the jacobian of an array of 2D square matrices.
- `laplacian(array)` : Returns the jacobian of an array of scalars.
- `ln(array)` : Returns the natural logarithm of an array of scalars/vectors/tensors.
- `log(array)` : Returns the natural logarithm of an array of scalars/vectors/tensors.
- `log10(array)` : Returns the base 10 logarithm of an array of scalars/vectors/tensors.
- `make_point_mask_from_NaNs(dataset, array)` : This function will create a ghost array corresponding to an input with NaN values. For each NaN value, the output array will have a corresponding value of `vtk.vtkDataSetAttributes.HIDDENPOINT`. These values are also combined with any ghost values that the dataset may have.
- `make_cell_mask_from_NaNs(dataset, array)` : This function will create a ghost array corresponding to an input with NaN values. For each NaN value, the output array will have a corresponding value of `vtk.vtkDataSetAttributes.HIDDENCELL`. These values are also combined with any ghost values that the dataset may have.

- `max(array)` : Returns the maximum value of the array as a single value. In parallel, compute the max accross processes.
- `max_angle(dataset)` : Returns the maximum angle of each cell in a dataset.
- `mag(a)` : Returns the magnitude of an array of scalars/vectors.
- `mean(array)` : Returns the mean value of an array of scalars/vectors/tensors. In parallel, compute the mean accross processes.
- `min(array)` : Returns the minimum value of the array as a single value. In parallel, compute the min accorss processes.
- `min_angle(dataset)` : Returns the minimum angle of each cell in a dataset.
- `mod(x, y)` : Same as remainder $(x, y)$.
- `multiply(x, y)` : Returns the product of $x$ and $y$. $x$ and $y$ can be single values or arrays. Note that this is an element-by-element operation when $x$ and $y$ are both arrays. This is the same as $x \times y$.
- `negative(x)` : Same as $-x$.
- `norm(a)` : Returns the normalized values of an array of scalars/vectors.
- `power(x, a)` : Exponentiation of $x$ with $a$. Here, both $x$ and $a$ can either be a single value or an array. If $x$ and $y$ are both arrays, a one-by-one mapping is used between two arrays.
- `reciprocal(x)` : Returns $\frac{1}{x}$.
- `remainder(x, y)` : Returns $x - y \times floor(\frac{x}{y})$. $x$ and $y$ can be single values or arrays.
- `rint(x)` : Rounds $x$ to the nearest integer(s).
- `shear(dataset)` : Returns the shear of each cell in a dataset.
- `skew(dataset)` : Returns the skew of each cell in a dataset.
- `square(x)` : Returns $x * x$.
- `sqrt(x)` : Returns $\sqrt[2]{x}$.
- `strain(array)` : Returns the strain of an array of 3D vectors.
- `subtract(x, y)` : Returns the difference between two values. $x$ and $y$ can be single values or arrays. This is the same as $x - y$.
- `surface_normal(dataset)` : Returns the surface normal of each cell in a dataset.
- `trace(array)` : Returns the trace of an array of 2D square matrices.
- `volume(dataset)` : Returns the volume normal of each cell in a dataset.
- `vorticity(array)` : Returns the vorticity/curl of an array of 3D vectors.
- `vertex_normal(dataset)` : Returns the vertex normal of each point in a dataset.

### Trigonometric Functions

Below is a list of supported trigonometric functions:

| | | | |
|---|---|---|---|
| `sin(x)` | `arccos(x)` | `cosh(x)` | `arctanh(x)` |
| `cos(x)` | `arctan(x)` | `tanh(x)` | |
| `tan(x)` | `hypot(x1, x2)` | `arcsinh(x)` | |
| `arcsin(x)` | `sinh(x)` | `arccosh(x)` | |

### Gradient

The `Gradient` filter computes the gradient of a cell or point data array for any type of dataset.

For unstructured grids, the gradient for cell data corresponds to the cell derivatives. For point data, the gradient at a given point is computed as the average of the derivatives of the cells to which the point belongs.

For structured grids, the gradient is computed using central differencing, except on the boundary of the dataset where forward and backward differencing is used for the boundary elements.

This filter can optionally compute the divergence, vorticity (also known as the curl), and Q-criterion. A 3-component array is required in order to compute these quantities. By default, only the gradient computation is enabled.

In the case of a uniform rectilinear grid (see Section 1.3.1), a specific implementation which efficiently computes the gradient of point data arrays is also available. This implementation extends the use of central differencing on the boundary elements after duplication of the boundary values. To activate this option, set the `Boundary Method` property to `Smoothed`, as shown in Fig. 1.100.

### Mesh Quality

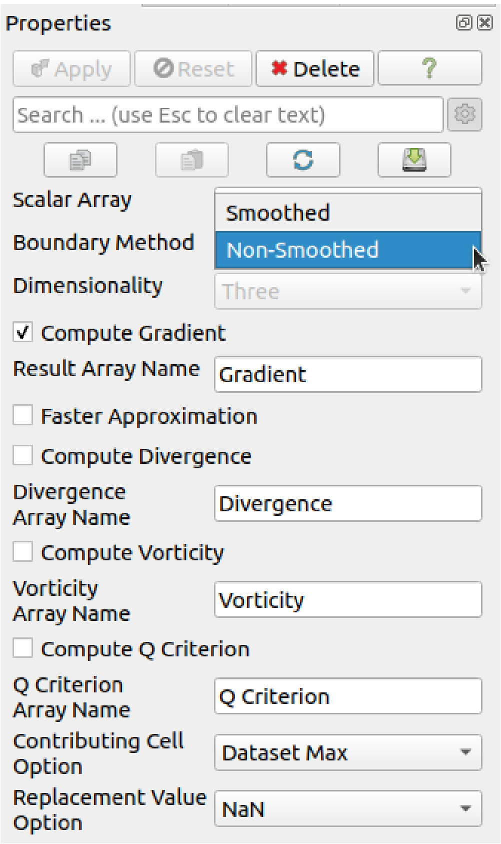
The `Mesh Quality` filter creates a new cell array containing a geometric measure of each cell's fitness. Different quality measures can be chosen for different cell shapes.

`Triangle Quality` indicates which quality measure will be used to evaluate triangle quality. The `Radius Ratio` is the size of a circle circumscribed by a triangle's three vertices divided by the size of a circle tangent to a triangle's three edges. The `Edge Ratio` is the ratio of the longest edge length to the shortest edge length.

`Quad Quality` indicates which quality measure will be used to evaluate quad cells.

`Tet Quality` indicates which quality measure will be used to evaluate tetrahedral quality. The `Radius Ratio` is the size of a sphere circumscribed by a tetrahedron's four vertices divided by the size of a circle tangent to a tetrahedron's four faces. The `Edge Ratio` is the ratio of the longest edge length to the shortest edge length. The `Collapse Ratio` is the minimum ratio of height of a vertex above the triangle opposite it, divided by the longest edge of the opposing triangle across all vertex/triangle pairs.

`HexQualityMeasure` indicates which quality measure will be used to evaluate quality of hexahedral cells.



Scalar Array

Boundary Method: Smoothed / Non-Smoothed

Dimensionality: Three

☑ Compute Gradient

Result Array Name: Gradient

☐ Faster Approximation

☐ Compute Divergence

Divergence Array Name: Divergence

☐ Compute Vorticity

Vorticity Array Name: Vorticity

☐ Compute Q Criterion

Q Criterion Array Name: Q Criterion

Contributing Cell Option: Dataset Max

Replacement Value Option: NaN

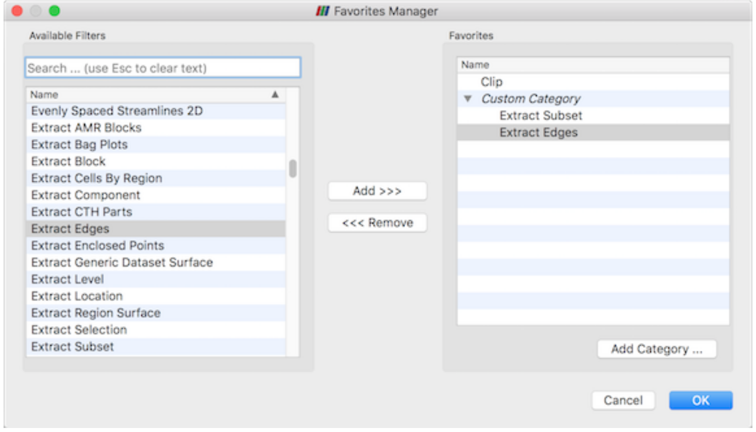
Fig. 1.100: The Properties Panel for the `Gradient` filter applied to a uniform structured grid.

## 1.5.10 White-box filters

This includes the `Programmable Filter` and `Programmable Source`. For these filters/sources, you can add Python code to do the data generation or processing. We'll cover writing Python code for these in Section 2.5.

## 1.5.11 Favorite filters

If you use some filters more than others, you can organize them in the *Filters* > *Favorites* menu. This can be done from the context menu in the pipeline or through the *Filters* > *Manage Favorites* menu as shown in Fig. 1.101. In this dialog you can create categories and subcategories. It supports drag'n'drop operation to sort and move filters and categories. Moreover, `Favorites` are highlighted in the other filter submenus on supported platforms. Favorites are saved in user settings so they can be used in other subsequent ParaView sessions.



Name
Evenly Spaced Streamlines 2D
Extract AMR Blocks
Extract Bag Plots
Extract Block
Extract Cells By Region
Extract Component
Extract CTH Parts
Extract Edges
Extract Enclosed Points
Extract Generic Dataset Surface
Extract Level
Extract Location
Extract Region Surface
Extract Selection
Extract Subset

Name
Clip
▼ *Custom Category*
Extract Subset
Extract Edges

Fig. 1.101: The Favorites Manager dialog. Left: the list of available filters. Right: the favorites, organized into categories.

## 1.5.12 Best practices

### Avoiding data explosion

The pipeline model that **ParaView** presents is very convenient for exploratory visualization. The loose coupling between components provides a very flexible framework for building unique visualizations, and the pipeline structure allows you to tweak parameters quickly and easily.

The downside of this coupling is that it can have a larger memory footprint. Each stage of this pipeline maintains its own copy of the data. Whenever possible, **ParaView** performs shallow copies of the data so that different stages of the pipeline point to the same block of data in memory. However, any filter that creates new data or changes the values or topology of the data must allocate new memory for the result. If **ParaView** is filtering a very large mesh, inappropriate use of filters can quickly deplete all available memory. Therefore, when visualizing large datasets, it is important to understand the memory requirements of filters.

Please keep in mind that the following advice is intended only for when dealing with very large amounts of data and the remaining available memory is low. When you are not in danger of running out of memory, the following advice is not relevant.

When dealing with structured data, it is absolutely important to know what filters will change the data to unstructured. Unstructured data has a much higher memory footprint, per cell, than structured data because the topology must be explicitly written out. There are many filters in **ParaView** that will change the topology in some way, and these filters will write out the data as an unstructured grid, because that is the only dataset that will handle any type of topology that is generated. The following list of filters will write out a new unstructured topology in its output that is roughly equivalent to the input. These filters should never be used with structured data and should be used with caution on unstructured data.

- *Append Datasets*
- *Append Geometry*
- *Clean*
- *Clean to Grid*
- *Connectivity*
- *D3*
- *Delaunay 2D/3D*
- *Extract Edges*
- *Linear Extrusion*
- *Loop Subdivision*
- *Reflect*
- *Rotational Extrusion*
- *Shrink*
- *Smooth*
- *Subdivide*
- *Tessellate*
- *Tetrahedralize*
- *Triangle Strips*
- *Triangulate*

Technically, the `Ribbon` and `Tube` filters should fall into this list. However, as they only work on 1D cells in poly data, the input data is usually small and of little concern.

This similar set of filters also outputs unstructured grids, but also tends to reduce some of this data. Be aware though that this data reduction is often smaller than the overhead of converting to unstructured data. Also note that the reduction is often not well balanced. It is possible (often likely) that a single process may not lose any cells. Thus, these filters should be used with caution on unstructured data and extreme caution on structured data.

- *Clip*
- *Decimate*
- *Extract Cells by Region*
- *Extract Selection*
- *Quadric Clustering*
- *Threshold*

Similar to the items in the preceding list, `Extract Subset` performs data reduction on a structured dataset, but also outputs a structured dataset. So the warning about creating new data still applies, but you do not have to worry about converting to an unstructured grid.

This next set of filters also outputs unstructured data, but it also performs a reduction on the dimension of the data (for example 3D to 2D), which results in a much smaller output. Thus, these filters are usually safe to use with unstructured data and require only mild caution with structured data.

- *Cell Centers*
- *Contour*
- *Extract CTH Parts*
- *Extract Surface*
- *Feature Edges*
- *Mask Points*
- *Outline Curvilinear DataSet*
- *Slice*
- *Stream Tracer*

The filters below do not change the connectivity of the data at all. Instead, they only add field arrays to the data. All the existing data is shallow copied. These filters are usually safe to use on all data.

- *Block Ids*
- *Calculator*
- *Cell Data to Point Data*
- *Curvature*
- *Elevation*
- *Surface Normals*
- *Gradient*
- *Level Scalars*
- *Median*
- *Mesh Quality*
- *Point Data to Cell Data*
- *Process Ids*
- *Random Vectors*
- *Resample with Dataset*
- *Surface Flow*
- *Surface Vectors*
- *Texture Map to…*
- *Transform*
- *Warp By Scalar*
- *Warp By Vector*

This final set of filters either add no data to the output (all data of consequence is shallow copied) or the data they add is generally independent of the size of the input. These are almost always safe to add under any circumstances (although they may take a lot of time).

| | |
|---|---|
| *Annotate Time* | *Outline Corners* |
| *Append Attributes* | *Plot Global Variables Over Time* |
| *Extract Block* | *Plot Over Line* |
| *Glyph* | *Plot Selection Over Time* |
| *Group Datasets* | *Probe Location* |
| *Histogram* | *Temporal Shift Scale* |
| *Integrate Variables* | *Temporal Snap-to-Time-Steps* |
| *Normal Glyphs* | *Temporal Statistics* |
| *Outline* | |

There are a few special case filters that do not fit well into any of the previous classes. Some of the filters, currently `Temporal Interpolator` and `Particle Tracer` , perform calculations based on how data changes over time. Thus, these filters may need to load data for two or more instances of time, which can double or more the amount of data needed in memory. The `Temporal Cache` filter will also hold data for multiple instances of time. Keep in mind that some of the temporal filters such as the Temporal Statistics and the filters that plot over time may need to iteratively load all data from disk. Thus, it may take an impractically long amount of time even if does not require any extra memory.

The `Programmable Filter` is also a special case that is impossible to classify. Since this filter does whatever it is programmed to do, it can fall into any one of these categories.

### Culling data

When dealing with large data, it is best to cull out data whenever possible and do so as early as possible. Most large data starts as 3D geometry and the desired geometry is often a surface. As surfaces usually have a much smaller memory footprint than the volumes that they are derived from, it is best to convert to a surface early on. Once you do that, you can apply other filters in relative safety.

A very common visualization operation is to extract isosurfaces from a volume using the Contour filter. The `Contour` filter usually outputs geometry much smaller than its input. Thus, the `Contour` filter should be applied early if it is to be used at all. Be careful when setting up the parameters to the `Contour` filter because it still is possible for it to generate a lot of data which can happen if you specify many isosurface values. High frequencies such as noise around an isosurface value can also cause a large, irregular surface to form.

Another way to peer inside of a volume is to perform a `Slice` on it. The `Slice` filter will intersect a volume with a plane and allow you to see the data in the volume where the plane intersects. If you know the relative location of an interesting feature in your large dataset, slicing is a good way to view it.

If you have little *a priori* knowledge of your data and would like to explore the data without the long memory and processing time for the full dataset, you can use the `Extract Subset` filter to subsample the data. The subsampled data can be dramatically smaller than the original data and should still be well load balanced. Of course, be aware that you may miss small features if the subsampling steps over them and that once you find a feature you should go back and visualize it with the full dataset.

There are also several features that can pull out a subset of a volume: `Clip` , `Threshold` , `Extract Selection` , and `Extract Subset` can all extract cells based on some criterion. Be aware, however, that the extracted cells are almost never well balanced; expect some processes to have no cells removed. All of these filters, with the exception of `Extract Subset` , will convert structured data types to unstructured grids. Therefore, they should not be used unless the extracted cells are of at least an order of magnitude less than the source data.

When possible, replace the use of a filter that extracts 3D data with one that will extract 2D surfaces. For example, if you are interested in a plane through the data, use the `Slice` filter rather than the `Clip` filter. If you are interested in knowing the location of a region of cells containing a particular range of values, consider using the `Contour` filter to generate surfaces at the ends of the range rather than extract all of the cells with the `Threshold` filter. Be aware that substituting filters can have an effect on downstream filters. For example, running the `Histogram` filter after `Threshold` will have an entirely different effect than running it after the roughly equivalent `Contour` filter.

## 1.6 Selecting Data

A typical visualization process has two components: setting up the visualization scene and performing the analysis of the results to gain insight. It is not uncommon for this process to be iterative. Often, what you are looking for drives from what filters you should use to extract the relevant information from the input datasets and what views will best represent that data. One of the ways of evaluating the results is inspecting the data or probing into it by identifying elements of interest. **ParaView** data selection mechanisms are designed specifically for such use-cases. In this chapter, we take a closer look at various ways of selecting data in **ParaView** and making use of these selections for data analysis.

### 1.6.1 Understanding selection

Broadly speaking, selection refers to selecting elements (either cells, points, table rows, etc.) from datasets. Since data is ingested into **ParaView** using readers or sources and transformed using filters, when you create a selection, you are selecting elements from the dataset produced as the output of source, filter, or any such pipeline module.

There are many ways to create selections. Several views provide means to create specific selections. For example, in the `SpreadSheet View` , which shows the data attributes as a spreadsheet, you can simply click on any row to *select* that row. You can, of course, use the  and `CTRL` (or ) keys to select multiple rows, as in typical spreadsheet-based applications.

While this seems like an exercise in futility, you are hardly achieving anything by highlighting rows in a spreadsheet. What transforms this into a key tool is the fact that selections are *linked* among views (whenever possible). Linked selection means that you select an element from a dataset in a specific view. All other views that are showing the *same* dataset will also highlight the selected elements.

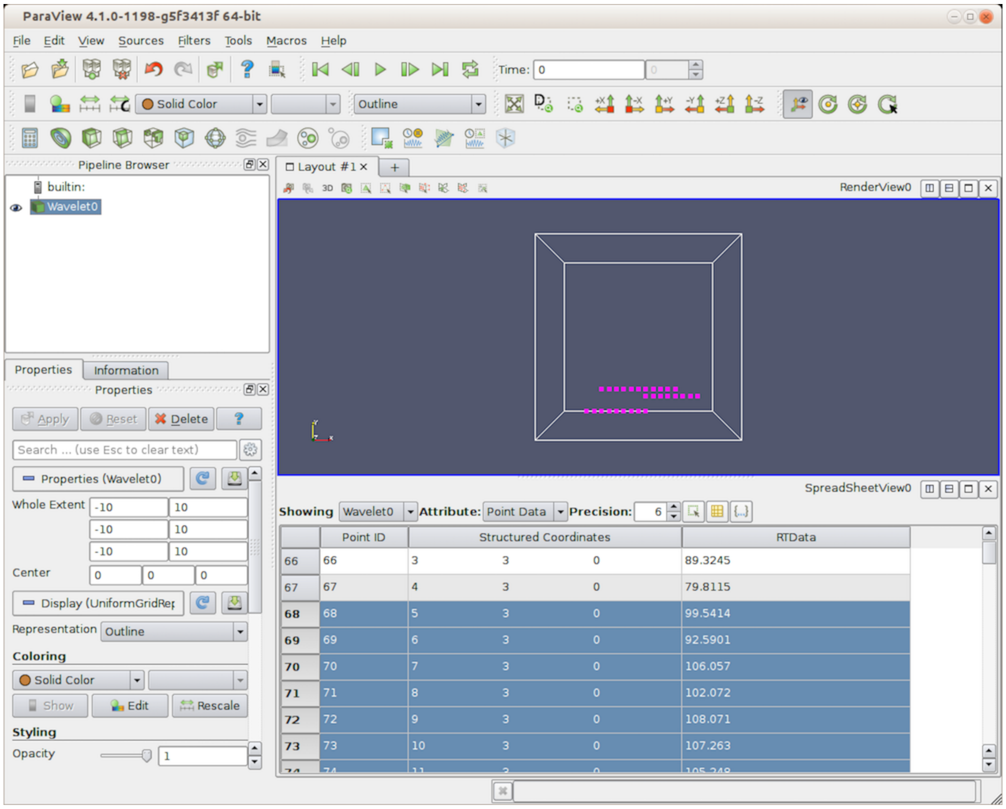
To make this easier, let's try a quick demo:

Starting with a fresh `paraview` session, create a sample dataset using the *Sources* > *Alphabetical* > *Wavelet* menu, and then click the `Apply` button. If you are using `paraview` with a default setup, that should result in a dataset outline being shown in the default `Render View` . Next, let's split the view and create `SpreadSheet View` . The `SpreadSheet View` will automatically show the data produced by the `Wavelet` source. Upon closer inspection of the header in the `SpreadSheet View` , we see that the view is showing the `Point Data` or point attributes associated with the dataset. Now we have the same dataset, the data produced by the `Wavelet` source, shown in two views. Now, highlight a few rows in the `SpreadSheet View` by clicking on them. As soon as you start selecting rows, the `Render View` will start highlighting some points in space as tiny magenta specks ( Fig. 1.102). That's linked selection in action! What is happening is that, as you highlight rows in the `SpreadSheet View` , you are creating a selection for selecting points (since the view is showing `Point Data` ) corresponding to the rows. Due to the linking of selections between views, any other view that is showing the dataset (in this case, the `Render View` ) will also highlight the selected points.

Of course, if you want to select cells instead of points, switch the `SpreadSheet View` to show cells by flipping the `Attribute` combo-box to `Cell Data` and then highlight rows. The `Render View` will show the selected cells as a wireframe, rather than points.

Conversely, you could have created the selection in the `Render View` , and the `SpreadSheet View` will also highlight the selected elements. We will see how to create such selection later in this chapter.

The first thing to note is that, when you create a new selection, the existing selection is cleared. Thus, there is at most one active selection in the application at any given time. As we shall see, certain views provide ways of expanding on the existing selection.



Time: 0

Whole Extent: -10 10, -10 10, -10 10

Center: 0 0 0

Representation: Outline

Coloring: Solid Color

Styling

Opacity: 1

Showing: Wavelet0 Attribute: Point Data Precision: 6

| | Point ID | Structured Coordinates | | | RTData |
|---|---|---|---|---|---|
| 66 | 66 | 3 | 3 | 0 | 89.3245 |
| 67 | 67 | 4 | 3 | 0 | 79.8115 |
| 68 | 68 | 5 | 3 | 0 | 99.5414 |
| 69 | 69 | 6 | 3 | 0 | 92.5901 |
| 70 | 70 | 7 | 3 | 0 | 106.057 |
| 71 | 71 | 8 | 3 | 0 | 102.072 |
| 72 | 72 | 9 | 3 | 0 | 108.071 |
| 73 | 73 | 10 | 3 | 0 | 107.263 |
| 74 | 74 | 11 | 3 | 0 | 105.248 |

Fig. 1.102: Linked selection between views allows you to select elements in a view and view them in all other views showing the selected data. In this demo, as we select rows in the `SpreadSheet View`, the corresponding points in the `3D View` get highlighted.

The second thing to note is that selections are *transient*, i.e., they cannot be undone/redone or saved in state files and loaded back. Nor can you apply filters or other transformation to the selections themselves. There are cases, however, where you may want to subset your dataset using the selection defined interactively and then apply filters and other analysis to that extracted subset. For that, there are filters available, namely `Extract Selection` and `Plot Selection Over Time` , that can capture the active selection as filter parameters and then produce a new dataset that consists of the selected elements.

The third thing to note is that there are different types of selections, e.g., id-based selections, where the selected elements are identified by their indices; frustum-based selections, where the selected elements are those that intersect a frustum defined in 3D space; query-based selections, where the selected elements are those that match the specified query string; and so on.

## 1.6.2 Creating selections using views

Views provide a convenient mechanism for creating selections interactively. Views like `Render View` can create multiple types of selection (id- or frustum-based selections for selecting points and cells), while others like the `SpreadSheet View` and `Line Chart View` only support one type (id-based selections for points or cells).

### Selecting in Render View

To create a selection in the `Render View` , you use the toolbar at the top of the view frame. There are two ways of selecting cells, points or blocks in **ParaView**: interactive and non-interactive.

**ParaView** enters a *non-interactive selection mode* when you click one of the non-interactive selection buttons: The type of selection you are creating will depend on the button you clicked. Once in non-interactive selection mode, the cursor will switch to cross-hair and you can click and drag to create a selection region. Once you release the mouse, **ParaView** will attempt to create a selection for any elements in the selection region and will go back to default interaction mode.

To create a selection for cells visible in the view, use the button. For selecting visible points, use the button instead. Visible cells (and points) are only those cells (or points) that are currently rendered on the screen. Thus, elements that are occluded or are too small to be rendered on the screen will not be selected. If you want to select all data elements that intersect the view frustum formed by the selection rectangle you drew on the screen, use the button (or for points). In this case, all elements, visible or otherwise, that are within the 3D space defined by the selection frustum are selected.

To create a selection for blocks visible in the view use the button. Note that there is no frustum selection for blocks.

While most selection modes allow you to define the selection region as a rectangle, (and for points) enables you to define the selection region as a closed polygon. However, this is limited to surface elements (i.e., no frustum-based selection).

**ParaView** enters an *interactive selection mode* when you click on one of the interactive selection buttons: . In interactive selection mode, you act on visible elements (cells or points). **ParaView** highlights elements of the dataset as you move the cursor over them. An element can be selected by clicking on it. Clicking repeatedly on different elements adds them to the selection. End the interactive selection mode by clicking on the interactive selection button pushed in or by pressing the `Esc` key. This mode is also ended when you enter a non-interactive selection mode. Use to select all cells with the same value in the current color array as the cell underneath the cursor (only available for `idtype` arrays). Use to do the same as the previous icon for point data arrays. You can use the button to interactively select cells of the dataset and use the button to interactively select points.

When there are selected elements, the button can be used to clear the selection.

Several of these buttons have hotkeys too, such as `S` for visible cell selection, `D` for visible points selection, `F` for frustum-based cell selection, and `G` for frustum-based point selection. If you notice, these are keys are right next to each other on the keyboard, starting with `S`, and are in the same order as the toolbar buttons themselves.

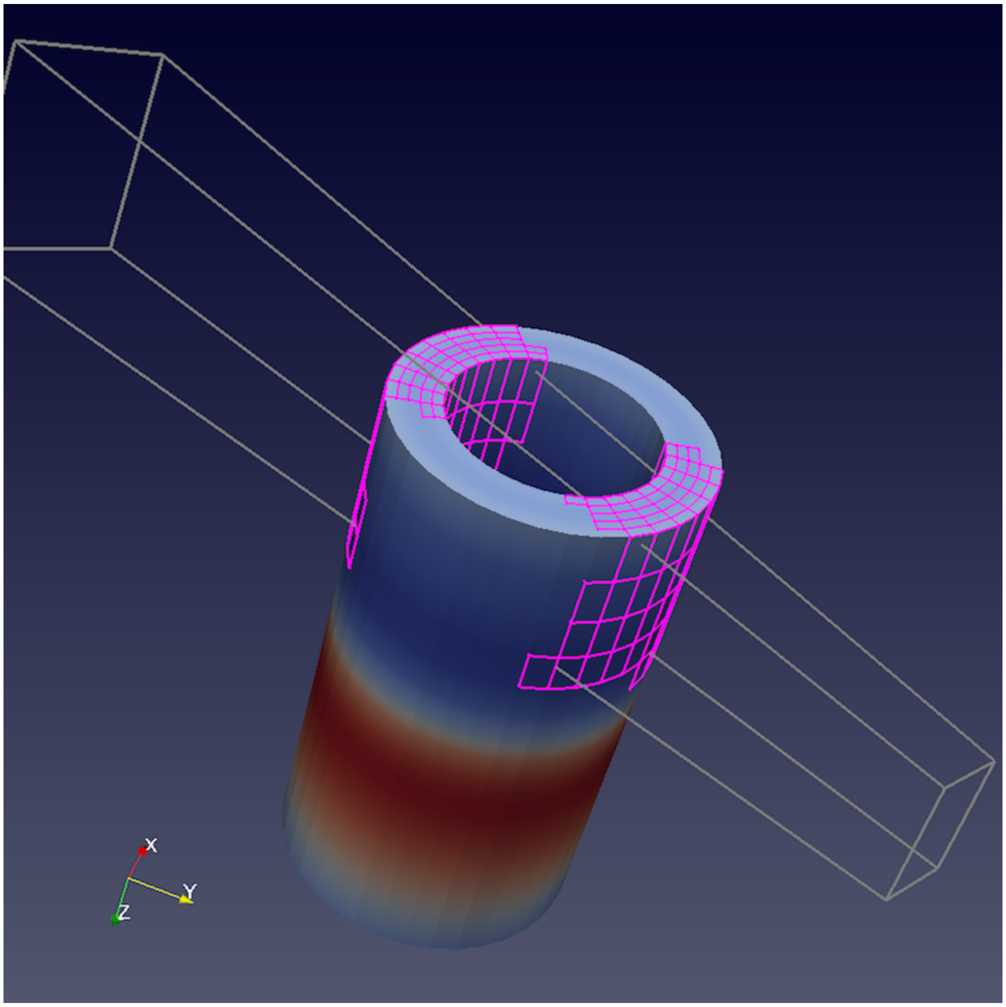


Fig. 1.103: Result of a frustum cell selection on disk_out_ref.ex2 dataset showing the frustum used to identify selected cells. All cells that fall in that frustum or that intersect it are selected, irrespective of whether they were visible from the view angle when the selection was made.

**Did you know?**

You can expand the current selection by keeping the `CTRL` (or ) key pressed when clicking and dragging in selection mode. `paraview` will then add the current selection. You can also subtract from the current selection using the , or even toggle using `CTRL` (or ) + . Selection modifier buttons, in the toolbar, can be used for the same effect. Add : , remove : , toggle : . These modifiers do not work, however, if the selected data is different from the current selection. If so, the current selection will be cleared (as is the norm) and then the new selection will be created.

#### Selecting in SpreadSheet View

To create a selection in the `SpreadSheet View`, you simply click on the corresponding rows in the spreadsheet. You can use the `CTRL` (or ) and keys to add to the selection. Based on which data attribute the view is currently showing, i.e., `Point Data`, `Cell Data`, or `Row Data`, the selection will select points, cells, or rows, respectively.

#### Selecting in Line Chart View

`Line Chart View` enables you to select the elements corresponding to the plotted data values. The selection interaction is similar to `Render View`. By default, you are in the interaction mode. You enter selection mode to create a selection by using the buttons in the view toolbar for creating a rectangular selection or a polygonal selection . Once in selection mode, you can click and drag to define the selection region. The selection is created once you release the mouse press.

When a new selection is created, by default, it will clear any existing selection in the view. The selection modifier buttons in the view toolbar can be used to control whether a new selection adds to selected elements , removes points from the selected elements , or toggles it . These modifier buttons are mutually exclusive and modal, i.e., they remain pressed until you click to unpress them or until you press another modifier button. `CTRL` (or ) and can also be used to add to/subtract from the selection.

### 1.6.3 Creating selections using the *Find Data* panel

Views provide mechanisms to create selections interactively. Selections in chart views and `SpreadSheet View` can be used to select elements with certain data properties, rather than spatial locations ( Fig. 1.104). For a richer data-based selection for selecting elements matching certain criteria, you can use the `Find Data` mechanism in `paraview`.

The `Find Data` panel can be accessed from the *Edit* menu, *View* menu, or by using the keyboard shortcut `V` or the button on the `Main Controls` toolbar. The `Find Data` panel can be split into three sections, reflecting how you would use this dialog. The `Create Selection` section helps you define the selection criteria. This identifies which elements, cells or points, are to be selected. The `Selected Data` section shows the results from the most recent selection. They are shown in a tabular view similar to the Spreadsheet view. Finally, the `Selection Display` section lets you change how the selected elements are displayed in the active view.

You can create selections in the `Find Data` panel using the widgets under the `Create Selection` section. First, choose the *data producer*. This is the source or filter from which you want to select elements from. Next choose the *element type*. If you want to select cells, choose **Cell**, for points choose **Point** and so on. The next step is to define the selection criteria. The left-most combo-box is used to select the array of interest. The available options reflect the data array currently available on the dataset. The next combo-box is used to select the operator. Options include the following:

- `is` matches a single value
- `is in range` matches a range of values specified by min and max
- `is one of` matches a list of comma-separated values

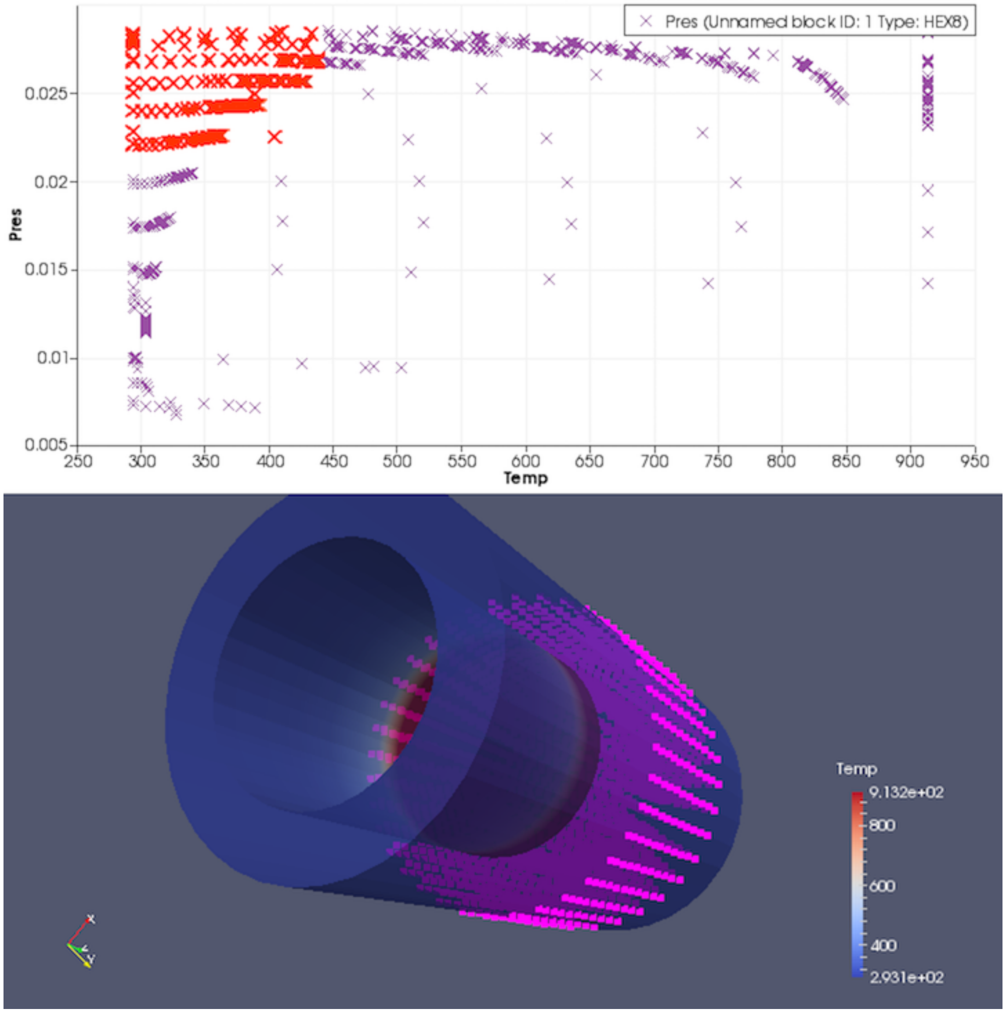
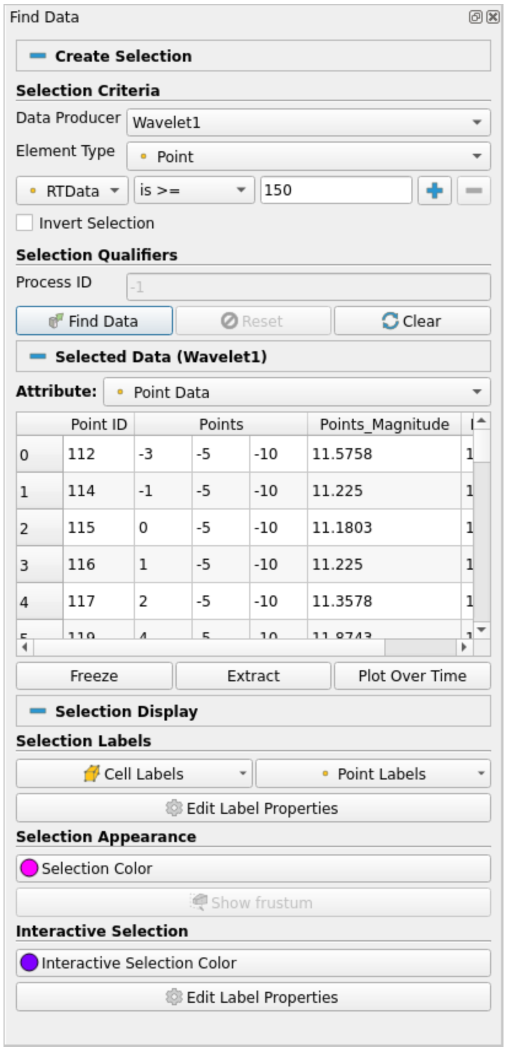


Fig. 1.104: Selection in `Line Chart View` can be used to locate elements matching attribute criteria. In this visualization, by generating a scatter plot plotting Pres against Temp in the disk_out_ref.ex2 dataset by selecting the top-left corner of the `Line Chart View` , we can easily locate elements in the high Pres, low Temp regions in the dataset.



Find Data

**Create Selection**

**Selection Criteria**

Data Producer: Wavelet1

Element Type: Point

RTData | is >= | 150

□ Invert Selection

**Selection Qualifiers**

Process ID: -1

**Selected Data (Wavelet1)**

**Attribute:** Point Data

| | Point ID | Points | | | Points_Magnitude | I |
|---|---|---|---|---|---|---|
| 0 | 112 | -3 | -5 | -10 | 11.5758 | 1 |
| 1 | 114 | -1 | -5 | -10 | 11.225 | 1 |
| 2 | 115 | 0 | -5 | -10 | 11.1803 | 1 |
| 3 | 116 | 1 | -5 | -10 | 11.225 | 1 |
| 4 | 117 | 2 | -5 | -10 | 11.3578 | 1 |
| 5 | 119 | 4 | -5 | -10 | 11.8743 | 1 |

**Selection Display**

**Selection Labels**

**Selection Appearance**

**Interactive Selection**

Fig. 1.105: The `Find Data` panel can be used to find data elements matching specific conditions. In this example, we are selecting all *Points* in *Wavelet1* dataset where *RTData* is >= 150.

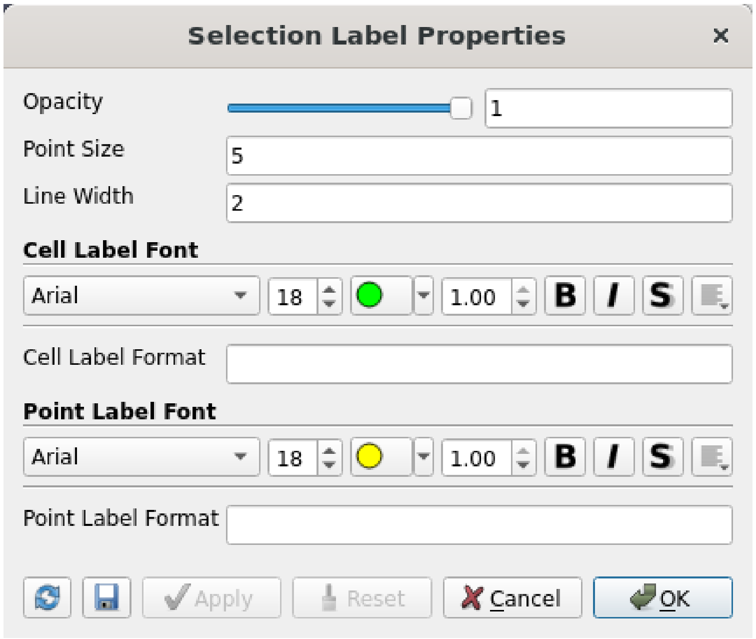
- `is >=` matches all values greater than or equal to the specified value
- `is <=` matches all values lesser than or equal to the specified value
- `is min` matches the minimum value for the array for the current time step
- `is max` matches the maximum value for the array for the current time step
- `is <= mean` matches values lesser than or equal to the mean
- `is >= mean` matches values greater than or equal to the mean
- `is mean` matches values equal to the mean within the specified tolerance

Based on your selection of the operator, input widgets will be shown next to this combo-box, where you enter the corresponding values. For example, for `is between`, you enter the min and max values for defining the range in the two text entry widgets.

Multiple selection criteria can be combined together. For example, you want to select all points with *Temp >= 100* **and** *Pres <= mean*, simply setup two expressions using the ✚ button.

Once you are satisfied with the selection criteria, hit the `Find Data` button. On success, the `Current Selection` spreadsheet will update to show the selected elements. Use the `Attribute` combo-box to change which element types are shows in the spreadsheet.

Similar to selecting in views, once you create a selection, any view showing the selected data will also highlight the selected elements, if possible. For example, the `Render View` will show a colored wireframe marking the selected elements, `SpreadSheet View` will highlight the rows, and so on. The `Selection Display` section lets you change how the selection is displayed in the active view. Currently, it is primarily designed for `Render View`. In the future, however, it could support changing selection attributes for other views as well. The available options allow you select the color to use to show the selected elements, as well as the data attributes to use to label the cells/points. For finer control on the label formatting, color, font, etc., toggle the  button on. That will pop up the `Edit Label Properties` dialog ( Fig. 1.106).


Selection Label Properties

Opacity: 1

Point Size: 5

Line Width: 2

**Cell Label Font**

Arial | 18 | 🟢 | 1.00

Cell Label Format:

**Point Label Font**

Arial | 18 | 🟡 | 1.00

Point Label Format:

Fig. 1.106: `Selection Label Properties` dialog for controlling selection labelling parameters.

**Did you know?**

Besides creating new selections, the Find Data dialog can also be used to inspect the current selection made from outside the dialog. For example, if you select elements in the Render View using the options described in Section Section 1.6.2, the Current Selection component in the Find Data dialog will indeed update to reflect the newly selected elements. Furthermore, you can change its display properties and extract this selection using the extraction buttons (which we will cover in Section Section 1.6.6).

### 1.6.4 Creating selections in Python

Another way to create selections is through ParaView's Python scripting interface. Python functions exist that are analogous to the selection operations available in the ParaView Render View and Find Data dialog. Let's take a look at an example.

```python
# import the selection module
from paraview.selection import *

renderView1 = GetActiveView()

# Create an initial rectangular selection in the render view
SelectSurfacePoints(Rectangle=[200, 321, 600, 744], View=renderView1)

# Add points within a polygon in the active view
SelectSurfacePoints(Polygon=[180, 200, 190, 400, 322, 300], Modifier='ADD')

# Subtract points with another rectangle
SelectSurfacePoints(Rectangle=[300, 400, 500, 700], Modifier='SUBTRACT')

# Now extract and show the selected points into another dataset
ExtractSelection()
Show()

# Clear the selection
ClearSelection()
```

The script starts out by importing functions from the paraview.selection module. Next, it creates a reference to the active render view and passes it into the selection functions. The first selection function selects points visible in the render view within a rectangular region. The rectangle is defined by bottom left and upper right points, (200, 321) and (600, 744), given in pixel coordinates.

The second selection is of visible points within a polygon defined by the points (180, 200), (190, 400), and (322, 300). In this call, the selection function modifies the existing selection so that newly selected points are added to the selection. This is controlled with the Modifier named function parameter. Other options for the Modifier parameter are 'SUBTRACT' , 'TOGGLE' , and None . When the Modifier is set to None , the previous selection gets replaced with the new selection. The last call to SelectSurfacePoints subtracts points from the current selection, which is the combination of the first two selections.

The last lines in this example script extract the currently selected points from the currently active source and shows them on the screen. Lastly, the selection is cleared with the ClearSelection function.

Selections by point or cell ID numbers are also possible, as shown in this example:

```python
from paraview.selection import *

# Select cell 1 from all blocks in a multiblock dataset on process 0
```

(continues on next page)

(continued from previous page)

```
SelectIDs(IDs=[0, 1], FieldType='CELL')

# Add cell 3 from block 4 on process 0 and cell 5 from block 6 on process 1
# to the selection
SelectCompositeDataIDs(IDs=[4, 0, 3, 6, 1, 5], Modifier='ADD')
```

Finally, selections by query expressions are also possible via the Python selection API. As an example, the following selects cells that have the maximum value for a cell variable named EQPS in the currently active source:

```
from paraview.selection import *

QuerySelect(QueryString='EQPS == max(EQPS)', FieldType='CELL')
```

The complete list of selection functions are briefly described below. For full documentation on these functions, you can invoke the help function on any of the functions, e.g., help(SelectSurfaceCells) .

- SelectSurfacePoints - Select visible points within a rectangular or polygon region.
- SelectSurfaceCells - Select visible cells within a rectangular or polygon region.
- SelectSurfaceBlocks - Select visible blocks within a rectangular region.
- SelectPointsThrough - Select all points within a rectangular region regardless of their visibility.
- SelectCellsThrough - Select all cells within a rectangular region regardless of their visibility.
- SelectGlobalIDs - Select attributes by global IDs.
- SelectPedigreeIDs - Select attributes by Pedigree IDs.
- SelectIDs - Select attributes by attribute IDs.
- SelectCompositeDataIDs - Select attributes by composite attribute IDs.
- SelectHierarchicalDataIDs - Select attributes by hierarchical data IDs.
- SelectThresholds - Select attributes in a source by thresholding on values in an associated array.
- SelectLocation - Select points by location.
- QuerySelect - Selection by query expression.
- ClearSelection - Clears the selection on the source passed in as a parameter.

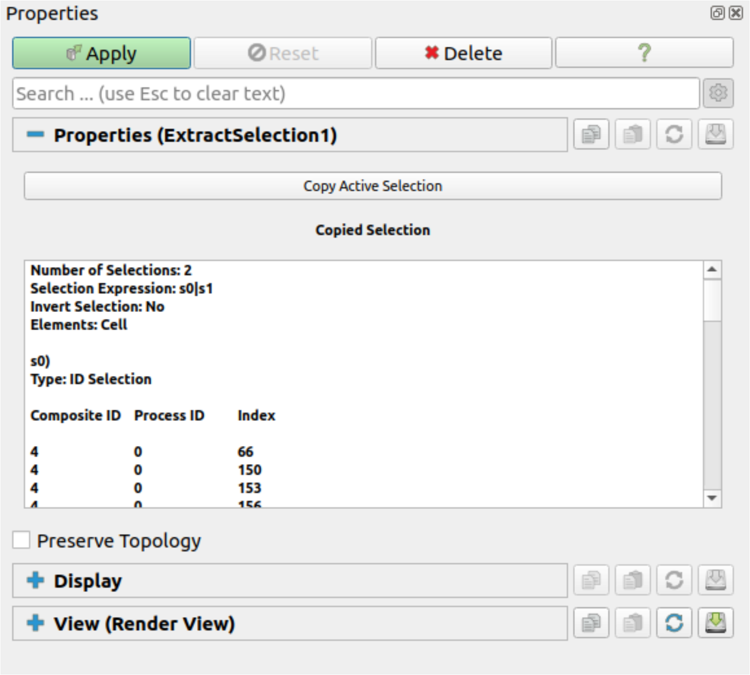
### 1.6.5 Displaying selections

The Find Data panel provides easy access to changing the Selection Display Properties for the selection in the active view. The Current Selection section in the Find Data dialog shows the selected elements in a spreadsheet view. You can also make a regular SpreadSheet View do the same by checking the ⬚ button in the view toolbar to show only selected elements.

### 1.6.6 Extracting selections

All the types of selections created through mechanisms discussed so far are transient and primarily used for highlighting data. If you want to do further operations on the selected subset, such as extract the selected elements and then save the result out as a new dataset or apply other filters only on the selected elements, then you need to use one of the extract selection filters. The `Extract Selection` and `Plot Selection Over Time` filters fall in this category of filters.

#### Extract selection



```
Number of Selections: 2
Selection Expression: s0|s1
Invert Selection: No
Elements: Cell

s0)
Type: ID Selection

Composite ID  Process ID  Index

4             0           66
4             0           150
4             0           153
4             0           156
```

□ Preserve Topology

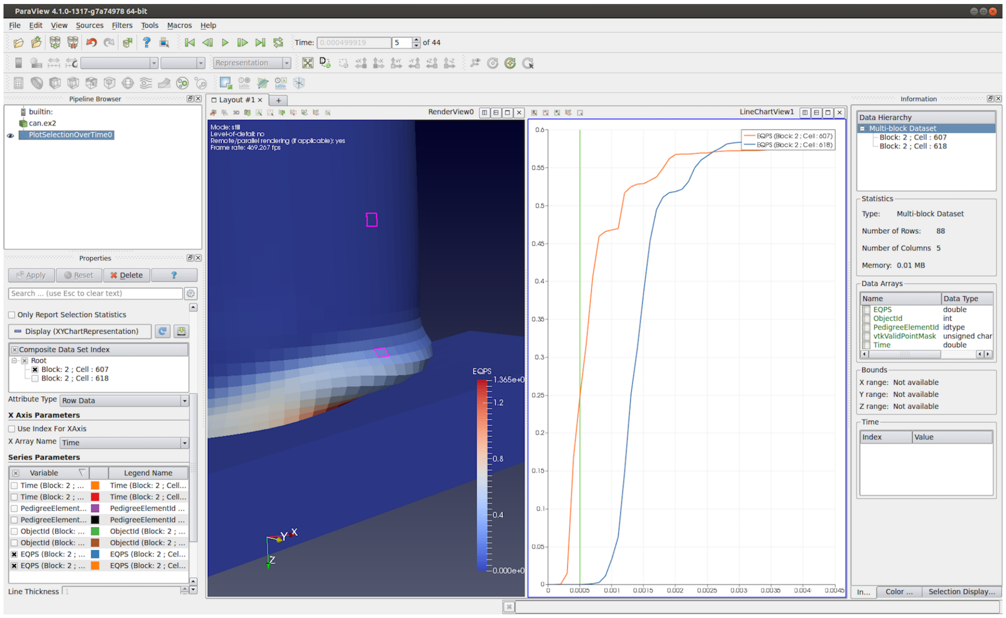
Fig. 1.107: `Properties` panel showing the properties for the `Extract Selection` filter.}

The `Extract Selection` filter is used to extract the selected elements as a new dataset for further filtering. There are multiple ways of creating this filter. You can use the conventional method for creating filters, i.e., using the *Filters* menu. When the filter is created, if there is any active selection, the filter will automatically copy that selection for convenience. Another way to extract the active selection is using the `Extract` button in the `Find Data` panel ( Fig. 1.105).

The `Properties` panel shows what defines the selection. You can update the selection by making a new active selection using any of the mechanisms described earlier in this chapter and then clicking on the `Copy Active Selection` button on the `Properties` panel for the `Extract Selection` filter.

By default, the filter is set up to extract the selected elements alone. This filter also supports passing the entire input dataset through by simply marking which elements are selected. For that, check the `Preserve Topology` check box on the `Properties` panel.

### Plot selection over time



□ Only Report Selection Statistics

Display (XYChartRepresentation)

☒ Composite Data Set Index
- ☒ Root
  - ☒ Block: 2 ; Cell : 607
  - □ Block: 2 ; Cell : 618

Attribute Type: Row Data

**X Axis Parameters**

□ Use Index For XAxis

X Array Name: Time

**Series Parameters**

| ☒ Variable | | Legend Name |
|---|---|---|
| □ Time (Block: 2 ; ... | 🟧 | Time (Block: 2 ; Cell... |
| □ Time (Block: 2 ; ... | 🟥 | Time (Block: 2 ; Cell... |
| □ PedigreeElement... | 🟪 | PedigreeElementId ... |
| □ PedigreeElement... | ⬛ | PedigreeElementId ... |
| □ ObjectId (Block: ... | 🟩 | ObjectId (Block: 2 ; ... |
| □ ObjectId (Block: ... | 🟫 | ObjectId (Block: 2 ; ... |
| ☒ EQPS (Block: 2 ; ... | 🟦 | EQPS (Block: 2 ; Cel... |
| ☒ EQPS (Block: 2 ; ... | 🟧 | EQPS (Block: 2 ; Cel... |

Line Thickness 1

Data Hierarchy
- Multi-block Dataset
  - Block: 2 ; Cell : 607
  - Block: 2 ; Cell : 618

Statistics

Type: Multi-block Dataset

Number of Rows: 88

Number of Columns 5

Memory: 0.01 MB

Data Arrays

| Name | Data Type |
|---|---|
| EQPS | double |
| ObjectId | int |
| PedigreeElementId | idtype |
| vtkValidPointMask | unsigned char |
| Time | double |

Bounds

X range: Not available

Y range: Not available

Z range: Not available

Time

| Index | Value |
|---|---|

Fig. 1.108: `Plot Selection Over Time` in action in `paraview`. The filter provides a convenient way to plot changes in attributes over time for the selected set of cells or points in a temporal dataset.

`Plot Selection Over Time` is similar to `Extract Selection` in the sense that it too extracts the selected elements from the input dataset. However, instead of simply extracting the result, the goal here is to plot attributes at the selected elements over time.

Fig. 1.108 shows an example use of this filter. In this case, we wanted to see how the strain (or *EQPS*) cell attribute changes over time for two specific cells that we selected in the `Render View` using the view-based selection mechanism. The selected cells are the highlighted elements in the left view. After having selected the cells, we create the `Plot Selection Over Time` filter using the *Filters > Data Analysis* menu. (You could also use the from the `Data Analysis` toolbar.) Similar to the `Extract Selection` filter, when this filter is created, it copies the active selection. You can change it afterwards using the `Copy Active Selection` button on the filter's `Properties` panel. On hitting `Apply` , `paraview` will show a visualization similar to the one shown here.

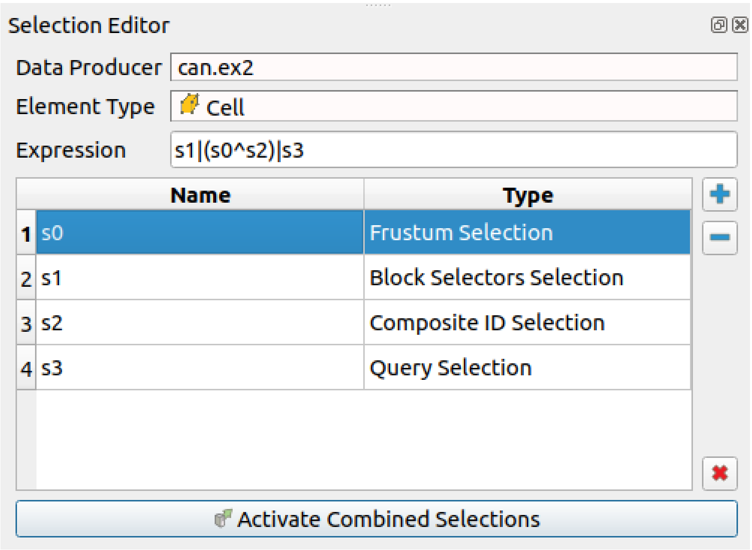
Instead of using the view for defining the selection, you could have used the `Find Data` panel. In that case, instead of being able to plot each element over time, you will be plotting summaries for the selected subset over time. This is essential since the selected subset can have a varying number of elements over time. The summaries include quantities like mininum, maximum, and median of available variables. You can make the filter always produce these statics alone (even when the selection is created by selecting specific elements in a view) by checking the `Only Report Selection Statistics` property on the `Properties` panel for the `Plot Selection Over Time` filter.

### 1.6.7 Freezing selections

When extracting selections, you can use views or the `Find Data` panel to define the selection. Since the extraction filters are indeed like any other filters in **ParaView**, they are re-executed any time the input dataset changes, properties on the filter change, or the current time changes. Every time the filter re-executes, it performs the *selection* and *extraction* operations. Thus, if you created the selection using `Render View` to create an id-based selection, the filter will identify which of the elements are of the requested ids and then pass those. For frustum-based selection, it will determine what elements fall within the frustum and extract those. Similarly, with query-based selections created using the `Find Data` panel, the query is re-evaluated. This can result in the selection of different elements with changes in timestep. For example, if you are selecting the cells where the strain is maximum, the selected cell(s) will potentially be different for each time step. Suppose you want to plot the changes in a cell that has maximum strain at the last time step – how can we do that? The answer is using the `Freeze Selection` button on the `Find Data` panel. What that does is convert any type of selection (frustum, query-based) to an id-based selection matching the currently selected element ids. Now you can use this frozen, id-based selection for `Extract Selection` or `Plot Selection Over Time`.

### 1.6.8 Saving and combining selections using the *Selection Editor* panel

Views and `Find Data` panel can be used to create different types of selections. To save and combine the created selections, you can use the `Selection Editor` panel. This panel can be accessed from the View → Selection Editor.



Selection Editor

Data Producer: can.ex2

Element Type: Cell

Expression: s1|(s0^s2)|s3

| | Name | Type |
|---|---|---|
| 1 | s0 | Frustum Selection |
| 2 | s1 | Block Selectors Selection |
| 3 | s2 | Composite ID Selection |
| 4 | s3 | Query Selection |

Fig. 1.109: The `Selection Editor` panel can be used to combine selections. In this example, we are combining frustum, block selector, composite ID and query selections.

The `Selection Editor` panel allows you to save selections and combine them using a boolean expression. This panel shows several pieces of static and editable information:

- An information-only `Data Producer` field is set based on the source of the active selection. If the selected object changes, the data producer will change as well, and any saved selections from the previous data producer will be deleted.
- An information-only `Element Type` field (cell or point) that is set based on the element type of the active selection. If a selection is saved and a new active selection is made that has a different element type, ParaView prompts for confirmation to delete all existing saved selections and changes the element type to that of the new active selection.

- An editable `Expression` string that defines how to combine the saved selections into a single active selection. This string is automatically filled while adding new selections to the saved selections, meaning that the combined selection will be the union of all saved selections. Selections can be combined using the not (`!`) operator, or (`|`) operator, and (&) operator, and xor (`^`) operator. Parentheses are available to define precedence as well.
- An editable table lists an automatically assigned name for the selection (s0, s1, s2, etc.), which is used in the `Expression` property, and the type of the selection. When a selection is highlighted in the saved selections table and ParaView's active view is a `Render View`, the highlighted selection will be shown in that `Render View`. When a selection is unhighlighted, it is no longer shown in the `Render View`.

Several buttons next to the saved selection table control the addition and removal of selections in the saved list:

- `+` (Add Active Selection) button that adds the active selection to the list of saved selections.
- `-` (Remove Selected Selection) button that removes the selected saved selection from the list of saved selections.
- `X` (Remove All Selections) button that removes all saved selections from the list of saved selections.
- `Activate Combined Selections` button that sets the combined saved selections as the active selection.

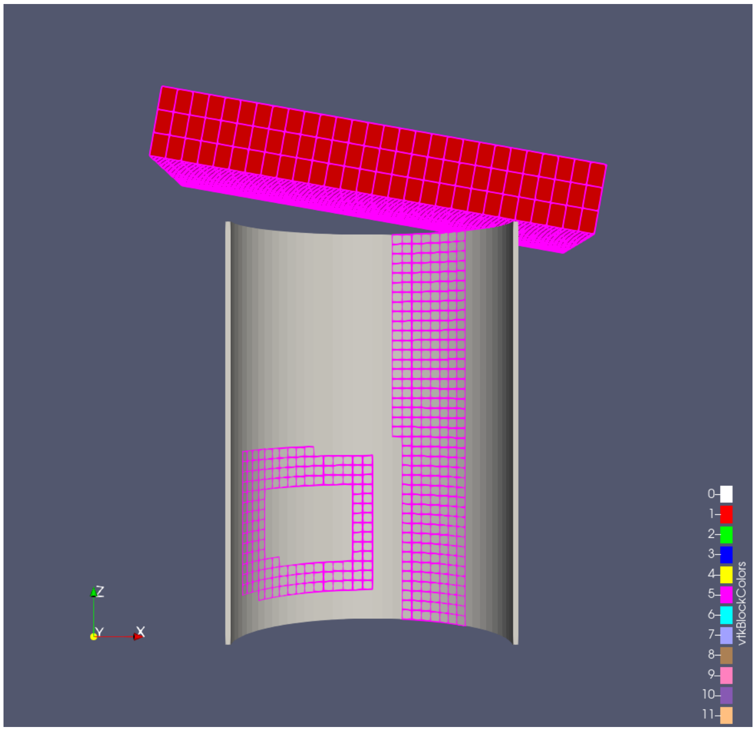


Fig. 1.110: The result of combining the selections as shown in Fig. 1.109.

## 1.7 Animation

In **ParaView**, you can create an animated scene where input data but also analysis and visualization parameters can change at each timestep.

When loading a time varying dataset, **ParaView** automatically loads the timesteps information. Then you can choose the current time, and the pipeline will be recomputed accordingly.

A property animation is created by recording a series of keyframes. At each keyframe, you set values for the properties of the readers, sources, and filters that make up the visualization pipeline, as well as the position and orientation of the camera. Once you have chosen the parameters, you can play through the animation.

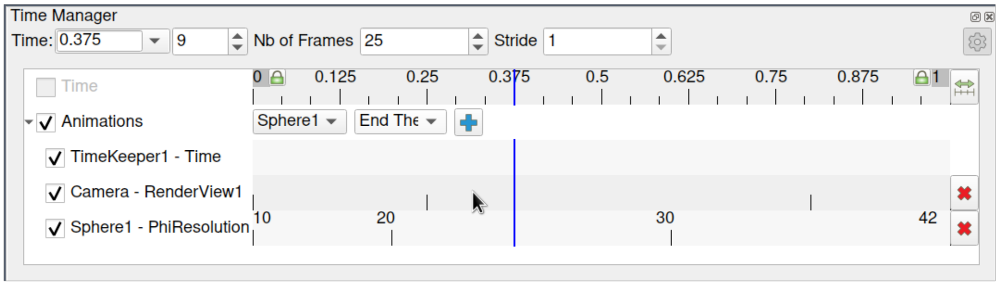
Also, the results of the animation can be saved to different format from the *File*: menu.

- *Save Animation*: images files (one image per animation frame) or movies (depends on available codecs)
- *Save Geometry*: all geometries in scene with **ParaView**'s PVD file format
- *Save Extracts*: save datasets and screenshots as configured in the pipeline with *Extractors* sources.
- *Export Animated Scene*: for external viewer, as vtk.js format

image files or to a movie file. The geometry rendered at each frame can also be saved in file format, which can be loaded back into **ParaView** as a time varying dataset.

### 1.7.1 Time Manager

`Time Manager` is the main place where to control time. This is accessible from the *View* menu.



Time Manager

Time: 0.375 9 Nb of Frames 25 Stride 1

Fig. 1.111: Time Manager.

This panel is presented as a table. Above the table are controls that administer how time progresses in the animation.

Within the table, the tracks are organized into two sections. The first displays information about time varying data present in the pipeline. Each row match one temporal pipeline source, and its associated timesteps. The first row in the table, simply labeled `Time`, shows the scene times, *i.e.* the times that will be used when playing the scene. That can be a combination of pipeline source data times. The current displayed time is indicated both in the Time field at the top and with a thick, vertical line within the table.

Note that both the first and the last time labels of the timeline have special background. Click on them to enter `Start Time` and `End Time` entry-boxes and configure the start and end times for the animation. By default, when you load time varying datasets, the start and end times are automatically adjusted to cover the entire time range present in the data. The lock check-buttons just next to the `Start Time` and `End Time` labels will prevent this from happening, so that you can ensure that your animation covers a particular time domain of your choosing.

In the second part come the animation tracks. The `Animations` row contains widget to creates animations.

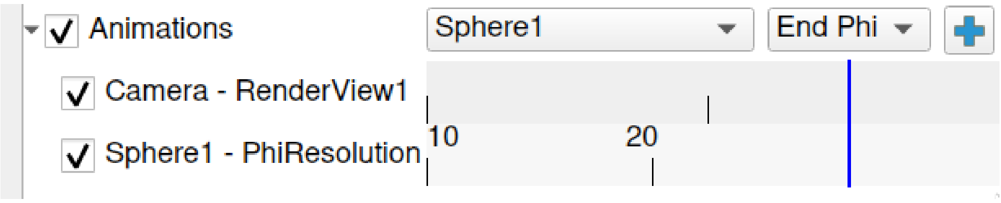
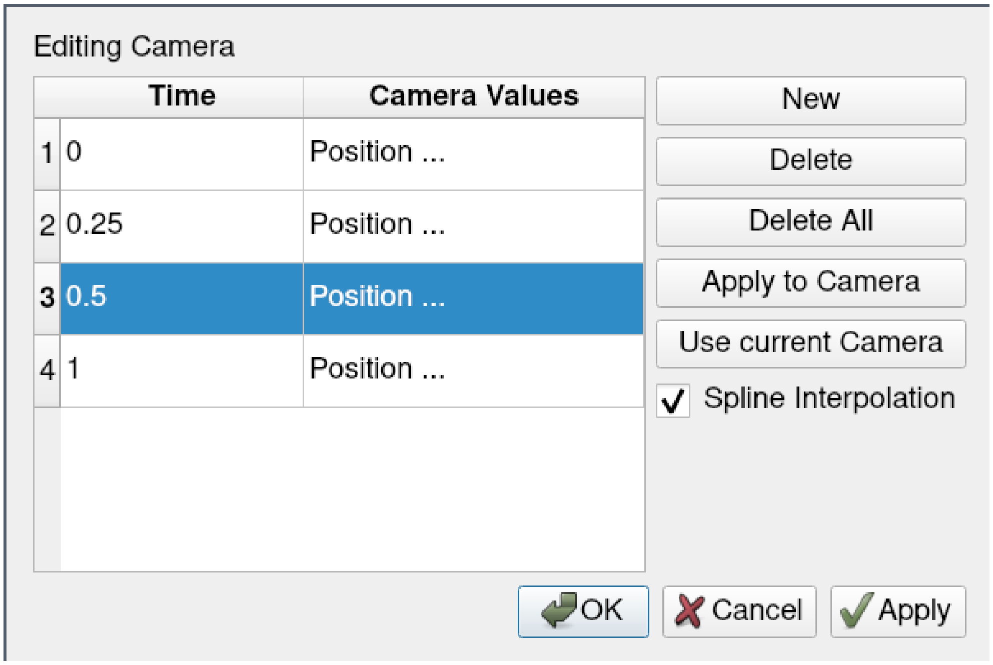


Fig. 1.112: Animation Tracks.

You choose a data source and then a particular property of the data source in the bottom row. To create an animation track with keyframes for that property, click the + on the right-hand side; this will create a new track. In the figure, tracks already exist for *SphereSource1*'s `Phi Resolution` property and for the camera's position. To delete a track, press the *X* button. You can temporarily disable a track by unchecking the check box on the left of the track. To enter values for the property, double-click within the white area to the right of the track name. This will bring up the `Animation Keyframes` dialog. Double-clicking in the camera entry brings up a dialog like the one in Fig. 1.113.



Editing Camera

| | Time | Camera Values |
|---|---|---|
| 1 | 0 | Position ... |
| 2 | 0.25 | Position ... |
| 3 | 0.5 | Position ... |
| 4 | 1 | Position ... |

☑ Spline Interpolation

Fig. 1.113: Editing the camera track.

From the `Animation Keyframes` dialog, you can press *New* to create new keyframes. You can also press *Delete* or *Delete All* to delete some or all of the keyframes. Clicking *New* will add a new row to the table. In any row, you can click within the `Time` column to choose a particular time for the keyframe, and you can click in the right-hand column to enter values for the parameter. The exact user interface components that let you set values for the property at the keyframe time vary. When available, you can change the interpolation between two keyframes by double-clicking on the central interpolation column.

Within the tracks of the `Time Manager` , the place in time where each keyframe occurs is shown as a vertical line. The values chosen for the property at that time and the interpolation function used between that value and the next are shown as text, when appropriate. In the previous figure, for example, the sphere resolution begins at 10 and then changes to 20, varying by linear interpolation between them. The camera values are too lengthy to show as text so they are not displayed in the track, but we can easily see that there are four keyframes spaced throughout the animation. The vertical lines in the tracks themselves may be dragged, so you can easily adjust the time at which each keyframe occurs.

**Did you know?**

> You can quickly add a simple animation track and edit the keyframe without using the `Time Manager` by using `Animation Shortcut` . First, enable `Show Animation Shortcut` from `Settings` dialog (on the `General` tab, search for the option by name or switch to advanced view). Then, several of the animatable properties on the `Properties` panel will have a  icon. Click this icon to add a new animation track for this property and edit it.

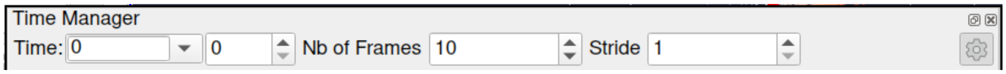
Properties (Shrink1)

Shrink Factor 0.5

## 1.7.2 Scene time configuration

### Time controls

The Time Manager has a header bar that lets you control some properties of the animation itself, as you can see in Fig. 1.114.



Time Manager

Time: 0 | 0 | Nb of Frames 10 | Stride 1

Fig. 1.114: Time Manager Widgets.

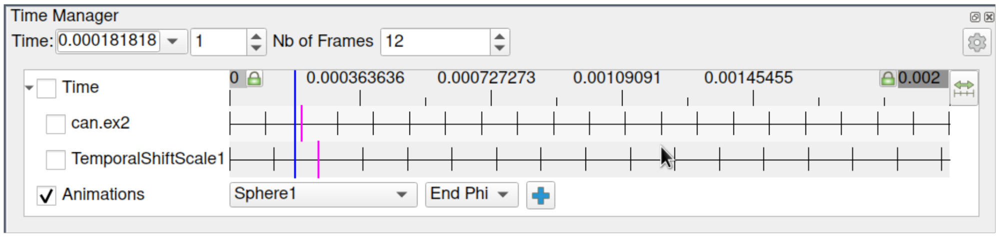
The Time entry-box shows the current animation time, which is the same as shown by a vertical marker in this view. You can change the current animation time by either entering a value in this box, if available, or by dragging the vertical marker. The second entry-box is the timestep index. You can also edit it.

If the scene does not use any pipeline sources to get its list of times (see below), you can also edit Number of frames to set the total number of frames for the scene.

The Stride parameter, available in advanced mode, allows to jump by this number of timesteps when navigating through times.

### Time list

**ParaView** supports two modes to configure scene times. The time list can either come from the data themself or it can be generated. We call them Snap to Timestep and Sequence respectively.



Time Manager

Time: 0.000181818 | 1 | Nb of Frames 12

□ Time

□ can.ex2

□ TemporalShiftScale1

☑ Animations | Sphere1 | End Phi

Fig. 1.115: Time Manager Sequence.

In Sequence mode, the number of frames is controlled by the No. Frames spinbox. Times are evenly spaced between Start and End time. This is the preferred mode when working with non-temporal data. This is activated when none of the Time tracks are enabled, i.e. the checkbox left to the Time label of first track is disabled.

In the figure, the main timeline has 12 ticks. They do not match the times from the data, as expected for a sequence.

In Snap To TimeSteps mode, the number of frames in the animation is determined by the number of time values in the dataset being animated and thus cannot be changed. This is the animation mode used by default by **ParaView** when a dataset with time values is loaded. Then playing the animation means playing through the time values of the dataset one after the other. This is enabled when at least one Time track is checked.

In the figure, only can.ex2 is checked so the scene knows only about its timesteps: the main timeline has same ticks as the can.ex2 track.

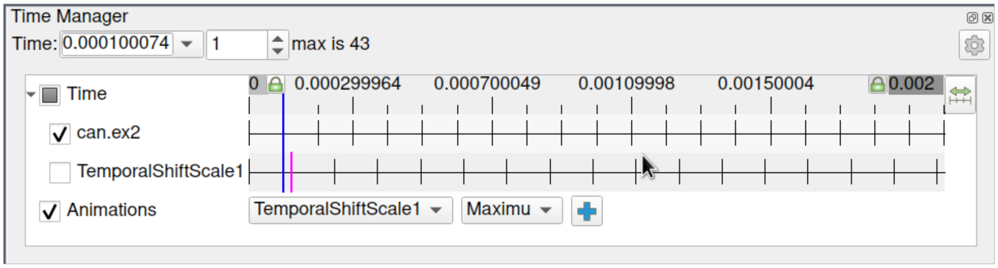
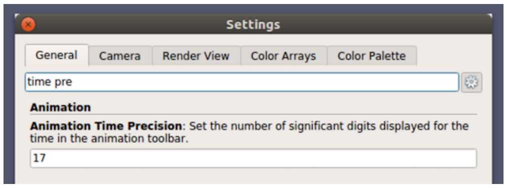


Fig. 1.116: Time Manager Snap to timesteps.

**Did you know?**

> You can change the precision (number of significant digits) displayed by the animation clock by changing the `Animation Time Precision` value in the `Settings` under the `General` tab.



time pre

**Animation**

**Animation Time Precision**: Set the number of significant digits displayed for the time in the animation toolbar.

17

## 1.7.3 Animating time-varying data

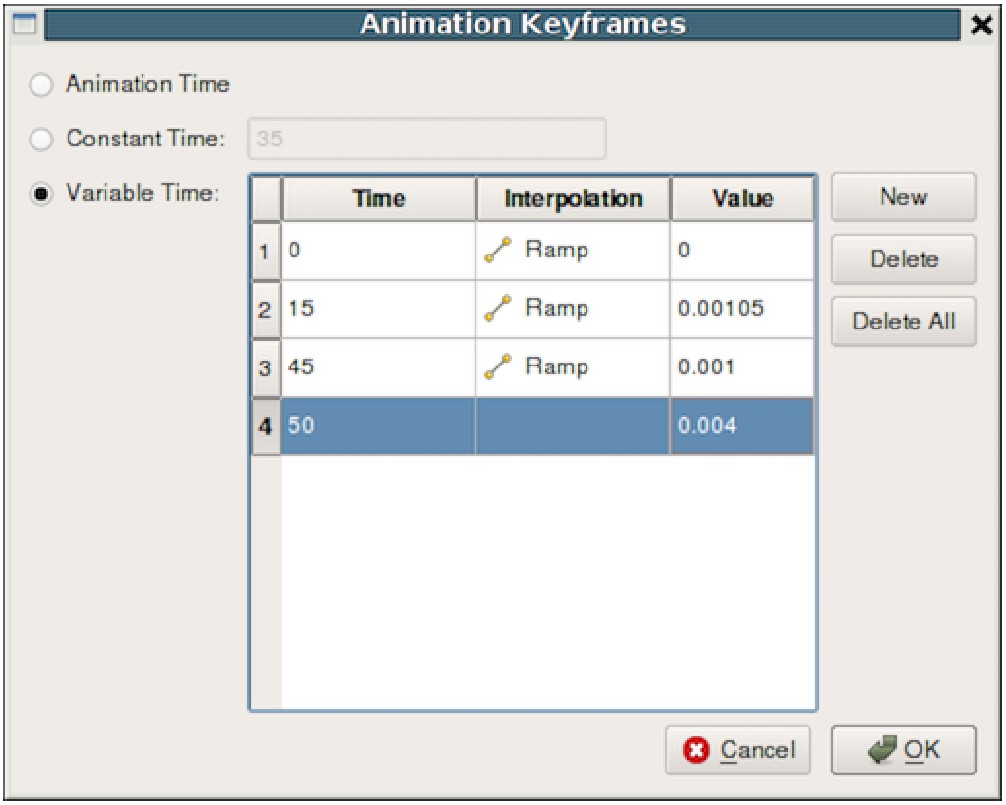
We saw that the main scene of **ParaView** can know about the times available from a pipeline source, such as a reader that produces time varying data. Then a current time is set at the scene level, and by default the pipeline is updated to the relevant time.

But what happen when asking for a time that does not exist in the data? How to do some properties animation at a fixed data time? Can I animate the time itself?

**ParaView** offers one answer to all those questions: the `Time Keeper`. This is an animation track, visible only in *advanced* mode. The `Time Keeper` maps the scene time to a new time used to update the pipeline. The default mapping simply maps the scene time to itself.

But you can uncouple the data time from the animation time so that you can create complex animations.

If you double-click in the `TimeKeeper - Time` track, the `Animation Keyframes` dialog, an example of which is shown in Fig. 1.117, appears. In this dialog, you can make data time progress in three fundamentally different ways. If the `Animation Time` radio-button is selected, the data time will be tied to and scaled with the animation time so that, as the animation progresses, you will see the data evolve naturally. If you want to ignore the time varying nature of the data, you can select `Constant Time` instead. In this case, you choose a particular time value at which the data will be displayed for the duration of the animation. Finally, you can select the `Variable Time` radio-button to have full control over data time and to control it as you do any other animatable property in the visualization pipeline. In the example shown in Fig. 1.117, time is made to progress forward for the first 15 frames of the animation, backward for the next 30, and forward for the final 15.



○ Animation Time

○ Constant Time: 35

● Variable Time:

|   | Time | Interpolation | Value |
|---|---|---|---|
| 1 | 0 | Ramp | 0 |
| 2 | 15 | Ramp | 0.00105 |
| 3 | 45 | Ramp | 0.001 |
| 4 | 50 |  | 0.004 |

Fig. 1.117: Controlling Data Time with keyframes.

### 1.7.4 Animating the camera

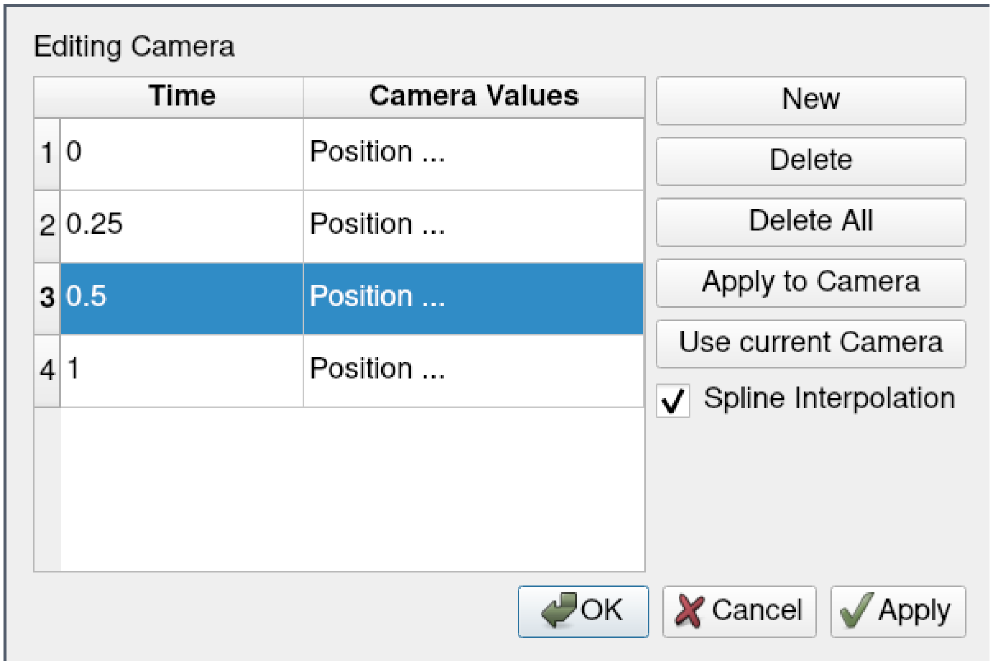
Just like you can change parameters on sources and filters in an animation, you can also change the camera parameters. You can add animation tracks to animate the camera for all the 3D render views in the setup separately. To add a camera animation track for a view, with the view selected, choose `Camera` from the first drop-down menu. The second drop-down list allows you to choose how to animate the camera. Then click on the + button. There are three possible options, each of which provides different mechanisms to specify the keyframes. It's not possible to change the mode after the animation track has been added, but you can simply delete the track and create a new one.

#### Follow data

In this mode, the camera focal point and position is changed to keep the data centered in the view. It does not change the camera zoom level, however.

### Interpolate cameras

In this mode, you specify camera position, focal point, view angle, and up direction at each keyframe. The animation player interpolates between these specified locations. As with other parameters, to edit the keyframes, double-click on the track. It is also possible to capture the current location as a keyframe by using the `Use Current` button.



Editing Camera

|   | Time | Camera Values |
|---|---|---|
| 1 | 0 | Position ... |
| 2 | 0.25 | Position ... |
| 3 | 0.5 | Position ... |
| 4 | 1 | Position ... |

☑ Spline Interpolation

Fig. 1.118: Setting camera animation parameters.

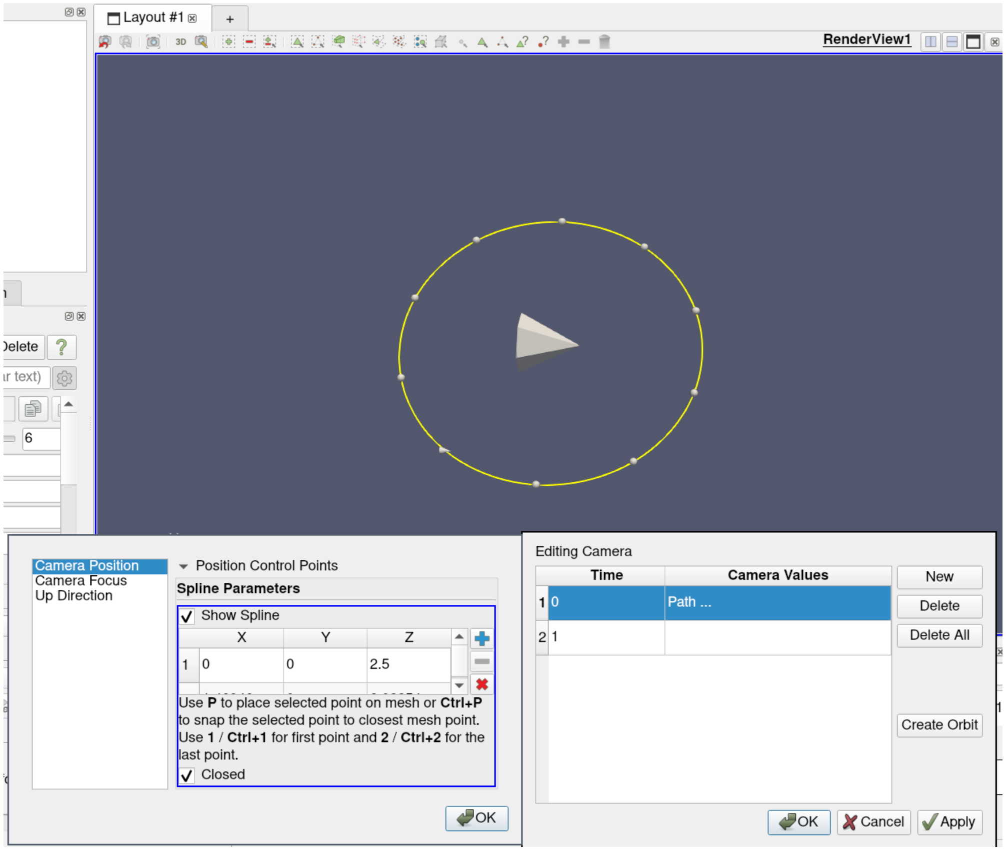
It can be quite challenging to add enough keyframes correctly to ensure that the animation results in a smooth visualization using this mode. However, it is the preferred mode if you want fine control on a few cameras.

### Follow path

In this mode, you get the opportunity to specify the path taken by the camera position and the camera focal point. By default, the path is set up to orbit around the selected objects. You can then edit the keyframe to change the paths. This is the preferred mode if you want to control the global motion more than exact points of view.

Fig. 1.119 shows the dialog for editing these paths for a keyframe. When Camera Position or Camera Focus is selected, a widget is shown in the 3D view that can be used to set the path. Use `CTRL` + `Left Click` to insert new control points, and + `Left Click` to remove control points. You can also toggle when the path should be closed or not.

This mode makes it possible to quickly create a camera animation in which the camera revolves around objects of interest. Clicking on the `Orbit` button will pop up a dialog where you can edit the orbit parameters such as the center of revolution, the normal for the plane of revolution, and the origin (i.e., a point on the plane where the revolution begins). By default, the `Center` is the center of the bounds of the selected objects, the `Normal` is the current up direction used by the camera, while the origin is the current camera position.



☑ Show Spline

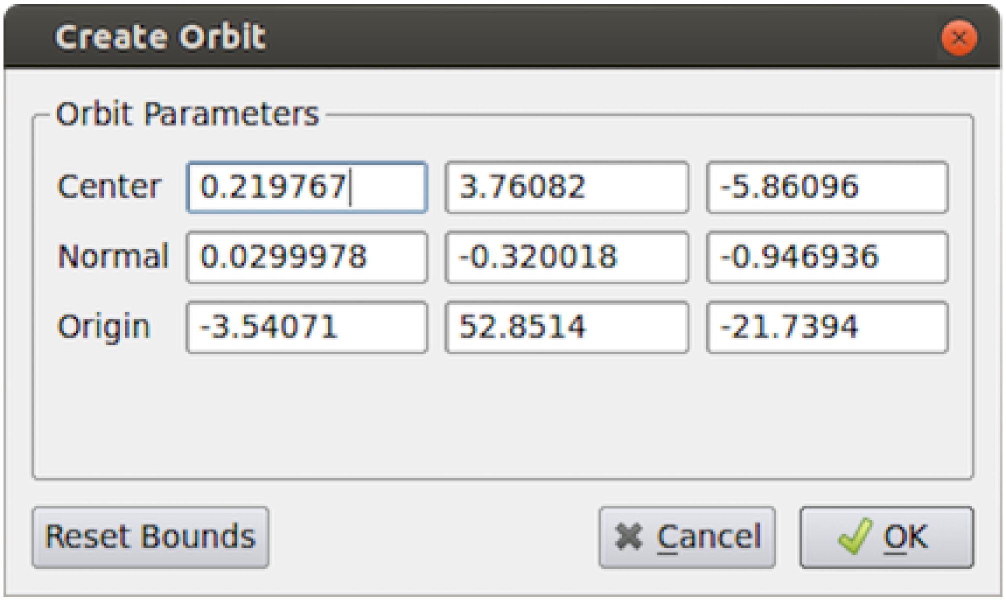
|   | X | Y | Z |
|---|---|---|---|
| 1 | 0 | 0 | 2.5 |

Use **P** to place selected point on mesh or **Ctrl+P** to snap the selected point to closest mesh point. Use **1** / **Ctrl+1** for first point and **2** / **Ctrl+2** for the last point.

☑ Closed

|   | Time | Camera Values |
|---|---|---|
| 1 | 0 | Path ... |
| 2 | 1 | |

Fig. 1.119: Creating a camera path.



Orbit Parameters

|   |   |   |   |
|---|---|---|---|
| Center | 0.219767 | 3.76082 | -5.86096 |
| Normal | 0.0299978 | -0.320018 | -0.946936 |
| Origin | -3.54071 | 52.8514 | -21.7394 |

Fig. 1.120: Creating a camera orbit.

### 1.7.5 Playing an animation

Once you have designed your animation, you can play through it with the VCR controls toolbar seen in Fig. 1.15.

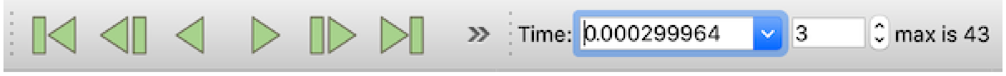


Fig. 1.121: `VCR Controls` and `Current Time Controls` toolbars in `paraview`.

Note that the frames are rendered as fast as possible. Thus, the viewing frame rate depends on the time needed to generate and render each frame.

When you play the animation, you can cache the geometric output of the visualization pipeline in memory. When you replay the animation, playback will be much faster because very little computation must be done to generate the images.

### 1.7.6 Explore an animation using Python

The following Python commands allow you to get and explore an animation scene.

Just like you can navigate through timesteps in `paraview`'s user interface, you can use Python commands to do the same. These commands are available through an animation scene object, retrieved with:

```
>>> scene = GetAnimationScene()
```

This object has several methods available to change which animation timestep is shown:

```
>>> scene.GoToFirst()
>>> scene.GoToLast()
>>> scene.GoToNext()
>>> scene.GoToPrevious()
```

Additional methods are available to start and stop playback of the animation:

*Play()* plays the animation from current timestep to the last. *Reverse()* plays the animation in reverse from the current timestep to the first.

Finally, you can access the available times in all loaded datasets using

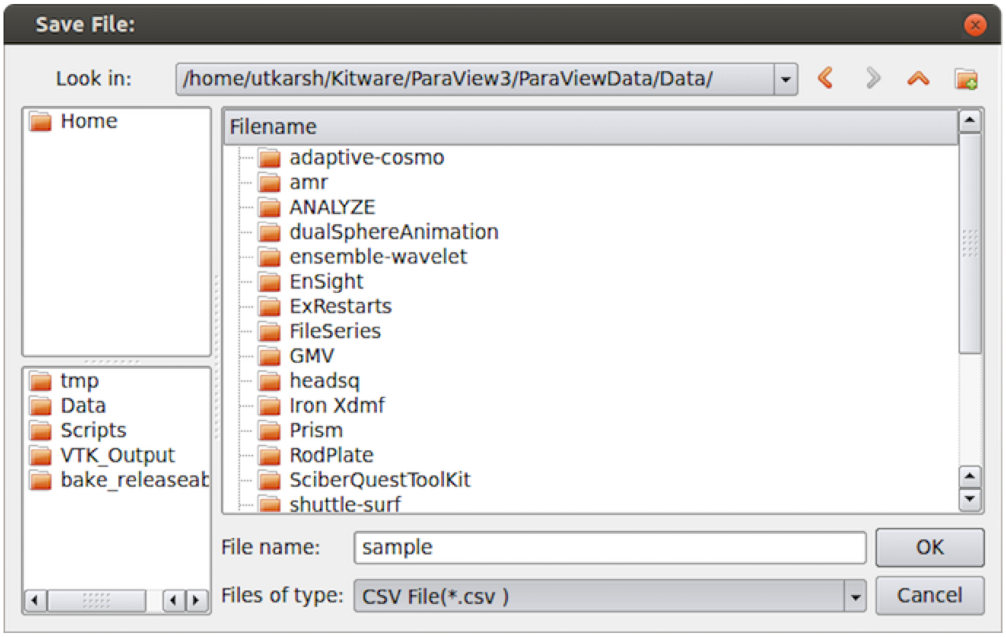
The times are returned in a Python list.

## 1.8 Saving Results

In this chapter, we will introduce various ways of saving visualization results in **ParaView**. Results generated throughout the visualization process not only include the images and the rendering results, but also include the datasets generated by filters, the scene representations that will be imported into other rendering applications, and the movies generated from animations.

### 1.8.1 Saving datasets

You can save the dataset produced by any pipeline module in **ParaView**, including sources, readers, and filters. To save the dataset in `paraview`, begin by selecting the pipeline module in the `Pipeline Browser` to make it the active source. For modules with multiple output ports, select the output port producing the dataset of interest. To save the dataset, use the *File* > *Save Data* menu or the  button in the `Main Controls` toolbar. You can also use the keyboard shortcut CTRL + S (or + S). The `Save File` dialog (Fig. 1.122) will allow you to select the filename and the file format. The available list of file formats depends on the type of the dataset you are trying to save.



Look in: /home/utkarsh/Kitware/ParaView3/ParaViewData/Data/

File name: sample

Files of type: CSV File(*.csv )

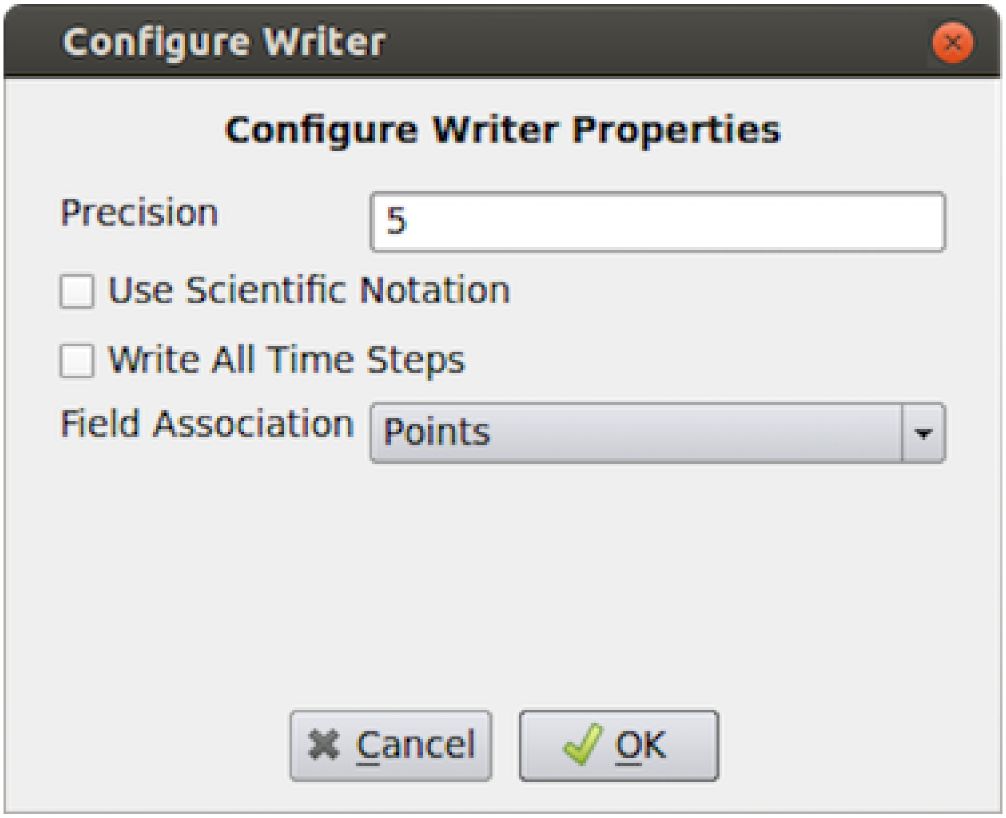
Fig. 1.122: `Save File` dialog in `paraview`.

On accepting a filename and file format to use, `paraview` may show the `Configure Writer` dialog (Fig. 1.123). This dialog allows you to further customize the writing process. The properties shown in this dialog depend on the selected file format and range from enabling you to `Write All Time Steps` , to selecting the attributes to write in the output file.

In `pvpython` too, you can save the datasets as follows:

```
# Saving the data using the default properties for
# the used writer, if any.
>>> SaveData("sample.csv", source)

# the second argument is optional, and refers to the pipeline module
# to write the data from. If none is specified the active source is used.

# To pass parameters to configure the writer
>>> SaveData("sample.csv", source,
             Precision=2,
             FieldAssociation='Cells')
```

`pvpython` will pick a writer based on the file extension and the dataset type selected for writing, similar to what it does in `paraview`. Admittedly, it can be tricky to figure out what options are available for the writer. The best way is to use the *Python tracing* capabilities in `paraview` and to use the generated sample script as a reference ( Section 1.1.6). Make sure you use a similar type of dataset and the same file format as you want to use in your Python script, when



Configure Writer

Configure Writer Properties

Precision: 5

□ Use Scientific Notation

□ Write All Time Steps

Field Association: Points

Fig. 1.123: `Configure Writer` dialog in `paraview` shown when saving a dataset as a csv file.

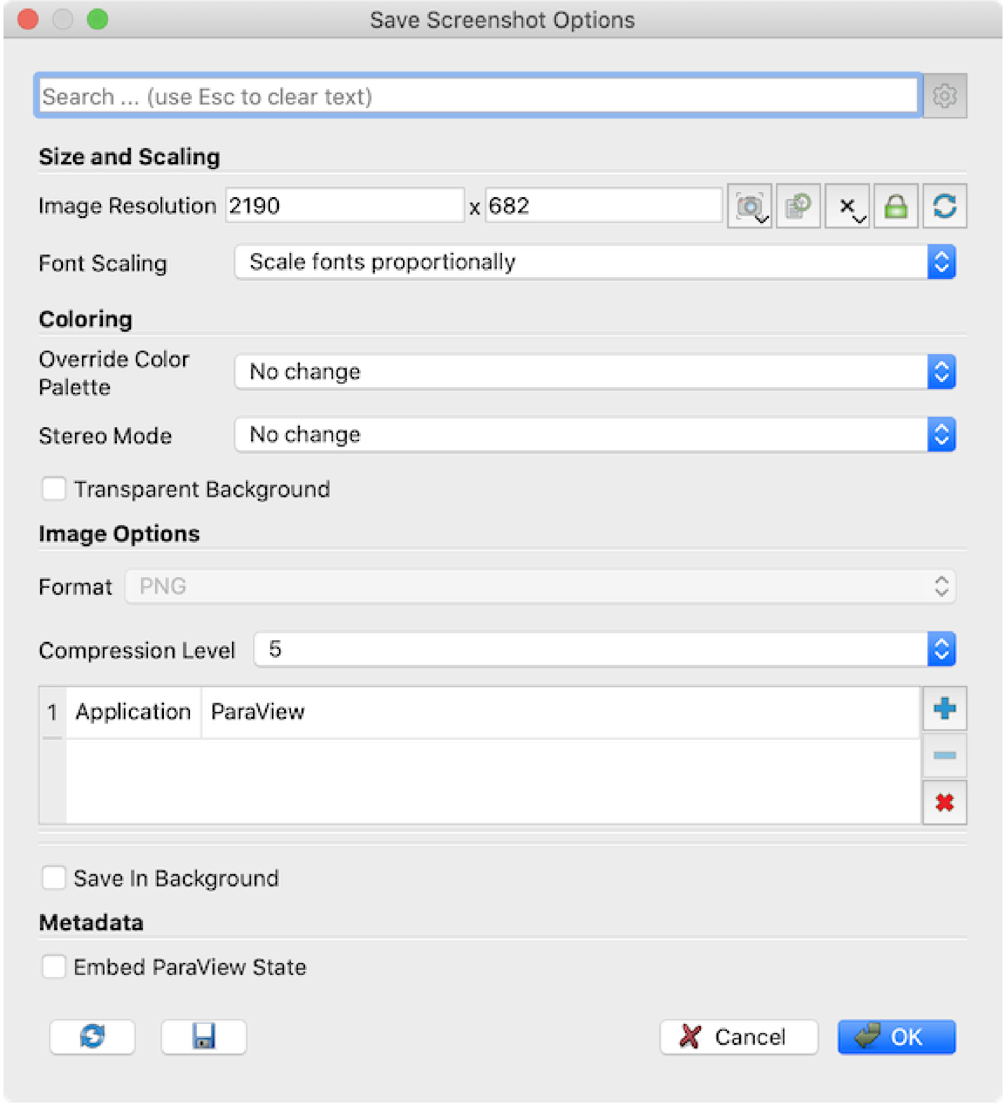
tracing, to avoid runtime issues.

## 1.8.2 Saving rendered results

Views that render results (this includes almost all of the views, except `SpreadSheet View` ) support saving images (or screenshots) in one of the standard image formats (PNG, JPEG, TIFF, BMP, PPM). Certain views also support exportings the results in several formats such as PDF, X3D, and VRML.

### Saving screenshots

To save the render image from a view in `paraview`, use the *File* > *Save Screenshot* menu option. When selected, a file dialog will appear where you can select the file path and format to which the screenshot should be saved. After selecting the image file, the `Save Screenshot Options` dialog (Fig. 1.124) will be shown. This dialog allows you to select various parameters that controls what image is saved out and how.



Size and Scaling

Image Resolution 2190 x 682

Font Scaling: Scale fonts proportionally

Coloring

Override Color Palette: No change

Stereo Mode: No change

□ Transparent Background

Image Options

Format: PNG

Compression Level: 5

| | | |
|---|---|---|
| 1 | Application | ParaView |

□ Save In Background

Metadata

□ Embed ParaView State

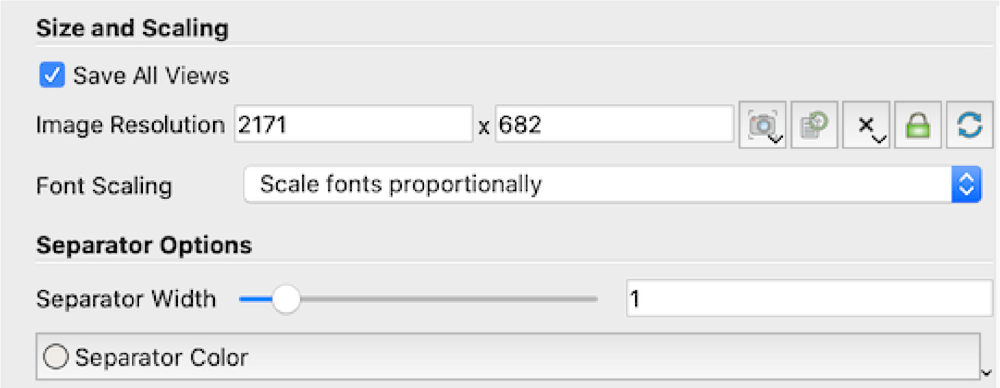
Fig. 1.124: The `Save Screenshot Options` dialog, which is used to customize saving screenshots in `paraview`.

If your visualization setup only has one view the active tab, then you'll be presented with options shown in (Fig. 1.124). The available options are as follows.

- `Image Resolution` : This is the target image resolution in pixels. By default, it is set to the current view dimensions. You can change it as needed. If the resolution larger than the current resolution, then ParaView will use *tiling* to render the full image in multiple stages. For reliable results, you may want to present the current aspect ratio. You can use *Tools > Lock View Size Custom* to lock the view size to a suitable aspect ratio.
- `Font Scaling` : When a resolution larger than the current resolution is specified, this allows you to control how the fonts are to be scaled. Default `Scale fonts proportionally` tries to achieve WYSIWYG as long as the aspect ratio is maintained. This is suitable for saving images targeted for higher DPI (or PPI) display than your screen. `Do not scale fonts` may be used to avoid font scaling and keep their size in pixels the same as what is currently on the screen. This is suitable for saving images targeted for a larger display with the same pixel resolution.
- `Override Color Palette` : Optionally change the color palette just for saving the screenshot using this dropdown.
- `Stereo Mode` : This option lets you save the image using one of the supported stereo modes.
- `Transparent Background` : If the file format supports it, you can check this option to save the images with a transparent background rather than the current background color.
- `Format` : This shows the file format selected in the file save dialog.

For formats that have different options like compression levels, format-specific options are presented in the `Save Screenshot Options` dialog. The PNG format has a `Compression Level` option that ranges from 0 (no compression) to 9 (maximum compression). The JPEG format options are `Quality` , which ranges from 0 to 100, and `Progressive` , which enables saving the file as a progressive JPEG. The TIFF file format has a `Compression` option with possible values `None` , `PackBits` , and `Deflate` . The BMP file format has no options.

If the active tab has more than one view, then the `Save Screenshot Options` dialog has a few more options as shown in Fig. 1.125.



☑ Save All Views

Image Resolution 2171 x 682

Font Scaling: Scale fonts proportionally

Separator Width 1

○ Separator Color

Fig. 1.125: Extra options in `Save Screenshot Options` dialog available when the active tab has more than 1 view.

- `Save All Views` : Check this to save all the views in the active tab laid out exactly as in the UI. If unchecked, only the active view will be saved.
- `Separator Options` : These control the separator drawn between the views in the generated image. You can specify the `Separator Width` in approximate pixels as well as the `Separator Color` .

To save a screenshot in `pvpython`, you use `SaveScreenshot` .

```
# Save a screenshot from a specific view.
>>> myview = GetActiveView()
>>> SaveScreenshot("aview.png", myview)

# Save all views in a tab
>>> layout = GetLayout()
>>> SaveScreenshot("allviews.png", layout)

# To save a specific target resolution, rather than using the
# the current view (or layout) size, and override the color palette.
>>> SaveScreenshot("aviewResolution.png", myview,
        ImageResolution=[1500, 1500],
        OverrideColorPalette="Black Background")
```

As always, you can use *Python tracing* in `paraview` to trace the exact form of the method to use to save a specific screenshot image.

### Exporting scenes

When available, you can export a visualization in a view in several of the supported formats using the *File > Export Scene…* menu option in `paraview`. For a `Render View` (or similar), the available formats include Cinema Database, EPS, PDF, PS, SVG, POV, VRML, WebGL, X3D, and X3DB. On selecting a file as which to export, `paraview` may pop up an `Export Options` dialog that allows you to set up parameters for the exporter, similar to saving datasets ( Section 1.8.1).

In addition, from `pvpython`, exporting takes the following form (again, just use *Python trace* to figure out the proper form – that's the easiest way).

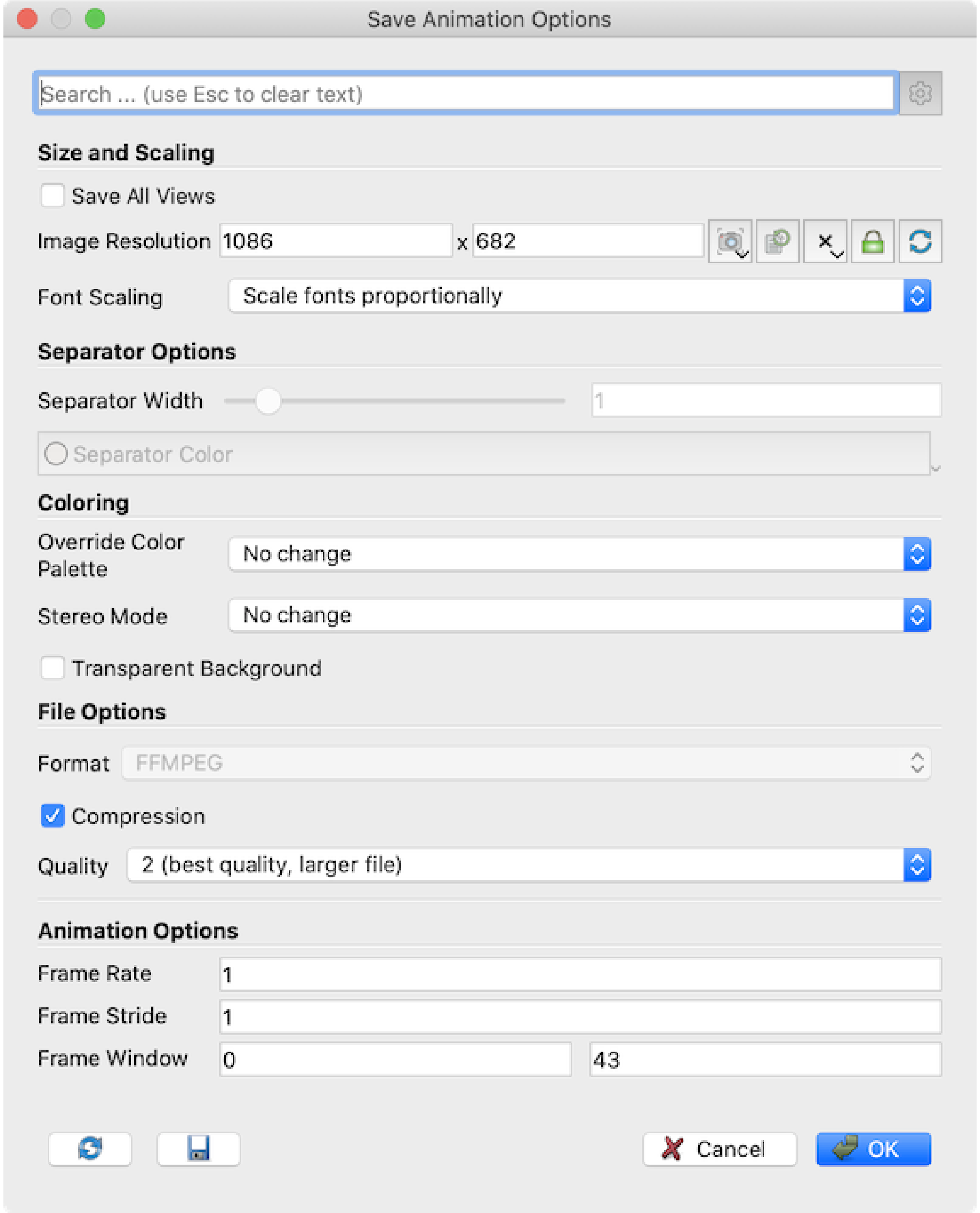
```
>>> myview = GetActiveView()
>>> ExportView('/tmp/sample.svg', view=myview,
               Plottitle='ParaView GL2PS Export',
               Compressoutputfile=1)
# the arguments after 'view' depend on the exporter selected.
```

## 1.8.3 Saving animation

To save an animation as a series of images or a video file, you use the *File > Save Animation* menu option. This raises a file save dialog where you choose where to save the file and which format to use. The available file formats include AVI (on Windows and Linux), MP4 (on Windows only), and Ogg video formats, as well as image formats such PNG, JPEG, and TIFF. If saving an animation as image frames, **ParaView** will generate a series of image files sequentially numbered using the frame number as a suffix to the specified filename.

After selecting the file and format, the `Save Animation Options` dialog (Fig. 1.126) is displayed. This dialog is nearly a clone of the `Save Screenshot Options` dialog (Fig. 1.124), including, optionally, the extra multiview options from Fig. 1.125, with additional format-specific compression options and a few animation-specific parameters. These are as follows:

- `Frame Rate`: When saving the animation as a video file (AVI or Ogg) rather than a series of images, this lets you specify the frame rate for the generated video. It has no effect when saving as a series of images.
- `Frame Stride`: Defines how many timesteps to jump forward after recording a screenshot of the current timestep.



Save Animation Options

Search ... (use Esc to clear text)

Size and Scaling

□ Save All Views

Image Resolution 1086 x 682

Font Scaling: Scale fonts proportionally

Separator Options

Separator Width 1

○ Separator Color

Coloring

Override Color Palette: No change

Stereo Mode: No change

□ Transparent Background

File Options

Format: FFMPEG

☑ Compression

Quality: 2 (best quality, larger file)

Animation Options

Frame Rate 1

Frame Stride 1

Frame Window 0 43

Fig. 1.126: The `Save Animation Options` dialog in `paraview`, which is used to customize saving of animation.

- `Frame Window`: If you didn't want to save out the full animation, instead limit to a specific window, you can use this to specify the range of frames to save. If you are generating a animation from a temporal dataset with timesteps, the frame generally corresponds to the timestep number.

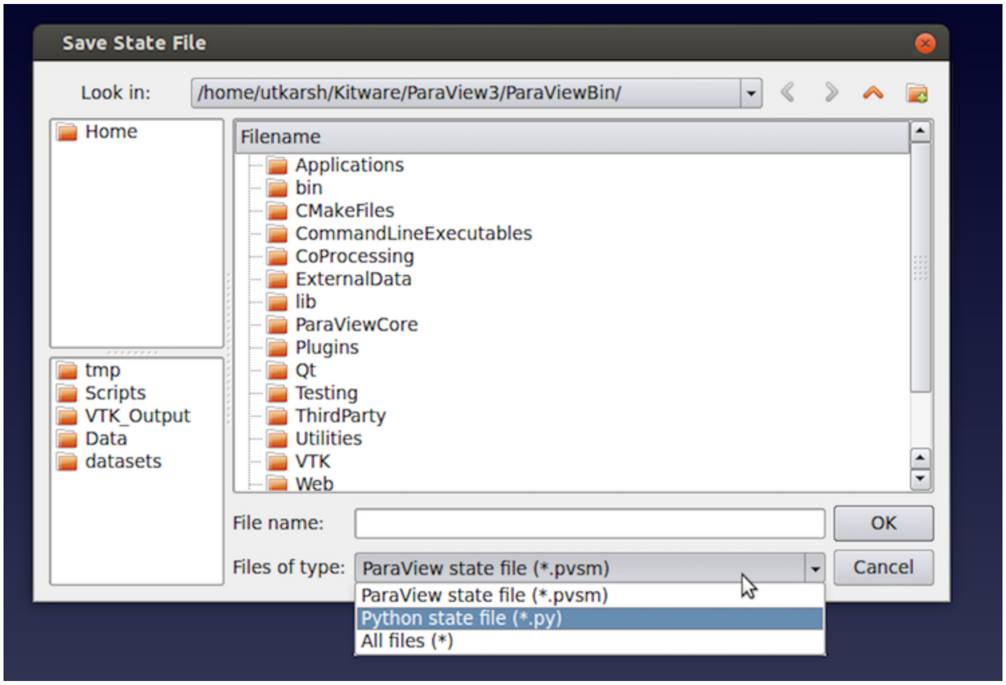
To save animations in `pvpython`, you use `SaveAnimation` . The arguments to this function are same as the `SaveScreenshot` with additional parameters for the animation specific options.

```
>>> SaveAnimation('animation.avi', GetActiveView(),
      FrameWindow = [1, 100],
      FrameRate = 1)
```

### 1.8.4 Saving state

Besides saving the results produced by your visualization setup, you can save the state of the visualization pipeline itself, including all the pipeline modules, views, their layout, and their properties. This is referred to as the `Application State` , or just `State` . In `paraview`, you can save the state using the *File > Save State...* menu option. Conversely, to load a saved state file, you can use *File > Load State...*.

There are two types of state files that you can save in `paraview`: *ParaView state file (*.pvsm)* and *Python state file (*.py)*. The PVSM files are XML-based text files that are human and machine readable, although not necessarily human friendly for a novice user. However, if you don't plan to read and make sense of the state files, PVSM is the most robust and reliable way to save the application state. For those who want to save the state and then modify it manually, using Python state files may be better, as using Python trace simply traces the actions that you perform in the UI as a Python script. Python state files, on the other hand, save the entire current state of the application as a Python script that you can use in `paraview` or the `Python Shell` .


Look in: /home/utkarsh/Kitware/ParaView3/ParaViewBin/

Files of type: ParaView state file (*.pvsm)

Fig. 1.127: The `Save State File` dialog in `paraview`.

To load a state file, you use the *File > Load State...* menu. Note that loading a state file will affect the current visualization state.

If you load a PVSM file this way you will be asked where to search for the data files. There are three available options:

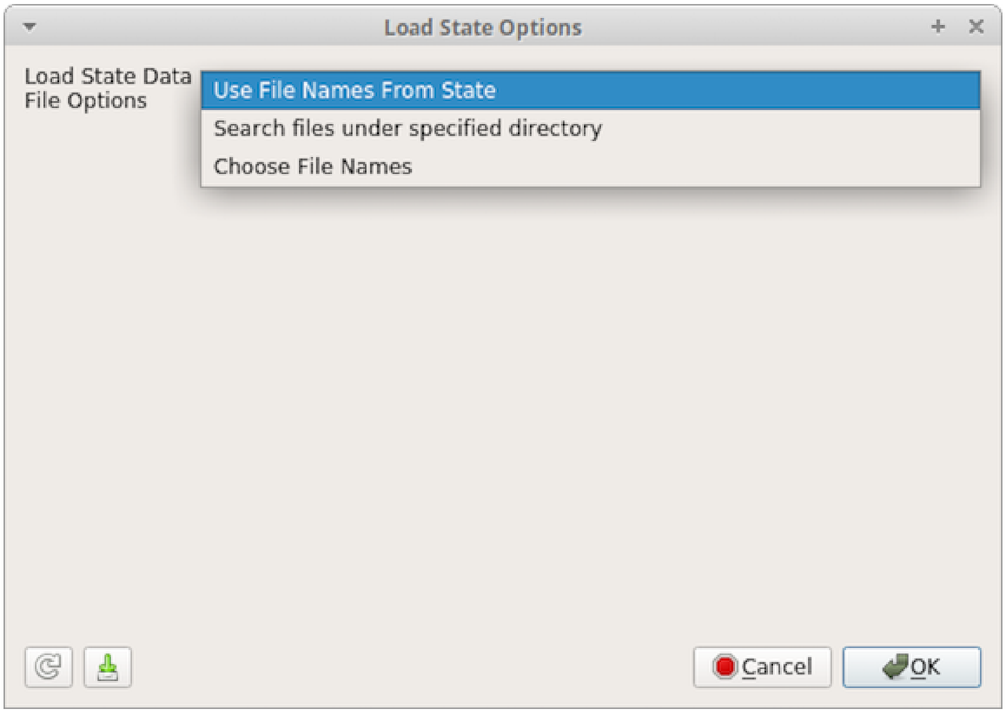

Load State Options
Load State Data File Options
Use File Names From State
Search files under specified directory
Choose File Names
Cancel
OK

Fig. 1.128: The `Load State Options` dialog in `paraview` showing the options for where to find data files.

`Use File Names From State` , `Search files under specified directory` and `Choose File Names` . If you select `Use File Names From State` then **ParaView** will look for the data at the absolute paths saved in the state file. If you select `Search files under specified directory` then you will see an option to browse for a directory that ParaView will search for the files before looking for them in the absolute path in the state file. This defaults to the location of the state file to make sharing state files between computers easier. If you select `Choose File Names` then you will be given a list of file names in the state file and can override each one individually.

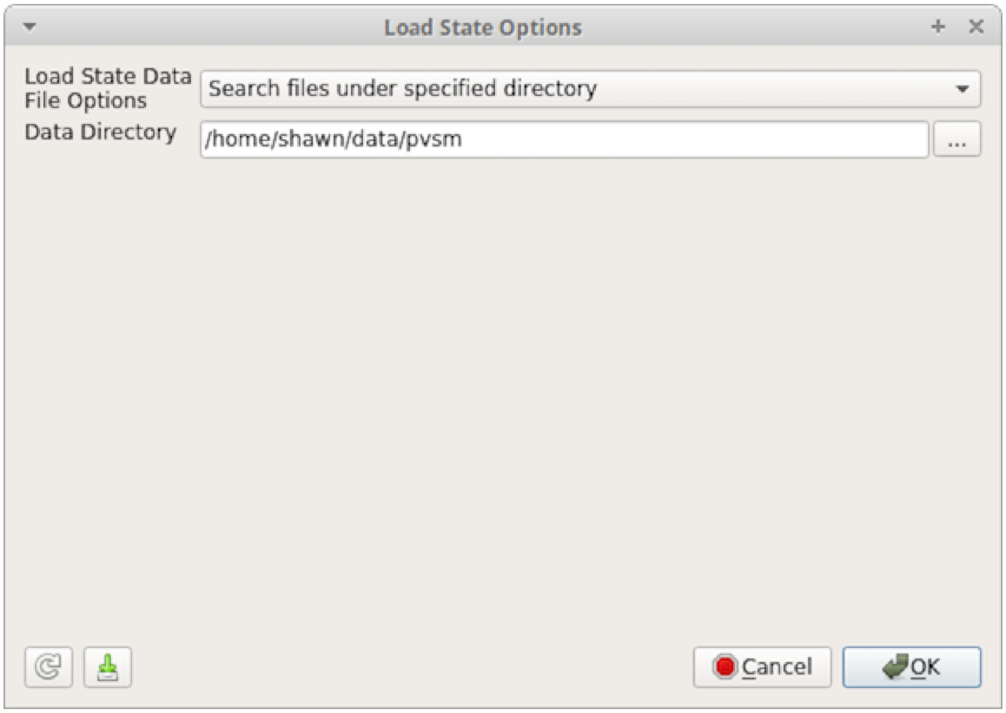
You can save/load the PVSM state file in `pvpython` as follows:

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

# Save the PVSM state file. Currently, this doesn't support
# saving Python state files.
>>> SaveState("sample.pvsm")

# To load a PVSM state file.
>>> LoadState("sample.pvsm")
```

To replace all data files used by state with those under a specific directory, you use the following form:

```
>>> LoadState("sample.pvsm",
              data_directory="[directory path]",

              # optionally, restrict to specified directory
              restrict_to_data_directory=True)
```

The function signature can become a little more complex if you want to explicitly override filenames used in the state file. It may be easier to use the Python trace capabilities to generate the function call for specific state files. It takes the following form:



Load State Data File Options: Search files under specified directory

Data Directory: /home/shawn/data/pvsm

Fig. 1.129: The Load State Options dialog in paraview showing the Search files under specified directory option.

```
>>> LoadState("sample.pvsm",
              filenames = [

                # a `dict` object for each reader in statefile to update.
                {
                  "name": "[reader name as shown in the pipeline browser]",

                  # if multiple readers have the same name, 'id' may be used
                  # instead of 'name' where the value is "id" used in the
                  # state file for this reader.

                  # filename properties and their overridden values for this
                  # reader, for example:
                  "FileName" : "foo.vtk",
                },

                # multiple such `dict`s can be specified.
              ])

# here's an example
>>> LoadState(statefile,
              filenames=[
                  {
                      'name' : 'can.ex2',
                      'FileName' : data_dir + 'can.ex2',
                  },
```

(continues on next page)

(continued from previous page)

```
                  {
                      'name' : 'dataset',
                      'FileName' : data_dir + 'disk_out_ref.ex2',
                  },
                  {
                      'name' : 'timeseries',
                      'FileName' : [ data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0000.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0001.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0002.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0003.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0004.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0005.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0006.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0007.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0008.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0009.vtp',
                                     data_dir + 'dualSphereAnimation/dualSphereAnimation_
↪P00T0010.vtp']
                  },
              ])
```

**Did you know? :class: tip**

Not all user interface settings are saved in ParaView state files. For example, the `Show Plane` and `Show Outline and Intersection Edges` settings in the Plane widget in the `Slice` and `Clip` filters are not saved because they are temporary interface state, not part of the data pipeline or display properties. end{didyouknow}

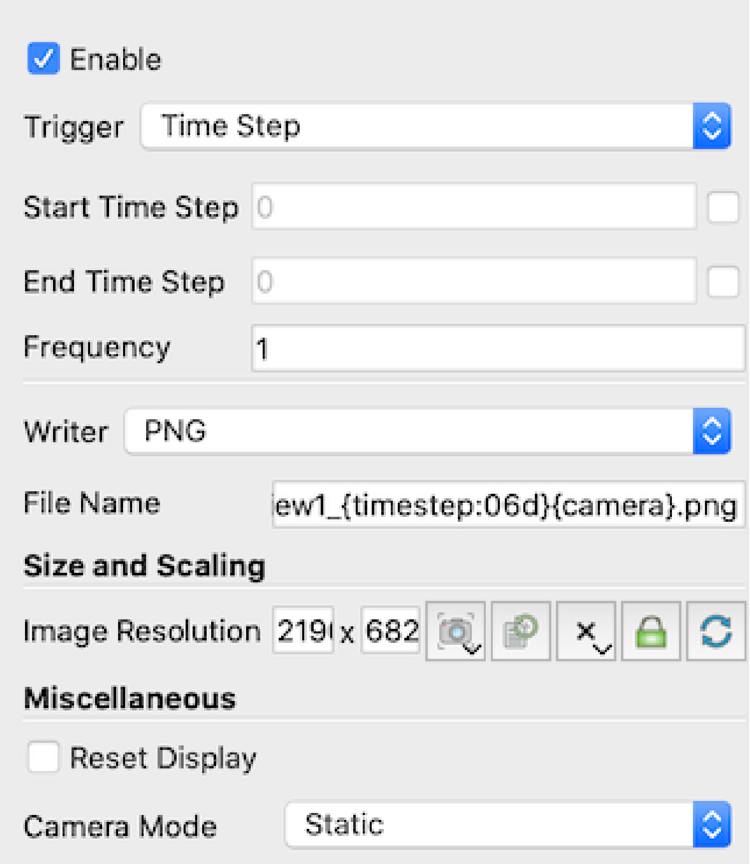
### 1.8.5 Extractors

Section 1.8.1 and Section 1.8.2 are two ways of saving datasets and images using actions, i.e., you click a button (or in Python, invoke a function) and the results are saved out immediately. If, for example, you now want to generate the results for another timestep, you have to repeat all the actions. One way to avoid this is to put together a Python script to generate the data and image files and then use that as a macro. An easier way is to use `extractors` . Extractors are a type of pipeline module, similar to sources and filters, but behave more like writers. Similar to filters, they have inputs. Unlike sources or filters, however, they produce no output that can be consumed by another pipeline module. Instead, when *activated*, they generate files – which we call extracts.

Since they are just another pipeline module, you use similar mechanisms as sources and filters for creating and configuring these. You use the *Extractors* menu to create them. The `Pipeline Browser` shows all the extractors present in the visualization. You select one of them by clicking on it in the `Pipeline Browser` at which point the `Properties` panel will update to show parameters on the selected extractor.

There are two types of extractors: data extractors and image extractors. The former generate files from datasets produced by sources and filters, while the latter save out rendering results from views. When created, a data extractor by default uses the active source as the input (similar to filters) which an image extractor uses the active view instead.

### Extractor Properties



☑ Enable

Trigger: Time Step

Start Time Step: 0 □

End Time Step: 0 □

Frequency: 1

Writer: PNG

File Name: ew1_{timestep:06d}{camera}.png

**Size and Scaling**

Image Resolution: 219( x 682

**Miscellaneous**

□ Reset Display

Camera Mode: Static

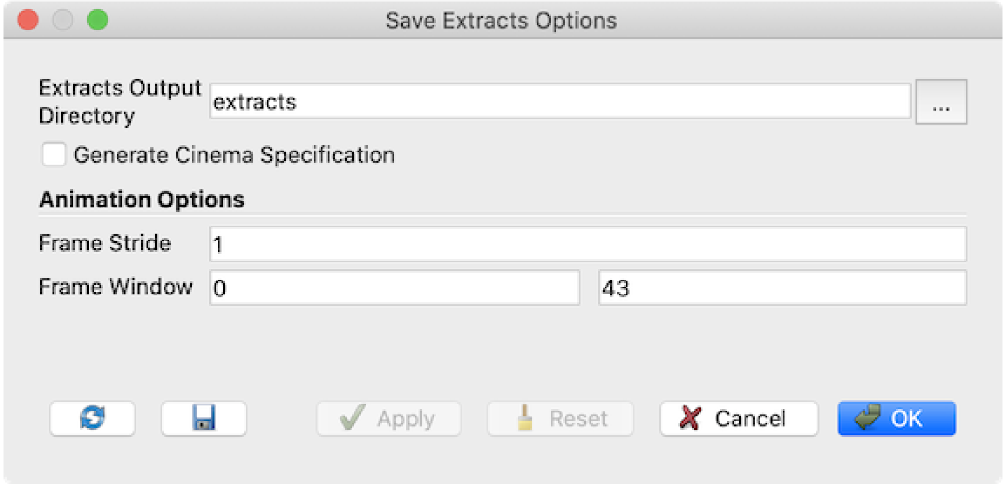
Fig. 1.130: Properties of an image extractor for PNG files.

You use the `Properties` panel view and change extractor properties. The available properties can be grouped into two major groups: first are `Trigger` properties which are common to all extractors, and the second are the `Writer` properties which are parameters specific to type of writer the extractor uses.

`Trigger` properties define when the extractor is activated i.e. under what conditions does the extractor produce extracts. Currently, we support time-based controls. You can select the `Start Time Step`, `End Time Step` or the `Frequency` at which to generate the results. `Frequency` is the number of timesteps per activation. To write every other timestep, set the `Frequency` to 2, to write every 3rd timestep, set it to 3, and so on.

`Writer` properties are specific to the writer. For data extractors, these will be similar to the writer properties shown in the `Configure Writer` dialog for the writer described in Section Section 1.8.1. For image extractors, they are similar to the `Save Screenshot Options` dialog described in Section Section 1.8.2. The `Writer` properties also lets you set a `File Name` . This is the file name to use to save the extracts. Since extractors are designed to generate a new extract every time they are activated, the `File Name` supports patterns that let you make the filename unique per activation. `{timestep}` or `{time}` in the filename are replaced by the timestep index and the time value for each activation. You add leading zeros (or other prefixes) to the numbers using a form such as `{timestep:06d}`. There the timestep will be padded with zeros if the number of digits is less than 6. You should not use absolute paths for specifying the filenames here. We will see how to select prefix to store these extracts under in the next section.

### Saving Extracts


Save Extracts Options

Extracts Output Directory: extracts

□ Generate Cinema Specification

Animation Options

Frame Stride: 1

Frame Window: 0 | 43

Fig. 1.131: `Save Extracts Options` dialog shown on *File* > *Save Extracts…*.

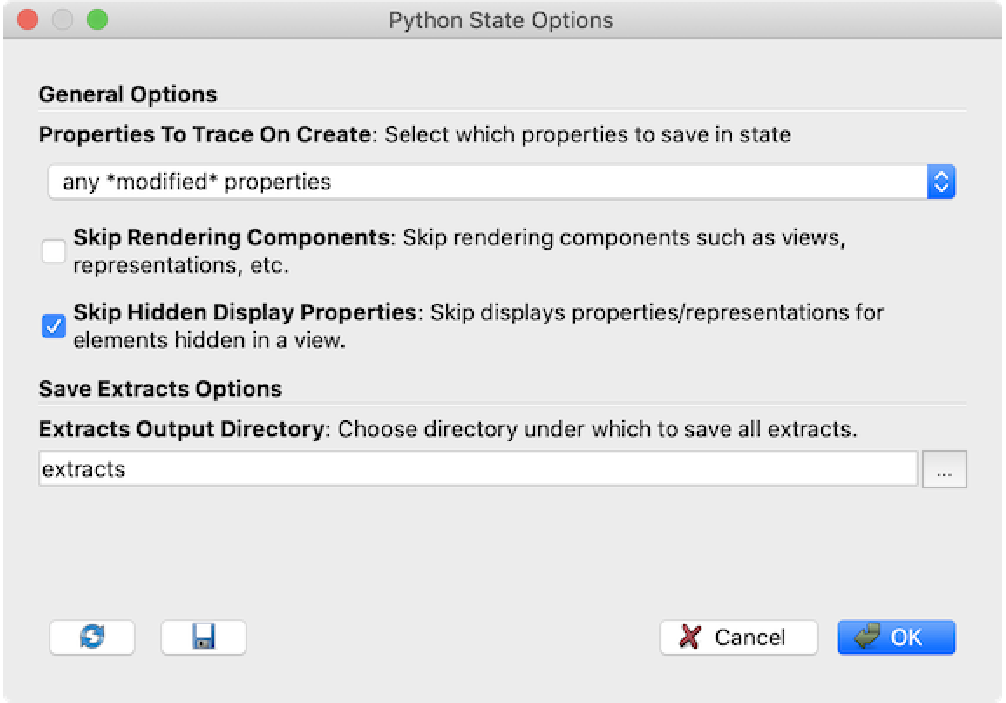
Once the extractors have been setup, you can trigger the saving of extract using *File* > *Save Extracts…*. This will pop up the `Save Extract Options` dialog which lets you configure the extract generation. `Extracts Output Directory` specifies the root directories under which all extracts are saved. Check `Generate Cinema Specification` to generate a *data.csv* file under the chosen extracts output directory that summarizes the generated extracts. This can be then used with viewers provided by the Cinema Science project to explore the generated extracts. `Frame Stride` and `Frame Window` control the timesteps for which extracts are generated.

Hit `Ok` and ParaView will animate through all timesteps (similar to using the `VCR Controls`), activating extractors based on their trigger criteria and then generating extracts. On successful completion you should have files under the chosen root directories.

In addition to generating extracts using the GUI, you can use `pvpython` or `pvbatch` to generate extract offline. Thus is especially handy for HPC use-cases; you can setup your state using an interactive session and once done save out the state and schedule a non-interactive job for the potentially time-consuming extract generation stage. To do so, setup your visualization pipeline including the extractors as normal. Then, instead of using `Save Extracts` , use *File* > *Save State…* and save out a Python state file. The `Python State Options` dialog has a section similar to `Save Extracts Options` dialog for choosing `Extracts Output Directory` and `Generate Cinema Specification`. Click `Ok` to save the Python script. The Python script has a section near the end of the end as follows:

```python
if __name__ == '__main__':
    # generate extracts
    SaveExtracts(ExtractsOutputDirectory='extracts')
```

This is what causes the Python script to save the extracts when the script is executed using `pvbatch` or `pvpython`.



General Options

**Properties To Trace On Create**: Select which properties to save in state

any *modified* properties

□ **Skip Rendering Components**: Skip rendering components such as views, representations, etc.

☑ **Skip Hidden Display Properties**: Skip displays properties/representations for elements hidden in a view.

Save Extracts Options

**Extracts Output Directory**: Choose directory under which to save all extracts.

extracts

Fig. 1.132: `Python State Options` dialog shown when saving a Python state file using *File* > *Save State*.

CHAPTER

TWO

# PARAVIEW REFERENCE MANUAL

## 2.1 Properties Panel

The Properties panel is perhaps the most often used panel in paraview. This is the panel you would use to change properties on modules in the visualization pipeline, including sources and filters, to control how they are displayed in views using the Display properties, and to customize the view itself. In this chapter, we take a closer look at the Properties panel to understand how it works.

### 2.1.1 Anatomy of the Properties panel

Before we start dissecting the Properties , remember that the Properties panel works on *active* objects, i.e., it shows the properties for the active source and active view, as well as the display properties, if any, for active source in the active view.

#### Buttons

Fig. 2.1 shows the various parts of the Properties panel. At the top is the group of buttons that let you accept, reject your changes to the panel or Delete the active source.

**Did you know?**

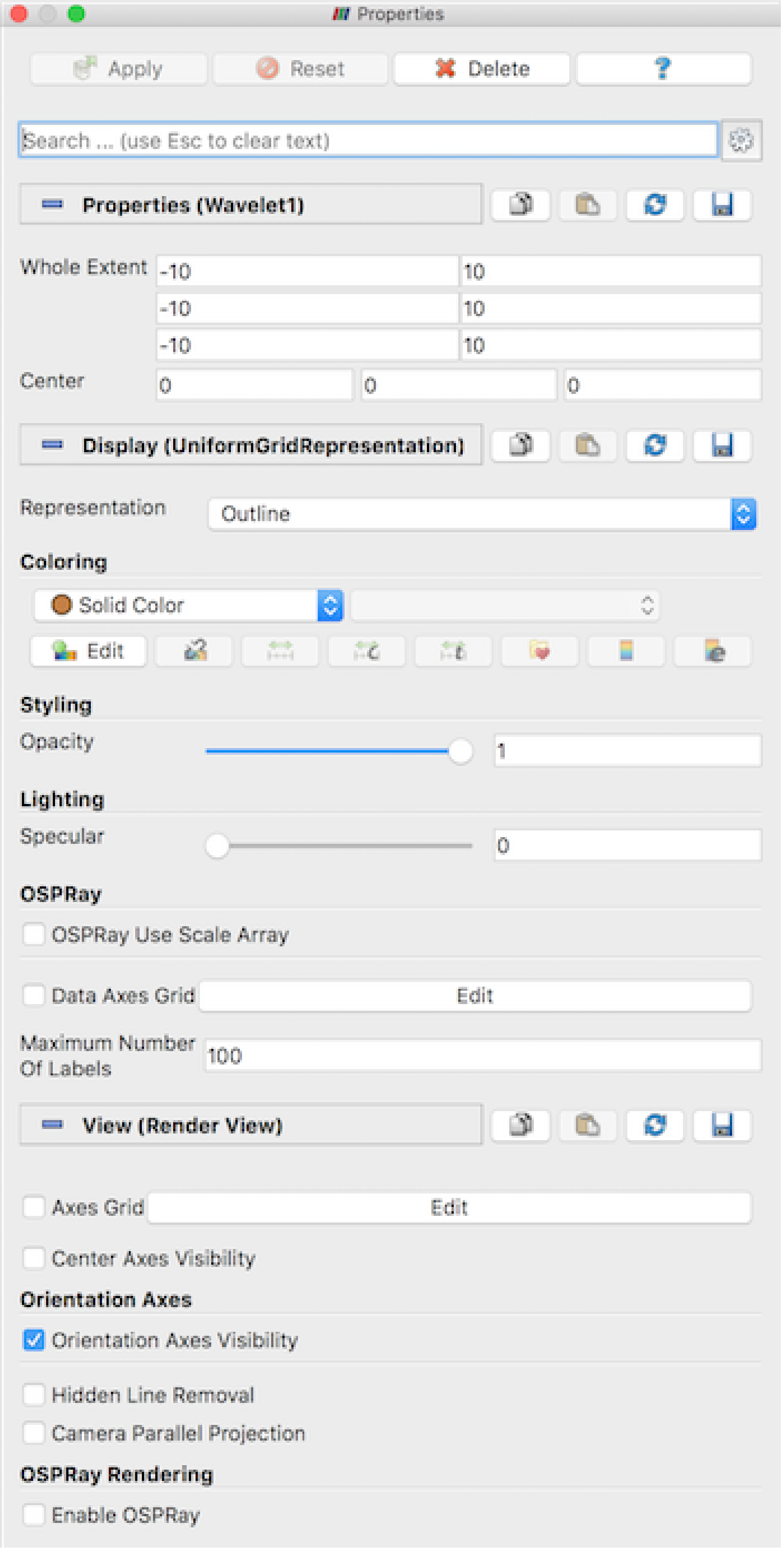
You can delete multiple sources by selecting them using the CTRL (or ) key when selecting in the Pipeline Browser and then clicking on the Delete button. Sometimes, the Delete button may be disabled. That happens when the selected source(s) have other filters connected to them. You will need to delete those filters first.

#### Search box

The Search box allows you to search for a property by using the name or the label for the property. Simply start typing text in the Search box, and the panel will update to show widgets for the properties matching the text.

The Properties panel has two modes that control the verbosity of the panel: default and advanced. In the default mode, a smaller subset of the available properties is shown. These are generally the frequently used properties for the pipeline modules. In advanced mode, you can see all the available properties. You can toggle between default and advanced modes using the  button next the Search box.

When you start searching for a property by typing text in the Search box, irrespective of the current mode of the panel (i.e., default or advanced), all properties that match the search text will be shown.



**Properties (Wavelet1)**

Whole Extent: -10 10 / -10 10 / -10 10

Center: 0 0 0

**Display (UniformGridRepresentation)**

Representation: Outline

**Coloring**

● Solid Color

Edit

**Styling**

Opacity: 1

**Lighting**

Specular: 0

**OSPRay**

□ OSPRay Use Scale Array

□ Data Axes Grid — Edit

Maximum Number Of Labels: 100

**View (Render View)**

□ Axes Grid — Edit

□ Center Axes Visibility

**Orientation Axes**

☑ Orientation Axes Visibility

□ Hidden Line Removal

□ Camera Parallel Projection

**OSPRay Rendering**

□ Enable OSPRay

Fig. 2.1: Properties Panel in paraview

**Did you know?**

The Search box is a recurring widget in paraview. Several other panels and dialog boxes, including the Settings dialog and the Color Map Editor , show a similar widget. Its behavior is also exactly the same as in the case of the Properties panel. Don't forget that the  button can be used to toggle between default and advanced modes.

### Properties

The Properties , Display , and View sections in the panel show widgets for properties on the active source, its display properties in the active view, and the properties for the active view, respectively. You can collapse/expand each of these sections by clicking on the section name.

To the right of each section name is a set of four buttons. Clicking the  copies the current set of property values to the clipboard while clicking the  will paste those property values into another compatible panel section. Note that the paste icon is enabled only for panel sections where the copied properties can be pasted.

The next two buttons  and  enable customizing the default values used for those properties. Refer to Section 2.12.2 to learn more about customizing default property values.

## 2.1.2 Customizing the layout

The Properties panel, by default, is set up to show the source, display, and view properties on the same panel. You may, however, prefer to have each of these sections in a separate dockable panel. You can indeed do so using the Settings dialog accessible from the *Edit* > *Settings* menu.

On the General tab search of the properties panel using the Search box, you should see the setting that lets you pick whether to combine all the sections in one (default), to separate out either the Display or View sections in a panel, or to create separate panels for each of the sections. You will need to restart paraview for this change to take effect. Also, since the Apply and Reset buttons only apply to the Properties section, they will only be shown in the dock panel that houses it.

# 2.2 Object Shading Properties

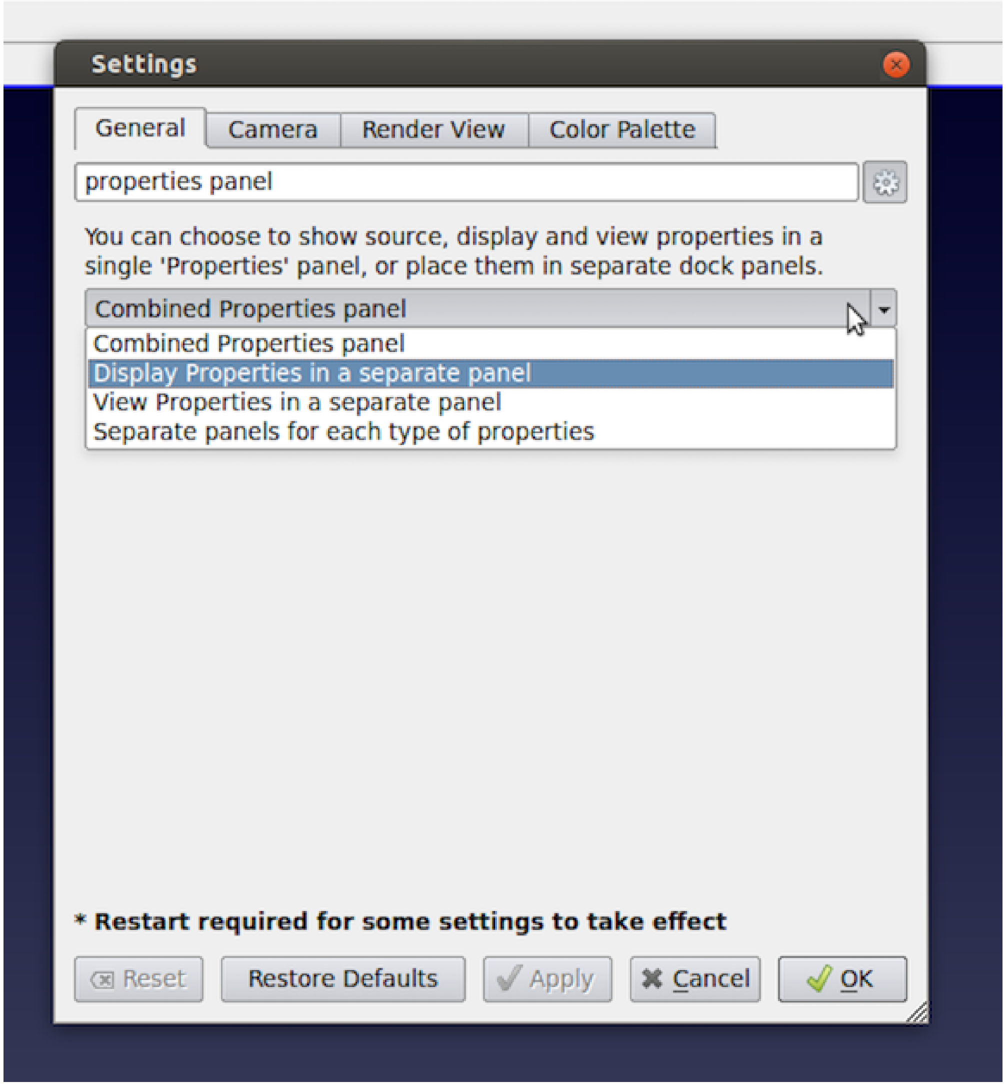
When visualizing any data it is important to control how the data is actually rendered. Most of the properties controlling the appearance of an object can be found in the Display Properties section in the object inspector (see section *display properties*). This chapter tries to entirely cover the lighting properties of an object when using the Surface representation.

## 2.2.1 Flat and Gouraud Lighting

These lighting models are intended to be easy-to-use basic models for general visualisation purposes. Both of these models are mainly controlled by the same parameters. The only difference between the two is that the flat shading model does not interpolate the normals of the rendered surfaces, meaning that a Gouraud lighting with no normal array will produce the same result as the flat shading model.

The parameters for these two models are :

- Specular: how specular the object is, that is how much of the light it will reflect.
- Specular Color: the color of the resulting specular.


Settings
General
Camera
Render View
Color Palette
Combined Properties panel
Display Properties in a separate panel
View Properties in a separate panel
Separate panels for each type of properties
Reset
Restore Defaults
Apply
Cancel
OK

properties panel

You can choose to show source, display and view properties in a single 'Properties' panel, or place them in separate dock panels.

Combined Properties panel

* Restart required for some settings to take effect

Fig. 2.2: Options for customizing the `Properties` panel layout using the `Settings` (left). View properties in a separate dock panel (right).

- Specular Power: how much the specular is spread on the object. The lower the wider it spreads.
- Luminosity: how much light the object emits. Only takes effect with a pathtracer backend.
- Ambient: coefficient for the ambient lighting.
- Diffuse: coefficient for the diffuse lighting.
- Texture Coordinates: the texture coordinates to use when applying a texture. This is needed to be able to use the *Texture* property.
- Texture: also called albedo, this is the perceived color of the object.
- Show Texture On Backface: whether or not to show the texture on the backface of the surface.
- Flip Texture: if on, flip the color texture.

Some other properties specific to the Gouraud shading :

- Normal Array: what array to use as the normal array
- Tangent Array: what array to use as the tangent array. Only needed when using the *Normal Texture*.
- Normal Texture: a normal map. This allows to have more precise shading without the need to subdivide the geometry. Stores the normal direction *{x,y,z}* in the RGB channels.
- Normal Scale: scale factor for the normal map.

### 2.2.2 PBR Lighting

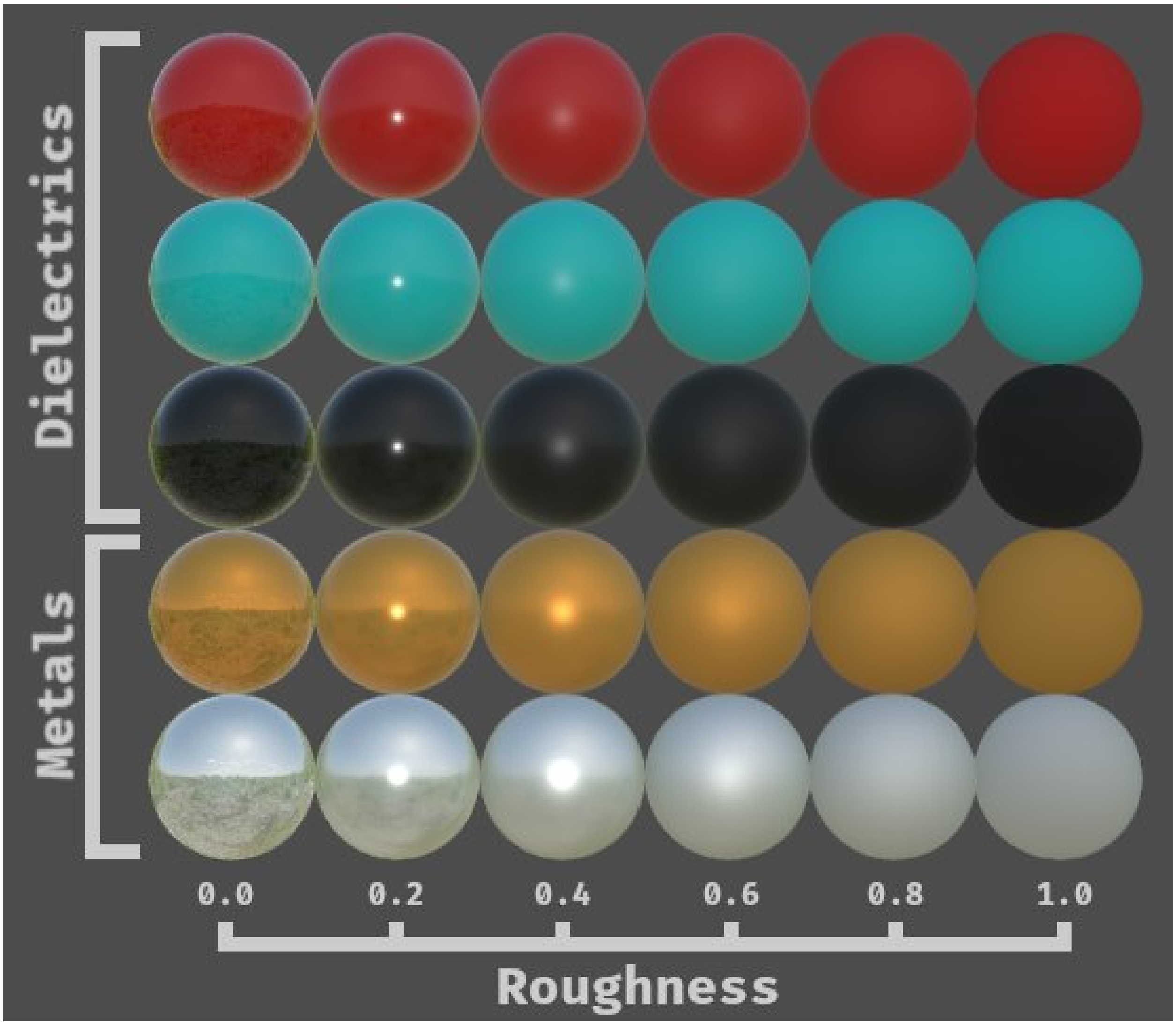
This lighting model is more complex than the previous ones but allows a wider range of effect on the objects and a more realistic rendering. Since this model is more complex to grasp, several blog posts have been written to explain how to use it (links to blogs are provided in the relevant sections below).

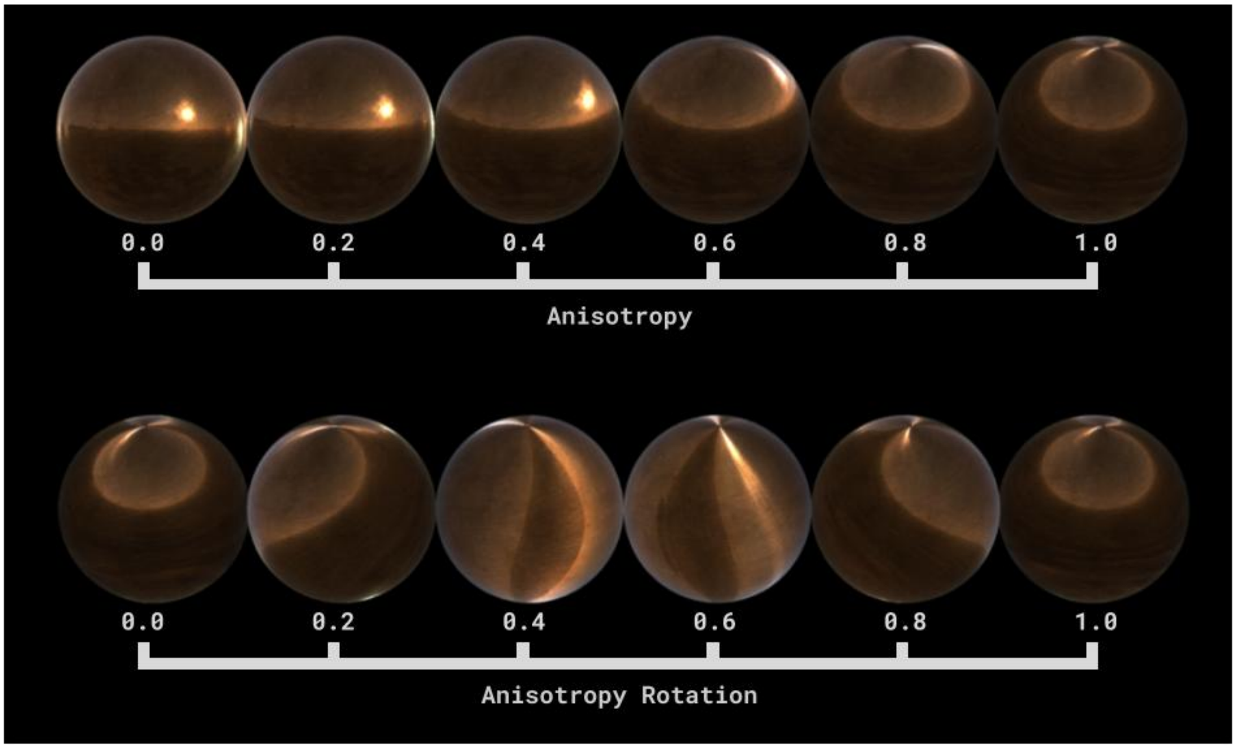
Basic parameters for this model are :

- Metallic: whether the object is metallic (= 1.0) or non-metallic / dielectric (= 0.0). For most materials, the value is either 0.0 or 1.0 but any value in between is valid.
- Roughness: parameter used to specify how glossy an object is.
- Luminosity: how much light the object emits. Only takes effect with a pathtracer backend.
- Diffuse: the global amount of light reflected by the object. Should usually be set to 1.0 as it is not an actual parameter of the theorical PBR shading model but it can be used for artistic effects.

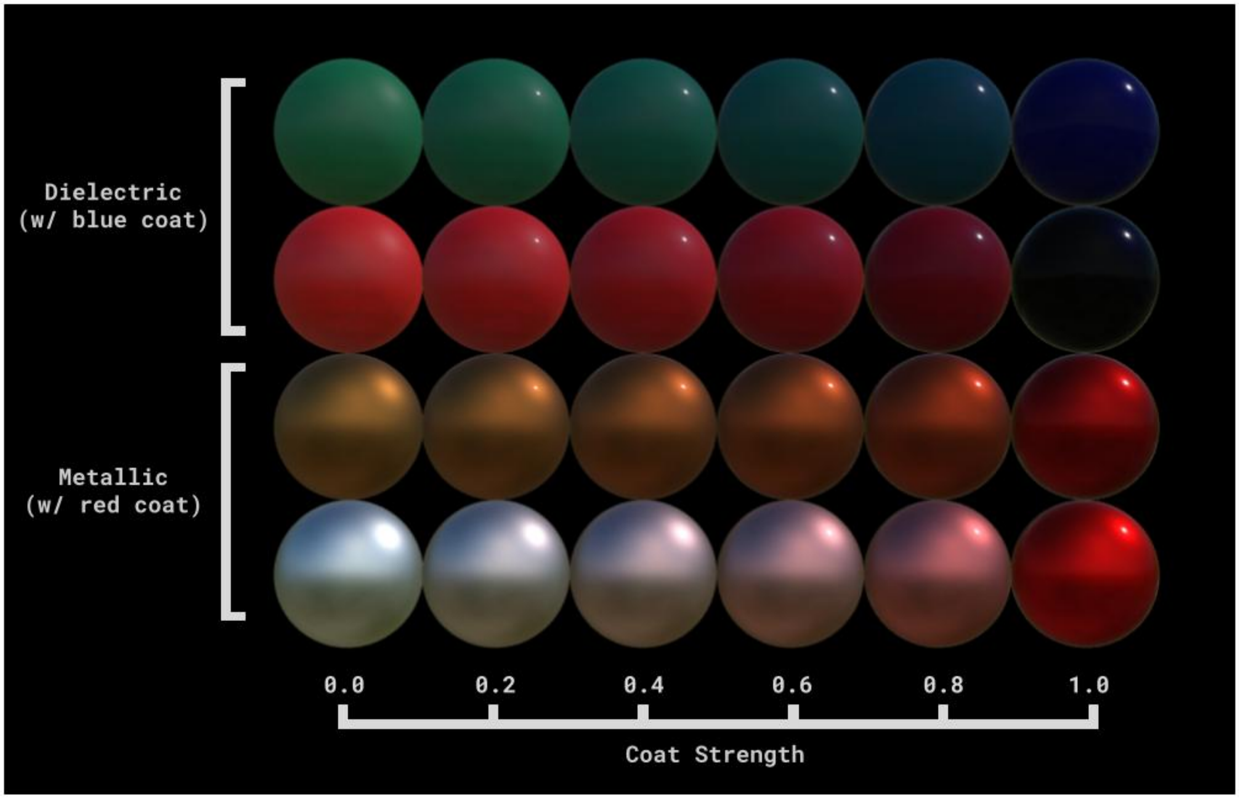
Following are more advanced properties. They can be hard to understand visually without knowing the theory behind it so it is advised to refer to the linked blog posts when using them for the first time.

- Anisotropy: the strength of the anisotropy (between 0.0 and 1.0). 0.0 means an isotropic material, 1.0 means a fully anisotropic material.
- Anisotropy Rotation: rotates the direction of the anisotropy (ie. the tangent) around the normal counter-clockwise (between 0.0 and 1.0). A value of 1.0 corresponds to a rotation of *2 * PI*.
  - anisotropy parameters are fully detailed here: anisotropy blog
- Coat Strength: the strength of the coat layer, between 0.0 and 1.0. 0 means no clear coat. This parameter can be considered as the thickness of the coating.
- Coat Roughness: the roughness of the coat layer.
- Coat Color: the color of the coat layer. Specular reflections on the coat layer are always white, but this parameter modifies the radiance that passes through it.
- Base IOR: the refractive index of the base layer.


Dielectrics
Metals
0.0
0.2
0.4
0.6
0.8
1.0
Roughness


0.0
0.2
0.4
0.6
0.8
1.0
Anisotropy
0.0
0.2
0.4
0.6
0.8
1.0
Anisotropy Rotation

- Coat IOR: the refractive index of the coat layer.
- Edge Tint: the color of the reflection on the edges at grazing angles for a metallic material.
  - coat parameters are fully detailed here: clear coat blog



Finally, some parameters can also be mapped using textures. As in the Gouraud shading model, a proper value for the *Texture Coordinates* and *Normal Array* properties will be needed in order to apply textures.

- Base Color Texture: also called albedo, this is the perceived color of the object, the diffuse color for non-metallic objects or the specular color for metallic objects.
- Normal Texture: a normal map. This allows to have more precise shading without the need to subdivide the geometry. Stores the normal direction *{x,y,z}* in the RGB channels.
- Material Texture: ambient Occlusion/Roughness/Metallic factors on the Red/Green/Blue channels. Also called ORM texture.
- Coat Normal Texture: a normal map for the clear coat layer.
- Anisotropy Texture: texture that controls the anisotropy strength on the red channel, while the green channel holds the anisotropy rotation. The blue channel is discarded.
- Emissive Texture: for light-emitting texture. Note that the material will not actually emit light, and the texture is ignored by OSPRay and NVidia pathtracing backends.

## 2.3 Color maps and transfer functions

One of the first things that any visualization tool user does when opening a new dataset and looking at the mesh is to color the mesh with some scalar data. Color mapping is a common visualization technique that maps data to color, and displays the colors in the rendered image. Of course, to map the data array to colors, we use a transfer function. A transfer function can also be used to map the data array to opacity for rendering translucent surfaces or for volume rendering. This chapter describes the basics of mapping data arrays to color and opacity.

### 2.3.1 The basics

Color mapping (which often also includes opacity mapping) goes by various names including scalar mapping and pseudo-coloring. The basic principle entails mapping data arrays to colors when rendering surface meshes or volumes. Since data arrays can have arbitrary values and types, you may want to define to which color a particular data value maps. This mapping is defined using what are called *color maps* or *transfer functions*. Since such mapping from data values to rendering primitives can be defined for not just colors, but opacity values as well, we will use the more generic term *transfer functions*.

Of course, there are cases when your data arrays indeed specify the red-green-blue color values to use when rendering (i.e., not using a transfer function at all). This can controlled using the `Map Scalars` display property. Refer to Chapter Section 1.4.3 for details. This chapter relates to cases when `Map Scalars` is enabled, i.e., when the transfer function is being used to map arrays to colors and/or opacity.

In **ParaView**, you can set up a transfer function for each data array for both color and opacity separately. **ParaView** associates a transfer function with the data array identified by its name. The same transfer function is used when coloring with the same array in different 3D views or results from different stages in the pipeline. You can also use Section 2.3.2 to have independant color map by array name and representation.

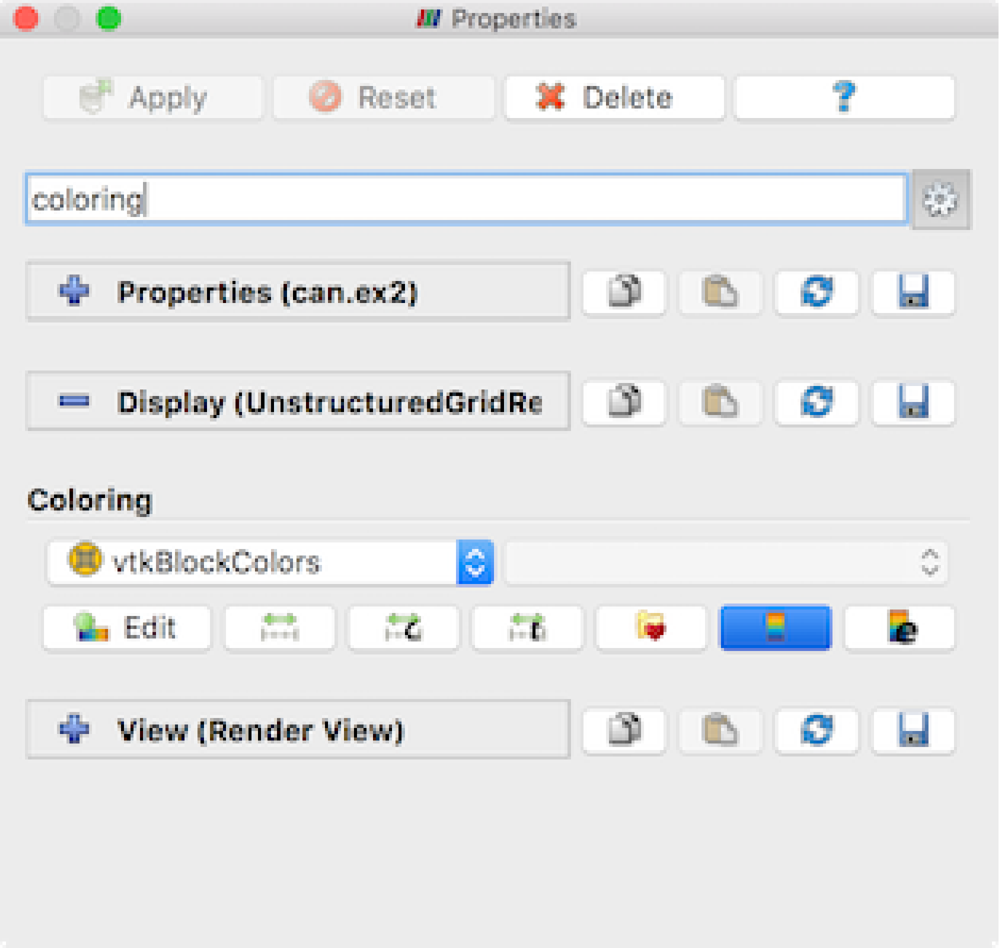
For arrays with more than one component, such as vectors or tensors, you can specify whether to use the magnitude or a specific component for the color/opacity mapping. Similar to the transfer functions themselves, this selection of how to map a multi-component array to colors is also associated with the array name. Thus, two pipeline modules being colored with the arrays that have the same name will not only be using the same transfer functions for opacity and color, but also the component/magnitude selection.

**Common Errors**

Beginners find it easy to forget that the transfer function is associated with an array name and, hence, are surprised when changing the transfer function for a dataset being shown in one view affects other views as well. Using different transfer functions for the same variable is discouraged by design in **ParaView**, since it can lead to the misinterpretation of values. If you want to use different transfer functions, despite this caveat, you can use the Separate Color Map feature (see Section 2.3.2).

There are separate transfer functions for color and opacity. The opacity transfer function is used for volume rendering, and it is optional when used for surface renderings.

### Color mapping in paraview



You can pick an array to use for color mapping, using either the Properties panel or the Active Variables Controls toolbar. You first select the array with which to color and then select the component or magnitude for multi-component arrays. **ParaView** will either use an existing transfer function or create a new one for the selected array.

### Color mapping in pvpython

Here's a sample script for coloring using a data array from the disk_out_ref.ex2 dataset.

```
from paraview.simple import *

# create a new 'ExodusIIReader'
reader = ExodusIIReader(FileName=['disk_out_ref.ex2'])
reader.PointVariables = ['V']
reader.ElementBlocks = ['Unnamed block ID: 1 Type: HEX8']

# show data in view
display = Show(reader)

# set scalar coloring
```

(continues on next page)

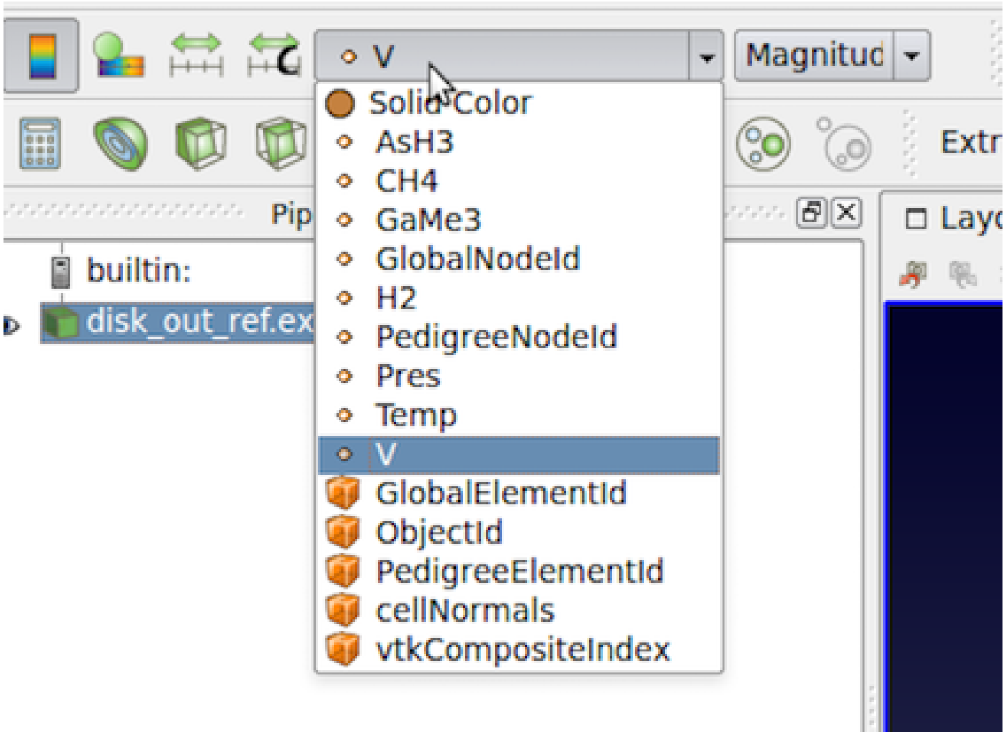


Fig. 2.3: The controls used for selecting the array to color within the `Properties` panel (top) and the `Active Variables Controls` toolbar (bottom).

(continued from previous page)

```
ColorBy(display, ('POINTS', 'V'))

# rescale color and/or opacity maps used to include current data range
display.RescaleTransferFunctionToDataRange(True)
```

The `ColorBy` function provided by the `simple` module ensures that the color and opacity transfer functions are set up correctly for the selected array, which is using an existing one already associated with the array name or is creating a new one. Passing `None` as the second argument to `ColorBy` will display scalar coloring.

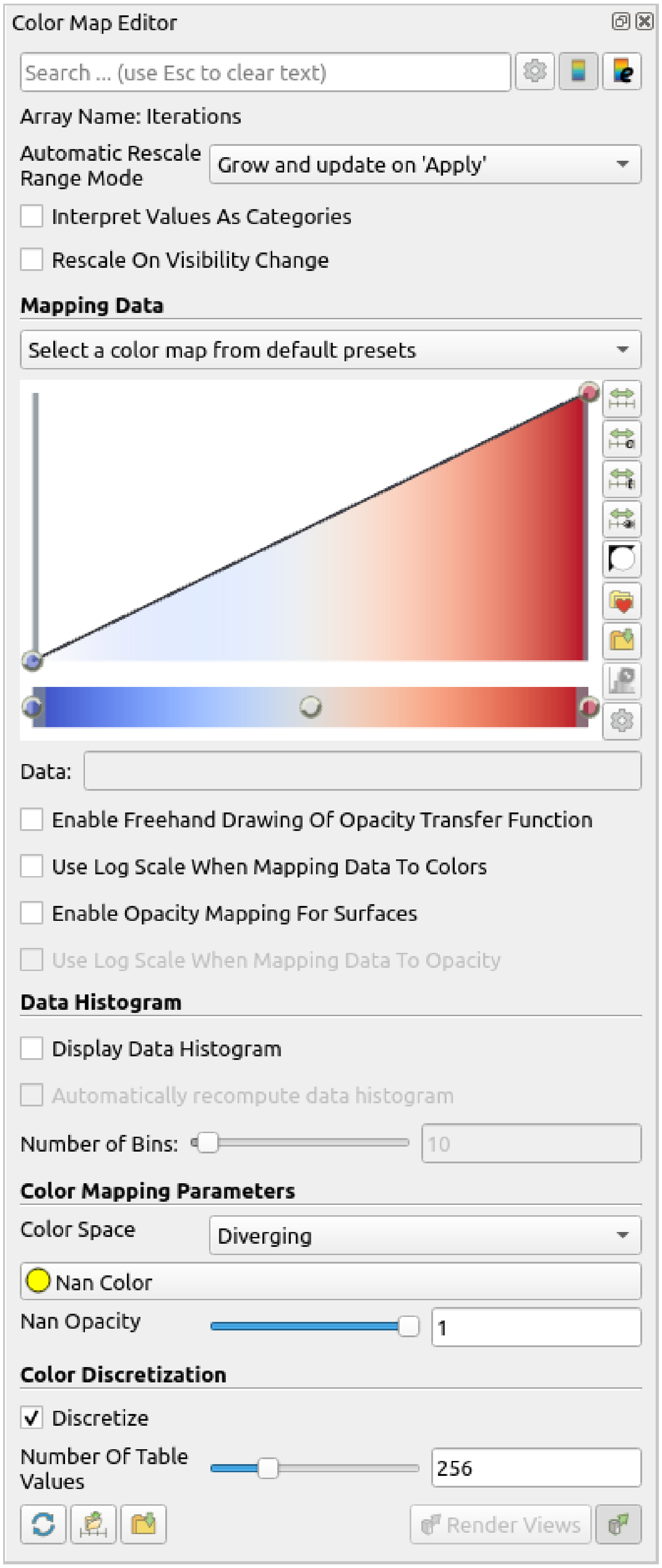
## 2.3.2 Editing the transfer functions in `paraview`

In `paraview`, you use the `Color Map Editor` to customize the color and opacity transfer functions. You can toggle the `Color Map Editor` visibility using the *View > Color Map Editor* menu option.

As shown in Fig. 2.4, the panel follows a layout similar to the `Properties` panel. The panel shows the properties for the transfer function, if any, used for coloring the active data source (or filter) in the active view. If the active source if not visible in the active view, or is not employing scalar coloring, then the panel will be empty.

Similar to the Properties panel, by default, the commonly used properties are shown. You can toggle the visibility of advanced properties by using the  button. Additionally, you can search for a particular property by typing its name in the `Search` box.

Whenever the transfer function is changed, we need to re-render, which may be time consuming. By default, the panel requests a render on every change. To avoid this, you can toggle the  button. When unchecked, you will need to manually update the panel using the `Render Views` button.



Color Map Editor

Search ... (use Esc to clear text)

Array Name: Iterations

Automatic Rescale Range Mode: Grow and update on 'Apply'

□ Interpret Values As Categories

□ Rescale On Visibility Change

**Mapping Data**

Select a color map from default presets

Data:

□ Enable Freehand Drawing Of Opacity Transfer Function

□ Use Log Scale When Mapping Data To Colors

□ Enable Opacity Mapping For Surfaces

□ Use Log Scale When Mapping Data To Opacity

**Data Histogram**

□ Display Data Histogram

□ Automatically recompute data histogram

Number of Bins: 10

**Color Mapping Parameters**

Color Space: Diverging

Nan Color

Nan Opacity: 1

**Color Discretization**

☑ Discretize

Number Of Table Values: 256

Fig. 2.4: `Color Map Editor` panel in `paraview` showing the major components of the panel.

The  button restores the application default settings for the current color map.

The  and  buttons save the current color and opacity transfer function, with all its properties, as the default transfer function. **ParaView** will use it next time it needs to set up a transfer function to color a new data array. The  button saves the transfer function as default for an array of the same name while the  button saves the transfer function as default for all arrays. Note that this will not affect transfer functions already setup. Also this is saved across sessions, so **ParaView** will remember this even after restart.

### Separate Color Map

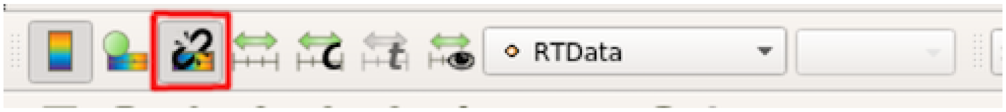


Fig. 2.5: The Separate Color Map button

In order to force **ParaView** to use a separate color map on the current Active Representation, click on the button shown in Fig. 2.5. A separate color map is not shared across representations by name, but is instead uniquely associated with the array name and the representation.

This can also easily be done in Python:

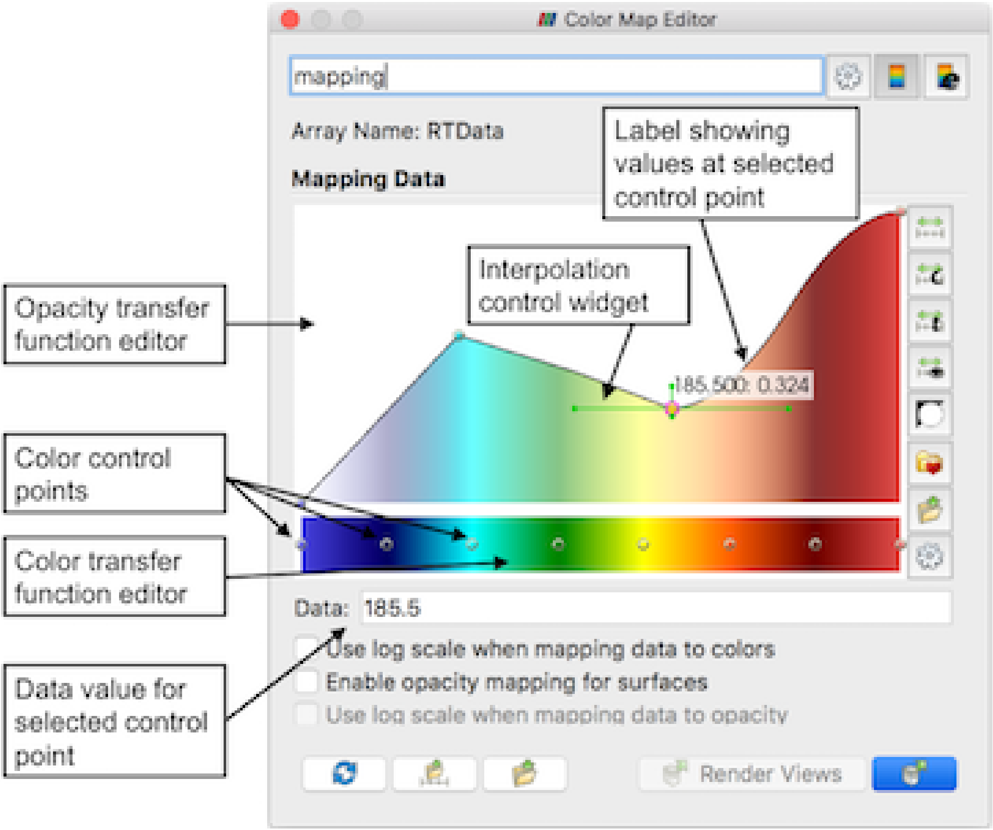
```python
from paraview.simple import *

Wavelet()
wavelet1Display = Show()
wavelet1Display.SetRepresentationType('Surface')

# set scalar coloring
ColorBy(wavelet1Display, 'RTData')

# set the usage of a Separate Color Map
wavelet1Display.UseSeparateColorMap = True

# or use the ColorBy interface directly
ColorBy(wavelet1Display, 'RTData', separate = True)

# display the same data in another view for comparison with different color map
# get layout
layout1 = GetLayout()

# split cell
layout1.SplitHorizontal(0, 0.5)

renderView1 = GetActiveView()

# Create a new 'Render View'
renderView2 = CreateView('RenderView')

# place view in the layout
layout1.AssignView(2, renderView2)
```

(continues on next page)

(continued from previous page)

```
# set active view
SetActiveView(renderView2)

wavelet2Display = Show()
wavelet2Display.SetRepresentationType('Surface')

# Use the ColorBy interface to create a separated color map
ColorBy(wavelet2Display, 'RTData', separate = True)

# get separate color transfer function/color map for 'RTData'
separate_wavelet2Display_RTDataLUT = GetColorTransferFunction('RTData', wavelet2Display, separate=True)

# Apply a preset using its name.
separate_wavelet2Display_RTDataLUT.ApplyPreset('Cold and Hot', True)

ResetCamera(renderView1)
ResetCamera(renderView2)
RenderAllViews()
```

### Mapping data


Color Map Editor
Label showing values at selected control point
Interpolation control widget
185.500: 0.324
Opacity transfer function editor
Color control points
Color transfer function editor
Data value for selected control point
Render Views

mapping

Array Name: RTData

Mapping Data

Data: 185.5

□ Use log scale when mapping data to colors

□ Enable opacity mapping for surfaces

□ Use log scale when mapping data to opacity

Fig. 2.6: Transfer function editor and related properties

The Mapping Data group of properties controls how the data is mapped to colors or opacity. The transfer function editor widgets are used to control the transfer function for color and opacity. The panel always shows both the transfer functions. Whether the opacity transfer function gets used depends on several things:

- When doing surface mesh rendering, it will be used only if Enable opacity mapping for surfaces is checked
- When doing volume rendering, the opacity mapping will always be used.

To map the data to color using a log scale, rather than a linear scale, check the Use log scale when mapping data to colors . It is assumed that the data is in the non-zero, positive range. **ParaView** will report errors and try to automatically fix the range if it is ever invalid for log mapping.

The range of a color map is a very important property that controls the mapping of data values to colors. The range can be automatically updated in a number of situations for convenience. How the range is updated is controlled by the Automatic Rescale Range Mode property in the Color Map Editor . When Never is selected, the data range will never be updated automatically. When Grow and update on 'Apply' is selected, **ParaView** will grow the color/opacity map range to include the current data range every time you hit Apply on the Properties panel. Thus, when the data range changes, if the timestep is changed, the color/opacity map range won't be affected. To grow the range on change in timestep as well, use the Grow and update every timestep option. Now the range will be updated on Apply as well as when the timestep changes. *Grow* indicates that the color/opacity map range will only be increased, never shrunk, to include the current data range. If you want the range to match the current data range exactly, then you should use the Clamp and update every timestep option. Now the range will be clamped to the exact data range each time you hit Apply on the Properties panel or when the timestep changes. The initial value for the Automatic Rescale Range Mode is controlled by the General setting Transfer Function Reset Mode in the Settings dialog (see Section 2.12.1 ).

### Transfer function editor

Using the transfer function editors is pretty straightforward. Control points in the opacity editor widget and the color editor widget are independent of each other. To select a control point, click on it. When selected, the control point is highlighted with a red circle and data value associated with the control point is shown in the Data input box under the widget. Clicking in an empty area will add a control point at that location. To move a control point, click on the control point and drag it. You can fine tune the data value associated with the selected control point using the Data input box. To delete a control point, select the control point and then type the Delete key. Note that the mouse pointer should be within the transfer function widget. While the end control points cannot be moved or deleted, if you drag the bars at either end, you can change the range of the transfer function. You can also rescale the entire transfer function to move the control points, as is explained later.

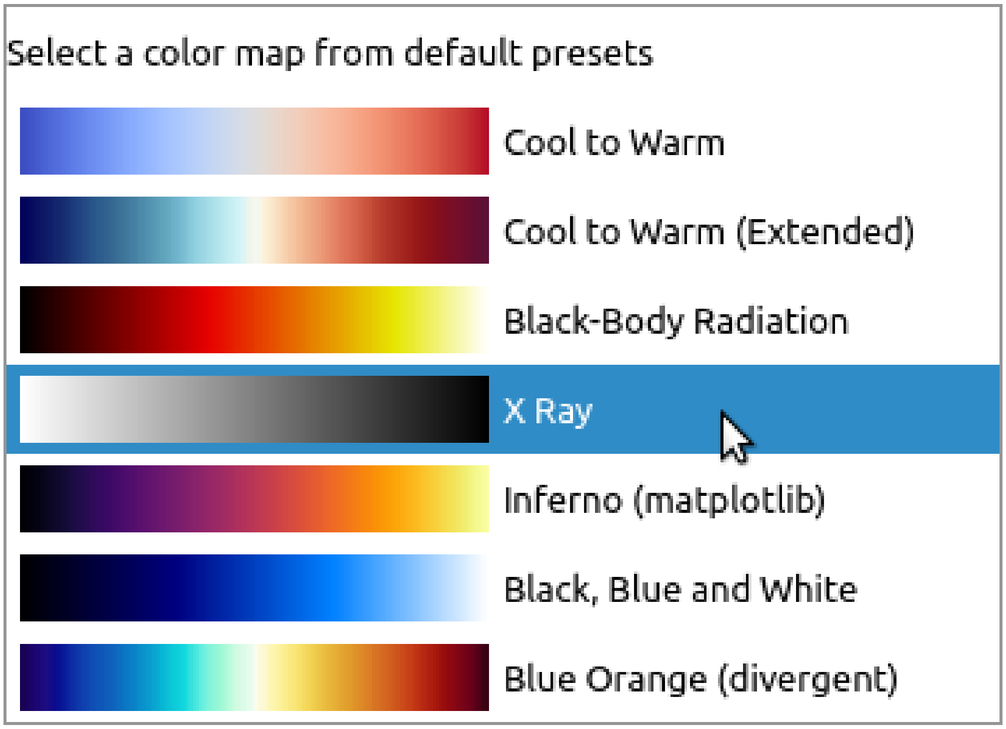
In the opacity transfer function widget, you can move the control points vertically to control the opacity value associated with that control point. In the color transfer function widget, you can select the control point and then type the Enter or Return key to pop up a color chooser dialog to set the color associated with that control point.

The opacity transfer function widget also offers some control over the interpolation between the control points. Double click on a control point to show the interpolation control widget, which allows for changing the sharpness and midpoint that affect the interpolation. Click and drag the control handles to see the change in interpolation.

The combo box at the top of the transfer function editor is used to quickly switch between the "Default" presets. Which presets are default ones can be configured from the Color Preset manager, which can be accessed with the Favorites button  described later.

The several control buttons on the right side of the transfer function widgets support the following actions:

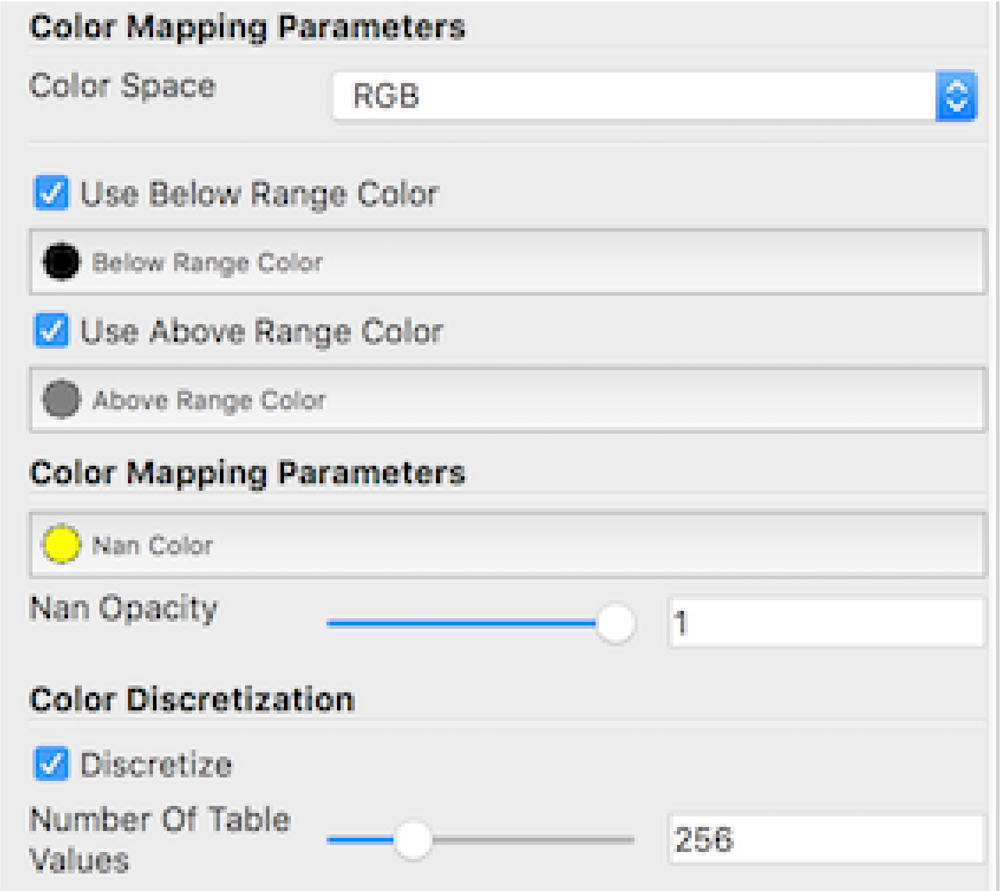
- : Rescales the color and opacity transfer functions using the data range from the data source selected in the Pipeline browser, i.e., the active source. This rescales the entire transfer function. Thus, all control points including the intermediate ones are proportionally adjusted to fit the new range.



Select a color map from default presets

- : Rescales the color and opacity transfer functions using a range provided by the user. A dialog will be popped up for the user to enter the custom range.
- : Rescales the color and opacity transfer functions to the range of values for data over all timesteps. This operation may be costly as data for all timesteps needs to be read.
- : Rescales the color and opacity transfer functions using the range of values for the elements (cells or points) visible in the view. This operations assigns the entire range of colors to visible elements which may reveal patterns not visible otherwise.
- : Inverts the color transfer function by moving the control points, e.g,. a red-to-green transfer function will be inverted to a green-to-red one. This only affects the color transfer function and leaves the opacity transfer function untouched.
- : Loads the color transfer function from a preset. The `Color Preset` manager dialog pops up to enable you to choose one of the color maps included with ParaView or import presets from a file.
- : Saves the current color transfer function to presets. The `Color Preset` manager dialog pops up to let you name the transfer function and export the transfer function to a file. The opacity function can also be saved with the transfer function. The preset will be added under the `Default` and `User` groups.
- : This toggles the detailed view for the transfer function control points. This is useful to manually enter values for the control points rather than using the UI.

### Color mapping parameters



Color Mapping Parameters

Color Space: RGB

☑ Use Below Range Color

● Below Range Color

☑ Use Above Range Color

● Above Range Color

Color Mapping Parameters

● Nan Color

Nan Opacity: 1

Color Discretization

☑ Discretize

Number Of Table Values: 256

Fig. 2.7: Color Mapping Parameters, including advanced properties. Advanced properties are enabled by clicking the gear icon at the top right of the `Color Map Editor` (not shown).}

The `Color Mapping Parameters` group of properties provides additional control over the color transfer function, including control over the color interpolation space, which is either RGB, HSV, Lab, Diverging, or Lab/CIEDE2000. To color data values falling below or above the range of the color map with special colors, enable the advanced `Use Below Range Color` and `Use Above Range Color` options, respectively. You can choose different colors for data falling on either side of the range. When color mapping floating point arrays with NaNs, you can select the color and opacity to use for NaN values. You can also affect whether the color transfer function uses smooth interpolation or discretizes the map into a fixed number of colors.

## 2.3.3 Editing the transfer functions in `pvpython`

In `pvpython`, you control the transfer functions by getting access to the transfer function objects and then changing properties on those. The following script shows how you can access transfer functions objects.

```python
from paraview.simple import *

# You can access the color and opacity transfer functions
# for a particular array as follows. These functions will
# create new transfer functions if none exist.
```

(continues on next page)

(continued from previous page)

```
# The argument is the array name used to locate the transfer
# functions.
>>> colorMap = GetColorTransferFunction('Temp')
>>> opacityMap = GetOpacityTransferFunction('Temp')
```

Once you have access to the color and opacity transfer functions, you can change properties on these similar to other sources, views, etc. Using the Python tracing capabilities to discover this API is highly recommended.

```
# Rescale transfer functions to a specific range
>>> colorMap.RescaleTransferFunction(1.0, 19.9495)
>>> opacityMap.RescaleTransferFunction(1.0, 19.9495)

# Invert the color map.
>>> colorMap.InvertTransferFunction()

# Map color map to log-scale preserving relative positions for
# control points
>>> colorMap.MapControlPointsToLogSpace()
>>> colorMap.UseLogScale = 1

# Return back to linear space.
>>> colorMap.MapControlPointsToLinearSpace()
>>> colorMap.UseLogScale = 0

# Change using of opacity mapping for surfaces
>>> colorMap.EnableOpacityMapping = 1

# Explicitly specify color map control points
# The value is a flattened list of tuples
# (data-value, red, green, blue). The color components
# must be in the range [0.0, 1.0]
>>> colorMap.RGBPoints = [1.0, 0.705, 0.015, 0.149,
                          5.0, 0.865, 0.865, 0.865,
                         10.0, 0.627, 0.749, 1.0,
                         19.9495, 0.231373, 0.298039, 0.752941]

# Similarly, for opacity map. The value here is
# a flattened list of (data-value, opacity, mid-point, sharpness)
>>> opacity.Points = [1.0, 0.0, 0.5, 0.0,
                      9.0, 0.404, 0.5, 0.0,
                     19.9495, 1.0, 0.5, 0.0]

# Note, in both these cases the controls points are assumed to be sorted
# based on  the data values. Also, not setting the first and last
# control point to have same data value can have unexpected artifacts
# in the 'Color Map Editor' panel.
```

Oftentimes, you want to rescale the color and opacity maps to fit the current data ranges. You can do this as follows:

```
>>> source = GetActiveSource()

# Update the pipeline, if it hasn't been updated already.
```

(continues on next page)

(continued from previous page)

```
>>> source.UpdatePipeline()

# First, locate the display properties for the source of interest.
>>> display = GetDisplayProperties()

# Reset the color and opacity maps currently used by 'display' to
# use the range for the array 'display' is using for color mapping.
# This requires that the 'display' has been set to use scalar coloring
# using an array that is available in the data generated. If not, you will
# get errors.
>>> display.RescaleTransferFunctionToDataRange()
```

### 2.3.4 Color legend

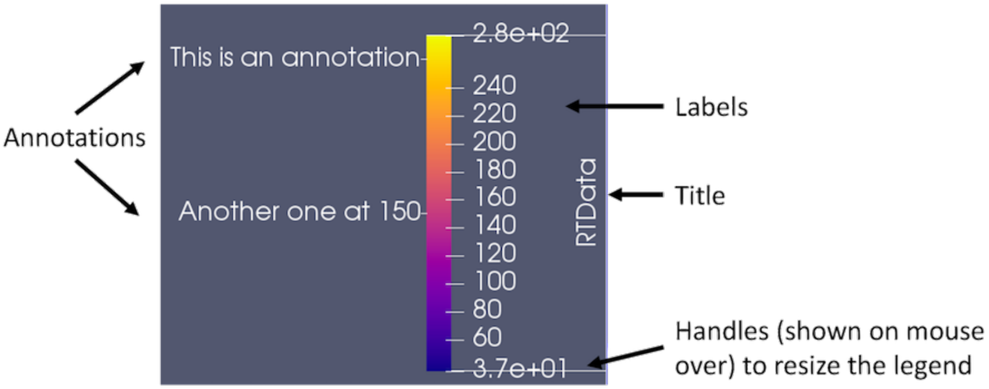


Fig. 2.8: Color legend in **ParaView**.

The color legend, also known as scalar bar or color bar, is designed to provide the user information about which color corresponds to what data value in the rendered view. You can toggle the visibility of the color legend corresponding to the transfer function being shown/edit in the `Color Map Editor` by using the  button near the top of the panel. This button affects the visibility of the legend in the active view.

Fig. 2.8 shows the various components of the color legend. By default, the title is typically the name of the array (and component number or magnitude for non-scalar data arrays) being mapped. Automatically generated labels appear on one side of the color legend, while on the other side are annotations, optionally including start and end annotations depicting the minimum and maximum of the color legend range.

The color legend can be manipulated with the mouse. You can click and drag the legend to place it at any position in the view. Additionally, you can change the length of the legend by clicking and dragging the end-markers shown when you hover the mouse pointer over the legend.

### Color legend parameters

You can edit the color legend parameters by clicking on the  button on the `Color Map Editor` panel. This will pop up the `Edit Color Legend Properties` dialog that shows the available parameters. Any changes made will affect only the particular color legend in the active view.

The first few options in the `Edit Color Legend Properties` control the orientation and location of the color legend in the render view. `Auto Orient` turns on automatic determination of the color legend's orientation. The color legend will change orientation to horizontal when it is dragged to the bottom or the top of the render view, and it will change to vertical when it is dragged to the left or right side. When disabled, you can choose the orientation you want the color legend to have by choosing an option in the `Orientation` combo-box. The `Window Location` option controls the location of the color legend in the window; if the value `AnyLocation` is selected, then the color legend will not be forced into any particular position. The color legend can be positioned by clicking and dragging it with the mouse in the render view, or the `Position` property can be specified explicitly with fractional coordinates that range from [0, 1] and represent the fraction of the window width (or height) where the color legend's bottom left corner should be placed. Note that if the color legend is placed interactively with the mouse, the `Window Location` option will automatically change to `AnyLocation` .

Besides the obvious changing of title text and font properties for the title, labels, and annotations, there are some other parameters that control the appearance of the color legend.

By default, the title is rotated 90 degrees counter-clockwise when the legend is oriented vertically to better align with the legend. Checking the `Horizontal Title` box forces the title of the color legend to be horizontal regardless of color legend orientation.

`Draw Annotations` determines whether the annotations (including the start and end annotations) are to be drawn at all.

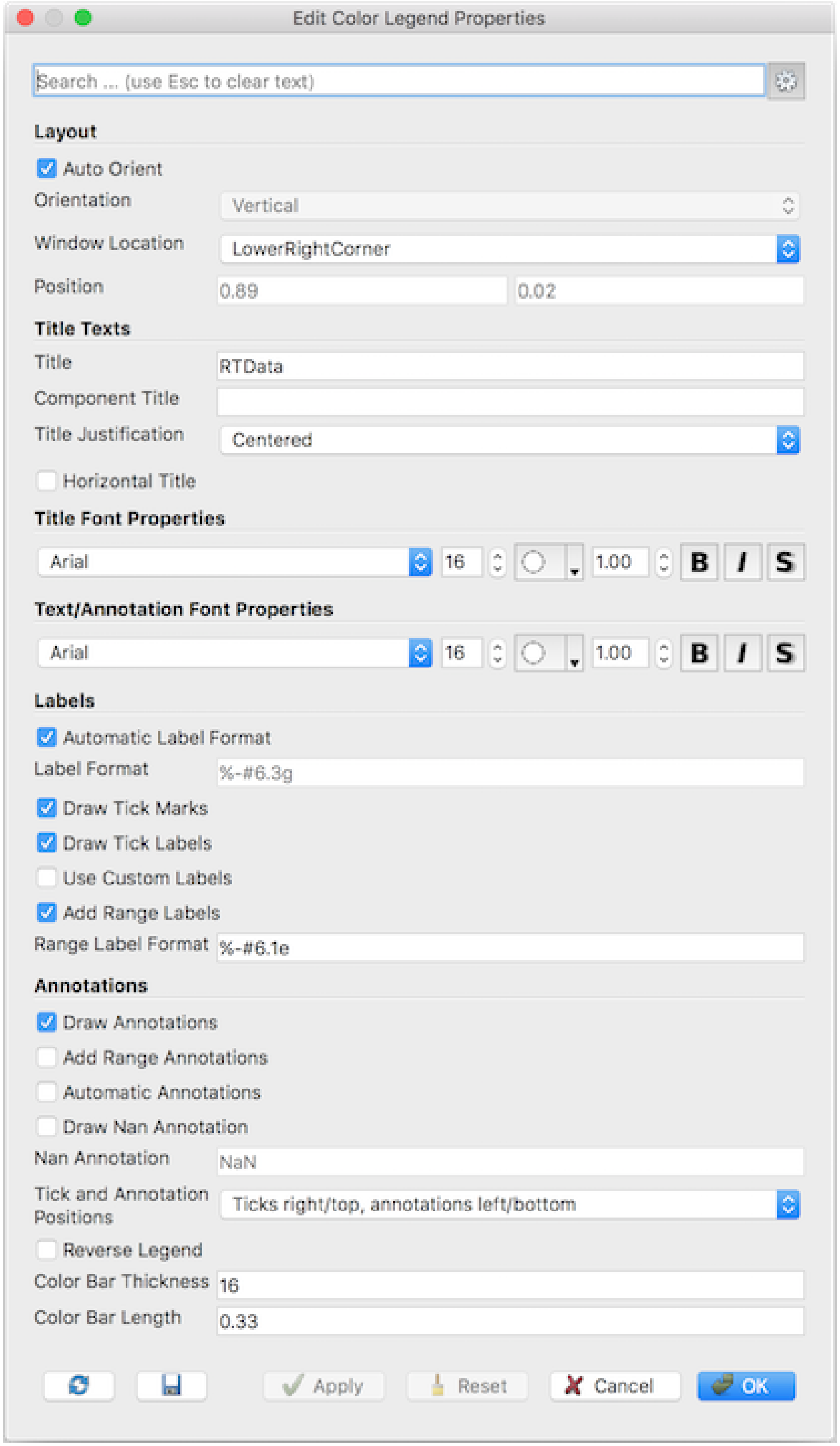
When checked, `Draw Nan Annotation` results in the color legend showing the NaN color set in the `Color Map Editor` panel in a separate color box right beside the color legend. The annotation text shown for that box can be modified by changing the `Nan Annotation` text.

If `Automatic Label Format` is checked, **ParaView** will try to pick an optimal representation for numerical values based on the value and available screen space. By unchecking it, you can explicitly specify the `printf`-style format to use for numeric values. To explicitly label values of interest, enable the `Use Custom Labels` option. You can specify exactly the labeled values you wish to display in the table that appears when this option is chosen. `Color Bar Thickness` is used to control the thickness of the legend. It is defined in terms of points just like how font sizes are specified. Use `Color Bar Length` to explicitly set the length of the color bar. This property is defined as a fraction in the range [0, 1] of the window width (when the color legend is oriented horizontally) or height (when oriented vertically).

### Color legend in `pvpython`

To show the color legend or scalar bar for the transfer function used for scalar mapping a source in a view, you can use API on its display properties:

```
>>> source = ...
>>> display = GetDisplayProperties(source, view)

# to show the color legend
>>> display.SetScalarBarVisibility(view, True)

# to hide the same
>>> display.SetScalarBarVisibility(view, False)
```



Edit Color Legend Properties

Search ... (use Esc to clear text)

**Layout**

☑ Auto Orient

Orientation: Vertical

Window Location: LowerRightCorner

Position: 0.89 0.02

**Title Texts**

Title: RTData

Component Title:

Title Justification: Centered

□ Horizontal Title

**Title Font Properties**

Arial 16 ○ 1.00 B I S

**Text/Annotation Font Properties**

Arial 16 ○ 1.00 B I S

**Labels**

☑ Automatic Label Format

Label Format: %-#6.3g

☑ Draw Tick Marks

☑ Draw Tick Labels

□ Use Custom Labels

☑ Add Range Labels

Range Label Format: %-#6.1e

**Annotations**

☑ Draw Annotations

□ Add Range Annotations

□ Automatic Annotations

□ Draw Nan Annotation

Nan Annotation: NaN

Tick and Annotation Positions: Ticks right/top, annotations left/bottom

□ Reverse Legend

Color Bar Thickness: 16

Color Bar Length: 0.33

Fig. 2.9: `Edit Color Legend Parameters` dialog in `paraview`.

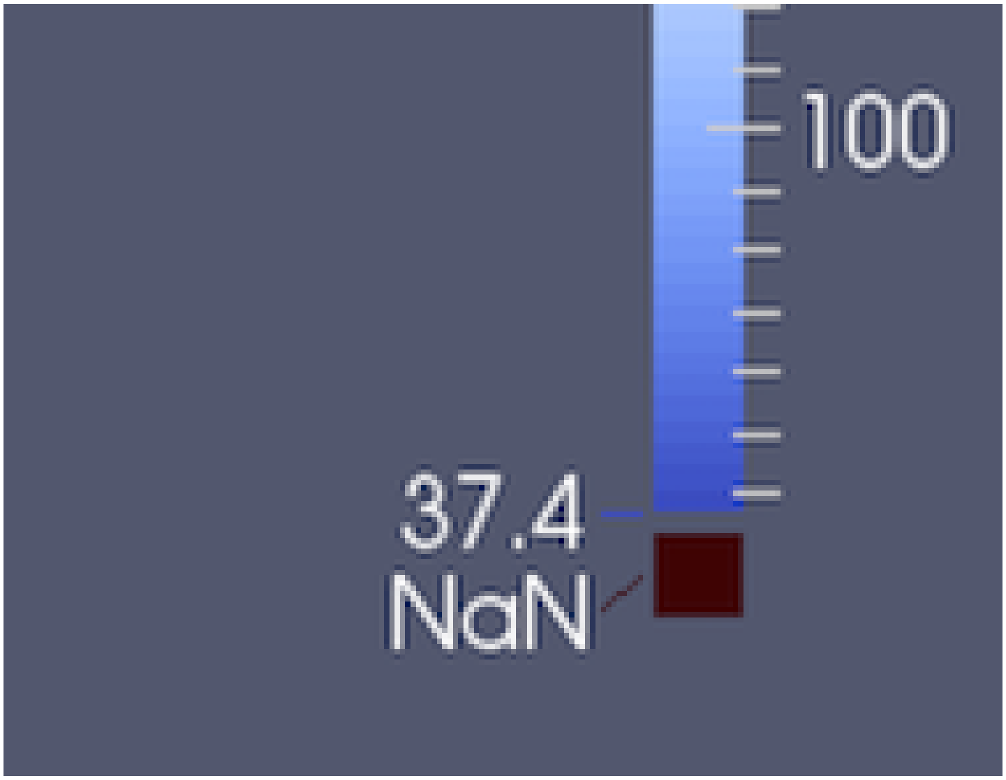


Fig. 2.10: Color legend showing NaN annotation

To change the color legend properties as in Section 2.3.4, you need to first get access to the color legend object for the color transfer function in a particular view. These are analogous to display properties for a source in a view.

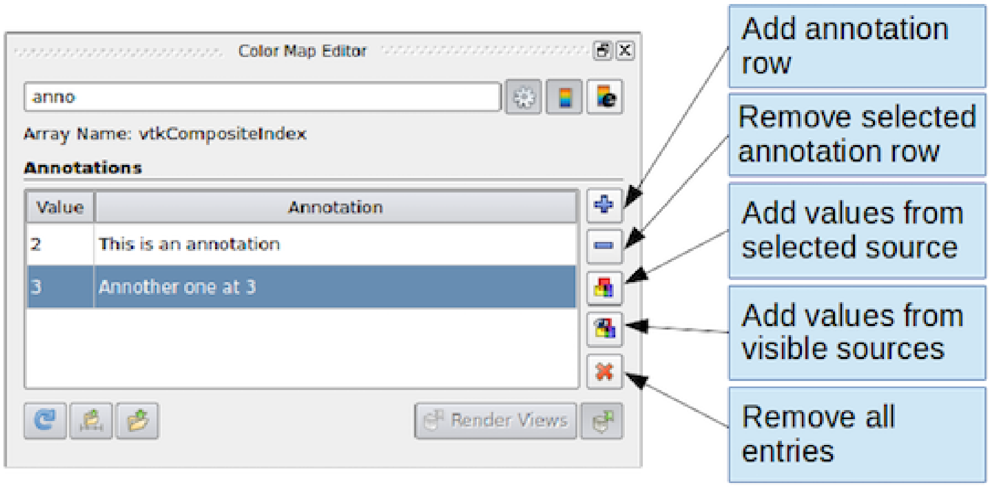
```
>>> colorMap = GetColorTransferFunction('Temp')

# get the scalar bar in a view (akin to GetDisplayProperties)
>>> scalarBar = GetScalarBar(colorMap, view)

# Now, you can change properties on the scalar bar object.
>>> scalarBar.TitleFontSize = 8
>>> scalarBar.DrawNanAnnotation = 1
```

### 2.3.5 Annotations

Simply put, annotations allow users to put custom text at particular data values in the color legend. The min and max data mapped value annotations are automatically added. To add any other custom annotations, you can use the `Color Map Editor`.

Since the list of annotations is an advanced property, you need to either toggle the visibility of advanced properties using the  icon near the top of the panel or type `annotations` in the search box. That will show the `Annotations` widget, which is basically a list widget where users can enter key-value pairs, rather than value-annotation pairs, as shown in Fig. 2.11.

You can use the buttons of the right on the widget to add/remove entries. Enter the data value to annotate under the `Value` column and then enter the text to display at that value under the `Annotation` column.


Color Map Editor
Render Views
Add annotation row
Remove selected annotation row
Add values from selected source
Add values from visible sources
Remove all entries

anno

Array Name: vtkCompositeIndex

Annotations

| Value | Annotation |
|---|---|
| 2 | This is an annotation |
| 3 | Another one at 3 |

Fig. 2.11: Widget to add/edit annotations on the `Color Map Editor` panel

You can use the `Tab` (tab) key to edit the next entry. Hitting `Tab` after editing the last entry in the table will automatically result in adding a new row, thus, making it easier to add bunch of annotations without having to click any buttons.

Some annotation texts may not show up on the legend. There are two possible reasons an annotation may not be shown. First, the value added is outside the mapped range of the color transfer function. Second, `Draw Annotations` is unchecked in the `Color Legend Parameters dialog` .

The ▦ and ▦ buttons can be used to fill the annotations widget with unique discrete values from a data array, if possible. Based on the number of distinct values present in the data array, this may not yield any result (Instead, a warning message will be shown). The data array values come either from the selected source object if you use the ▦ button or it comes from the visible pipeline objects if you use the ▦ button.

### Annotations in `pvpython`

Annotations is a property on the color map object. You simply get access to the color map of interest and then change the `Annotations` property.

```
>>> colorMap = GetColorTransferFunction('Temp')

# Annotations are specified as a flattened list of tuples
# (data-value, annotation-text)
>>> colorMap.Annotations = ['1', 'Slow',
                            '10', 'Fast']
```

### 2.3.6 Categorical colors

A picture is worth a thousand words, they say, so let's just let the picture do the talking. Categorical color maps allow you to render visualizations as shown in Fig. 2.12.

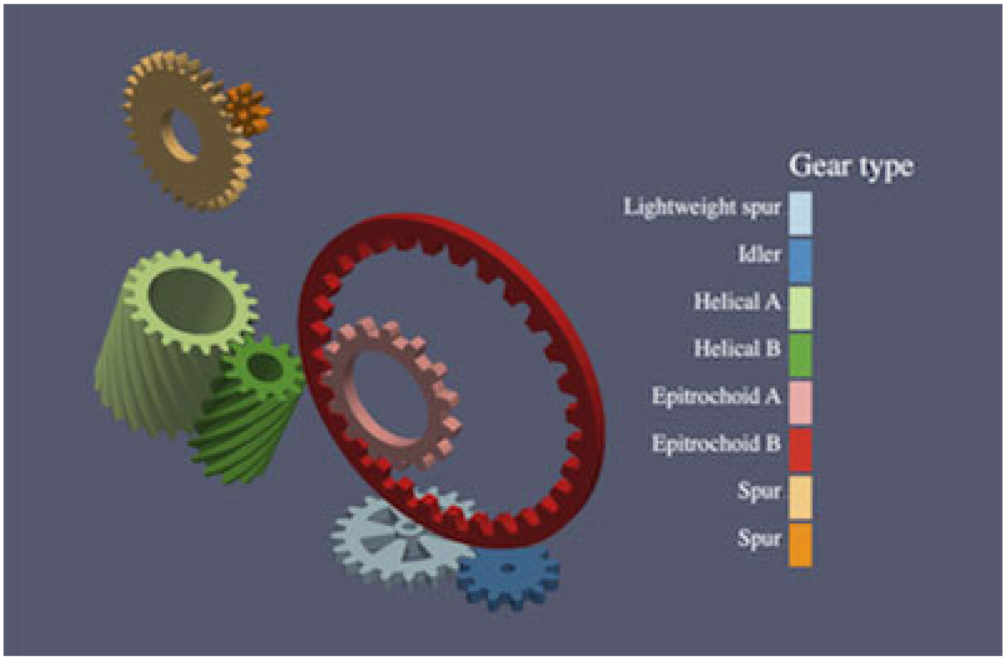


Fig. 2.12: Visualization using a categorical color map for discrete coloring

When one thinks of scalar coloring, one is typically talking of mapping numerical values to colors. However, in some cases, the numerical values are not really numbers, but enumerations such as elements types and gear types (as in Fig. 2.12) or generally, speaking, categories. The traditional approach of using an interpolated color map specifying the range of values with which to color doesn't really work here. While users could always play tricks with the number of discrete steps, multiple control points, and annotations, it is tedious and cumbersome.

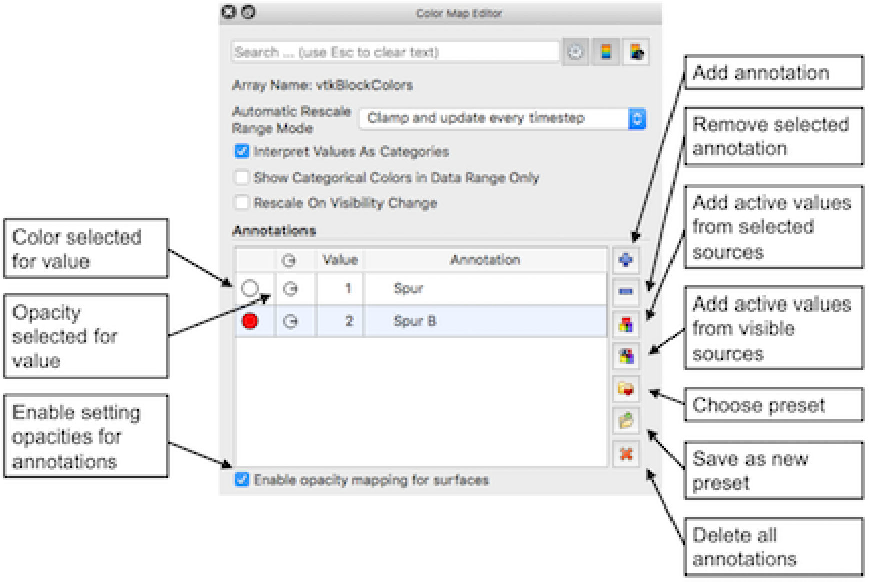
Categorical color maps provide an elegant solution for such cases. Instead of a continuous color transfer function, the user specifies a set of discrete values and colors to use for those values. For any element where the data value matches the values in the lookup table exactly, `paraview` renders the specified color; otherwise, the NaN color is used.

The color legend or scalar bar also switches to a new mode where it renders swatches with annotations, rather than a single bar. You can add custom annotations for each value in the lookup table.

#### Categorical Color: User Interface

To tell `paraview` that the data array is to be treated as categories for coloring, check the `Interpret Values As Categories` checkbox in the `Color Map Editor` panel. As soon as that's checked, the panel switches to categorical mode: The `Mapping Data` group is hidden, and the `Annotations` group becomes a non-advanced group, i.e., the annotations widget is visible even if the panel is not showing advanced properties, as is shown in Fig. 2.13.

The annotations widget will still show any annotations that may have been added earlier, or it may be empty if none were added. You can add annotations for data values as was the case before using the buttons on the side of the widget. This time, however, each annotation entry also has a column for color. If color has not been specified, a question mark icon will show up; otherwise, a color swatch will be shown. You can double click the color swatch or the question mark icon to specify the color to use for that entry. Alternatively, you can choose from a preset collection of categorical color maps by clicking the  button.



Color Map Editor

Search ... (use Esc to clear text)

Array Name: vtkBlockColors

Automatic Rescale Range Mode: Clamp and update every timestep

☑ Interpret Values As Categories

□ Show Categorical Colors in Data Range Only

□ Rescale On Visibility Change

Annotations

| | | Value | Annotation |
|---|---|---|---|
| ○ | ◷ | 1 | Spur |
| ● | ◷ | 2 | Spur B |

☑ Enable opacity mapping for surfaces

Color selected for value
Opacity selected for value
Enable setting opacities for annotations
Add annotation
Remove selected annotation
Add active values from selected sources
Add active values from visible sources
Choose preset
Save as new preset
Delete all annotations

Fig. 2.13: Default `Color Map Editor` when `Interpret Values As Categories` is checked.

As before, you can use `Tab` key to edit and add multiple values. Hence, you can first add all the values of interest in one pass and then pick a preset color map to set colors for the values added. If the preset has fewer colors than the annotated values, then the user may have to manually set the colors for those extra annotations.

**Common Erros**

Categorical color maps are designed for data arrays with enumerations, which are typically integer arrays. However, they can be used for arrays with floating point numbers as well. With floating point numbers, the value specified for annotation may not match the value in the dataset exactly, even when the user expects it to match. In that case, the NaN color will be used.

### Categorical colors in `pvpython`

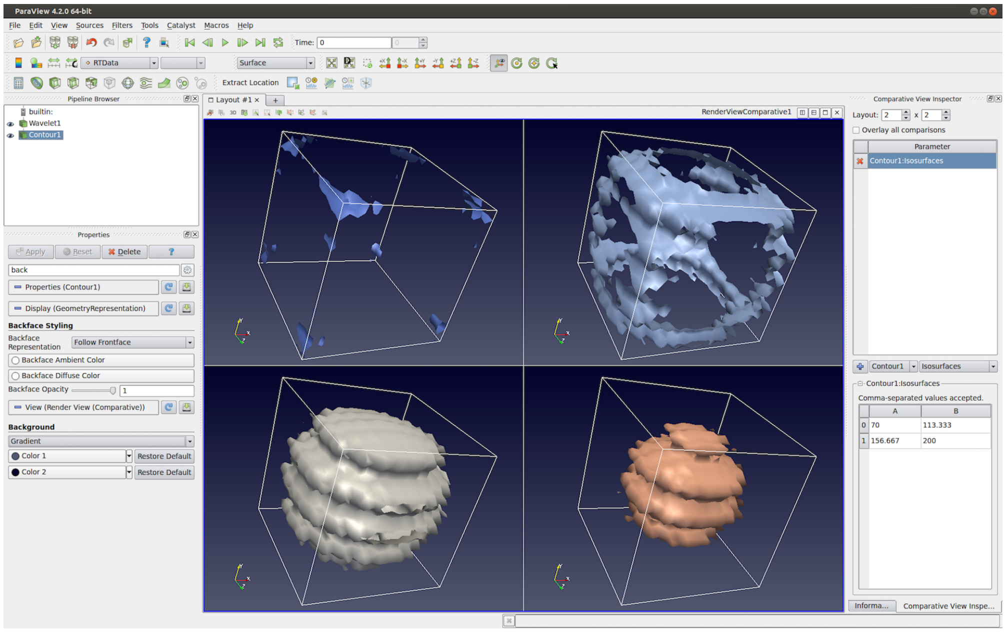
```
>>> categoricalColorMap = GetColorTransferFunction('Modes')
>>> categoricalColorMap.InterpretValuesAsCategories = 1

# specify the labels for the categories. This is similar to how
# other annotations are specified.
>>> categoricalColorMap.Annotations = ['0', 'Alpha', '1', 'Beta']

# now, set the colors for each category. These are an ordered list
# of flattened tuples (red, green, blue). The first color gets used for
# the first annotation, second for second, and so on
>>> categoricalColorMap.IndexedColors = [0.0, 0.0, 0.0,
                                         0.89, 0.10, 0.10]
```

## 2.4 Comparative visualization

Comparative visualization in **ParaView** refers to the ability to create side-by-side visualizations for comparing with one another. In its most basic form, you can indeed use **ParaView**'s ability to show multiple views side by side to set up simultaneous visualizations. But, that can get cumbersome too quickly. Let's take a look at a simple example: Let's say you want to do a parameter study where you want to compare isosurfaces generated by a set of isovalues in a dataset. To set up such a visualization, you'll need to first create as many `Render View` s as isovalues. Then, create just as many `Contour` filters, setting each one up with a right isovalue for the contour to generate and display the result in one of the views. As the number of isovalues increases, you can see how this can get tedious. It is highly error prone, as you need to keep track of which view shows which isovalue. Now, imagine after having set up the entire visualization that you need to change the `Representation` type for all the of the isosurfaces to `Wireframe` !

`Comparative View` s were designed to handle such use-cases. Instead of creating separate views, you create a single view `Render View (Comparative)` . The view itself comprises of a configurable $m \times n$ `Render View` s. Any data that you show in this view gets shown in all the internal views simultaneously. Any display property changes, such as scalar coloring and representation type are also maintained consistently between these internal views. The interactions in the internal views are linked, so when you interact with one, all other views update as well. While this is all nice and well, the real power of `Comparative View` s becomes apparent when you set up a parameter to vary across the views. This parameter can be any of the properties on the pipeline modules such as filter properties and opacity, or it could be the data time. Each of these internal views will now render the result obtained by setting the parameter as per your selection. Going back to our original example, we will create a single `Contour` filter that we show in `Render View (Comparative)` with as many internal views as the isovalue to compare. Next, we will set up a parameter study for varying the `Isosurfaces` property on the `Contour` filter, and, viola! The view will generate the comparative visualization for us!

In this chapter, we look at how to configure this view and how to set these parameters to compare. We limit our discussion to `Render View (Comparative)` . However, the same principles are applicable to other comparative views, including `Bar Chart View (Comparative)` and `Line Chart View (Comparative)` .



Properties search: back

- Properties (Contour1)
- Display (GeometryRepresentation)

**Backface Styling**

Backface Representation: Follow Frontface

○ Backface Ambient Color

○ Backface Diffuse Color

Backface Opacity: 1

- View (Render View (Comparative))

**Background**

Gradient

● Color 1 — Restore Default

● Color 2 — Restore Default

Layout: 2 x 2

□ Overlay all comparisons

Parameter: Contour1:Isosurfaces

Contour1 — Isosurfaces

Contour1:Isosurfaces

Comma-separated values accepted.

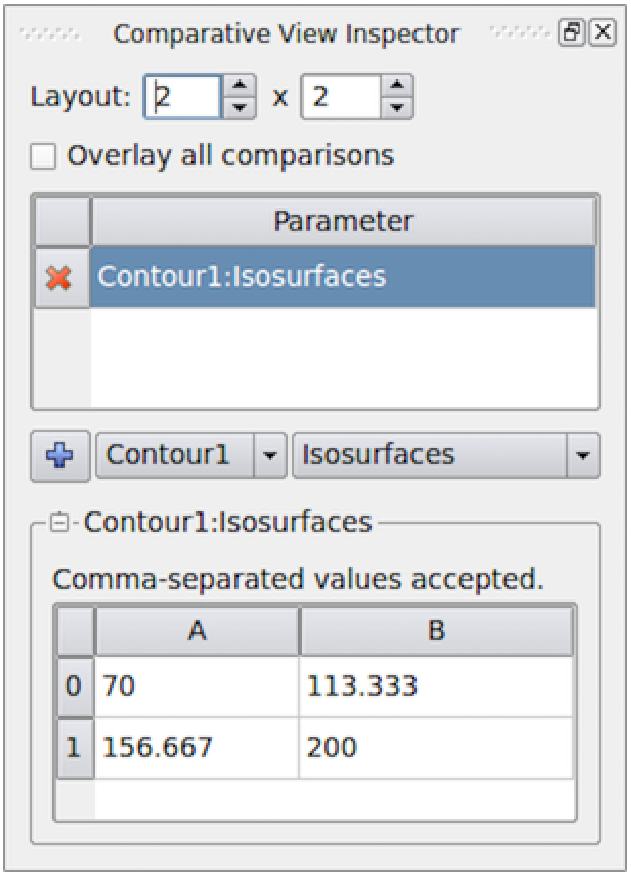
| | A | B |
|---|---|---|
| 0 | 70 | 113.333 |
| 1 | 156.667 | 200 |

Fig. 2.14: `Render View (Comparative)` in `paraview` showing a parameter study. In this case, we are comparing the visualization generated using different isovalues for the `Contour` filter. The `Comparative View Inspector` dockpanel (on the right) is used to configure the parameter study.

### 2.4.1 Setting up a comparative view

To create `Render View (Comparative)` in `paraview`, split or close the active view and select `Render View (Comparative)` from the new view creation widget. `paraview` will immediately show four `Render View` s laid out in a 2 × 2 grid. While you cannot resize these internal views, notice that you can still split the view frame and create other views if needed.

The `Properties` panel will show properties similar to those available on the `Render View` under the `View` properties section. If you change any of these properties, they will affect all these internal views, e.g., setting the `Background` color to `Gradient` will make all the views show a gradient background.



Layout: 2 x 2

□ Overlay all comparisons

| | Parameter |
|---|---|
| ✖ | Contour1:Isosurfaces |

Contour1 ▾ Isosurfaces ▾

Contour1:Isosurfaces

Comma-separated values accepted.

| | A | B |
|---|---|---|
| 0 | 70 | 113.333 |
| 1 | 156.667 | 200 |

Fig. 2.15: `Comparative View Inspector` in `paraview` used to configure the active comparative view.

To configure the comparative view itself, you must to use the `Comparative View Inspector` ( Fig. 2.15) accessible from the *View* menu. The `Comparative View Inspector` is a dockable panel that is enabled when the active view is a comparative view.

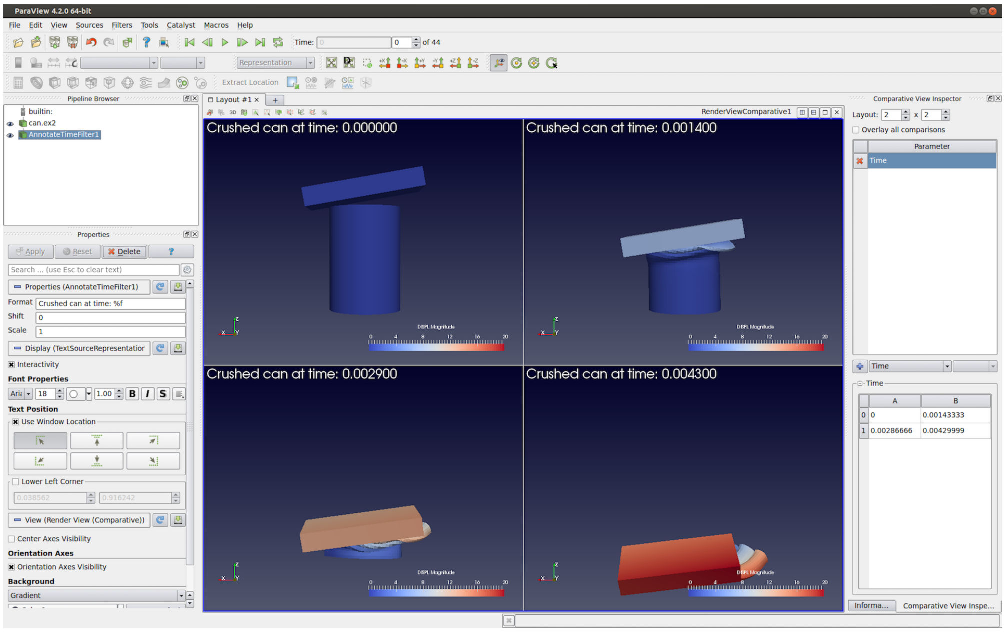
To change how many internal views are used in the active comparative view and how they are laid out, use the `Layout` . The first value is the number of views in the horizontal direction and the second is the count in the vertical direction.

Besides doing a parameter study in side-by-side views, you can also show all the results in a single view. For that, simply check the `Overlay all comparisons` checkbox. When checked, instead of getting a grid of $m \times n$ views, you will only see one view with all visible data displayed $m \times n$ times.

To show data in this view is the same as any other view: Make this view active and use the `Pipeline Browser` to toggle the eyeball icons to show data produced by the selected pipeline module(s). Showing any dataset in this view will result in the data being shown in all the internal views simultaneously. As is true with `View` properties, with `Display` properties, changing `Coloring` , `Styling` , or any other properties will reflect in all internal views.

Since the cameras among the internal views are automatically linked, if you interact with one of the views, all views will also update simultaneously when you release the mouse button.

### 2.4.2 Setting up a parameter study


ParaView 4.2.0 64-bit
File Edit View Sources Filters Tools Catalyst Macros Help
Pipeline Browser
builtin:
can.ex2
AnnotateTimeFilter1
Properties
Apply
Reset
Delete
Search ... (use Esc to clear text)
Layout #1
RenderViewComparative1
Crushed can at time: 0.000000
Crushed can at time: 0.001400
Crushed can at time: 0.002900
Crushed can at time: 0.004300
DISPL Magnitude
Comparative View Inspector
Parameter
Informa...
Comparative View Inspe...

Time: 0 of 44

Properties (AnnotateTimeFilter1)

Format: Crushed can at time: %f

Shift: 0

Scale: 1

Display (TextSourceRepresentation

☒ Interactivity

**Font Properties**

Arial 18 1.00

**Text Position**

☒ Use Window Location

☐ Lower Left Corner

0.038562 0.916242

View (Render View (Comparative))

☐ Center Axes Visibility

**Orientation Axes**

☒ Orientation Axes Visibility

**Background**

Gradient

Layout: 2 x 2

☐ Overlay all comparisons

| | Parameter |
|---|---|
| ✖ | Time |

Time

| | A | B |
|---|---|---|
| 0 | 0 | 0.00143333 |
| 1 | 0.00286666 | 0.00429999 |

Fig. 2.16: `Render View (Comparative)` with `Annotate Time` filter showing the time for each of the views. In this case, the parameter study is varying `Time` across the views.

To understand how to setup a parameter study, let's go back to our original example. Our visualization pipeline is simply `Wavelet` → `Contour` (assuming default property values), and we are showing the result from the `Contour` filter in the $2 \times 2$ `Render View (Comparative)` with `Overlay all comparisons` unchecked.

Now, we want to vary the `Isosurfaces` property value for the visualization in each of the internal views. That's our parameter to study. Since that's a property on the `Contour` filter, we select the `Contour` filter instance. Then, its property `Isosurfaces` is in the parameter selection combo-boxes. To add the parameter to the study, you must hit the ✚ button.

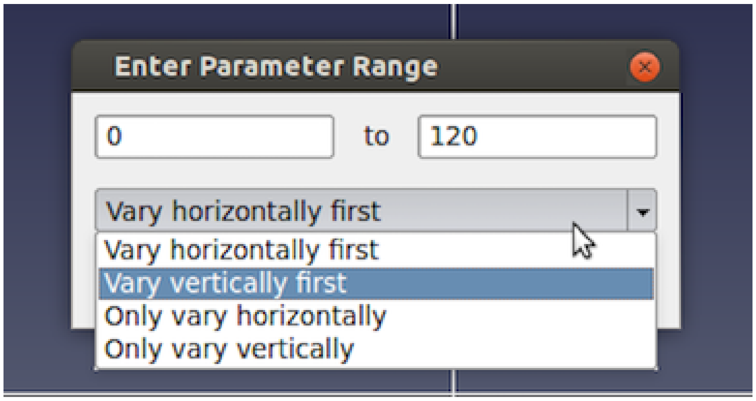
The parameter `Contour1:Isosurfaces` will then show up in the `Parameter` list above the combo-boxes. You can delete this parameter by using the ✖ button next to the parameter name.

**Did you know?**

Notice how this mechanism for setting up a parameter study is similar to how animations are set up. In fact, under the cover, the two mechanisms are not that different, and they share a lot of implementation code.

Simultaneously, the table widget below the combo-boxes is populated with a default set of values for the `Isosurfaces` . This is where you specify the parameter values for each of the views. The location of the views matches the location in the table. Hence, $0 - A$ is the top-left view, $0 - B$ is the second view from the left in the topmost row, and so on. The parameter value at a specific location in the table is the value used to generate the visualization in the corresponding view.

To change the parameter (in our case `Isosurfaces` ) value, you can double click on the cell in the table and change it one at a time. Also, to make it easier to fill the cells with a range of values, you can click and drag over multiple cells. When you release the mouse, a dialog will prompt you to enter the data value range ( Fig. 2.17). In case you selected a combination of rows and columns, the dialog will also let you select in which direction the parameter is varied using the range specified. You can choose to vary the parameter horizontally first, vertically first, or only along one of the directions while keeping the other constant. This is useful when doing a study with multiple parameters.



Enter Parameter Range

0 to 120

Vary horizontally first

Fig. 2.17: Dialog used to select a range of parameter values and control how to vary them in `paraview`.

As soon as you change the parameter values, the view will update to reflect the change.

### 2.4.3 Adding annotations

You add annotations like color legends, text, and cube-axes exactly as you would with a regular `Render View` . As with other properties, annotations will show up in all of the internal views.

**Did you know?**

You can use the `Annotate Time` source or filter to show the data time or the view time in each of the internal views. If `Time` is one of the parameters in your study, the text will reflect the time values for each of the individual views, as expected ( Fig. 2.16)!

## 2.5 Programmable Filter

A pipeline module in **ParaView** does one of two things: it either generates data or processes input data. To generate data, the module may use a mathematical model e.g. *Sources* > *Sphere* or read a file from disk. Processing data entails transforming input data by applying defined operations to generate a new output. **ParaView** provides a large set of readers, data sources and data filters that cover the needs of many users. For the cases where the available collection does not satisfy your needs, **ParaView** provides a mechanism to add new modules via plugins. Conventional plugins, however, are intended for hardcore developers. They are written in C++, using the data processing APIs provided by **ParaView** and VTK. The complexity of building and packaging C++ plugins that work on all distributed versions of **ParaView** can be daunting and thus a huge barrier for even advanced **ParaView** users. Python-based programmable filters, sources and annotations provide an easy alternative to this. New modules can be written as Python scripts that are executed by **ParaView** to generate, process and/or display data, just like conventional C++ modules. Since the scripts are standard Python scripts, you have access to Python packages such as NumPy that provide several numeric operations useful for data transformation.

In this chapter, we will explore how to use Python to add new data processing modules to ParaView through examples. For additional explanation of the data processing API, see Section 2.6.

**Common Errors**

In this guide so far, we have been looking at examples of Python scripts for `pvpython`. These scripts are used to script the actions you would perform using the `paraview` UI. The scripts you would write for `Programmable Source` , `Programmable Filter`, and `Programmable Annotation` are entirely different. The data processing API executes within the data processing pipeline and, thus, has access to the data being processed. In client-server mode, this means that such scripts are indeed executed on the server side, potentially in parallel, across several MPI ranks. Therefore, attempting to import the `paraview.simple` Python module in the `Programmable Source` script, for example, is not supported and will have unexpected consequences.

## 2.5.1 Understanding the programmable modules

With programmable modules, you are writing custom code for filters and sources. You are expected to understand the basics of a VTK (and **ParaView**) data processing pipeline, including the various stages of the pipeline execution as well as the data model. Refer to Section 1.3.1 for an overview of the VTK data model. While a detailed discussion of the VTK pipeline execution model is beyond the scope of this book, the fundamentals covered in this section, along with the examples in the rest of this chapter, should help you get started and write useful modules. For a primer on the details of the VTK pipeline execution stages, see [BerkGeveci].

To create the programmable source, filter, or annotation in `paraview`, you use the *Sources* > *Programmable Source*, *Filters* > *Programmable Filter* or *Sources* > *Programmable Annotation* menus, respectively. Since the `Programmable Filter` and `Programmable Annotation` are filters, like other filters, they get connected to the currently active source(s), i.e., the currently active source(s) become the input to this new filter. `Programmable Source` , on the other hand, does not have any inputs.

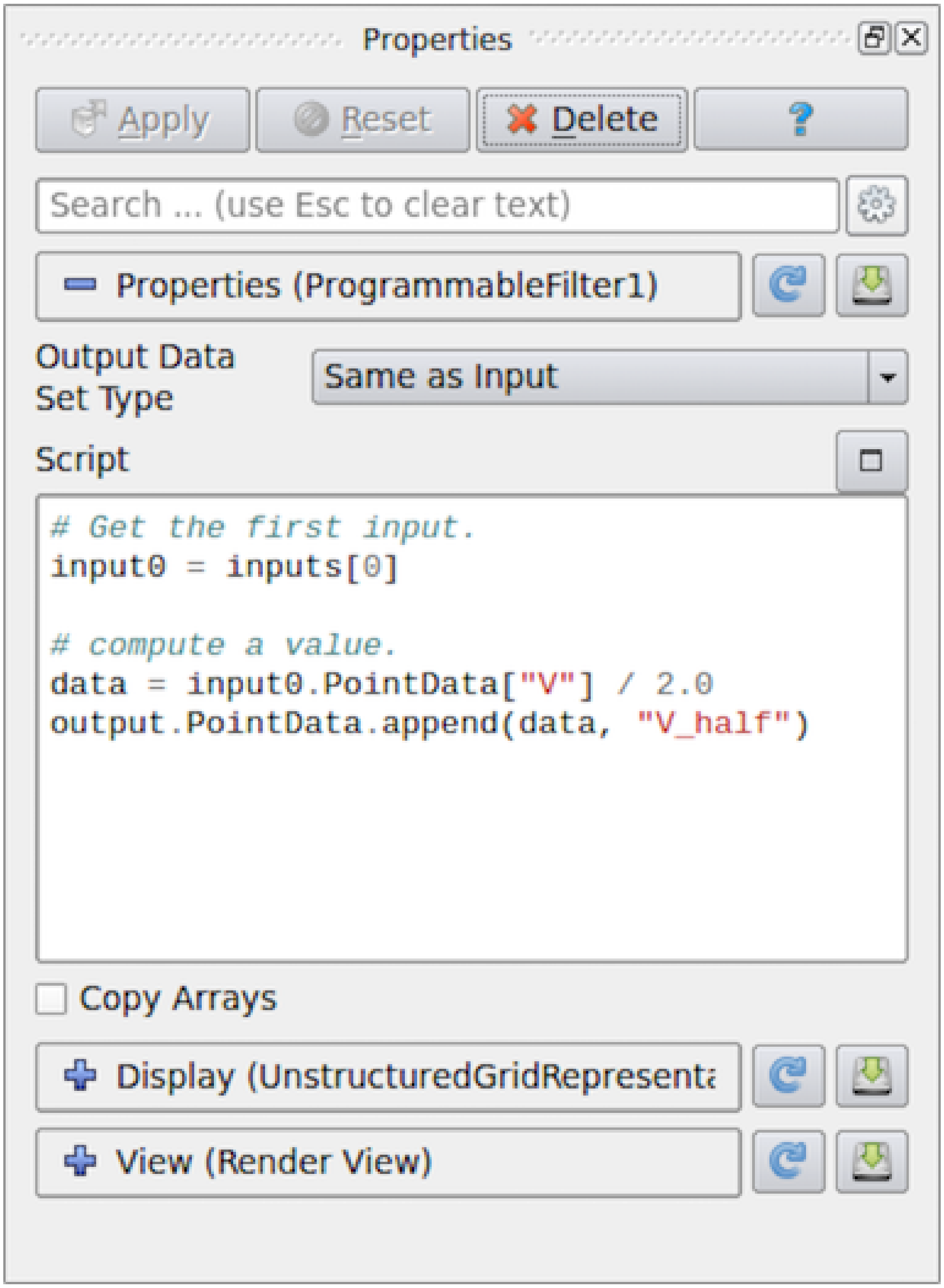
One of the first things that you specify after creating a programmable filter or source is the `Output Data Set Type` . This option lets you select the type of dataset this module will produce. The options provided include several of the data types discussed in Section 1.3.1. Additionally, for the `Programmable Filter` , you can select `Same as Input` to indicate that the filter preserves the input dataset type.

Next is the primary part: the `Script` . This is where you enter the Python script to generate or process from the inputs the dataset that the module will produce. As with any Python script, you can import other Python packages and modules in this script. Just be aware that when running in client-server mode, this script is going to be executed on the server side. Accordingly, any modules or packages you import must be available on the server side to avoid errors.

The script gets executed in what's called the *RequestData* pass of the pipeline execution. This is the pipeline pass in which an algorithm is expected to produce the output dataset.

There are several other passes in a pipeline's execution. The ones for which you can specify a Python script to execute in the programmable filter and source are:

- *RequestInformation*: In this pass, the algorithm is expected to provide the pipeline with any meta-data available about the data that will be produced by it. This includes things like number of timesteps in the dataset and their time values for temporal datasets or extents for structured datasets. This gets called before *RequestData* pass. In the *RequestData* pass, the pipeline could potentially qualify the request based on the meta-data provided in this pass. For example, if an algorithm announces that the temporal dataset has multiple timesteps, the pipeline could request that the algorithm produce data for one of those timesteps in `RequestData` .
- *RequestUpdateExtent*: In this pass, a filter gets the opportunity to qualify requests for execution passed on to the upstream pipeline. As an example, if an upstream reader announced in its *RequestInformation* script that it can produce several timesteps, in *RequestUpdateExtent*, this filter can make a request to the upstream reader for a specific timestep. This pass gets called after *RequestInformation*, but before `RequestData` . It's not very common to provide a script for this pass.


Properties
Apply
Reset
Delete
?
Search ... (use Esc to clear text)
Properties (ProgrammableFilter1)
Script
Display (UnstructuredGridRepresent
View (Render View)

Output Data Set Type: Same as Input

Script

```
# Get the first input.
input0 = inputs[0]

# compute a value.
data = input0.PointData["V"] / 2.0
output.PointData.append(data, "V_half")
```

☐ Copy Arrays

Fig. 2.18: `Properties` panel for `Programmable Filter` in `paraview`.

You can specify the script for the *RequestInformation* pass in `RequestInformation Script` and for the *RequestUpdateExtent* pass in `RequestUpdateExtent Script` . Since the *RequestUpdateExtent* pass does not make much sense for an algorithm that does not have any inputs, `RequestUpdateExtent Script` is not available on `Programmable Source` . `Programmable Annotation` only has a *RequestData* script as this is the only one that make sense in this context.

## 2.5.2 Recipes for Programmable Source

In this section, we look at several recipes for `Programmable Source` . A common use of `Programmable Source` is to prototype readers. If your reader library already provides a Python API, then you can easily import the appropriate Python package to read your dataset using `Programmable Source` .

**Did you know?**

Most of the examples in this chapter use a NumPy-centric API for accessing and creating data arrays. Additionally, you can use VTK's Python wrapped API for creating and accessing arrays. However, given the ubiquity of NumPy, there is rarely any need for using VTK's API directly.

### Reading a CSV file

For this example, we will read in a CSV file to produce a Table ( Section 1.3.1) using `Programmable Source` . We will use NumPy to do the parsing of the CSV files and pass the arrays read in directly to the pipeline. Note that the `Output DataSet Type` must be set to `vtkTable` .

```python
# Code for 'Script'

# We will use NumPy to read the csv file.
# Refer to NumPy documentation for genfromtxt() for details on
# customizing the CSV file parsing.

import numpy as np
# assuming data.csv is a CSV file with the 1st row being the names names for
# the columns
data = np.genfromtxt("data.csv", dtype=None, names=True, delimiter=',', autostrip=True)
for name in data.dtype.names:
   array = data[name]

   # You can directly pass a NumPy array to the pipeline.
   # Since ParaView expects all arrays to be named, you
   # need to assign it a name in the 'append' call.
   output.RowData.append(array, name)
```

### Reading a CSV file series

Building on the example from Section 2.5.2, let's say we have a series of files that we want to read in as a temporal series. Recall from Section 2.5.1 that meta-data about data to be produced, including timestep information, is announced in *RequestInformation* pass. Hence, for this example, we will need to specify the `RequestInformation Script` as well.

As was true earlier, `Output DataSet Type` must be set to `vtkTable` . Now, to announce the timesteps, we use the following as the `RequestInformation Script` .

```
# Code for 'RequestInformation Script'.
def setOutputTimesteps(algorithm, timesteps):
    "helper routine to set timestep information"
    executive = algorithm.GetExecutive()
    outInfo = executive.GetOutputInformation(0)

    outInfo.Remove(executive.TIME_STEPS())
    for timestep in timesteps:
        outInfo.Append(executive.TIME_STEPS(), timestep)

    outInfo.Remove(executive.TIME_RANGE())
    outInfo.Append(executive.TIME_RANGE(), timesteps[0])
    outInfo.Append(executive.TIME_RANGE(), timesteps[-1])

# As an example, let's say we have 4 files in the file series that we
# want to say are producing time 0, 10, 20, and 30.
setOutputTimesteps(self, (0, 10, 20, 30))
```

The `Script` is similar to earlier, except that we will read a specific CSV file based on which timestep was requested.

```
# Code for 'Script'
def GetUpdateTimestep(algorithm):
    """Returns the requested time value, or None if not present"""
    executive = algorithm.GetExecutive()
    outInfo = executive.GetOutputInformation(0)
    return outInfo.Get(executive.UPDATE_TIME_STEP()) \
              if outInfo.Has(executive.UPDATE_TIME_STEP()) else None

# This is the requested time-step. This may not be exactly equal to the
# timesteps published in RequestInformation(). Your code must handle that
# correctly.
req_time = GetUpdateTimestep(self)

# Now, use req_time to determine which CSV file to read and read it as before.
# Remember req_time need not match the time values put out in
# 'RequestInformation Script'. Your code need to pick an appropriate file to
# read, irrespective.

...
# TODO: Generate the data as you want.

# Now mark the timestep produced.
output.GetInformation().Set(output.DATA_TIME_STEP(), req_time)
```

### Reading a CSV file with particles

This is similar to Section 2.5.2. Now, however, let's say the CSV has three columns named `X`, `Y` and `Z` that we want to treat as point coordinates and produce a `vtkPolyData` with points instead of a `vtkTable` . For that, we first ensure that `Output DataSet Type` is set to `vtkPolyData` . Next, we use the following `Script` :

```
# Code for 'Script'

from vtk.numpy_interface import algorithms as algs
from vtk.numpy_interface import dataset_adapter as dsa
import numpy as np

# assuming data.csv is a CSV file with the 1st row being the names names for
# the columns
data = np.genfromtxt("/tmp/points.csv", dtype=None, names=True, delimiter=',',␣
↪autostrip=True)

# convert the 3 arrays into a single 3 component array for
# use as the coordinates for the points.
coordinates = algs.make_vector(data["X"], data["Y"], data["Z"])

# create a vtkPoints container to store all the
# point coordinates.
pts = vtk.vtkPoints()

# numpyTovtkDataArray is needed to called directly to convert the NumPy
# to a vtkDataArray which vtkPoints::SetData() expects.
pts.SetData(dsa.numpyTovtkDataArray(coordinates, "Points"))

# set the pts on the output.
output.SetPoints(pts)

# next, we define the cells i.e. the connectivity for this mesh.
# here, we are creating merely a point could, so we'll add
# that as a single poly vextex cell.
numPts = pts.GetNumberOfPoints()
# ptIds is the list of point ids in this cell
# (which is all the points)
ptIds = vtk.vtkIdList()
ptIds.SetNumberOfIds(numPts)
for a in range(numPts):
    ptIds.SetId(a, a)

# Allocate space for 1 cell.
output.Allocate(1)
output.InsertNextCell(vtk.VTK_POLY_VERTEX, ptIds)

# We can also pass all the array read from the CSV
# as point data arrays.
for name in data.dtype.names:
    array = data[name]
    output.PointData.append(array, name)
```

The thing to note is that this time, we need to define the geometry and topology for the output dataset. Each data type has different requirements on how these are specified. For example, for unstructured datasets like vtkUnstructuredGrid and vtkPolyData, we need to explicitly specify the geometry and all the connectivity information. For vtkImageData, the geometry is defined using origin, spacing, and extents, and connectivity is implicit.

### Reading binary 2D image

This recipe shows how to read raw binary data representing a 3D volume. Since raw binary files don't encode information about the volume extents and data type, we will assume that the extents and data type are known and fixed.

For producing image volumes, you need to provide the information about the structured extents in *RequestInformation*. Ensure that the `Output Data Set Type` is set to `vtkImageData` .

```
# Code for 'RequestInformation Script'.
executive = self.GetExecutive()
outInfo = executive.GetOutputInformation(0)
# we assume the dimensions are (48, 62, 42).
outInfo.Set(executive.WHOLE_EXTENT(), 0, 47, 0, 61, 0, 41)
outInfo.Set(vtk.vtkDataObject.SPACING(), 1, 1, 1)
```

The `Script` to read the data can be written as follows.

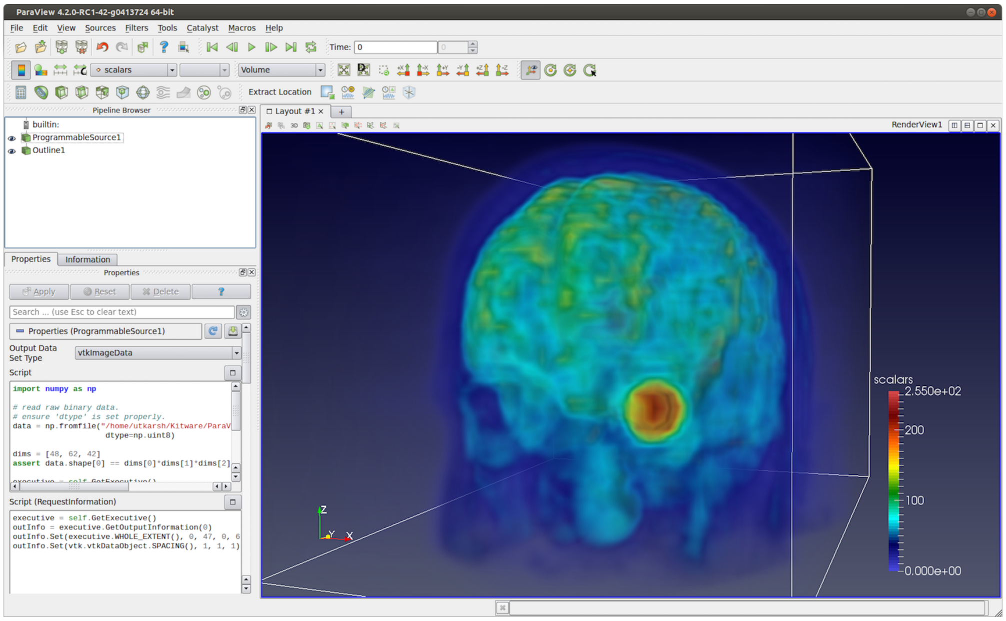
```
# Code for 'Script'
import numpy as np

# read raw binary data.
# ensure 'dtype' is set properly.
data = np.fromfile("HeadMRVolume.raw", dtype=np.uint8)

dims = [48, 62, 42]
assert data.shape[0] == dims[0]*dims[1]*dims[2], "dimension mismatch"

output.SetExtent(0, dims[0]-1, 0, dims[1]-1, 0, dims[2]-1)
output.PointData.append(data, "scalars")
output.PointData.SetActiveScalars("scalars")
```

### Helix source

Here is another polydata source example. This time, we generate the data programmatically.

```
# Code for 'Script'

#This script generates a helix curve.
#This is intended as the script of a 'Programmable Source'
import math
import numpy as np
from vtk.numpy_interface import algorithms as algs
from vtk.numpy_interface import dataset_adapter as dsa

numPts = 80  # Points along Helix
length = 8.0 # Length of Helix
rounds = 3.0 # Number of times around
```

(continues on next page)



Output Data Set Type: vtkImageData

Script

```
import numpy as np

# read raw binary data.
# ensure 'dtype' is set properly.
data = np.fromfile("/home/utkarsh/Kitware/ParaV
                   dtype=np.uint8)

dims = [48, 62, 42]
assert data.shape[0] == dims[0]*dims[1]*dims[2]
```

Script (RequestInformation)

```
executive = self.GetExecutive()
outInfo = executive.GetOutputInformation(0)
outInfo.Set(executive.WHOLE_EXTENT(), 0, 47, 0, 6
outInfo.Set(vtk.vtkDataObject.SPACING(), 1, 1, 1)
```

Fig. 2.19: `Programmable Source` used to read *HeadMRVolume.raw* file available in the VTK data repository.

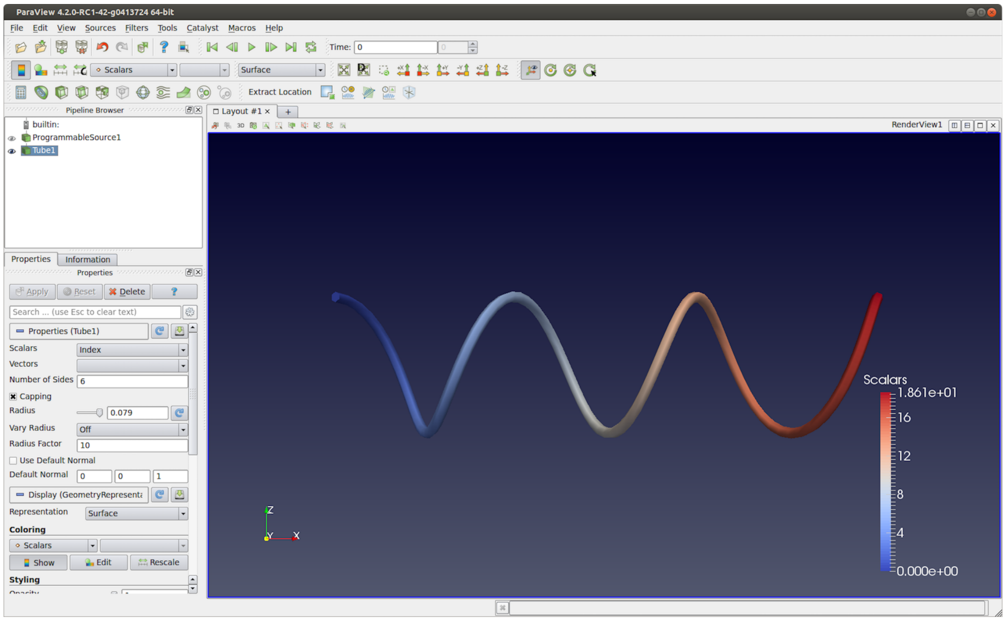
(continued from previous page)

```
# Compute the point coordinates for the helix.
index = np.arange(0, numPts, dtype=np.int32)
scalars = index * rounds * 2 * math.pi / numPts
x = index * length / numPts;
y = np.sin(scalars)
z = np.cos(scalars)

# Create a (x,y,z) coordinates array and associate that with
# points to pass to the output dataset.
coordinates = algs.make_vector(x, y, z)
pts = vtk.vtkPoints()
pts.SetData(dsa.numpyTovtkDataArray(coordinates, 'Points'))
output.SetPoints(pts)

# Add scalars to the output point data.
output.PointData.append(index, 'Index')
output.PointData.append(scalars, 'Scalars')

# Next, we need to define the topology i.e.
# cell information. This helix will be a single
# polyline connecting all the  points in order.
ptIds = vtk.vtkIdList()
ptIds.SetNumberOfIds(numPts)
for i in range(numPts):
   #Add the points to the line. The first value indicates
   #the order of the point on the line. The second value
   #is a reference to a point in a vtkPoints object. Depends
   #on the order that Points were added to vtkPoints object.
   #Note that this will not be associated with actual points
```

(continues on next page)

(continued from previous page)

```
    #until it is added to a vtkPolyData object which holds a
    #vtkPoints object.
    ptIds.SetId(i, i)

# Allocate the number of 'cells' that will be added. We are just
# adding one vtkPolyLine 'cell' to the vtkPolyData object.
output.Allocate(1, 1)

# Add the poly line 'cell' to the vtkPolyData object.
output.InsertNextCell(vtk.VTK_POLY_LINE, ptIds)
```



Properties (Tube1)
Scalars: Index
Vectors:
Number of Sides: 6
☒ Capping
Radius: 0.079
Vary Radius: Off
Radius Factor: 10
☐ Use Default Normal
Default Normal: 0 0 1
Display (GeometryRepresenta
Representation: Surface
Coloring: Scalars
Styling

Fig. 2.20: `Programmable Source` output generated using the script in Section 2.5.2. This visualization uses `Tube` filter to make the output polyline more prominent.

### 2.5.3 Recipes for Programmable Filter

One of the differences between the `Programmable Source` and the `Programmable Filter` is that the latter expects at least 1 input. Of course, the code in the `Programmable Filter` is free to disregard the input entirely and work exactly as `Programmable Source` . `Programmable Filter` is designed to customize data transformations. For example, it is useful when you want to compute derived quantities using expressions not directly possible with `Python Calculator` and `Calculator` or when you want to use other Python packages or even VTK filters not exposed in **ParaView** for processing the inputs.

In this section, we look at various recipes for `Programmable Filter` s.

### Adding a new point/cell data array based on an input array

Python Calculator provides an easy mechanism of computing derived variables. You can also use the Programmable Filter . Typically, for such cases, ensure that the Output DataSet Type is set to Same as Input .

```
# Code for 'Script'

# 'inputs' is set to an array with data objects produced by inputs to
# this filter.

# Get the first input.
input0 = inputs[0]

# compute a value.
dataArray = input0.PointData["V"] / 2.0

# To access cell data, you can use input0.CellData.

# 'output' is a variable set to the output dataset.
output.PointData.append(dataArray, "V_half")
```

The thing to note about this code is that it will work as expected even when the input dataset is a composite dataset such as a Multiblock dataset ( Section 1.3.1). Refer to Section 2.6 for details on how this works. There are cases, however, when you may want to explicitly iterate over blocks in an input multiblock dataset. For that, you can use the following snippet.

```
input0 = inputs[0]
if input0.IsA("vtkCompositeDataSet"):
    # iterate over all non-empty blocks in the input
    # composite dataset, including multiblock and AMR datasets.
    for block in input0:
        processBlock(block)
else:
    processBlock(input0)
```

### Computing tetrahedra volume

This recipe computes the volume for each tetrahedral cell in the input dataset. You can always simply use the Python Calculator to compute cell volume using the expression volume(inputs[0]). This recipe is provided to illustrate the API.

Ensure that the output type is set to Same as Input , and this filter assumes the input is an unstructured grid ( Section 1.3.1).

```
# Code for 'Script'.

import numpy as np

# This filter computes the volume of the tetrahedra in an unstructured mesh.
# Note, this is just an illustration and not  the most efficient way for
# computing cell volume. You should use 'Python Calculator' instead.
input0 = inputs[0]
```

(continues on next page)

(continued from previous page)

```
numTets = input0.GetNumberOfCells()

volumeArray = np.empty(numTets, dtype=np.float64)
for i in range(numTets):
       cell = input0.GetCell(i)
       p1 = input0.GetPoint(cell.GetPointId(0))
       p2 = input0.GetPoint(cell.GetPointId(1))
       p3 = input0.GetPoint(cell.GetPointId(2))
       p4 = input0.GetPoint(cell.GetPointId(3))
       volumeArray[i] = vtk.vtkTetra.ComputeVolume(p1,p2,p3,p4)

output.CellData.append(volumeArray, "Volume")
```

### Labeling common points between two datasets

In this example, the Programmable Filter takes two input datasets: A and B. It outputs dataset B with a new scalar array that labels the points in B that are also in A. You should select two datasets in the pipeline browser and then apply the programmable filter.

```
# Code for 'Script'

# Get the two inputs
A = inputs[0]
B = inputs[1]
# use len(inputs) to determine now many inputs are connected
# to this filter.

# We use numpy.in1d to test all which point coordinate components
# in  B are present in A as well.
maskX = np.in1d(B.Points[:,0], A.Points[:,0])
maskY = np.in1d(B.Points[:,1], A.Points[:,1])
maskZ = np.in1d(B.Points[:,2], A.Points[:,2])

# Combining each component mask, we get the mask for point
# itself.
mask = maskX & maskY & maskZ

# Now convert it to uint8, since bool arrays
# cannot be passed back to the VTK pipeline.
mask = np.asarray(mask, dtype=np.uint8)

# Initialize the output and add the labels array

# This ShallowCopy is needed since by default the output is
# initialized to be a shallow copy of the first input (inputs[0]),
# but we want it to be a description of the second input.
output.ShallowCopy(B.VTKObject)
output.PointData.append(mask, "labels")
```

Note in the script above the two inputs are defined in an inputs array. The order of elements in this array is determined by the order the data sources were selected in the Pipeline Browser . Hence, inputs[0] is the first data source selected and `inputs[1]` is the second.

## 2.5.4 Recipes for Programmable Annotation

The main difference between the programmable annotation and other programmable modules is that output is intended to be shown as a text representation. The output is expected to be a `vtkTable` containing a single one-component string array containing a single tuple. This tuple will be shown as a text representation, similar to the output of the `Text` source or the `Python Annotation` filter.

By default, the `Programmable Annotation` script already contains all that is needed to create this table, it just need to be filled up.

Similar to the `Programmable Filter`, the `Programmable Annotation` is designed to customize data display. For example, it is useful when you want to compute derived quantities using expressions not directly possible with `Python Annotation` and other annotation filters or when you want to use other Python packages or even VTK filters not exposed in **ParaView** for processing the inputs.

In this section, we look at various recipes for `Programmable Annotation` s.

### Displaying the number of cells with a non-zero volume

`Python Annotation` provides easy mechanism to compute many values on datasets and arrays, but conditional operation like the one proposed here require the use of a `Programmable Annotation` .

```
# Code for 'Script'

# 'inputs' is set to an array with data objects produced by inputs to
# this filter.

# Get the first input.
input0 = inputs[0]

# compute the volume of each cell of the input
vols = volume(input0)

# the codepath for composite dataset and non composite dataset can't be shared
# with this operation
if input0.IsA("vtkCompositeDataSet"):

  # create a running sum to iterate over blocks
  num = 0;

  # iterate over blocks
  for i in range(size(vols.Arrays)):
    # count the number of cells with a non-zero volume in this block
    # and add it to the running sum
    num += sum(val > 0 for val in vols.Arrays[i])
else:
  # non-composite case : just  count the number of cells with a non-zero volume
  num = sum(val > 0 for val in vols)

# standard code to display the result
to = self.GetTableOutput()
```

(continues on next page)

(continued from previous page)

```
arr = vtk.vtkStringArray()
arr.SetName("Text")
arr.SetNumberOfComponents(1)
arr.InsertNextValue(str(num))
to.AddColumn(arr)
```

### Display system and date information

Python Annotation does not provides any mechanism to import Python modules while doing that in a Programmable Annotation is trivial.

```
# Code for 'Script'

# import needed python modules
from datetime import date
import platform

# construct the string to display
string = "Date: %s\n" % date.today()
string += "System: %s" % platform.platform()

# standard code to display the string
to = self.GetTableOutput()
arr = vtk.vtkStringArray()
arr.SetName("Text")
arr.SetNumberOfComponents(1)
arr.InsertNextValue(string)
to.AddColumn(arr)
```

## 2.5.5 Python Algorithm

Programmable Source and Programmable Filter are convenient ways to prototype a Python-based data processing module. If you want to distribute such modules, or package them into modules with user interfaces, for example, then VTKPythonAlgorithmBase -based approach is recommended instead. Here, you write a Python class by subclassing VTKPythonAlgorithmBase and implementing methods to do the data processing, just like any other VTK-based filter or source. Using Python syntactic add-ons called *decorators* to annotate your class, you can easily expose the class to ParaView as a filter or a source, and ParaView will automatically add UI widgets to control parameters, etc.

Let's start with the simple script featured in Fig. 2.18. Here's the Python script to create a VTKPythonAlgorithmBase subclass for the same operation.

```
from vtkmodules.vtkCommonDataModel import vtkDataSet
from vtkmodules.util.vtkAlgorithm import VTKPythonAlgorithmBase
from vtkmodules.numpy_interface import dataset_adapter as dsa

class HalfVFilter(VTKPythonAlgorithmBase):
    def __init__(self):
        VTKPythonAlgorithmBase.__init__(self)

    def RequestData(self, request, inInfo, outInfo):
```

(continues on next page)

(continued from previous page)

```
        # get the first input.
        input0 = dsa.WrapDataObject(vtkDataSet.GetData(inInfo[0]))

        # compute a value.
        data = input0.PointData["V"] / 2.0

        # add to output
        output = dsa.WrapDataObject(vtkDataSet.GetData(outInfo))
        output.ShallowCopy(input0.VTKObject)
        output.PointData.append(data, "V_half");
        return 1
```

To expose this filter in ParaView, you have to add decorators to the class definition as follows:

```
# same imports as earlier.
from vtkmodules.vtkCommonDataModel import vtkDataSet
from vtkmodules.util.vtkAlgorithm import VTKPythonAlgorithmBase
from vtkmodules.numpy_interface import dataset_adapter as dsa

# new module for ParaView-specific decorators.
from paraview.util.vtkAlgorithm import smproxy, smproperty, smdomain

@smproxy.filter(label="Half-V Filter")
@smproperty.input(name="Input")
class HalfVFilter(VTKPythonAlgorithmBase):
    # the rest of the code here is unchanged.
    def __init__(self):
        VTKPythonAlgorithmBase.__init__(self)

    def RequestData(self, request, inInfo, outInfo):
        # get the first input.
        input0 = dsa.WrapDataObject(vtkDataSet.GetData(inInfo[0]))

        # compute a value.
        data = input0.PointData["V"] / 2.0

        # add to output
        output = dsa.WrapDataObject(vtkDataSet.GetData(outInfo))
        output.ShallowCopy(input0.VTKObject)
        output.PointData.append(data, "V_half");
        return 1
```

To use this new filter, save this to a `*.py` file, and load it as a plugin using the `Plugin Manager` from *Tools > Plugin Manager*. On success, you will see a `Half-V Filter` in the *Filters* menu.

Besides exposing class as filters, or sources, you can use the decorators to add UI widgets to call methods on the class to set parameters.

The follow examples adds a new source named `Python-based Superquadric Source Example`, with UI to control various parameters.

```
# to add a source, instead of a filter, use the `smproxy.source` decorator.
@smproxy.source(label="Python-based Superquadric Source Example")
```

(continues on next page)

(continued from previous page)

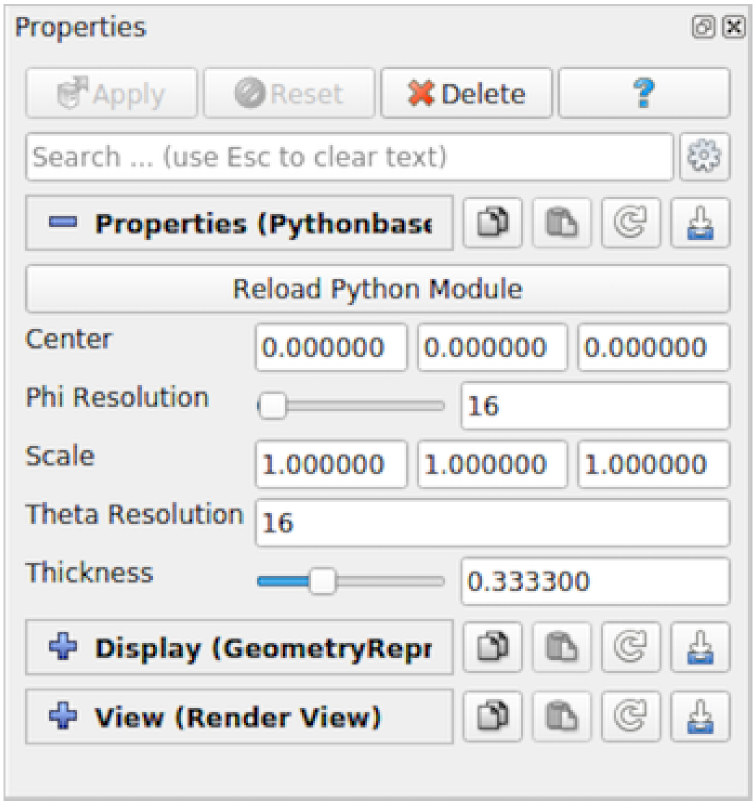
```
class PythonSuperquadricSource(VTKPythonAlgorithmBase):
    """This is dummy VTKPythonAlgorithmBase subclass that
    simply puts out a Superquadric poly data using a vtkSuperquadricSource
    internally"""
    def __init__(self):
        VTKPythonAlgorithmBase.__init__(self,
                nInputPorts=0,
                nOutputPorts=1,
                outputType='vtkPolyData')
        from vtkmodules.vtkFiltersSources import vtkSuperquadricSource
        self._realAlgorithm = vtkSuperquadricSource()

    def RequestData(self, request, inInfo, outInfo):
        from vtkmodules.vtkCommonDataModel import vtkPolyData
        self._realAlgorithm.Update()
        output = vtkPolyData.GetData(outInfo, 0)
        output.ShallowCopy(self._realAlgorithm.GetOutput())
        return 1

    # for anything too complex or not yet supported, you can explicitly
    # provide the XML for the method.
    @smproperty.xml("""
        <DoubleVectorProperty name="Center"
            number_of_elements="3"
            default_values="0 0 0"
            command="SetCenter">
            <DoubleRangeDomain name="range" />
            <Documentation>Set center of the superquadric</Documentation>
        </DoubleVectorProperty>""")
    def SetCenter(self, x, y, z):
        self._realAlgorithm.SetCenter(x,y,z)
        self.Modified()

    # In most cases, one can simply use available decorators.
    @smproperty.doublevector(name="Scale", default_values=[1, 1, 1])
    @smdomain.doublerange()
    def SetScale(self, x, y, z):
        self._realAlgorithm.SetScale(x,y,z)
        self.Modified()

    @smproperty.intvector(name="ThetaResolution", default_values=16)
    def SetThetaResolution(self, x):
        self._realAlgorithm.SetThetaResolution(x)
        self.Modified()

    @smproperty.intvector(name="PhiResolution", default_values=16)
    @smdomain.intrange(min=0, max=1000)
    def SetPhiResolution(self, x):
        self._realAlgorithm.SetPhiResolution(x)
        self.Modified()

    @smproperty.doublevector(name="Thickness", default_values=0.3333)
```

(continues on next page)

(continued from previous page)

```
    @smdomain.doublerange(min=1e-24, max=1.0)
    def SetThickness(self, x):
        self._realAlgorithm.SetThickness(x)
        self.Modified()
```

On loading this script as a plugin and creating the `Python-based Superquadric Source Example` source, the `Properties` panel will be populated as shown in Fig. 2.21.



Center 0.000000 0.000000 0.000000
Phi Resolution 16
Scale 1.000000 1.000000 1.000000
Theta Resolution 16
Thickness 0.333300

Fig. 2.21: `Properties` panel automatically generated from a decorated Python class, `PythonSuperquadricSource` .

The decorators also enable us to add new readers and writers. Here is an example writer that uses NumPy to write tables as compressed binary arrays.

```
# `smproxy.writer` decorator register the module as writer for the provided file
# extension.
@smproxy.writer(extensions="npz", file_description="NumPy Compressed Arrays", support_
↪reload=False)
@smproperty.input(name="Input", port_index=0)
# this domain lets ParaView know which types of data this writer can write.
@smdomain.datatype(dataTypes=["vtkTable"], composite_data_supported=False)
class NumpyWriter(VTKPythonAlgorithmBase):
    def __init__(self):
        VTKPythonAlgorithmBase.__init__(self, nInputPorts=1, nOutputPorts=0, inputType=
↪'vtkTable')
        self._filename = None
```

(continues on next page)

(continued from previous page)

```
    @smproperty.stringvector(name="FileName", panel_visibility="never")
    @smdomain.filelist()
    def SetFileName(self, fname):
        """Specify filename for the file to write."""
        if self._filename != fname:
            self._filename = fname
            self.Modified()

    def RequestData(self, request, inInfoVec, outInfoVec):
        from vtkmodules.vtkCommonDataModel import vtkTable
        from vtkmodules.numpy_interface import dataset_adapter as dsa

        table = dsa.WrapDataObject(vtkTable.GetData(inInfoVec[0], 0))
        kwargs = {}
        for aname in table.RowData.keys():
            kwargs[aname] = table.RowData[aname]

        import numpy
        numpy.savez_compressed(self._filename, **kwargs)
        return 1

    def Write(self):
        self.Modified()
        self.Update()
```

## 2.6 Using NumPy for processing data

In Section 2.5, we looked at several recipes for writing Python script for data processing that relied heavily on using NumPy for accessing arrays and performing operations on them. In this chapter, we take a closer look at the VTK-NumPy integration layer that makes it possible to use VTK and NumPy together, despite significant differences in the data representations between the two systems.

### 2.6.1 Teaser

Let's start with a teaser by creating a simple pipeline with `Sphere` connected to an `Elevation` filter, followed by the `Programmable Filter`. Let's see how we would access the input data object in the `Script` for the `Programmable Filter`.

```
from paraview.vtk.numpy_interface import dataset_adapter as dsa
from paraview.vtk.numpy_interface import algorithms as algs

data = inputs[0]
print(data.PointData.keys())
print(data.PointData['Elevation'])
```

This example prints out the following in the output window.

```
['Normals', 'Elevation']
[ 0.67235619  0.32764378  0.72819519  0.7388373   0.70217478  0.62546903
```

(continues on next page)

(continued from previous page)

```
  0.52391261  0.41762003  0.75839448  0.79325461  0.77003199  0.69332629
  0.57832992  0.44781935  0.72819519  0.7388373   0.70217478  0.62546903
  0.52391261  0.41762003  0.65528756  0.60746235  0.53835285  0.46164712
  0.39253765  0.34471244  0.58238     0.47608739  0.37453097  0.29782525
  0.2611627   0.27180481  0.55218065  0.42167011  0.30667371  0.22996798
  0.20674542  0.24160551  0.58238     0.47608739  0.37453097  0.29782525
  0.2611627   0.27180481  0.65528756  0.60746235  0.53835285  0.46164712
  0.39253765  0.34471244]
```

The importance lies in the last three lines. In particular, note how we used a different API to access the `PointData` and the `Elevation` array in the last two lines. Also note that, when we printed the `Elevation` array, the output didn't look like one from a `vtkDataArray`. In fact:

```
>>> elevation = data.PointData['Elevation']
>>> print(type(elevation))
<class paraview.vtk.numpy_interface.dataset_adapter.VTKArray>

>>> import numpy
>>> print(isinstance(elevation, numpy.ndarray))
True
```

So, a VTK array is a NumPy array? What kind of trickery is this, you say? What kind of magic makes the following possible?

```
data.PointData.append(elevation + 1, 'e plus one')
print(algs.max(elevation))
print(algs.max(data.PointData['e plus one']))
print(data.VTKObject)
```

The output here is:

```
0.7932546138763428
1.7932546138763428
vtkPolyData (0x7fa20d011c60)
...
Point Data:
...
Number Of Arrays: 3
Array 0 name = Normals
Array 1 name = Elevation
Array 2 name = e plus one
```

It is all in the `numpy_interface` module. It ties VTK datasets and data arrays to NumPy arrays and introduces a number of algorithms that can work on these objects. There is quite a bit to this module, and we will introduce it piece by piece in the rest of this chapter.

Let's wrap up this section with one final teaser.

```
print(algs.gradient(data.PointData['Elevation']))
```

Output:

```
[[ 0.32640398  0.32640398  0.01982867]
 [ 0.32640402  0.32640402  0.01982871]
```

(continues on next page)

(continued from previous page)

```
...
[ 0.41252578  0.20134845  0.2212007 ]
[ 0.41105482  0.21514832  0.0782456 ]]
```

Please note that this example is not very easily replicated by using pure NumPy. The gradient function returns the gradient of an unstructured grid – a concept that does not exist in NumPy. However, the ease-of-use of NumPy is there.

### 2.6.2 Understanding the dataset_adapter module

In this section, let's take a closer look at the dataset_adapter module. This module was designed to simplify accessing VTK datasets and arrays from Python and to provide a NumPy-style interface.

Let's continue with the example from the previous section. Remember, this script is being put in the Programmable Filter 's Script , connected to the Sphere , followed by the Elevation filter pipeline.

```
from vtk.numpy_interface import dataset_adapter as dsa
...
print(data)
print(isinstance(data, dsa.VTKObjectWrapper))
```

This will print:

```
<paraview.vtk.numpy_interface.dataset_adapter.PolyData object at 0x14b7caa50>
True
```

We can access the underlying VTK object using the VTKObject member:

```
print(type(data.VTKObject))
```

which produces:

```
<type 'vtkCommonDataModelPython.vtkPolyData'>
```

What we get as the $inputs$ in the Programmable Filter is actually a Python object that wraps the VTK data object itself. The Programmable Filter does this by manually calling the WrapDataObject function from the vtk.numpy_interface.dataset_adapter module on the VTK data object. Note that the WrapDataObject function will return an appropriate wrapper class for all vtkDataSet subclasses, vtkTable, and all vtkCompositeData subclasses. Other vtkDataObject subclasses are not currently supported.

VTKObjectWrapper forwards VTK methods to its VTKObject so the VTK API can be accessed directy as follows:

```
print(data.GetNumberOfCells())
96L
```

However, VTKObjectWrapper s cannot be directly passed to VTK methods as an argument.

```
from paraview.vtk.vtkFiltersGeneral import vtkShrinkPolyData
s = vtkShrinkPolyData()
s.SetInputData(data)
```

This attempt to set the data results in an error message.

```
TypeError: SetInputData argument 1: method requires a VTK object
```

Instead, we must pass the VTK object to the VTK filter, like so:

```
s.SetInputData(data.VTKObject)
```

An important thing to note in the example above is how the Python class `vtkShrinkPolyData` was imported for use in the script. In VTK, classes are organized into different groups of related functionality, and these groups can be invididually imported as Python modules. To use a class, you first identify the module in which it resides, which can be determined from the Doxygen documentation of the class [KitwareInc]. Go to the Doxygen page for the class, find the path of the file from which the documentation was generated at the bottom of the page, which has the form `dox/<first directory>/<second directory>/<class name>`. The module is then derived as `vtk<first directory><second directory>`. As an example, the documentation for `vtkShrinkPolyData` is generated from `dox/Filters/General/vtkShrinkPolyData`, hence its module is `vtkFiltersGeneral`. Then, you can import the class with a statement of the form.

### Dataset attributes

So far, we have a wrapper for VTK data objects that partially behaves like a VTK data object. This gets a little bit more interesting when we start looking at how to access the fields (arrays) contained within this dataset.

For simplicity, we will embed the output generated by the script in the code itself and use the >>> prefix to differentiate the code from the output.

```
>>> print(data.PointData)
<vtk.numpy_interface.dataset_adapter.DataSetAttributes at 0x110f5b750>

>>> print(data.PointData.keys())
['Normals', 'Elevation']

>>> print(data.CellData.keys())
[]

>>> print(data.PointData['Elevation'])
VTKArray([ 0.5       ,  0.        ,  0.45048442,  0.3117449 ,  0.11126047,
        0.        ,  0.        ,  0.        ,  0.45048442,  0.3117449 ,
        0.11126047,  0.        ,  0.        ,  0.        ,  0.45048442,
        ...,
        0.11126047,  0.        ,  0.        ,  0.        ,  0.45048442,
        0.3117449 ,  0.11126047,  0.        ,  0.        ,  0.        ], dtype=float32)

>>> elevation = data.PointData['Elevation']

>>> print(elevation[:5])
VTKArray([0.5, 0., 0.45048442, 0.3117449, 0.11126047], dtype=float32)
# Note that this works with composite datasets as well:

>>> mb = vtk.vtkMultiBlockDataSet()
>>> mb.SetNumberOfBlocks(2)
>>> mb.SetBlock(0, data.VTKObject)
>>> mb.SetBlock(1, data.VTKObject)
>>> mbw = dsa.WrapDataObject(mb)
>>> print(mbw.PointData)
<vtk.numpy_interface.dataset_adapter.CompositeDataSetAttributes instance at 0x11109f758>
```

(continues on next page)

(continued from previous page)

```
>>> print(mbw.PointData.keys())
['Normals', 'Elevation']

>>> print(mbw.PointData['Elevation'])
<vtk.numpy_interface.dataset_adapter.VTKCompositeDataArray at 0x1110a32d0>
```

It is possible to access `PointData` , `CellData` , `FieldData` , `Points` (subclasses of `vtkPointSet` only), and `Polygons` (`vtkPolyData` only) this way. We will continue to add accessors to more types of arrays through this API.

### 2.6.3 Working with arrays

For this section, let's change our test pipeline to consist of the `Wavelet` source connected to the `Programmable Filter` .

In the `Script` , we access the `RTData` point data array as follows:

```
# Code for 'Script'
from paraview.vtk.vtkFiltersGeneral import vtkDataSetTriangleFilter
image = inputs[0]
rtdata = image.PointData['RTData']

# Let's transform this data as well, using another VTK filter.
tets = vtkDataSetTriangleFilter()
tets.SetInputDataObject(image.VTKObject)
tets.Update()

# Here, now we need to explicitly wrap the output dataset to get a
# VTKObjectWrapper instance.
ugrid = dsa.WrapDataObject(tets.GetOutput())
rtdata2 = ugrid.PointData['RTData']
```

Here, we created two datasets: an image data (`vtkImageData`) and an unstructured grid (`vtkUnstructuredGrid`). They essentially represent the same data but the unstructured grid is created by tetrahedralizing the image data. So, we expect the unstructured grid to have the same points but more cells (tetrahedra).

```
from paraview.vtk.vtkFiltersGeneral import vtkDataSetTriangleFilter
```

#### The array API

`numpy_interface` array objects behave very similar to NumPy arrays. In fact, arrays from `vtkDataSet` subclasses are instances of VTKArray, which is a subclass of `numpy.ndarray` . Arrays from `vtkCompositeDataSet` and subclasses are not NumPy arrays, but they behave very similarly. We will outline the differences in a separate section. Let's start with the basics. All of the following work as expected.

As before, for simplicity, we will embed the output generated by the script in the code itself and use the >>> prefix to differentiate the code from the output.

```
>>> print(rtdata[0])
60.763466

```

(continues on next page)

(continued from previous page)

```
>>> print(rtdata[-1])
57.113735

>>> print(repr(rtdata[0:10:3]))
VTKArray([  60.76346588,   95.53707886,   94.97672272,  108.49817657], dtype=float32)

>>> print(repr(rtdata + 1))
VTKArray([ 61.76346588,  86.87795258,  73.80931091, ...,  68.51051331,
        44.34006882,  58.1137352 ], dtype=float32)

>>> print(repr(rtdata < 70))
VTKArray([ True , False, False, ...,  True,  True,  True])

# We will cover algorithms later. This is to generate a vector field.
>>> avector = algs.gradient(rtdata)

# To demonstrate that avector is really a vector
>>> print(algs.shape(rtdata))
(9261,)

>>> print(algs.shape(avector))
(9261, 3)

>>> print(repr(avector[:, 0]))
VTKArray([ 25.69367027,   6.59600449,   5.38400745, ...,  -6.58120966,
        -5.77147198,  13.19447994])
```

A few things to note in this example:

- Single component arrays always have the following shape: (n-tuples,) and not (n-tuples, 1)
- Multiple component arrays have the following shape: (n-tuples, n-components)
- Tensor arrays have the following shape: (n-tuples, 3, 3)
- The above holds even for images and other structured data. All arrays have one dimension (1 component arrays), two dimensions (multi-component arrays), or three dimensions (tensor arrays).

One more cool thing: It is possible to use boolean arrays to index arrays. Thus, the following works very nicely:

```
>>> print(repr(rtdata[rtdata < 70]))
VTKArray([ 60.76346588,  66.75043488,  69.19681549,  50.62128448,
        64.8801651 ,  57.72655106,  49.75050354,  65.05570221,
        57.38450241,  69.51113129,  64.24596405,  67.54656982,
        ...,
        61.18143463,  66.61872864,  55.39360428,  67.51051331,
        43.34006882,  57.1137352 ], dtype=float32)

>>> print(repr(avector[avector[:,0] > 10]))
VTKArray([[ 25.69367027,   9.01253319,   7.51076698],
       [ 13.1944809 ,   9.01253128,   7.51076508],
       [ 25.98717642,  -4.49800825,   7.80427408],
       ...,
       [ 12.9009738 , -16.86548471,  -7.80427504],
```

(continues on next page)

(continued from previous page)

```
       [ 25.69366837,  -3.48665428,  -7.51076889],
       [ 13.19447994,  -3.48665524,  -7.51076794]])
```

### Algorithms

You can do a lot simply using the array API. However, things get much more interesting when we start using the `numpy_interface.algorithms` module. We introduced it briefly in the previous examples. We will expand on it a bit more here. For a full list of algorithms, use `help(algs)` . Here are some self-explanatory examples:

```
>>> import paraview.vtk.numpy_interface.algorithms as algs
>>> print(repr(algs.sin(rtdata)))
VTKArray([-0.87873501, -0.86987603, -0.52497   , ..., -0.99943125,
       -0.59898132,  0.53547275], dtype=float32)

>>> print(repr(algs.min(rtdata)))
VTKArray(37.35310363769531)

>>> print(repr(algs.max(avector)))
VTKArray(34.781060218811035)

>>> print(repr(algs.max(avector, axis=0)))
VTKArray([ 34.78106022,  29.01940918,  18.34743023])

>>> print(repr(algs.max(avector, axis=1)))
VTKArray([ 25.69367027,   9.30603981,   9.88350773, ...,  -4.35762835,
        -3.78016186,  13.19447994])
```

If you haven't used the axis argument before, it is pretty easy. When you don't pass an axis value, the function is applied to all values of an array without any consideration for dimensionality. When `axis=0` , the function will be applied to each component of the array independently. When `axis=1` , the function will be applied to each tuple independently. Experiment if this is not clear to you. Functions that work this way include `sum` , `min` , `max` , `std` , and `var` .

Another interesting and useful function is where the indices of an array are returned where a particular condition occurs.

```
>>> print(repr(algs.where(rtdata < 40)))
(array([ 420, 9240]),)
# For vectors, this will also return the component index if an axis is not
# defined.

>>> print(repr(algs.where(avector < -29.7)))
(VTKArray([4357, 4797, 4798, 4799, 5239]), VTKArray([1, 1, 1, 1, 1]))
```

So far, all of the functions that we discussed are directly provided by NumPy. Many of the NumPy ufuncs are included in the algorithms module. They all work with single arrays and composite data arrays. Algorithms also provide some functions that behave somewhat differently than their NumPy counterparts. These include cross, dot, inverse, determinant, eigenvalue, eigenvector, etc. See a non-exhaustive list in Section 1.5.9. All of these functions are applied to each tuple rather than to a whole array/matrix. For example:

```
>>> amatrix = algs.gradient(avector)
>>> print(repr(algs.determinant(amatrix)))
VTKArray([-1221.2732624 ,  -648.48272183,    -3.55133937, ...,    28.2577152 ,
        -629.28507693, -1205.81370163])
```

Note that everything above only leveraged per-tuple information and did not rely on the mesh. One of VTK's biggest strengths is that its data model supports a large variety of meshes, while its algorithms work generically on all of these mesh types. The algorithms module exposes some of this functionality. Other functions can be easily implemented by leveraging existing VTK filters. We used gradient before to generate a vector and a matrix. Here it is again:

```
>>> avector = algs.gradient(rtdata)
>>> amatrix = algs.gradient(avector)
```

Functions like this require access to the dataset containing the array and the associated mesh. This is one of the reasons why we use a subclass of `ndarray` in dataset_adapter:

```
>>> print(repr(rtdata.DataSet))
<paraview.vtk.numpy_interface.dataset_adapter.DataSet at 0x11b61e9d0>
```

Each array points to the dataset containing it. Functions such as gradient use the mesh and the array together. NumPy provides a gradient function too, you say. What is so exciting about yours? Well, this:

```
>>> print(repr(algs.gradient(rtdata2)))
VTKArray([[ 25.46767712,   8.78654003,   7.28477383],
       [  6.02292252,   8.99845123,   7.49668884],
       [  5.23528767,   9.80230141,   8.3005352 ],
       ...,
       [ -6.43249083,  -4.27642155,  -8.30053616],
       [ -5.19838905,  -3.47257614,  -7.49668884],
       [ 13.42047501,  -3.26066017,  -7.28477287]])
>>> print(rtdata2.DataSet.GetClassName())
vtkUnstructuredGrid
```

Gradient and algorithms that require access to a mesh work whether that mesh is a uniform grid, a curvilinear grid, or an unstructured grid thanks to VTK's data model. Take a look at various functions in the algorithms module to see all the cool things that can be accomplished using it. In the remaining sections, we demonstrate how specific problems can be solved using these modules.

## 2.6.4 Handling composite datasets

In this section, we take a closer look at composite datasets. For this example, our pipeline is `Sphere` source, and `Cone` source is set as two inputs to the `Programmable Filter` .

We can create a multiblock dataset in the `Programmable Filter` 's `Script` as follows:

```
# Let's assume inputs[0] is the output from Sphere and
# inputs[1] is the output from Cone.
mb = vtk.vtkMultiBlockDataSet()
mb.SetBlock(0, inputs[0].VTKObject)
mb.SetBlock(1, inputs[1].VTKObject)
```

Many of VTK's algorithms work with composite datasets without any change. For example:

```
e = vtk.vtkElevationFilter()
e.SetInputData(mb)
e.Update()

mbe = e.GetOutputDataObject(0)
print(mbe.GetClassName())
```

This will output vtkMultiBlockDataSet.

Now that we have a composite dataset with a scalar, we can use numpy_interface . As before, for simplicity, we will embed the output generated by the script in the code itself and use the >>> prefix to differentiate the code from the output.

```
>>> from paraview.vtk.numpy_interface import dataset_adapter as dsa
>>> mbw = dsa.WrapDataObject(mbe)
>>> print(repr(mbw.PointData.keys()))
['Normals', 'Elevation']
>>> elev = mbw.PointData['Elevation']
>>> print(repr(elev))
<paraview.vtk.numpy_interface.dataset_adapter.VTKCompositeDataArray at 0x1189ee410>
```

Note that the array type is different than we have previously seen (VTKArray). However, it still works the same way.

```
>>> from paraview.vtk.numpy_interface import algorithms as algs
>>> print(algs.max(elev))
0.5
>>> print(algs.max(elev + 1))
1.5
```

You can individually access the arrays of each block as follows.

```
>>> print(repr(elev.Arrays[0]))
VTKArray([ 0.5       ,  0.        ,  0.45048442,  0.3117449 ,  0.11126047,
        0.        ,  0.        ,  0.        ,  0.45048442,  0.3117449 ,
        0.11126047,  0.        ,  0.        ,  0.        ,  0.45048442,
        0.3117449 ,  0.11126047,  0.        ,  0.        ,  0.        ,
        0.45048442,  0.3117449 ,  0.11126047,  0.        ,  0.        ,
        0.        ,  0.45048442,  0.3117449 ,  0.11126047,  0.        ,
        0.        ,  0.        ,  0.45048442,  0.3117449 ,  0.11126047,
        0.        ,  0.        ,  0.        ,  0.45048442,  0.3117449 ,
        0.11126047,  0.        ,  0.        ,  0.        ,  0.45048442,
        0.3117449 ,  0.11126047,  0.        ,  0.        ,  0.        ], dtype=float32)
```

Note that indexing is slightly different.

```
>>> print(elev[0:3])
[VTKArray([ 0.5,  0.,  0.45048442], dtype=float32),
 VTKArray([ 0.,  0.,  0.43301269], dtype=float32)]
```

The return value is a composite array consisting of two VTKArrays. The [] operator simply returned the first four values of each array. In general, all indexing operations apply to each VTKArray in the composite array collection. It is similar for algorithms, where:

```
>>> print(algs.where(elev < 0.5))
[(array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]),),
       (array([0, 1, 2, 3, 4, 5, 6]),)]
```

Now, let’s look at the other array called Normals.

```
>>> normals = mbw.PointData['Normals']
>>> print(repr(normals.Arrays[0]))
VTKArray([[  0.00000000e+00,   0.00000000e+00,   1.00000000e+00],
       [  0.00000000e+00,   0.00000000e+00,  -1.00000000e+00],
       [  4.33883727e-01,   0.00000000e+00,   9.00968850e-01],
       [  7.81831503e-01,   0.00000000e+00,   6.23489797e-01],
       [  9.74927902e-01,   0.00000000e+00,   2.22520933e-01],
       ...
       [  6.89378142e-01,  -6.89378142e-01,   2.22520933e-01],
       [  6.89378142e-01,  -6.89378142e-01,  -2.22520933e-01],
       [  5.52838326e-01,  -5.52838326e-01,  -6.23489797e-01],
       [  3.06802124e-01,  -3.06802124e-01,  -9.00968850e-01]], dtype=float32)
>>> print(repr(normals.Arrays[1]))
<paraview.vtk.numpy_interface.dataset_adapter.VTKNoneArray at 0x1189e7790>
```

Notice how the second array is a `VTKNoneArray` . This is because `vtkConeSource` does not produce normals. Where an array does not exist, we use a VTKNoneArray as placeholder. This allows us to maintain a one-to-one mapping between datasets of a composite dataset and the arrays in the VTKCompositeDataArray. It also allows us to keep algorithms working in parallel without a lot of specialized code.

Where many of the algorithms apply independently to each array in a collection, some algorithms are global. Take `min` and `max` , for example, as we demonstrated above. It is sometimes useful to get per-block answers. For this, you can use `_per_block` algorithms.

```
>>> print(algs.max_per_block(elev))
[0.5, 0.4330127]
```

These work very nicely together with other operations. For example, here is how we can normalize the elevation values in each block.

```
>>> _min = algs.min_per_block(elev)
>>> _max = algs.max_per_block(elev)
>>> _norm = (elev - _min) / (_max - _min)
>>> print(algs.min(_norm))
0.0
>>> print(algs.max(_norm))
1.0
```

Once you grasp these features, you should be able to use composite arrays very similarly to single arrays.

A final note on composite datasets: The composite data wrapper provided by `numpy_interface.dataset_adapter` offers a few convenience functions to traverse composite datasets. Here is a simple example:

```
>>> for ds in mbw:
>>>    print(type(ds))
<class 'paraview.vtk.numpy_interface.dataset_adapter.PolyData'>
<class 'paraview.vtk.numpy_interface.dataset_adapter.PolyData'>
```

## 2.7 Remote and parallel visualization

One of the goals of the **ParaView** application is enabling data analysis and visualization for large datasets. **ParaView** was born out of the need for visualizing simulation results from simulations run on supercomputing resources that are often too big for a single desktop machine to handle. To enable interactive visualization of such datasets, **ParaView** uses remote and/or parallel data processing. The basic concept is that if a dataset cannot fit on a desktop machine due to memory or other limitations, we can split the dataset among a cluster of machines, driven from your desktop. In this chapter, we will look at the basics of remote and parallel data processing using **ParaView**. For information on setting up clusters, please refer to the **ParaView** Wiki [ThePCommunity].

**Did you know?**

Remote and parallel processing are often used together, but they refer to different concepts, and it is possible to have one without the other.

In the case of ParaView, remote processing refers to the concept of having a client, typically `paraview` or `pvpython`, connecting to a `pvserver`, which could be running on a different, remote machine. All the data processing and, potentially, the rendering can happen on the `pvserver`. The client drives the visualization process by building the visualization pipeline and viewing the generated results.

Parallel processing refers to a concept where instead of single core — which we call a `rank` — processing the entire dataset, we split the dataset among multiple ranks. Typically, an instance of `pvserver` runs in parallel on more than one rank. If a client is connected to a server that runs in parallel, we are using both remote and parallel processing.

In the case of `pvbatch`, we have an application that operates in parallel but without a client connection. This is a case of parallel processing without remote processing.

### 2.7.1 Understanding remote processing

Let's consider a simple use-case. Let's say you have two computers, one located at your office and another in your home. The one at the office is a nicer, beefier machine with larger memory and computing capabilities than the one at home. That being the case, you often run your simulations on the office machine, storing the resulting files on the disk attached to your office machine. When you're at work, to visualize those results, you simply launch `paraview` and open the data file(s). Now, what if you need to do the visualization and data analysis from home? You have several options:

- You can copy the data files over to your home machine and then use `paraview` to visualize them. This is tedious, however, as you not only have to constantly keep copying/updating your files manually, but your machine has poorer performance due to the decreased compute capabilities and memory available on it!
- You can use a desktop sharing system like *Remote Desktop* or *VNC*, but those can be flaky depending on your network connection.

Alternatively, you can use **ParaView**'s remote processing capabilities. The concept is fairly simple. You have two separate processes: `pvserver` (which runs on your work machine) and a `paraview` client (which runs on your home machine). They communicate with each other over sockets (over an SSH tunnel, if needed). As far as using `paraview` in this mode, it's no different than how we have been using it so far – you create pipelines and then look at the data produced by those pipelines in views and so on. The pipelines themselves, however, are created remotely on the `pvserver` process. Thus, the pipelines have access to the disks on your work machine. The `Open File` dialog will in fact browse the file system on your work machine, i.e., the machine on which `pvserver` is running. Any filters that you create in your visualization pipeline execute on the `pvserver`.

While all the data processing happens on the `pvserver`, when it comes to rendering, `paraview` can be configured to either do the rendering on the server process and deliver only images to the client (remote rendering) or to deliver the geometries to be rendered to the client and let it do the rendering locally (local rendering). When remote rendering, you'll be using the graphics capabilities on your work machine (the machine running the `pvserver`). Every time a new rendering needs to be obtained (for example, when pipeline parameters are changed or you interact with the camera, etc.), the `pvserver` process will re-render a new image and deliver that to the client. When local rendering, the geometries to be rendered are delivered to the client and the client renders those locally. Thus, not all interactions require server-side processing. Only when the visualization pipeline is updated does the server need to deliver updated geometries to the client.

## 2.7.2 Remote visualization in `paraview`

### Starting a remote server

To begin using **ParaView** for remote data processing and visualization, we must first start the server application `pvserver` on the remote system. To do this, connect to your remote system using a shell and run:

```
> pvserver
```

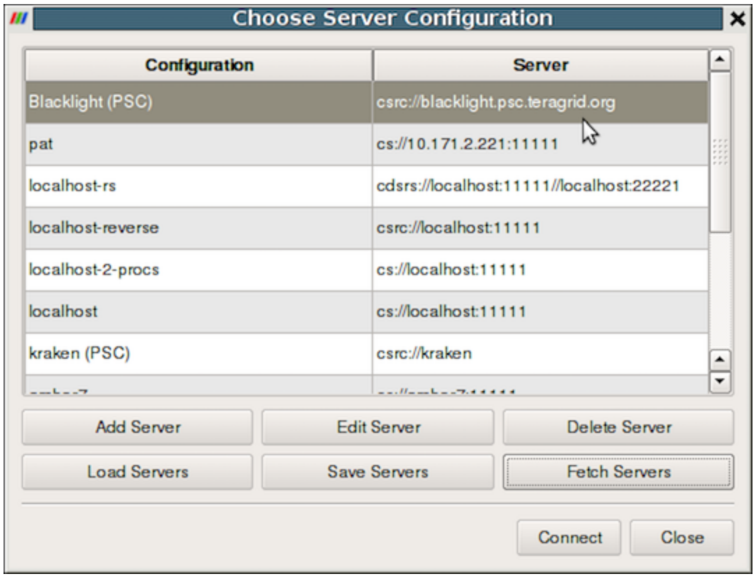
You will see this startup message on the terminal:

```
Waiting for client...
Connection URL: cs://myhost:11111
Accepting connection(s): myhost:11111
```

This means that the server has started and is listening for a connection from a client.

### Configuring a server connection

To connect to this server with the `paraview` client, select *File* > *Connect* or click the icon in the toolbar to bring up the `Choose Server Configuration` dialog.



| Configuration | Server |
|---|---|
| Blacklight (PSC) | csrc://blacklight.psc.teragrid.org |
| pat | cs://10.171.2.221:11111 |
| localhost-rs | cdsrs://localhost:11111//localhost:22221 |
| localhost-reverse | csrc://localhost:11111 |
| localhost-2-procs | cs://localhost:11111 |
| localhost | cs://localhost:11111 |
| kraken (PSC) | csrc://kraken |

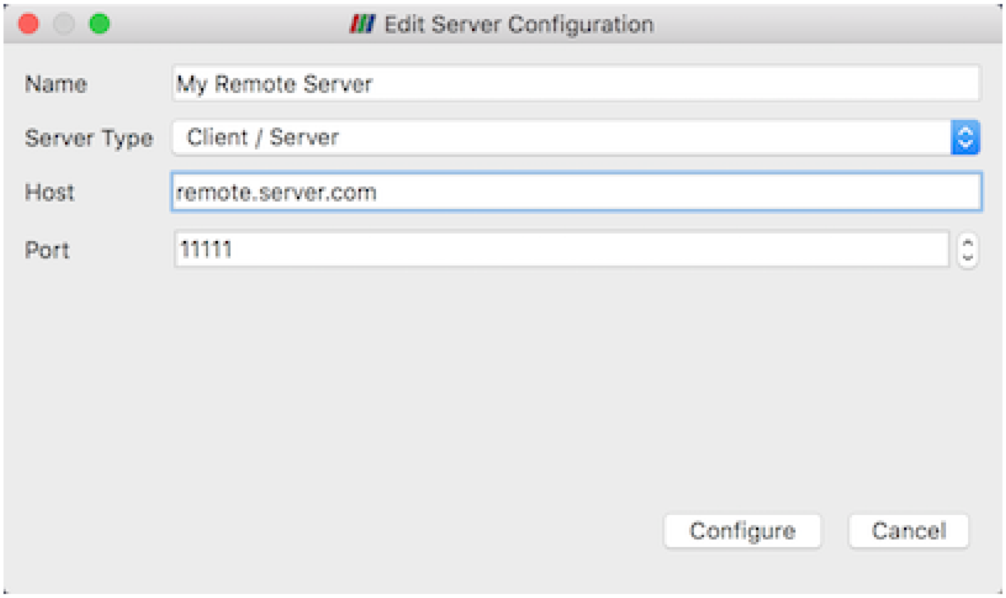
Fig. 2.22: The `Choose Server Configuration` dialog is used to connect to a server.

**Common Errors**

If your server is behind a firewall and you are attempting to connect to it from outside the firewall, the connection may not be established successfully. You may also try reverse connections ( Section 2.7.4) as a workaround for firewalls. Please consult your network manager if you have network connection problems.

---

Figure Fig. 2.22 shows the Choose Server Configuration dialog with a number of entries for remote servers. In the figure, a number of servers have already been configured, but when you first open this dialog, this list will be empty. Before you can connect to a remote server, you will need to add an entry to the list by clicking on the Add Server button. When you do, you will see the Edit Server Configuration dialog as in Figure Fig. 2.23.



Edit Server Configuration

Name: My Remote Server

Server Type: Client / Server

Host: remote.server.com

Port: 11111

Fig. 2.23: The Edit Server Configuration dialog is used to configure settings for connecting to remote servers.

You will need to set a name for the connection, the server type, the DNS name of the host on which you just started the server, and the port. The default Server Type is set to Client / Server, which means that the server will be listening for an incoming connection from the client. There are several other options for this setting that we will discuss later.

When you are done, click the Configure button. Another dialog, as shown in Fig. 2.24, will appear where you specify how to start the server. Since we started the server manually, we will leave the Startup Type on the default Manual setting. You can optionally set the Startup Type to Command and specify an external shell command to launch a server process.

When you click the Save button, this particular server configuration will be saved for future use. You can go back and edit the server configuration by selecting the entry in the list of servers and clicking the Edit Server button in the Choose Server Configuration dialog. You can delete it by clicking the Delete button.

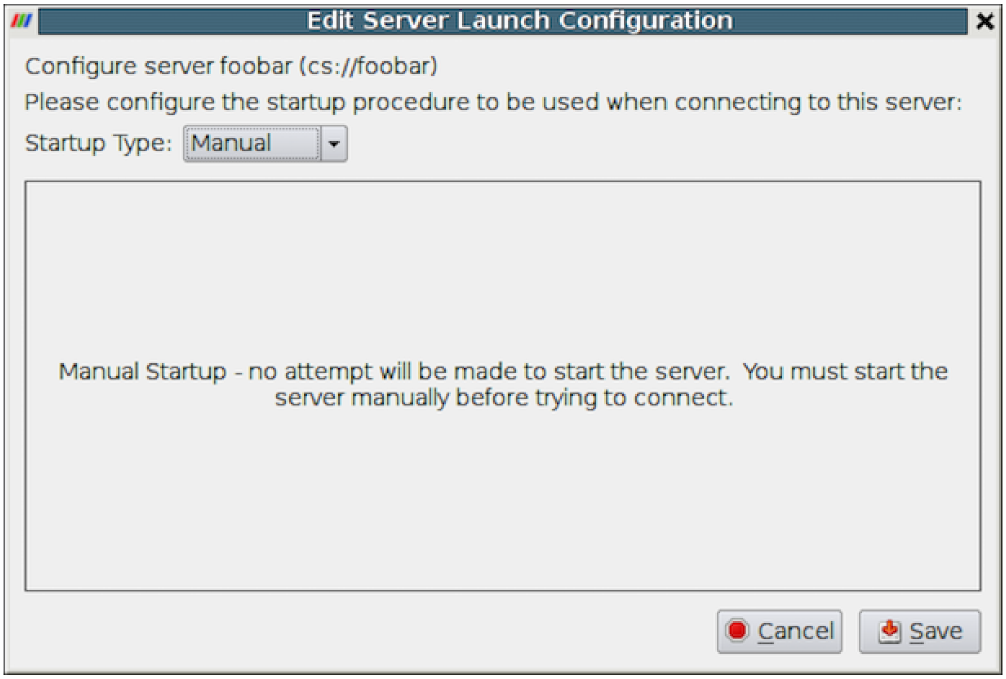
Server configurations can be imported and exported through the Choose Server Configuration dialog. Use the Load Servers button to load a server configuration file and the Save Servers button to save a server configuration file. Files can be exchanged with others to access the same remote servers.

---

**Did you know?**

Visualization centers can provide system-wide server configurations on web servers to allow non-experts to simply select an already configured **ParaView** server. These site-wide settings can be loaded with the Fetch Servers button. Advanced users may also want to specify their own servers in more details. These features are provided thanks to ParaView Server Configuration files (Section 2.7.5).

---



Edit Server Launch Configuration

Configure server foobar (cs://foobar)

Please configure the startup procedure to be used when connecting to this server:

Startup Type: Manual

Manual Startup - no attempt will be made to start the server. You must start the server manually before trying to connect.

Fig. 2.24: Configure the server manually. It must be started outside of ParaView.

### Connect to the remote server

To connect to the server, select the server configuration you just set up from the configuration list, modify the timeout in the timeout combo box if needed and click `Connect`. ParaView will try to connect to the server until it succeed or timeout is reached. In that case, you can just retry as needed. Once the connection steps succeed, we are now connected and ready to build the visualization pipelines.

**Common Errors**

ParaView does not perform any kind of authentication when clients attempt to connect to a server. For that reason, we recommend that you do not run `pvserver` on a computing resource that is open to the outside world.

ParaView also does not encrypt data sent between the client and server. If your data is sensitive, please ensure that proper network security measures have been taken. The typical approach is to use an SSH tunnel within your server configuration files using native SSH support (Section 2.7.5).

### Managing multiple clients

`pvserver` can be configured to accept connections from multiple clients at the same time. In this case only one, called the master, can interact with the pipeline. Others clients are only allowed to visualize the data. The `Collaboration Panel` shares information between connected clients.

To enable this mode, `pvserver` must be started with the `--multi-clients` flag:

```
pvserver --multi-clients
```

If your remote server is accessible from many users, you may want to restrict the access. This can be done with a connect id. If your client does not have the same connect-id as the server you want to connect to, you will be prompted for a connect-id. Then, if you are the master, you can change the connect-id in the `Collaboration Panel`.

Note that initial value for connect-id can be set by starting the pvserver (and respectively paraview) with the --connect-id flag, for instance:

```
pvserver --connect-id=147
```

The master client can also disable further connections in the Collaboration Panel so you can work alone, for instance. Once you are ready, you may allow other people to connect to the pvserver to share a visualization. This is the default feature when pvserver is started with --multi-clients --disable-further-connections.

### Setting up a client/server visualization pipeline

Using paraview when connected to a remote server is not any different than when it's being used in the default standalone mode. The only difference, as far as the user interface goes, is that the Pipeline Browser reflects the name of the server to which you are connected. The address of the server connection next to the icon changes from builtin to cs://myhost:11111 .

Since the data processing pipelines are executing on the server side, all file I/O also happens on the server side. Hence, the Open File dialog, when opening a new data file, will browse the file system local to the pvserver executable and not the paraview client.

## 2.7.3 Remote visualization in pvpython

The pvpython executable can be used by itself for visualization of local data, but it can also act as a client that connects to a remote pvserver. Before creating a pipeline in pvpython, use the Connect function:

```
# Connect to remote server "myhost" on the default port, 11111
>>> Connect("myhost") # Connect to remote server "myhost" on a
                      # specified port
>>> Connect("myhost", 11111)
```

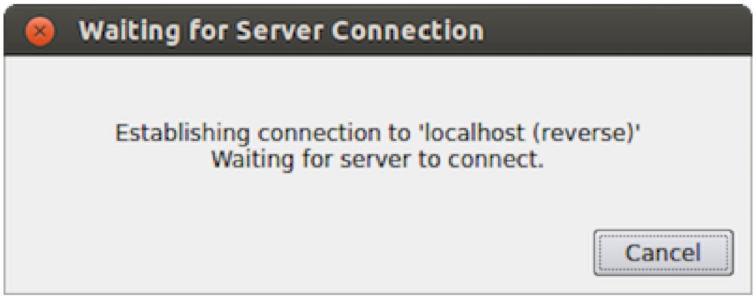
Now, when new sources are created, the data produced by the sources will reside on the server. In the case of pvpython, all data remains on the server and images are generated on the server too. Images are sent to the client for display or for saving to the local filesystem.

## 2.7.4 Reverse connections

It is frequently the case that remote computing resources are located behind a network firewall, making it difficult to connect a client outside the firewall to a server behind it. **ParaView** provides a way to set up a *reverse connection* that reverses the usual client server roles when establishing a connection.

To use a remote connection, two steps must be performed. First, in paraview, a new connection must be configured with the connection type set to reverse. To do this, open the Choose Server Configuration dialog through the *File* > *Connect* menu item. Add a new connection, setting the Name to myhost (reverse)'', and select ``Client / Server (reverse connection) for Server Type . Click Configure . In the Edit Server Launch Configuration dialog that comes up, set the Startup Type to Manual . Save the configuration. Next, select this configuration and click Connect . A message window will appear showing that the client is awaiting a connection from the server.

Second, pvserver must be started with the --reverse-connection (-rc) flag. To tell pvserver the name of the client, set the --client-host (-ch) command-line argument to the hostname of the machine on which the paraview client is running. You can specify a port with the --server-port (-sp) command-line argument.



Waiting for Server Connection

Establishing connection to 'localhost (reverse)'
Waiting for server to connect.

Fig. 2.25: Message window showing that the client is awaiting a connection from a server.

```
pvserver -rc --client-host=mylocalhost --server-port=11111
```

When the server starts, it prints a message indicating the success or failure of connecting to the client. When the connection is successful, you will see the following text in the shell:

```
Connecting to client (reverse connection requested)...
Connection URL: csrc://mylocalhost:11111
Client connected.
```

To wait for reverse connections from a `pvserver` in `pvpython`, you use `ReverseConnect` instead of `Connect` .

```
# To wait for connections from a 'pvserver' on the default port 11111
>>> ReverseConnect()

# Optionally, you can specify the port number as the argument.
>>> ReverseConnect(11111)
```

### 2.7.5 ParaView Server Configuration Files

In the `Choose Server Configuration` dialog, it is possible to `Load Servers` and `Save Servers` using the dedicated buttons. Server configurations are stored in ParView Server Configuration files (.pvsc).

These files make it possible to extensively customize the server connection process. During startup, ParaView looks at several locations for server configurations to load by default.

- **On Unix-based systems and macOS**
  - `default_servers.pvsc` in the ParaView executable directory (you can do a `ls -l /proc/<paraview PID here>/exe` to identify the executable directory)
  - `/usr/share/ParaView/servers.pvsc`
  - `$HOME/.config/ParaView/servers.pvsc` (ParaView will save user defined servers here)
- **On Windows**
  - `default_servers.pvsc` in the ParaView executable directory
  - `%COMMON_APPDATA%\ParaView\servers.pvsc`
  - `%APPDATA%\ParaView\servers.pvsc` (ParaView will save user defined servers here)

Here are a few examples of some common use-cases.

### Case One: Simple command server startup

In this use-case, we are connecting to a locally started pvserver (localhost) on the 11111 port, except that the command to start the server will be automatically called just before connecting to the server, we will wait for timeout seconds before aborting the connection.

```
<Server name="case01" resource="cs://localhost:11111" timeout="10">
  <CommandStartup>
    <Command process_wait="0" delay="5" exec="/path/to/pvserver"/>
  </CommandStartup>
</Server>
```

Here, CommandStartup element specify that a command will be run before connecting to the server. The Command element contains the details about this command, which includes process_wait, the time in seconds that paraview will wait for the process to start, delay, the time in seconds paraview will wait after running the command to try to connect and finally, exec, which is the command that will be run and usually contains the path to pvserver but could also contain a mpi command to start pvserver distributed or to any script or executable on the localhost filesystem.

### Case Two: Simple remote server connection

In this use-case, we are setting a configuration for a simple server connection (to a pvserver processes) running on a node named “amber1”, at port 20234. The pvserver process will be started manually by the user.

```
<Server name="case02" resource="cs://amber1:20234">
  <ManualStartup/>
</Server>
```

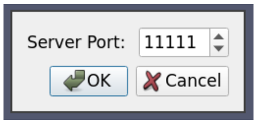
Here, name specify the name of the server as it will appear in the pipeline browser, resource identifies the type if the connection (cs – implying client-server), host name and port. If the port number i.e. :20234 part is not specified in the resource, then the default port number (which is 11111) is assumed. Since the user starts pvserver processes manually, we use ManualStartup.

### Case Three: Server connection with user-specified port

This is the same as case two except that we want to ask the user each time the port number to connect to the pvserver at.

```
<Server name="case03" resource="cs://amber1">
  <ManualStartup>
    <Options>
      <Option name="PV_SERVER_PORT" label="Server Port: ">
        <Range type="int" min="1" max="65535" step="1" default="11111" />
      </Option>
    </Options>
  </ManualStartup>
</Server>
```

Here the only difference is the Options element. This element is used to specify run-time options that the user specifies when connecting to the server, see this section for a list of available run-time options. In this case, we want to show the user an integral spin-box to select the port number, hence we use the Range element to specify the type of the option. When the user connects to this server, he is shown a dialog similarly to the following image:


Server Port: 11111

### Case Four: Simple connection to a data-server/render-server

This is the same as case two, except that instead of a single server (i.e. `pvserver`), we are connecting to a separate render-server/data-server with `pvdataserver` running on port 20230 on amber1 and `pvrenderserver` running port 20233 on node amber2.

```
<Server name="case04" resource="cdsrs://amber1:20230//amber2:20233">
  <ManualStartup />
</Server>
```

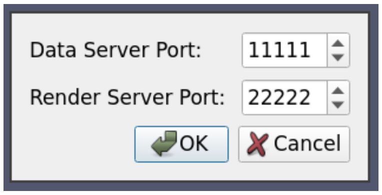
The only difference with case two, is the `resource` specification. `cdsrs` indicates that it is a client-dataserver-renderserver configuration. The first `host:port` pair is the dataserver while the second one is the render server.

### Case Five: Connection to a data-server/render-server with user specified server port

This is a combination of case three and case four, where we want to ask the user for the port number for both the render server and the data server.

```
<Server name="case05" resource="cdsrs://localhost//localhost">
  <ManualStartup>
    <Options>
      <Option name="PV_DATA_SERVER_PORT" label="Data Server Port: ">
        <Range type="int" min="1" max="65535" step="1" default="11111" />
      </Option>
      <Option name="PV_RENDER_SERVER_PORT" label="Render Server Port: ">
        <Range type="int" min="1" max="65535" step="1" default="22222" />
      </Option>
    </Options>
  </ManualStartup>
</Server>
```

The XML is quite self-explanatory given what we has already been explained above. The options dialog produced by this XML looks as follows:


Data Server Port: 11111

Render Server Port: 22222

### Case Six: Reverse Connection

By default the client connects to the server processes. However it is possible to tell the paraview client to wait for the server to connect to it instead. This is called a reverse connection. In such a case the server processes must be started with `--reverse-connection` or `--rc` flag.

To indicate reverse connection in the server configuration xml, the only change is suffixing the `resource` protocol part with `rc` (for reverse connection). eg.

```
resource="csrc://localhost" -- connect to pvserver on localhost using reverse connection
resource="cdsrsrc://localhost//localhost" -- connect to pvdataserver/pvrenderserver␣
↪using reverse connection.
```

So a simple local reverse connection server configuration, similarly to case one, would look like this

```xml
<Server name="case06" resource="csrc://localhost:11111">
  <CommandStartup>
    <Command exec="/path/to/pvserver --reverse-connection --client-host=localhost"/>
  </CommandStartup>
</Server>
```

Here the `--client-host=localhost` in the `exec` is actually not needed has this is the default.

### Case Seven: Server command with option

As we have seen in case one, the server can be started by ParaView on connection, but this can be combined with the `Option` element as seen in case three to create a dynamically generated server command.

```xml
<Server name="case07" resource="cs://localhost">
  <CommandStartup>
    <Options>
      <!-- The user chooses the port on which to start the server -->
      <Option name="PV_SERVER_PORT" label="Server Port: ">
        <Range type="int" min="1" max="65535" step="1" default="11111" />
      </Option>
    </Options>
    <Command delay="5" exec="/path/to/pvserver">
      <Arguments>
        <Argument value="--server-port=$PV_SERVER_PORT$" />
      </Arguments>
    </Command>
  </CommandStartup>
</Server>
```

As with case one, we are using `CommandStartup` and `Command` elements. Command line arguments can be passed to the command executed using the `Arguments` element. All runtime environment variables specified as `$name$` are replaced with the actual values. Eg. in this case `$PV_SERVER_PORT$` gets replaced by the port number chosen by the user in the options dialog.

### Case Eight: Using connection-id and random port

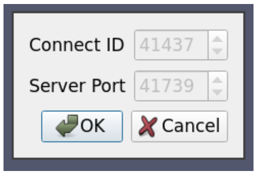
In many cases, a server cluster may be running multiple pvserver (or pvdataserver/pvrenderserver) processes for different users. In that case we need some level of authentication between the server and the client. This can be achieved (at a very basic level) with the connect-id option. If specified on the command line when starting the server processes (using `--connect-id`) then the server will allow only that client which reports the same connection id to connect.

We also want to avoid port collision with other users, so we use a random port for the server connection.

Here is an example similarly to case seven but with a connect-id option and random server port.

```
<Server name="case08" resource="cs://localhost">
  <CommandStartup>
    <Options>
      <Option name="PV_CONNECT_ID" label="Connect ID" readonly="true">
        <Range type="int" min="1" max="65535" default="random" />
      </Option>
      <Option name="PV_SERVER_PORT" label="Server Port" readonly="true">
        <Range type="int" min="11111" max="65535" default="random" />
      </Option>
    </Options>
    <Command exec="/path/to/pvserver" delay="5">
      <Arguments>
        <Argument value="--connect-id=$PV_CONNECT_ID$" />
        <Argument value="--server-port=$PV_SERVER_PORT$" />
      </Arguments>
    </Command>
  </CommandStartup>
</Server>
```

In this case, the `readonly` attribute on the `Option` indicates that the value cannot be changed by the user, it is only shown for information purposes. The default value for the `PV_CONNECT_ID` and `PV_SERVER_PORT` is set to random so that ParaView makes up a value at run time. Of course, in a production environnement they should be assigned by user instead of randomly generated.


Connect ID 41437
Server Port 41739

### Case Nine: Starting server using ssh

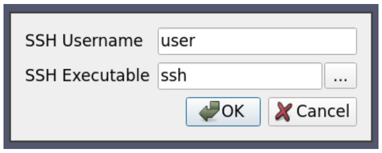
In this use case the server process is spawned on some remote host using specifically crafted ssh command. We want the user to be able to specify the ssh executable. We also want to preserve the ssh executable path across ParaView sessions so that the user does not have to enter it each time.

```
<Server name="case09" resource="cs://localhost:11111">
  <CommandStartup>
    <Options>
      <Option name="SSH_USER" label="SSH Username" save="true">
        <!-- choose the username. Since 'save' is true, this value will
             be maintained across sessions -->
```

(continues on next page)

(continued from previous page)

```
        <String default="user" />
      </Option>
      <Option name="SSH_EXE" label="SSH Executable" save="true">
        <!-- select the SSH executable. Since 'save' is true, this value will
             also be maintinaed across sessions -->
        <File default="ssh" />
      </Option>
    </Options>
    <Command exec="$SSH_EXE$" delay="5">
      <Arguments>
        <Argument value="-L8080:amber5:11111" /> <!-- port forwarding -->
        <Argument value="amber5" />
        <Argument value="-l" />
        <Argument value="$SSH_USER$" />
        <Argument value="/path/to/pvserver" />
      </Arguments>
    </Command>
  </CommandStartup>
</Server>
```


SSH Username user
SSH Executable ssh

Note here that the value for the exec attribute is set to $SSH_EXE$ hence it gets replaced by the user selected ssh executable. We use the optional attribute save on the Option element to tell ParaView to preserve the user chosen value across ParaView sessions so that the user doesn't have to enter the username and the ssh executable every time he wants to connect to this server.

**Did you know?**

While SSH connection can be started by crafting the command, ParaView now support SSH connection natively by specyfing a SSHCommand, see below for more information.

### Case Ten: Starting server using custom script with custom user-settable options

This example will illustrate the full capability of server configuration. Suppose we have a custom script "MyServerStarter" that takes in multiple arguments to start the server process. We want the user to be able to set up values for these arguments when he tries to connect to using this configuration. As an example, let's say MyServerStarter takes the following arguments:

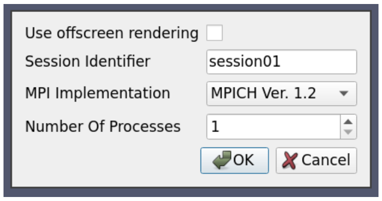
- --force-offscreen-rendering – to indicate use of offscreen rendering
- --force-onscreen-rendering – to indicate on-screen rendering (this can be assumed from absence of --force-offscreen-rendering, but we are using it as an example)
- --session-name=<string> – some string identifying the session
- --mpitype=<mpich1.2|mpich2|openmpi> – choose between available MPI implementations
- --num-procs=<num> – number of server processess

- `--server-port` – port number passed the pvserver processes

All (except the –server-port) of these must be settable by the user at the connection time. This can be achieved as follows:

```
<Server name="case10" resource="cs://localhost">
  <CommandStartup>
    <Options>
      <Option name="OFFSCREEN" label="Use offscreen rendering">
        <Boolean true="--use-offscreen" false="--use-onscreen" default="false" />
      </Option>
      <Option name="SESSIONID" label="Session Identifier">
        <String default="session01"/>
      </Option>
      <Option name="MPITYPE" label="MPI Implementation">
        <Enumeration default="mpich1.2">
          <Entry value="mpich1.2" label="MPICH Ver. 1.2" />
          <Entry value="mpich2" label="MPICH Ver 2.0" />
          <Entry value="openmpi" label="Open MPI" />
        </Enumeration>
      </Option>
      <Option name="NUMPROC" label="Number Of Processes">
        <Range type="int" min="1" max="256" step="4" default="1" />
      </Option>
    </Options>
    <Command exec="/path/to/MyServerStarter" delay="5">
      <Arguments>
        <Argument value="--server-port=$PV_SERVER_PORT$" />
        <Argument value="--mpitype=$MPITYPE$" />
        <Argument value="--num-procs=$NUMPROC$" />
        <Argument value="$OFFSCREEN$" />
        <Argument value="--session-name=$SESSIONID$" />
      </Arguments>
    </Command>
  </CommandStartup>
</Server>
```

Each `Option` defines a new run-time variable that can be accessed as `${name}$` in the `Command` section. When the user tries to connect using this configuration, he is shown the following options dialog:



Use offscreen rendering □
Session Identifier: session01
MPI Implementation: MPICH Ver. 1.2
Number Of Processes: 1

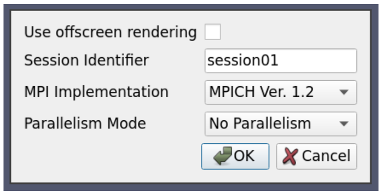
This can be extended to start the server processes using ssh or any batch scheduler etc. as may be the required by the server administrator. This can also be set up to use reverse connection (by changing the protocol in the resource attribute).

### Case Eleven: Case Ten + Switch Statement

This is same as case ten with one change: We no longer allow the user to choose the number of processes. Instead, the number of processes is automatically selected based on the value of the distribution combobox.

```
<Server name="case11" resource="cs://localhost">
  <CommandStartup>
    <Options>
      <Option name="OFFSCREEN" label="Use offscreen rendering">
        <Boolean true="--use-offscreen" false="--use-onscreen" default="false" />
      </Option>
      <Option name="SESSIONID" label="Session Identifier">
        <String default="session01"/>
      </Option>
      <Option name="MPITYPE" label="MPI Implementation">
        <Enumeration default="mpich1.2">
          <Entry value="mpich1.2" label="MPICH Ver. 1.2" />
          <Entry value="mpich2" label="MPICH Ver 2.0" />
          <Entry value="openmpi" label="Open MPI" />
        </Enumeration>
      </Option>
      <Option name="DISTRIBUTION" label="Distribution Mode">
        <Enumeration default="notDistributed">
          <Entry value="notDistributed" label="Not Distributed" />
          <Entry value="someDistribution" label="Some Distribution" />
          <Entry value="highDistribution" label="Highly Distributed" />
        </Enumeration>
      </Option>
      <Switch name="DISTRIBUTION">
        <Case value="notDistributed">
          <Set name="NUMPROC" value="1" />
        </Case>
        <Case value="someDistribution">
          <Set name="NUMPROC" value="2" />
        </Case>
        <Case value="highDistribution">
          <Set name="NUMPROC" value="10" />
        </Case>
      </Switch>
    </Options>
    <Command exec="/path/to/MyServerStarter" delay="5">
      <Arguments>
        <Argument value="--server-port=$PV_SERVER_PORT$" />
        <Argument value="--mpitype=$MPITYPE$" />
        <Argument value="--num-procs=$NUMPROC$" />
        <Argument value="$OFFSCREEN$" />
        <Argument value="--session-name=$SESSIONID$" />
      </Arguments>
    </Command>
  </CommandStartup>
</Server>
```

The `Switch` statement can only have `Case` statements as children, while the `Case` statement can only have `Set` statements as children. `Set` statements are not much different from `Option` except that the value is fixed and the user is not



Use offscreen rendering □
Session Identifier: session01
MPI Implementation: MPICH Ver. 1.2
Parallelism Mode: No Parallelism

prompted to set that value.

### Case Twelve: Simple SSH run server command

If Command element let you craft SSH commands, it can be quite complex to do so and the pipeline browser in ParaView may not show the correct server as it could connect through a ssh tunnel.

Here, similarly to case one, we use native ssh support to start a pvserver process remotely, on amber1, before connecting to it directly on the default port:

```
<Server name="case12" resource="cs://amber1">
  <CommandStartup>
    <SSHCommand exec="/path/to/pvserver" delay="5">
      <SSHConfig user="user"/>
    </SSHCommand>
  </CommandStartup>
</Server>
```

First SSHCommand element is used instead of Command so that ParaView knows to use native ssh support. Then the SSHConfig element is used to configure the ssh connection. The user attribute is the SSH user to use with SSH. If a password is needed, it will be asked on the terminal used to run ParaView, which may not be visible in certain cases.

### Case Thirteen: SSH run server command with complex config

Here, similarly to case twelve, we use native ssh support to start a pvserver process remotely, on amber1, before connecting to it directly, but we specify much more specifically the configuration to use.

```
<Server name="case13" resource="cs://amber1">
  <CommandStartup>
    <Options>
      <!-- The user chooses the port on which to start the server -->
      <Option name="PV_SERVER_PORT" label="Server Port: ">
        <Range type="int" min="1" max="65535" step="1" default="11111" />
      </Option>
    </Options>
    <SSHCommand exec="/path/to/pvserver" delay="5">
      <SSHConfig user="user">
        <Terminal exec="/usr/bin/xterm"/>
        <SSH exec="/usr/bin/ssh"/>
      </SSHConfig>
      <Arguments>
        <Argument value="--server-port=$PV_SERVER_PORT$"/>
      </Arguments>
    </SSHCommand>
  </CommandStartup>
</Server>
```

Inside the SSHConfig element, we use different elements. Here, Terminal element is used to specify that ParaView will try to open a terminal to ask the user for his password. Here, the terminal executable is specified using the exec attribute. If it was not, ParaView would try to find one automatically (Linux and Windows). When troubleshooting server configuration, not using Terminal element is suggested as the terminal will close as soon as the command finish executing. On Linux, it is also possible to replace the Terminal element by the AskPass element to specify the ParaView should use SSH_ASKPASS so that a ask-pass binary is used when asking for the SSH password. Finally, the SSH element specify the SSH binary to use thanks to its exec attribute.

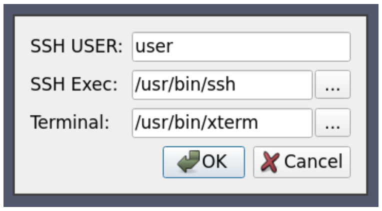
We also use PV_SERVER_PORT, similarly to case seven to let user select the port to connect to.

### Case Fourteen: SSH run server command with user chosen config

Here, similarly to case thirteen and five, we use native ssh support to start a pvserver process remotely, on amber1, before connecting to it directly, but we let the user choose interactively some SSH options.

```
<Server name="case14" resource="cs://amber1">
  <CommandStartup>
    <Options>
      <Option label="SSH USER:" name="SSH_USER" save="true">
        <String default="user"/>
      </Option>
      <Option label="SSH Exec:" name="SSH_EXEC" save="true">
        <File default="/usr/bin/ssh" />
      </Option>
      <Option label="Terminal:" name="TERMINAL" save="true">
        <File default="/usr/bin/xterm"/>
      </Option>
    </Options>
    <SSHCommand exec="/path/to/pvserver" delay="5">
      <SSHConfig user="$SSH_USER$">
        <Terminal exec="$TERMINAL$"/>
        <SSH exec="$SSH_EXEC$"/>
      </SSHConfig>
      <Arguments>
        <Argument value="--server-port=$PV_SERVER_PORT$"/>
      </Arguments>
    </SSHCommand>
  </CommandStartup>
</Server>
```

Similarly to all other options, SSH related options can be set interactively by the user. Here we let the user set the SSH user, the SSH executable as well as the Terminal executable to use when connecting through ssh.


SSH USER: user

SSH Exec: /usr/bin/ssh

Terminal: /usr/bin/xterm

### Case Fifteen: Ssh run server command with reverse connection

Similarly to case twelve and thirteen, we use native ssh support to start a reverse connection pvserver process remotely, on amber1, before letting it connect to ParaView using the hostname of the client on static non-default port.

```xml
<Server name="case15" resource="csrc://amber1:11112">
  <CommandStartup>
    <SSHCommand exec="/path/to/pvserver" delay="5">
      <SSHConfig user="user">
        <Terminal/>
      </SSHConfig>
      <Arguments>
        <Argument value="--reverse-connection"/>
        <Argument value="--client-host=$PV_CLIENT_HOST$"/>
        <Argument value="--server-port=$PV_SERVER_PORT$"/>
      </Arguments>
    </SSHCommand>
  </CommandStartup>
</Server>
```

The only difference with case twelve is in the `ressource`, which now contain the reverse connection as well as the usage of `$PV_CLIENT_HOST$` in the arguments for the reverse connection, automatically set to the hostname of the client which the server should be able to resolve to an ip to connect to.

### Case Sixteen: Secured Connection to a Server trough SSH tunnel

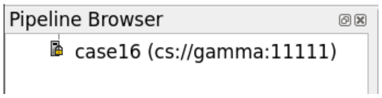
To communicate securely trough a ssh tunnel, something usually done with a crafted command looking like this: `ssh -L 8080:localhost:port user@remote /path/to/pvserver --sp=port`

You would then connect on a server on `localhost:8080` within ParaView. This is complex to set up either manually of with a `Command` element. Also, the true server and port will not appear in the pipeline browser in ParaView.

This is however natively supported with `SSHCommand` element. Here we create a secured SSH tunnel to amber1 before connecting through the SSH tunnel on the 11111 port.

```xml
<Server name="case16" resource="cs://amber1:11111">
  <CommandStartup>
    <SSHCommand exec="/path/to/pvserver" delay="5">
      <SSHConfig user="user">
        <Terminal/>
        <PortForwarding local="8080"/>
      </SSHConfig>
      <Arguments>
        <Argument value="--server-port=$PV_SERVER_PORT$"/>
      </Arguments>
    </SSHCommand>
  </CommandStartup>
</Server>
```

Similarly to case thirteen, we only add a `PortForwarding` element in the `SSHConfig` element with a `local` attribute port, so that ParaView creates a SSH tunnel to connect through. The `$PV_SERVER_PORT$` is automatically set to the value of the port to use within the SSH tunnel. In ParaView, the tunnel will be integrated nicely in the UI with the correct port and hostname in the pipeline browser, the server icon will look different with a small lock to note the secured nature of this connection:



Pipeline Browser
case16 (cs://gamma:11111)

### Case Seventeen: Secured Reverse Connection from a Server trough SSH tunnel

Similarly to case sixteen, a reverse connection through a SSH tunnel would require to craft a command like this one: `ssh -R 8080:localhost:port user@remote /path/to/pvserver --rc --ch=localhost --sp=8080`

You would then connect on a reverse connection server on localhost:8080 on ParaView. This is complex to set up either manually or with a `Command` element. Also, the true server and port will not appear in the pipeline browser in ParaView.

This is however natively supported with `SSHCommand`. Here we create a reverse secured SSH tunnel to amber1 before reverse connecting through the SSH tunnel on a specific port.

```
<Server name="case17" resource="csrc://amber1:11115">
  <CommandStartup>
    <SSHCommand exec="/path/to/pvserver" delay="5">
      <SSHConfig user="user">
        <Terminal/>
        <PortForwarding local="8080"/>
      </SSHConfig>
      <Arguments>
        <Argument value="--reverse-connection"/>
        <Argument value="--client-host=localhost"/>
        <Argument value="--server-port=$PV_SSH_PF_SERVER_PORT$"/>
      </Arguments>
    </SSHCommand>
  </CommandStartup>
</Server>
```

We also specify a `PortForwarding` element in the `SSHConfig` with a `local` port to trigger the creation of the SSH tunnel. Finally, `$PV_SSH_PF_SERVER_PORT$` variable should be use by the server to connect through the SSH tunnel to the client.

**Did you know?**

While SSH native support can simplify the configuration file, some cases are still not covered and require complex custom command. Client/DataServer/RenderServer SSH setup are not supported natively, nested SSH tunnels are not supported natively either. To create such setup, use of complex Command is needed.

### PVSC file XML Schema

Here is the exhaustive PVSC file XML schema

- The `<Servers>` tag is the root element of the document, which contains zero-to-many `<Server>` tags.
- Each `<Server>` tag represents a configured server:
  - The `name` attribute uniquely identifies the server configuration, and is displayed in the user interface.

- The `timeout` attribute specifies the maximum amount of time (in seconds) that the client will wait for the server to start, -1 means forever, default to 60.
- The `resource` attribute specifies the type of server connection, server host(s) and optional port(s) for making a connection. Values are
- `cs://<host>:<port>` - for client-pvserver configurations with forward connection i.e. client connects to the server. If not specified, port default to 11111.
- `csrc://<host>:<port>` - for client-pvserver configurations with reverse connection i.e. server connects to the client. If not specified, port default to 11111.
- `cdsrs://<ds-host>:<ds-port>//<rs-host>:<rs-port>` - for client-pvdataserver-pvrenderserver configurations with forward connection. If not specified, ds-port default to 11111, rs-port default to 22222.
- `cdsrsrc://<ds-host>:<ds-port>//<rs-host>:<rs-port>` - for client-pvdataserver-pvrenderserver configurations with reverse connection. If not specified, ds-port default to 11111, rs-port default to 22222.
- The `<CommandStartup>` tag is used to run an external command to start a server.
  - An optional `<Options>` tag can be used to prompt the user for options required at startup.
    - Each `<Option>` tag represents an option that the user will be prompted to modify before startup.
      - The `name` attribute defines the name of the option, which will become its variable name when used as a run-time environment variable, and for purposes of string-substitution in `<Argument>` tags.
      - The `label` attribute defines a human-readable label for the option, which will be used in the user interface.
      - The optional `readonly` attribute can be used to designate options which are user-visible, but cannot be modified.
      - The optional `save` attribute can be used to indicate that the value choosen by the user for this option will be saved in the ParaView settins so that it's preserved across ParaView sessions.
      - A `<Range>` tag designates a numeric option that is only valid over a range of values.
        - The `type` attribute controls the type of number controlled. Valid values are `int` for integers and `double` for floating-point numbers, respectively.
        - The `min` and `max` attributes specify the minimum and maximum allowable values for the option (inclusive).
        - The `step` attribute specifies the preferred amount to increment / decrement values in the user interface.
        - The `default` attribute specifies the initial value of the option.
        - As a special-case for integer ranges, a default value of `random` will generate a random number as the default each time the user is prompted for a value. This is particularly useful with PV_CONNECT_ID and PV_SERVER_PORT.

  - A `<String>` tag designates an option that accepts freeform text as its value.
  - The `default` attribute specifies the initial value of the option.
  - A `<File>` tag designates an option that accepts freeform text along with a file browse button to assist in choosing a filepath
  - The `default` attribute specifies the initial value of the option.
  - A `<Boolean>` tag designates an option that is either on/off or true/false.
  - The `true` attribute specifies what the option value will be if enabled by the user.
  - The `false` attribute specifies what the option value will be if disabled by the user.
  - The `default` attribute specifies the initial value of the option, either `true` or `false`.
  - An `<Enumeration>` tag designates an option that can be one of a finite set of values.
    - The `default` attribute specifies the initial value of the option, which must be one of its enumerated values.
    - Each `<Entry>` tag describes one allowed value.
    - The `name` tag specifies the value for that choice.
    - The `label` tag provides human-readable text that will be displayed in the user interface for that choice.

- A `<Command>` tag is used to specify the external command and its startup arguments.
  - The `exec` attribute specifies the filename of the command to be run. The system PATH will be used to search for the command, unless an absolute path is specified. If the value for this attribute is specified as $STRING$, then it will be replaced with the value of a predefined or user-defined (through <Option/>) variable.
  - The `process_wait` attribute specifies a waiting time (in seconds) that ParaView will wait for the exec command to start. Default to 0.
  - The `delay` attribute specifies a delay (in seconds) between the time the startup command completes and the time that the client attempts a connection to the server. Default to 0.
  - `<Argument>` tags are command-line arguments that will be passed to the startup command.
  - String substitution is performed on each argument, replacing each `$STRING$` with the value of a predefined or user-defined variable.
  - Arguments whose value is an empty string are not passed to the startup command.
- A `<SSHCommand>` tag is used to specify the external command to be started through ssh
  - All `<Command>` related attributes and tags still applies.

  - A <SSHConfig> tag is used to set the SSH configuration.
    - The user attribute is used to set the SSH username
    - A <Terminal> tag is used to inform ParaView to use a terminal to issue ssh commands and ask user for password when needed.
    - The exec attribute specifies the terminal executable to use, if not set, ParaView will try to find one automatically, on Windows and Linux only.
    - A <AskPass> tag, which should not be used with <Terminal> tag, can be used to inform ParaView to use a AskPass, using the SSH_ASKPASS environnement variable, on Linux only.
    - A <SSH> tag, used to specify
    - the exec attribute that specifies the SSH executable to use.
    - A <PortForwarding> tag, that indicates to ParaView that a SSH tunnel will need to be created, either forward or reverse depending on the connection type.
    - the local attribute that specified the local port to use the SSH tunel.
- The <ManualStartup> tag indicates that the user will manually start the given server prior to connecting.
- An optional <Options> tag can be used to prompt the user for options required at startup. Note that PV_SERVER_PORT, PV_DATA_SERVER_PORT, PV_RENDER_SERVER_PORT, and PV_CONNECT_ID are the only variables that make sense in this context.

### Startup Command Variables

When a startup command is run, its environment will include all of the user-defined variables specified in <Option> tags, plus the following predefined variables:

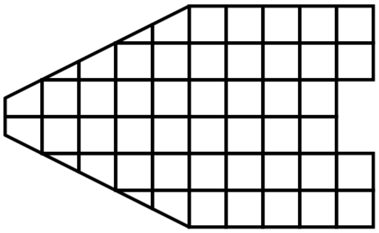
- PV_CLIENT_HOST
- PV_CONNECTION_URI
- PV_CONNECTION_SCHEME
- PV_VERSION_MAJOR (e.g. 5)
- PV_VERSION_MINOR (e.g. 9)
- PV_VERSION_PATCH (e.g. 1)
- PV_VERSION (e.g. 5.9)
- PV_VERSION_FULL (e.g. 5.9.1)
- PV_SERVER_HOST
- PV_SERVER_PORT
- PV_SSH_PF_SERVER_PORT
- PV_DATA_SERVER_HOST
- PV_DATA_SERVER_PORT

- `PV_RENDER_SERVER_HOST`
- `PV_RENDER_SERVER_PORT`
- `PV_CLIENT_PLATFORM` (possible values are: `Windows`, `Apple`, `Linux`, `Unix`, `Unknown`)
- `PV_APPLICATION_DIR`
- `PV_APPLICATION_NAME`
- `PV_CONNECT_ID`

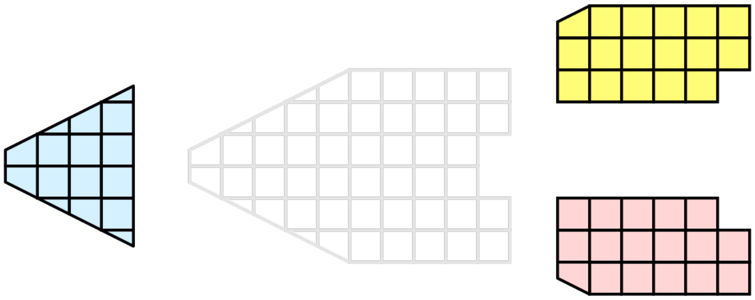
These options can be used in the <`Command`> or <`SSHCommand`> elements part of the PVSC files, as well as extracted from the environnement when running the command. If an <`Option`> element defines a variable with the same name as a predefined variable, the <`Option`> element value takes precedence. This can be used to override defaults that are normally hidden from the user. As an example, if a site wants users to be able to override default port numbers, the server configuration might specify an <Option> of PV_SERVER_PORT.

### 2.7.6 Understanding parallel processing

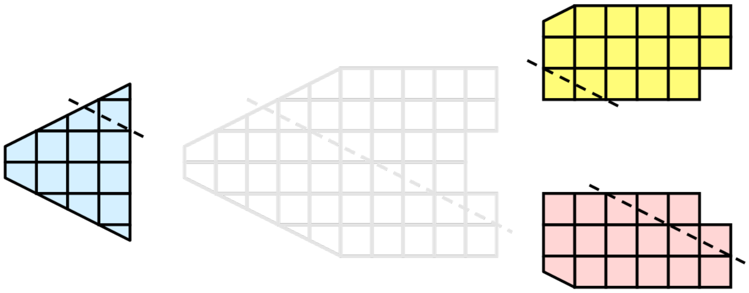
Parallel processing, put simply, implies processing the data in parallel, simultaneously using multiple workers. Typically, these workers are different processes that could be running on a multicore machine or on several nodes of a cluster. Let's call these ranks. In most data processing and visualization algorithms, work is directly related to the amount of data that needs to be processed, i.e., the number of cells or points in the dataset. Thus, a straight-forward way of distributing the work among ranks is to split an input dataset into multiple chunks and then have each rank operate only an independent set of chunks. Conveniently, for most algorithms, the result obtained by splitting the dataset and processing it separately is same as the result that we'd get if we processed the dataset in a single chunk. There are, of course, exceptions. Let's try to understand this better with an example. For demonstration purposes, consider this very simplified mesh.



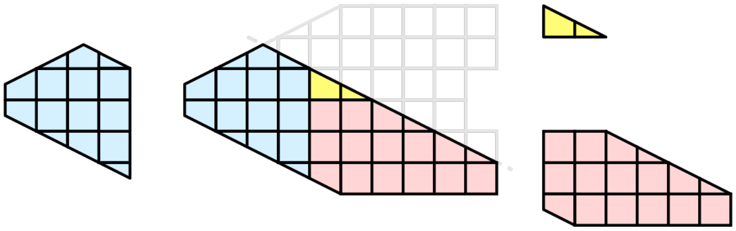
Now, let us say we want to perform visualizations on this mesh using three processes. We can divide the cells of the mesh as shown below with the blue, yellow, and pink regions.



Once partitioned, some visualization algorithms will work by simply allowing each process to independently run the algorithm on its local collection of cells. Take clipping as an example. Let's say that we define a clipping plane and give that same plane to each of the processes.

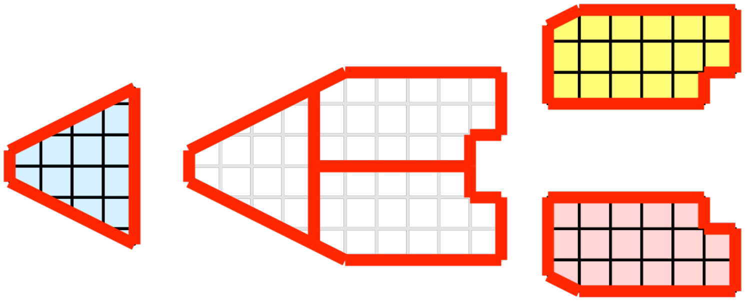


Each process can independently clip its cells with this plane. The end result is the same as if we had done the clipping serially. If we were to bring the cells together (which we would never actually do for large data for obvious reasons), we would see that the clipping operation took place correctly.



### Ghost levels

Unfortunately, blindly running visualization algorithms on partitions of cells does not always result in the correct answer. As a simple example, consider the **external faces** algorithm. The external faces algorithm finds all cell faces that belong to only one cell, thereby, identifying the boundaries of the mesh.

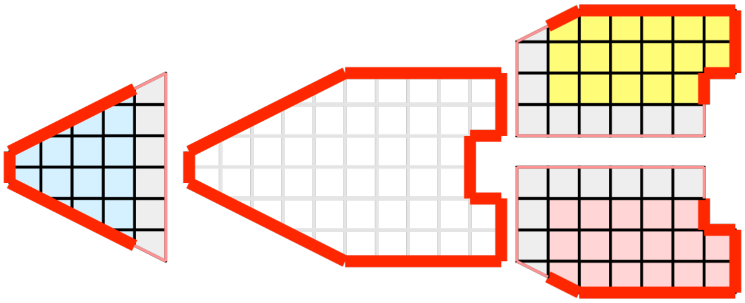


Oops! We see that when all the processes ran the external faces algorithm independently, many internal faces where incorrectly identified as being external. This happens where a cell in one partition has a neighbor in another partition. A process has no access to cells in other partitions, so there is no way of knowing that these neighboring cells exist.

The solution employed by ParaView and other parallel visualization systems is to use **ghost cells** . Ghost cells are cells that are held in one process but actually belong to another. To use ghost cells, we first have to identify all the neighboring cells in each partition. We then copy these neighboring cells to the partition and mark them as ghost cells, as indicated with the gray colored cells in the following example.

When we run the external faces algorithm with the ghost cells, we see that we are still incorrectly identifying some internal faces as external. However, all of these misclassified faces are on ghost cells, and the faces inherit the ghost status of the cell from which it came. ParaView then strips off the ghost faces, and we are left with the correct answer.

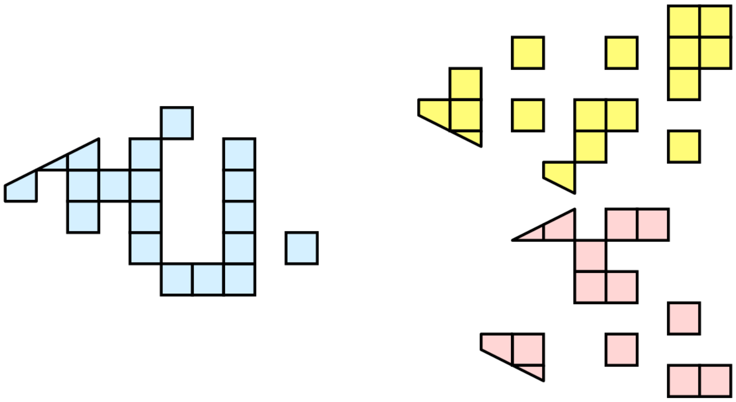
In this example, we have shown one layer of ghost cells: only those cells that are direct neighbors of the partition's cells. ParaView also has the ability to retrieve multiple layers of ghost cells, where each layer contains the neighbors



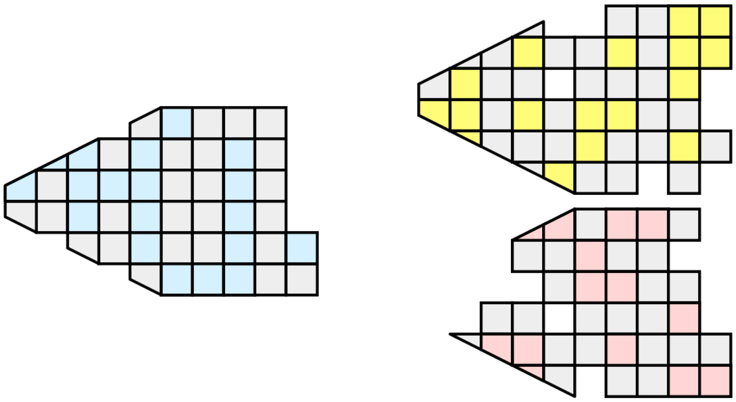
of the previous layer not already contained in a lower ghost layer or in the original data itself. This is useful when we have cascading filters that each require their own layer of ghost cells. They each request an additional layer of ghost cells from upstream, and then remove a layer from the data before sending it downstream.

### Data partitioning

Since we are breaking up and distributing our data, it is prudent to address the ramifications of how we partition the data. The data shown in the previous example has a **spatially coherent** partitioning. That is, all the cells of each partition are located in a compact region of space. There are other ways to partition data. For example, you could have a random partitioning.



Random partitioning has some nice features. It is easy to create and is friendly to load balancing. However, a serious problem exists with respect to ghost cells.



In this example, we see that a single level of ghost cells nearly replicates the entire dataset on all processes. We have thus removed any advantage we had with parallel processing. Because ghost cells are used so frequently, random partitioning is not used in ParaView.

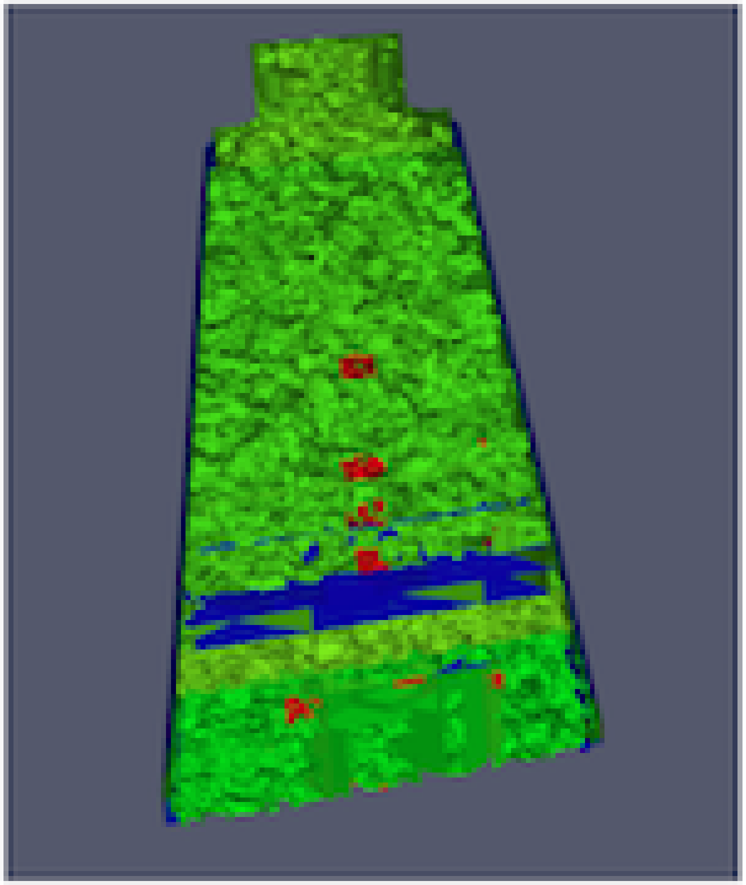
### D3 Filter

The previous section described the importance of load balancing and ghost levels for parallel visualization. This section describes how to achieve that.

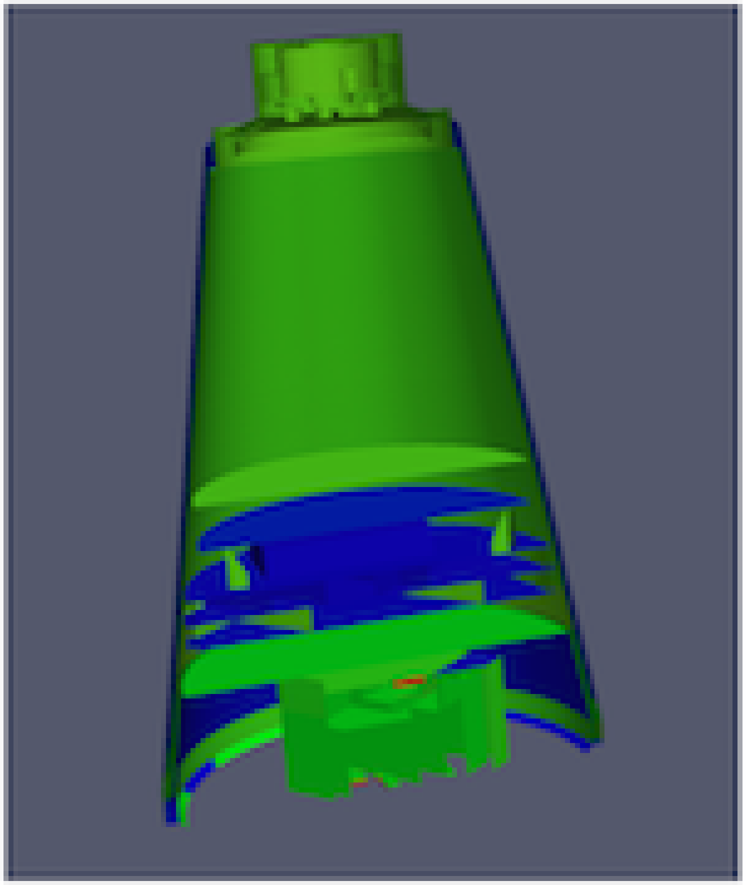
Load balancing and ghost cells are handled automatically by ParaView when you are reading structured data (image data, rectilinear grid, and structured grid). The implicit topology makes it easy to break the data into spatially coherent chunks and identify where neighboring cells are located.

It is an entirely different matter when you are reading in unstructured data (poly data and unstructured grid). There is no implicit topology and no neighborhood information available. ParaView is at the mercy of how the data was written to disk. Thus, when you read in unstructured data, there is no guarantee of how well-load balanced your data will be. It is also unlikely that the data will have ghost cells available, which means that the output of some filters may be incorrect.

Fortunately, ParaView has a filter that will both balance your unstructured data and create ghost cells. This filter is called D3, which is short for distributed data decomposition. Using D3 is easy; simply attach the filter (located in *Filters > Alphabetical > D3*) to whatever data you wish to repartition.



The most common use case for D3 is to attach it directly to your unstructured grid reader. Regardless of how well-load balanced the incoming data might be, it is important to be able to retrieve ghost cell so that subsequent filters will generate the correct data. The example above shows a cutaway of the extract surface filter on an unstructured grid. On the left, we see that there are many faces improperly extracted because we are missing ghost cells. On the right, the problem is fixed by first using the D3 filter.



### 2.7.7 Ghost Cells Generation

If your unstructured grid data is already partitioned satisfactorily but does not have ghost cells, it is possible to generate them using the `Ghost Cells` filter. This filter can be attached to a source just like the `D3` filter. Unlike `D3` , it will not repartition the dataset, it will only generate ghost cells, which is needed for some algorithms to execute correctly.

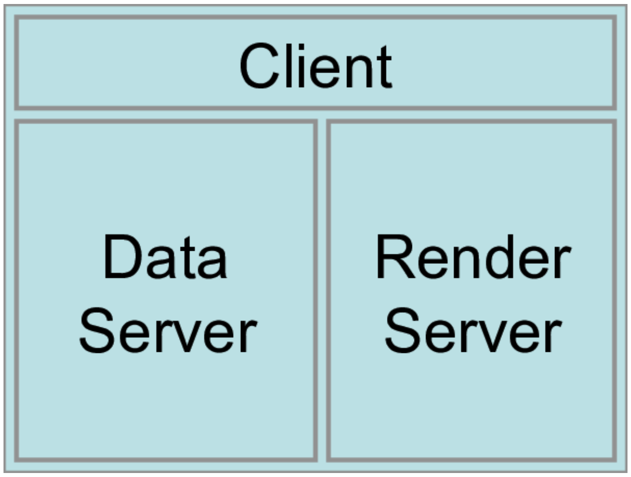
The `Ghost Cells` filter has several options. `Build If Required` tells the filter to generate ghost cells only if required by a downstream filter. Since computing ghost cells is a computationally and communications intensive process, turning this option on can potentially save a lot of processing time. The `Minimum Number Of Ghost Levels` specifies at least how many ghost levels should be generated if `Build If Required` is off. Downstream filters may request more ghost levels than this minimum, in which case the `Ghost Cells` will generate the requested number of ghost levels. The `Use Global Ids` option makes use of a GlobalIds array if it is present if on. If off, ghost cells are determined by coincident points.

### 2.7.8 ParaView architecture

Before we see how to use ParaView for parallel data processing, let's take a closer look at the ParaView architecture. ParaView is designed as a three-tier client-server architecture. The three logical units of ParaView are as follows.

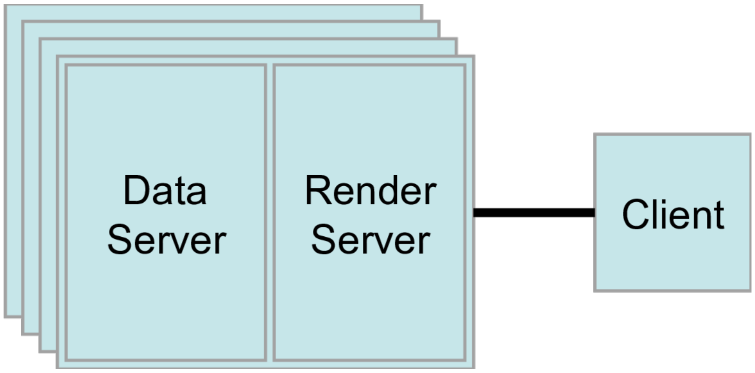
- **Data Server** The unit responsible for data reading, filtering, and writing. All of the pipeline objects seen in the pipeline browser are contained in the data server. The data server can be parallel.
- **Render Server** The unit responsible for rendering. The render server can also be parallel, in which case built-in parallel rendering is also enabled.
- **Client** The unit responsible for establishing visualization. The client controls the object creation, execution, and destruction in the servers, but does not contain any of the data (thus allowing the servers to scale without bottlenecking on the client). If there is a GUI, that is also in the client. The client is always a serial application.

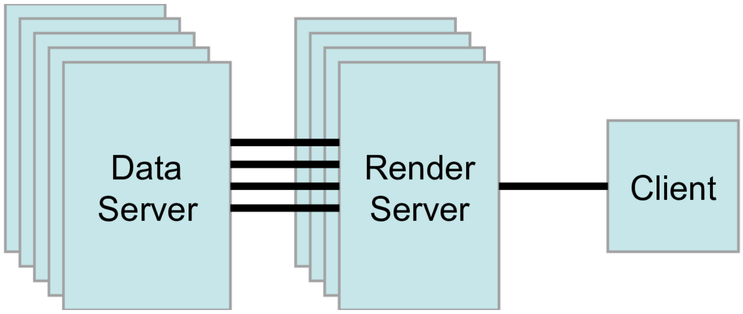
These logical units need not by physically separated. Logical units are often embedded in the same application, removing the need for any communication between them. There are three modes in which you can run ParaView.



The first mode, with which you are already familiar, is **standalone** mode. In standalone mode, the client, data server, and render server are all combined into a single serial application. When you run the `paraview` application, you are automatically connected to a **builtin** server so that you are ready to use the full features of ParaView.

The second mode is **client-server** mode. In client-server mode, you execute the `pvserver` program on a parallel machine and connect to it with the `paraview` client application (or `pvpython`). The `pvserver` program has both the data server and render server embedded in it, so both data processing and rendering take place there. The client and server are connected via a socket, which is assumed to be a relatively slow mode of communication, so data transfer over this socket is minimized. We saw this mode of operation in Section 2.7.2.





The third mode is **client-render server-data server** mode. In this mode, all three logical units are running in separate programs. As before, the client is connected to the render server via a single socket connection. The render server and data server are connected by many socket connections, one for each process in the render server. Data transfer over the sockets is minimized.

Although the client-render server-data server mode is supported, we almost never recommend using it. The original intention of this mode is to take advantage of heterogeneous environments where one might have a large, powerful computational platform and a second smaller parallel machine with graphics hardware in it. However, in practice, we find any benefit is almost always outstripped by the time it takes to move geometry from the data server to the render server. If the computational platform is much bigger than the graphics cluster, then use software rendering on the large computational platform. If the two platforms are about the same size, just perform all the computation on the graphics cluster. The executables used for this mode are `paraview` (or `pvpython`) (acting as the client), `pvdataserver` for the data-server, and `pvrenderserver` for the render-server.

### 2.7.9 Parallel processing in `paraview` and `pvpython`

To leverage parallel processing capabilities in `paraview` or `pvpython`, one has to use remote visualization, i.e., one has to connect to a `pvserver`. The processing for connecting to this `pvserver` is not different from what we say in Section 2.7.2 and Section 2.7.3. The only thing that changes is how the `pvserver` is launched.

You can start `pvserver` to run on more than one processing core using `mpirun` .

```
mpirun -np 4 pvserver
```

This will run `pvserver` on four processing cores. It will still listen for an incoming connection from a client on the default port. The big difference when running `pvserver` this way is that when data is loaded from a source, it will be distributed across the four cores if the data source is `parallel aware` and supports distributing the data across the different processing cores.

To see how this data is distributed, run `pvserver` as the command above and connect to it with `paraview`. Next, create another `Sphere` source using *Source > Sphere*. Change the array to color by to `vtkProcessId` . You will see an image like Figure Fig. 2.26.

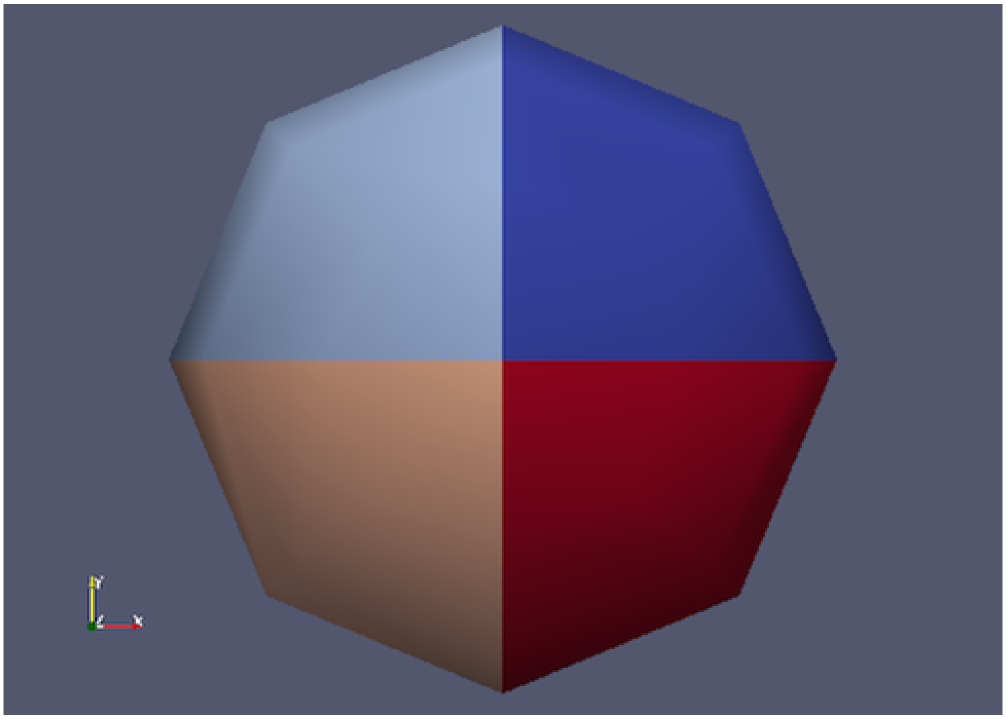


Fig. 2.26: Sphere source colored by `vtkProcessId` array that encodes the processing core on which the sphere data resides. Here, the sphere data is split among the four processing cores invoked by the command `mpirun -np 4 pvserver`.

If a data reader or source is not `parallel aware`, you can still get the benefits of spreading the data among processing cores by using the `D3` filter. This filter partitions a dataset into convex regions and transfers each region to a different processing core. To see an example of how D3 partitions a dataset, create a *Source* > *Wavelet* while `paraview` is still connected to the `pvserver`. Next, select *Filters* > *Alphabetical* > *D3* and click `Apply` . The output of `D3` will not initially appear different from the original wavelet source. If you color by `vtkProcessId` , however, you will see the four partitions that have been distributed to the server processing cores.

### 2.7.10 Using `pvbatch`

In Section 2.7.9, we said that to use parallel processing capabilities, one has to use remote visualization, i.e., one must use ParaView in a client-server mode with the client (`paraview` or `pvpython`) connecting to a server (`pvserver`) that is being run in parallel using `mpirun` . However, there is one exception: `pvbatch`. `pvpython` and `pvbatch` are quite similar in that both are similarly to the `python`

executable that can be used to run Python scripts. The extra thing that these executables do when compared with the standard `python` is that they initialize the environment so that any scripts that you run will be able to locate the ParaView Python modules and libraries automatically. `pvpython` is exactly like the `paraview` executable without the GUI. You can think of it as the GUI from `paraview` is replaced by a Python interpreter in `pvpython`. `pvbatch`, on the other hand, can be thought of a `pvserver` where, instead of taking the control command from a remote client (`paraview` or `pvpython`), in `pvbatch`, the commands are taken from a Python script that is executed in the `pvbatch` executable itself. Since `pvbatch` is akin to the `pvserver`, unlike `pvpython`, it can be run in parallel using `mpirun` . In that case, the root rank (or the first rank or the rank with index $0$) is the one that acts as the client, interpreting the Python script to execute the commands. Since `pvbatch` is designed to act is its own server, you cannot connect to a remote server in the Python script, i.e., you cannot use `simple.Connect` . Furthermore, `pvbatch` is designed

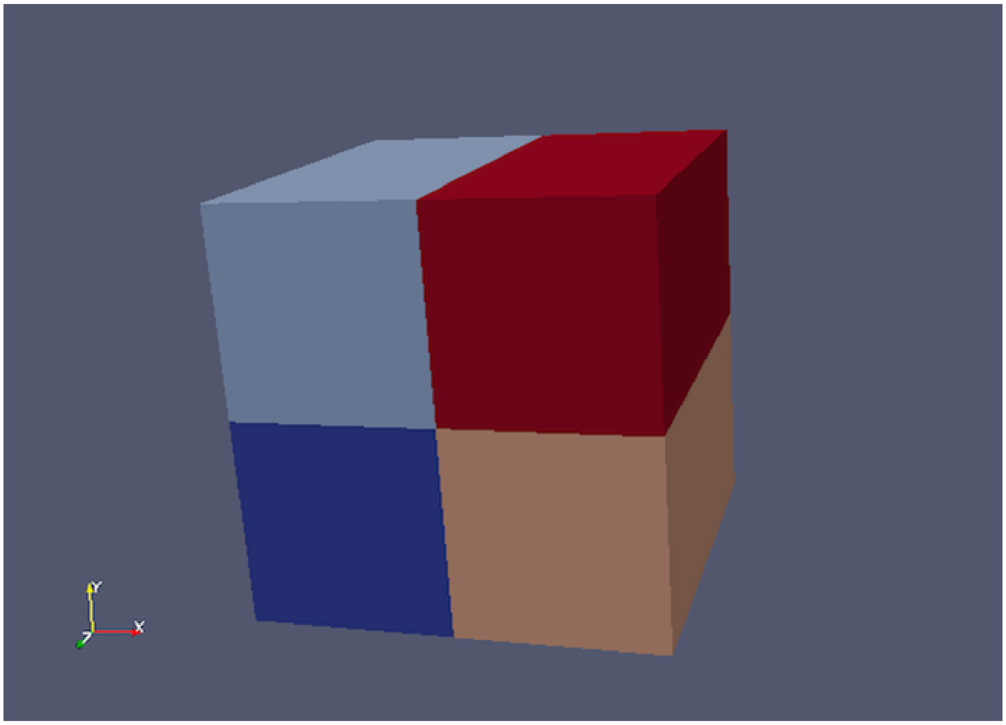


Fig. 2.27: Wavelet source processed by the D3 filter and colored by `vtkProcessId` array. Note how four regions of the image data are split evenly among the four processing cores when `pvserver` is run with `mpirun -np 4 pvserver`.

for batch operation, which means that you can only specify the Python script as a command line argument. Unlike `pvpython`, you cannot run this executable to get an interactive shell to enter Python commands.

```
# process the sample.py script in single process mode.
> pvbatch sample.py

# process the sample.py script in parallel.
> mpirun -np 4 pvbatch sample.py
```

In general, you should use `pvpython` if you will be using the interpreter interactively and `pvbatch` if you are running in parallel.

### 2.7.11 Fetching data to the client

Section 1.3.3 describes how to obtain information *about* a data object, but not how to access the data object itself. This section describes several ways to access data from within a Python script. The client/server nature of ParaView requires a couple steps to access the raw data. The Python script runs on the client side in either `pvpython` or `paraview`, so one step involves moving the data from the server to the client. This can be accomplished with the following:

```
from paraview.simple import *
Connect("myhost")

# Create a sphere source on myhost
s = Sphere()
full_sphere = servermanager.Fetch(s)
```

Here, the full dataset is moved from the server to the client.

The second step is required to deal with the fact that data on the remote server may be split across distributed processes. By default, `servermanager.Fetch(s)` appends all the pieces on the different remote processes and produces the appended dataset on the client. The exact append operation depends on the type of dataset being retrieved. Composite datasets are merged by treating the dataset piece on each distributed process as a block merged into a new multiblock dataset, polygonal datasets are appended into a single polygonal dataset, rectilinear grids are appended into a single rectilinear grid, and other datasets are appended into an unstructured grid. Distributed image datasets cannot currently be fetched to the client. Care must be taken when fetching an entire dataset to the client because the data that fits on many distributed processes on a remote system may not fit in client memory.

Another option is to fetch just a single piece of the dataset on one remote process to the client. To do this, pass the rank number of the remote process from which you want to retrieve the data to the `Fetch` function, e.g.,

```python
# Retrieve the piece of the dataset on remote process 2
s = Sphere()
sphere_piece = servermanager.Fetch(s, 2)
```

Lastly, `servermanager.Fetch` provides a way to apply helper filters to the dataset that run at two stages. The filter for the first stage is applied to the data on each remote process, and the filter for the second stage is applied to the results from the first stage after they are gathered to the root server process. The results from the second stage of filtering are then transferred from the root server process to the client.

In the next example, the `Extract Surface` filter is applied to a source with data on each process in the first stage. The results are then assembled with the `Append Geometry` filter and sent to the client.

```python
s = Sphere()
extract = servermanager.filters.ExtractSurface()
append = servermanager.filters.AppendGeometry()
full_surface = servermanager.Fetch(s, extract, append)
```

The second filter must be able to accept multiple connections and handle the output dataset type from the first filter.

### 2.7.12 Rendering

Rendering is the process of synthesizing the images that you see based on your data. The ability to effectively interact with your data depends highly on the speed of the rendering. Thanks to advances in 3D hardware acceleration, fueled by the computer gaming market, we have the ability to render 3D quickly even on moderately-priced computers. But, of course, the speed of rendering is proportional to the amount of data being rendered. As data gets bigger, the rendering process naturally gets slower.

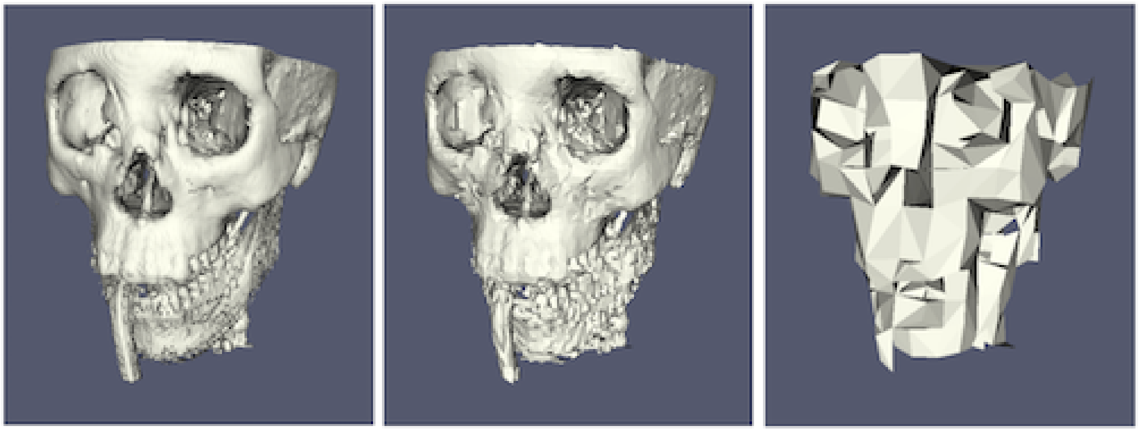
To ensure that your visualization session remains interactive, ParaView supports two modes of rendering that are automatically flipped as necessary. In the first mode, **still render** , the data is rendered at the highest level of detail. This rendering mode ensures that all of the data is represented accurately. In the second mode, **interactive render** , speed takes precedence over accuracy. This rendering mode endeavors to provide a quick rendering rate regardless of data size.

While you are interacting with a 3D view (for example, rotating, panning, or zooming with the mouse), ParaView uses an interactive render. This is because, during the interaction, a high frame rate is necessary to make these features usable and because each frame is immediately replaced with a new rendering while the interaction is occurring so that fine details are less important during this mode. At any time when interaction of the 3D view is not taking place, ParaView uses a still render so that the full detail of the data is available as you study it. As you drag your mouse in a 3D view to move the data, you may see an approximate rendering. The full detail will be presented as soon as you release the mouse button.

The interactive render is a compromise between speed and accuracy. As such, many of the rendering parameters concern when and how lower levels of detail are used.

### Basic Rendering Settings

Some of the most important rendering options are the LOD parameters. During interactive rendering, the geometry may be replaced with a lower **level of detail** ( **LOD** ), an approximate geometry with fewer polygons.



The resolution of the geometric approximation can be controlled. In the proceeding images, the left image is the full resolution, the middle image is the default decimation for interactive rendering, and the right image is ParaView's maximum decimation setting.

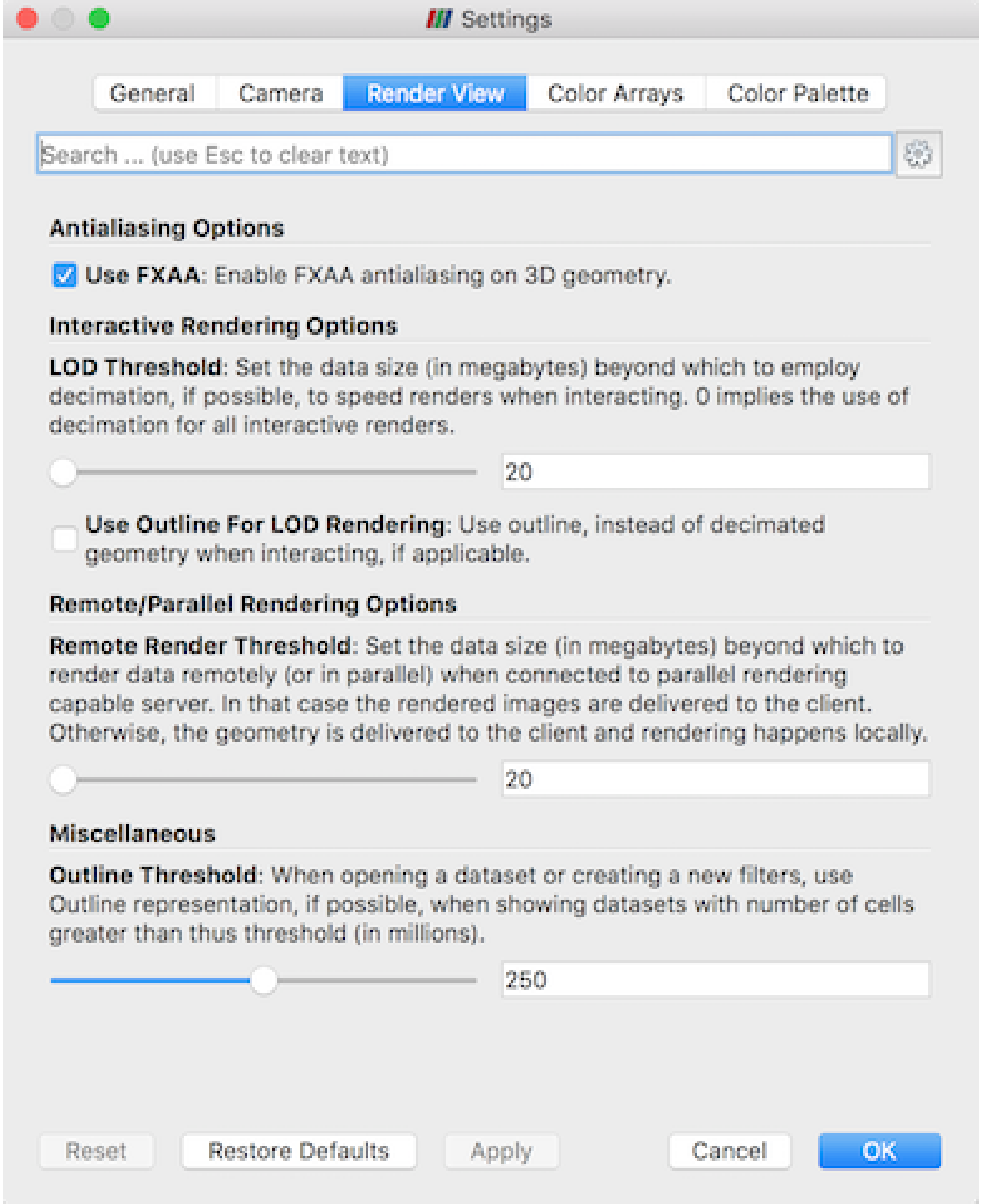
The 3D rendering parameters are located in the settings dialog box, which is accessed in the menu from the *Edit* > *Settings* menu (*ParaView* > *Preferences* on the Mac). The rendering options in the dialog are in the `Render View` tab.

The options pertaining to the geometric decimation for interactive rendering are located in a section labeled `Interactive Rendering Options` . Some of these options are considered advanced, so to access them, you have to either toggle on the advanced options with the  button or search for the option using the edit box at the top of the dialog. The interactive rendering options include the following.

- `LOD Threshold` : Set the data size at which to use a decimated geometry in interactive rendering. If the geometry size is under this threshold, ParaView always renders the full geometry. Increase this value if you have a decent graphics card that can handle larger data. Try decreasing this value if your interactive renders are too slow.
- `LOD Resolution` : Set the factor that controls how large the decimated geometry should be. This control is set to a value between 0 and 1. 0 produces a very small number of triangles but, possibly, with a lot of distortion. 1 produces more detailed surfaces but with larger geometry.
- `Non Interactive Render Delay` : Add a delay between an interactive render and a still render. ParaView usually performs a still render immediately after an interactive motion is finished (for example, releasing the mouse button after a rotation). This option can add a delay that can give you time to start a second interaction before the still render starts, which is helpful if the still render takes a long time to complete.
- `Use Outline For LOD Rendering` : Use an outline in place of decimated geometry. The outline is an alternative for when the geometry decimation takes too long or still produces too much geometry. However, it is more difficult to interact with just an outline.

ParaView contains many more rendering settings. Here is a summary of some other settings that can effect the rendering performance regardless of whether ParaView is run in client-server mode or not. These options are spread among several categories, and several are considered advanced.

- **`Translucent Rendering Options`**
  - `Depth Peeling` : Enable or disable depth peeling. Depth peeling is a technique ParaView uses to properly render translucent surfaces. With it, the top surface is rendered and then "peeled away" so that the next lower surface can be rendered and so on. If you find that making surfaces transparent really slows things down or renders completely incorrectly, then your graphics hardware may not be implementing the depth peeling extensions well; try shutting off depth peeling.



Antialiasing Options

☑ **Use FXAA**: Enable FXAA antialiasing on 3D geometry.

Interactive Rendering Options

**LOD Threshold**: Set the data size (in megabytes) beyond which to employ decimation, if possible, to speed renders when interacting. 0 implies the use of decimation for all interactive renders.

20

☐ **Use Outline For LOD Rendering**: Use outline, instead of decimated geometry when interacting, if applicable.

Remote/Parallel Rendering Options

**Remote Render Threshold**: Set the data size (in megabytes) beyond which to render data remotely (or in parallel) when connected to parallel rendering capable server. In that case the rendered images are delivered to the client. Otherwise, the geometry is delivered to the client and rendering happens locally.

20

Miscellaneous

**Outline Threshold**: When opening a dataset or creating a new filters, use Outline representation, if possible, when showing datasets with number of cells greater than thus threshold (in millions).

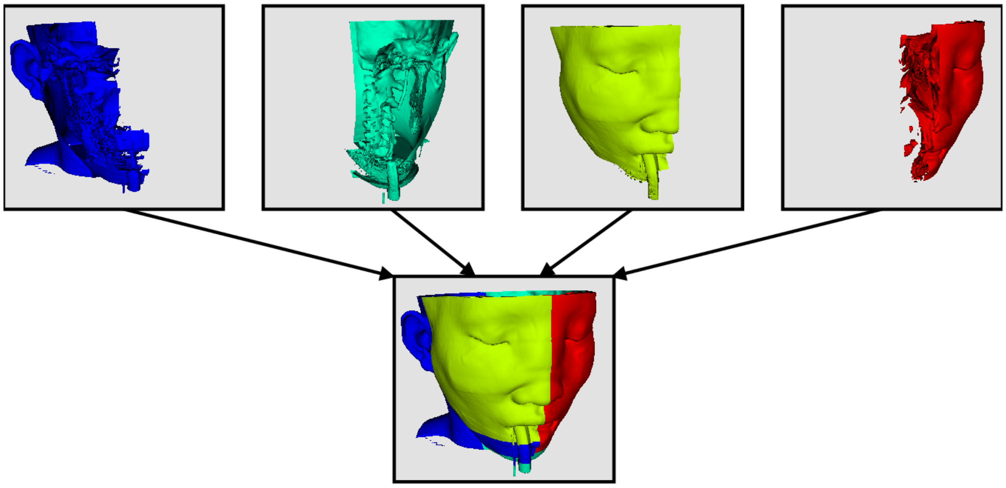
250

  - `Depth Peeling for Volumes` : Include volumes in depth peeling to correctly intermix volumes and translucent polygons.
  - `Maximum Number Of Peels` : Set the maximum number of peels to use with depth peeling. Using more peels allows more depth complexity, but allowing less peels runs faster. You can try adjusting this parameter if translucent geometry renders too slow or translucent images do not look correct.
- **`Miscellaneous`**
  - `Outline Threshold` : When creating very large datasets, default to the outline representation. Surface representations usually require ParaView to extract geometry of the surface, which takes time and memory. For data with sizes above this threshold, use the outline representation, which has very little overhead, by default instead.
  - `Show Annotation` : Show or hide annotation providing rendering performance information. This information is handy when diagnosing performance problems.

Note that this is not a complete list of ParaView rendering settings. We have left out settings that do not significantly affect rendering performance. We have also left out settings that are only valid for parallel client-server rendering, which are discussed in Section 2.7.12.

### Basic Parallel Rendering

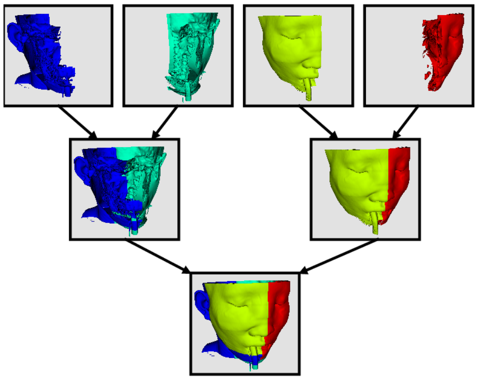
When performing parallel visualization, we are careful to ensure that the data remains partitioned among all of the processes up to and including the rendering processes. ParaView uses a parallel rendering library called **IceT** . IceT uses a **sort-last** algorithm for parallel rendering. This parallel rendering algorithm has each process independently render its partition of the geometry and then **composites** the partial images together to form the final image.

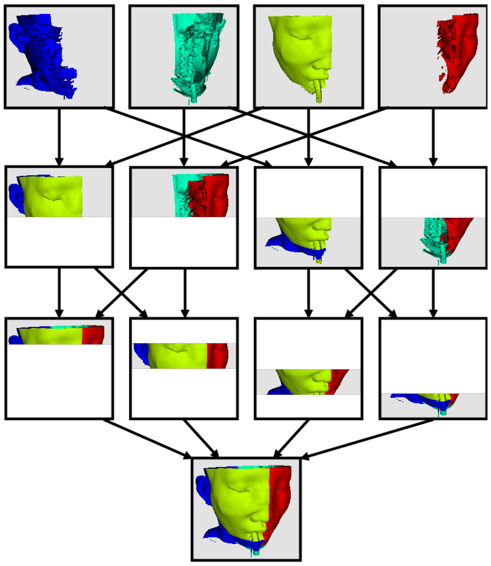


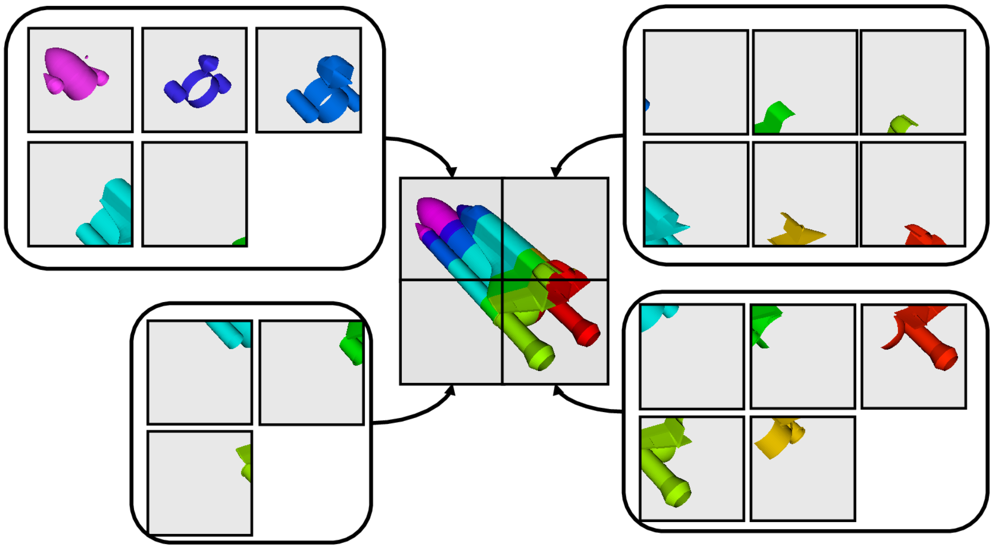
The preceding diagram is an oversimplification. IceT contains multiple parallel image compositing algorithms such as **binary tree** , **binary swap** , and **radix-k** that efficiently divide work among processes using multiple phases.

The wonderful thing about sort-last parallel rendering is that its efficiency is completely insensitive to the amount of data being rendered. This makes it a very scalable algorithm and well suited to large data. However, the parallel rendering overhead does increase linearly with the number of pixels in the image. Consequently, some of the rendering parameters deal with the image size.

IceT also has the ability to drive tiled displays, which are large, high-resolution displays comprising an array of monitors or projectors. Using a sort-last algorithm on a tiled display is a bit counterintuitive because the number of pixels to composite is so large. However, IceT is designed to take advantage of spatial locality in the data on each process to drastically reduce the amount of compositing necessary. This spatial locality can be enforced by applying the *Filters* > *Alphabetical* > *D3* filter to your data.





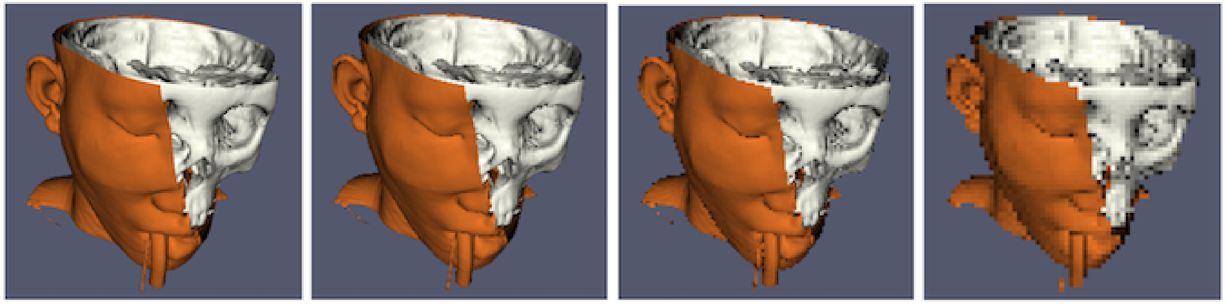


Because there is an overhead associated with parallel rendering, ParaView has the ability to turn off parallel rendering at any time. When parallel rendering is turned off, the geometry is shipped to the location where display occurs. Obviously, this should only happen when the data being rendered is small.

### Image Level of Detail

The overhead incurred by the parallel rendering algorithms is proportional to the size of the images being generated. Also, images generated on a server must be transfered to the client, a cost that is also proportional to the image size. To help increase the frame rate during interaction, ParaView introduces a new LOD parameter that controls the size of the images.

During interaction while parallel rendering, ParaView can optionally **subsample** the image. That is, ParaView will reduce the resolution of the image in each dimension by a factor during interaction. Reduced images will be rendered, composited, and transfered. On the client, the image is inflated to the size of the available space in the GUI.



The resolution of the reduced images is controlled by the factor with which the dimensions are divided. In the proceeding images, the left image has the full resolution. The following images were rendered with the resolution reduced by a factor of 2, 4, and 8, respectively.

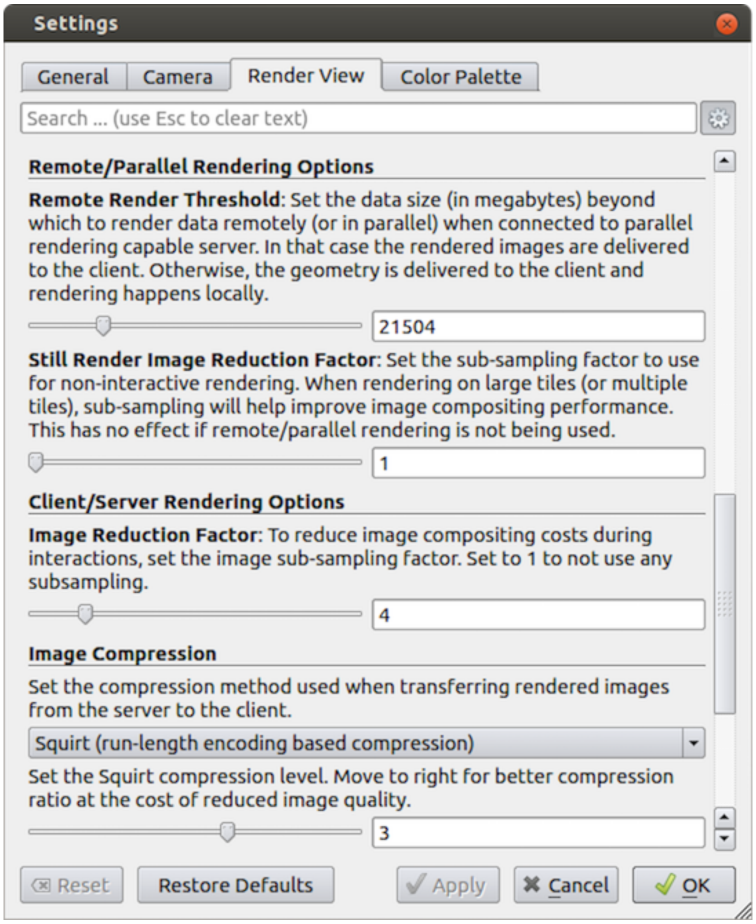
ParaView also has the ability to compress images before transferring them from server to client. Compression, of course, reduces the amount of data transferred and, therefore, makes the most of the available bandwidth. However, the time it takes to compress and decompress the images adds to the latency.

ParaView contains several different image compression algorithms for client-server rendering. The first uses LZ4 compression that is designed for high-speed compression and decompression. The second option is a custom algorithm called **Squirt** , which stands for Sequential Unified Image Run Transfer. Squirt is a run-length encoding compression that reduces color depth to increase run lengths. The third algorithm uses the **Zlib** compression library, which implements a variation of the Lempel-Ziv algorithm. Zlib typically provides better compression than Squirt, but it takes longer to perform and, hence, adds to the latency. `paraview` Windows and Linux executables include a compression option that uses NVIDIA's NVPipe library for hardware-accelerated compression and decompression if a Kepler-class or higher NVIDIA GPU is available.

### Parallel Render Parameters

Like the other 3D rendering parameters, the parallel rendering parameters are located in the `Settings` dialog. The parallel rendering options in the dialog are in the `Render View` tab (intermixed with several other rendering options such as those described in Section 2.7.12). The parallel and client-server options are divided among several categories, and several are considered advanced.

- `Remote/Parallel Rendering Options`
  - `Remote Render Threshold` : Set the data size at which to render remotely in parallel or to render locally. If the geometry is over this threshold (and ParaView is connected to a remote server), the data is rendered in parallel remotely, and images are sent back to the client. If the geometry is under this threshold, the geometry is sent back to the client, and images are rendered locally on the client.



**Remote/Parallel Rendering Options**

**Remote Render Threshold**: Set the data size (in megabytes) beyond which to render data remotely (or in parallel) when connected to parallel rendering capable server. In that case the rendered images are delivered to the client. Otherwise, the geometry is delivered to the client and rendering happens locally.

21504

**Still Render Image Reduction Factor**: Set the sub-sampling factor to use for non-interactive rendering. When rendering on large tiles (or multiple tiles), sub-sampling will help improve image compositing performance. This has no effect if remote/parallel rendering is not being used.

1

**Client/Server Rendering Options**

**Image Reduction Factor**: To reduce image compositing costs during interactions, set the image sub-sampling factor. Set to 1 to not use any subsampling.

4

**Image Compression**

Set the compression method used when transferring rendered images from the server to the client.

Squirt (run-length encoding based compression)

Set the Squirt compression level. Move to right for better compression ratio at the cost of reduced image quality.

3

  - `Still Render Image Reduction Factor` : Set the sub-sampling factor for still (non-interactive) rendering. Some large displays have more resolution than is really necessary, so this sub-sampling reduces the resolution of all images displayed.

- `Client/Server Rendering Options`

  - `Image Reduction Factor` : Set the interactive subsampling factor. The overhead of parallel rendering is proportional to the size of the images generated. Thus, you can speed up interactive rendering by specifying an image subsampling rate. When this box is checked, interactive renders will create smaller images, which are then magnified when displayed. This parameter is only used during interactive renders.

- `Image Compression`

  - Before images are shipped from server to client, they can optionally be compressed using one of three available compression algorithms: LZ4 , Squirt , or Zlib . To make the compression more effective, either algorithm can reduce the color resolution of the image before compression. The sliders determine the amount of color bits saved. Full color resolution is always used during a still render.

  - Suggested image compression presets are provided for several common network types. When attempting to select the best image compression options, try starting with the presets that best match your connection.

### Parameters for Large Data

The default rendering parameters are suitable for most users. However, when dealing with very large data, it can help to tweak the rendering parameters. While the optimal parameters depend on your data and the hardware on which ParaView is running, here are several pieces of advice that you should follow.

- If there is a long pause before the first interactive render of a particular dataset, it might be the creation of the decimated geometry. Try using an outline instead of decimated geometry for interaction. You could also try lowering the factor of the decimation to 0 to create smaller geometry.

- Avoid shipping large geometry back to the client. The remote rendering will use the power of the entire server to render and ship images to the client. If remote rendering is off, geometry is shipped back to the client. When you have large data, it is always faster to ship images than to ship data. (Although, if your network has a high latency, this could become problematic for interactive frame rates.)

- Adjust the interactive image sub-sampling for client-server rendering as needed. If image compositing is slow, if the connection between client and server has low bandwidth, or if you are rendering very large images, then a higher subsample rate can greatly improve your interactive rendering performance.

- Make sure `Image Compression` is on. It has a tremendous effect on desktop delivery performance, and the artifacts it introduces, which are only there during interactive rendering, are minimal. Lower bandwidth connections can try using Zlib instead of Squirt compression. Zlib will create smaller images at the cost of longer compression/decompression times.

- If the network connection has a high latency, adjust the parameters to avoid remote rendering during interaction. In this case, you can try turning up the remote rendering threshold a bit, and this is a place where using the outline for interactive rendering is effective.

- If the still (non-interactive) render is slow, try turning on the delay between interactive and still rendering to avoid unnecessary renders.

## 2.8 Memory Inspector

The **ParaView** Memory Inspector panel provides users with a convenient way to monitor **ParaView**'s memory usage during interactive visualization. It also provides developers with a point-and-click interface for attaching a debugger to local or remote client and server processes. As explained earlier, both the Information panel and the Statistics Inspector are prone to over and under estimate the total memory used for the current pipeline. The Memory Inspector addresses those issues through direct queries to the operating system. A number of diagnostic statistics are gathered and reported, including the total memory used by all processes on a per-host basis, the total cumulative memory use by **ParaView** on a per-host basis, and the individual per-rank use by each **ParaView** process. When memory consumption reaches a critical level, either cumulatively on the host or in an individual rank, the corresponding GUI element will turn red, alerting you that you are in danger of potentially being shut down. This gives you a chance to save state and restart the job with more nodes to avoid losing your work. On the flip side, knowing when you're not close to using the full capacity of available memory can be useful to conserve computational resources by running smaller jobs. Of course, the memory foot print is only one factor in determining the optimal run size.

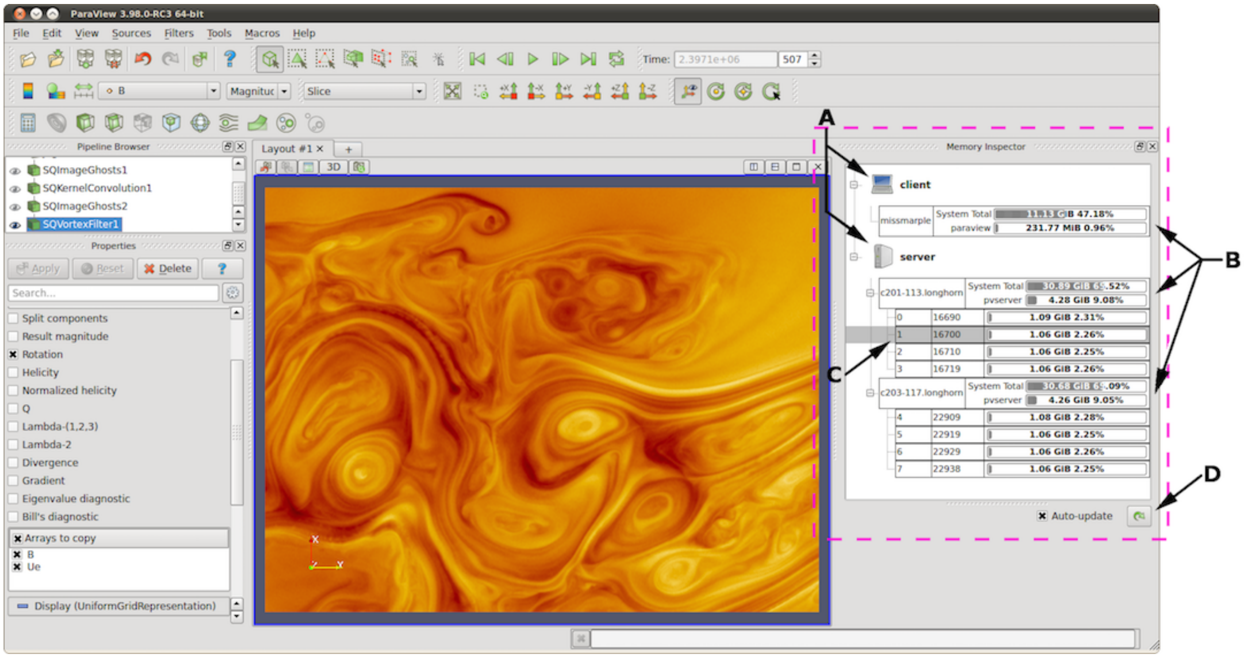


Fig. 2.28: The main UI elements of the Memory Inspector panel. A: Process Groups, B: Per-Host statistics, C: Per-Rank statistics, and D: Update controls.

### 2.8.1 User interface and layout

The Memory Inspector panel displays information about the current memory usage on the client and server hosts. Fig. 2.28 shows the main UI elements labeled A-D. A number of additional features are provided via specialized context menus accessible from the Client and Server group, Host, and Rank's UI elements. The main UI elements are:

A. *Process Groups*

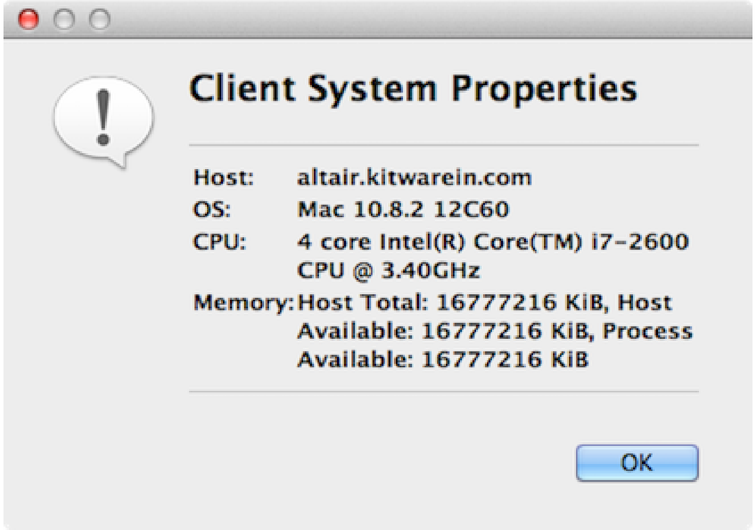
   1. *Client* : There is always a client group that reports statistics about the **ParaView** client.
   2. *Server* : When running in client-server mode, a server group reports statistics about the hosts where pvserver processes are running.
   3. *Data Server* : When running in client-data-render server mode, a data server group reports statistics about the hosts where pvdataserver processes are running.

4. *Render Server* : When running in client-data-render server mode, a render server group reports statistics about the hosts where `pvrenderserver` processes are running.

B. *Per-Host Statistics* : Per-host statics are reported for each host where a **ParaView** process is running. Hosts are organized by host name, which is shown in the first column. Two statics are reported: 1) total memory used by all processes on the host and 2) **ParaView**'s cumulative usage on this host. The absolute value is printed in a bar that shows the percentage of the total available memory used. On systems where job-wide resource limits are enforced, **ParaView** is made aware of the limits via the *PV_HOST_MEMORY_LIMIT* environment variable, in which case, **ParaView**'s cumulative percent used is computed using the smaller of the host total and the resource limit.

C. *Update Controls* : By default, when the panel is visible, memory use statistics are updated automatically as pipeline objects are created, modified, or destroyed and after the scene is rendered. Updates may be triggered manually by using the refresh button. Automatic updates may be disabled by un-checking the *Auto-update* check box. Queries to remote systems have proven to be very fast even for fairly large jobs. Hence, the auto-update feature is enabled by default.

D. *Host Properties Dialog* : The *Host* context menu provides the `Host Properties` dialog, which reports various system details such as the OS version and the CPU version, as well as the memory installed and available to the host context and process context. While the `Memory Inspector` panel reports memory use as a percent of the available in the given context, the `Host Properties` dialog reports the total memory installed and available in each context. Comparing the installed and available memory can be used to determine if you are impacted by resource limits.



Client System Properties

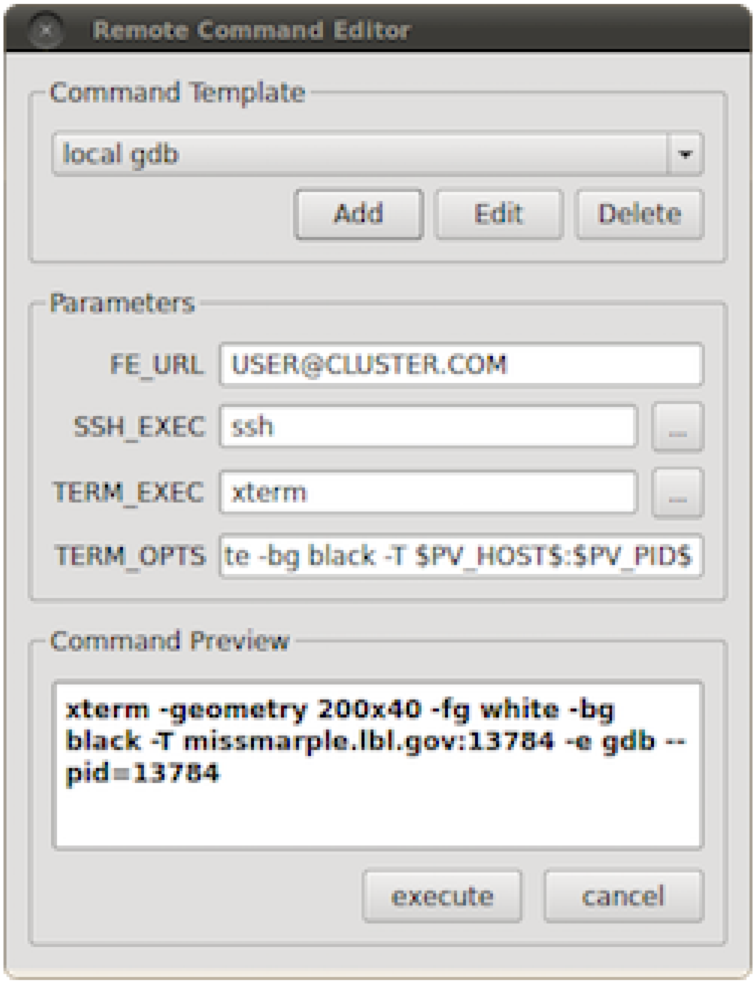
Host: altair.kitwarein.com
OS: Mac 10.8.2 12C60
CPU: 4 core Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
Memory: Host Total: 16777216 KiB, Host Available: 16777216 KiB, Process Available: 16777216 KiB

Fig. 2.29: Host properties dialog.

### 2.8.2 Advanced debugging features

#### Remote commands

The `Memory Inspector Panel` provides a remote (or local) command feature, allowing you to execute a shell command on a given host. This feature is exposed via a specialized Rank item context menu. Because we have information such as a rank's process id, individual processes may be targeted. For example, this allows you to quickly attach a debugger to a server process running on a remote cluster. If the target rank is not on the same host as the client, then the command is considered remote. Otherwise, it is considered local. Therefore, remote commands are executed via `ssh` , while local commands are not. A list of command templates is maintained. In addition to a number of pre-defined command templates, you may add templates or edit existing ones. The default templates allow you to:



Command Template: local gdb

FE_URL: USER@CLUSTER.COM

SSH_EXEC: ssh

TERM_EXEC: xterm

TERM_OPTS: te -bg black -T $PV_HOST$:$PV_PID$

```
xterm -geometry 200x40 -fg white -bg
black -T missmarple.lbl.gov:13784 -e gdb --
pid=13784
```

Fig. 2.30: The remote command dialog.

- Attach gdb to the selected process
- Run top on the host of the selected process
- Send a signal to the selected process

Prior to execution, the selected template is parsed, and a list of special tokens are replaced with runtime-determined or user-provide values. User-provided values can be set and modified in the dialog's parameter group. The command, with tokens replaced, is shown for verification in the dialog's preview pane.

The following tokens are available and may be used in command templates as needed:

1. *$TERM_EXEC$* : The terminal program that will be used to execute commands. On Unix systems, xterm is typically used. On Windows systems, `cmd.exe` is typically used. If the program is not in the default path, then the full path must be specified.
2. *$TERM_OPTS$* : Command line arguments for the terminal program. On Unix, these may be used to set the terminals window title, size, colors, and so on.
3. *$SSH_EXEC$* : The program to use to execute remote commands. On Unix, this is typically `ssh`. On Windows, one option is `plink.exe`. If the program is not in the default path, then the full path must be specified.
4. *$FE_URL$* : Ssh URL to use when the remote processes are on compute nodes that are not visible to the outside world. This token is used to construct command templates where two `ssh` hops are made to execute the command.
5. *$PV_HOST$* : The hostname where the selected process is running.
6. *$PV_PID$* : The process-id of the selected process.

Note: On Windows, the debugging tools found in Microsoft's SDK need to be installed in addition to Visual Studio (e.g., `windbg.exe`). The `ssh` program `plink.exe` for Windows doesn't parse ANSI escape codes that are used by Unix shell programs. In general, the Windows- specific templates need some polishing.

#### Stack trace signal handler

The Process Group's context menu provides a back trace signal handler option. When enabled, a signal handler is installed that will catch signals such as SEGV, TERM, INT, and ABORT and that will print a stack trace before the process exits. Once the signal handler is enabled, you may trigger a stack trace by explicitly sending a signal. The stack trace signal handler can be used to collect information about crashes or to trigger a stack trace during deadlocks when it's not possible to `ssh` into compute nodes. Sites that restrict users' `ssh` access to compute nodes often provide a way to signal running processes from the login node. Note that this feature is only available on systems that provide support for POSIX signals, and we currently only have implemented stack trace for GNU-compatible compilers.

### 2.8.3 Compilation and installation considerations

If the system on which **ParaView** will run has special resource limits enforced, such as job-wide memory use limits, or non-standard per-process memory limits, then the system administrators need to provide this information to the running instances of **ParaView** via the following environment variables. For example, those could be set in the batch system launch scripts.

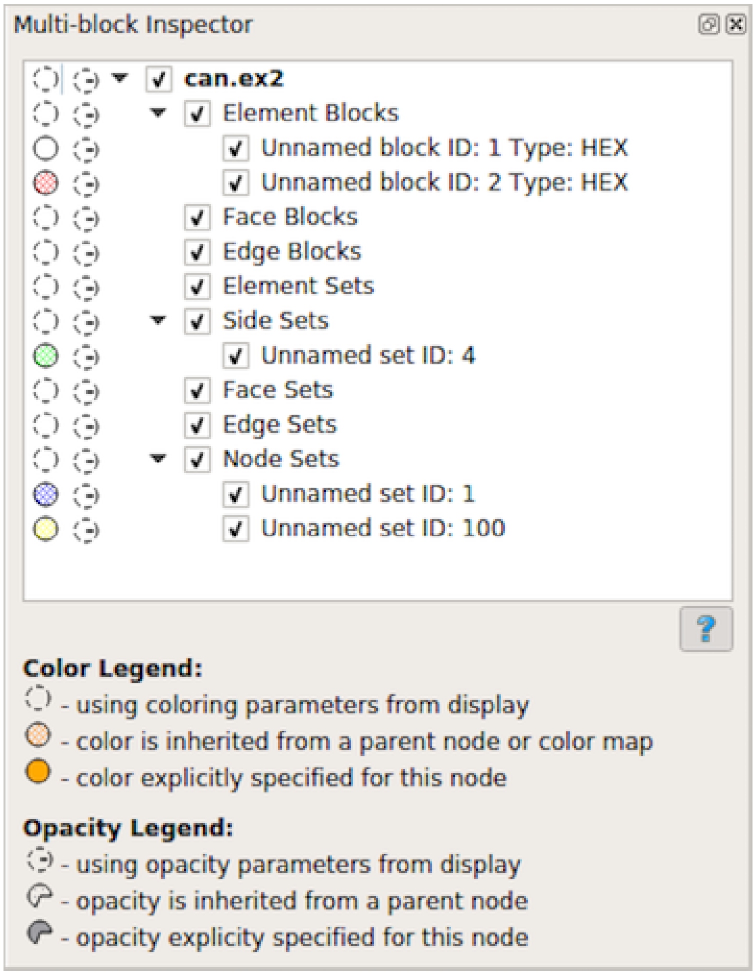
1. *PV_HOST_MEMORY_LIMIT* : For reporting host-wide resource limits.
2. *PV_PROC_MEMORY_LIMIT* : For reporting per-process memory limits. that are not enforced via standard Unix resource limits.

A few of the debugging features (such as printing a stack trace) require debug symbols. These features will work best when **ParaView** is built with `CMAKE_BUILD_TYPE=Debug` or, for release builds, `CMAKE_BUILD_TYPE=RelWithDebugSymbols`.

## 2.9 Multiblock Inspector

Composite datasets (Section 1.3) such as multiblock datasets and AMR are often encountered when visualizing results from several scientific simulation codes e.g. OpenFOAM, Exodus, etc. Readers for several of these simulation file formats support selecting which blocks to read. Additionally, you may want to control display properties such as visibility, opacity, and color for individual blocks or subtress. You can use the `Multiblock Inspector` panel for this.

Fig. 2.31 shows the `Multiblock Inspector` showing the hierarchy from an Exodus dataset. The panel tracks the active source and reflects the structure for the data produced by the active source. The display properties reflect their state in the active view.



Multi-block Inspector

- ☑ can.ex2
  - ☑ Element Blocks
    - ☑ Unnamed block ID: 1 Type: HEX
    - ☑ Unnamed block ID: 2 Type: HEX
  - ☑ Face Blocks
  - ☑ Edge Blocks
  - ☑ Element Sets
  - ☑ Side Sets
    - ☑ Unnamed set ID: 4
  - ☑ Face Sets
  - ☑ Edge Sets
  - ☑ Node Sets
    - ☑ Unnamed set ID: 1
    - ☑ Unnamed set ID: 100

**Color Legend:**

◌ - using coloring parameters from display
◍ - color is inherited from a parent node or color map
● - color explicitly specified for this node

**Opacity Legend:**

◌ - using opacity parameters from display
◔ - opacity is inherited from a parent node
◕ - opacity explicity specified for this node

Fig. 2.31: The `Multiblock Inspector`

To show or hide a block you have to click on the checkbox next to the block name. Toggling the visibility of a non-leaf block, affects the visibility of the entire subtree.

The first column reflects the color used for the block. The color for a block can be specified in multiple ways. First, you can choose to not use any block specific overrides and simply let the default display properties ( Section 1.4.3) affect the rendering. This is the default behavior. When that is the case for any block, it is indicated in the color column by an empty dotted circle ◌. Second, you can explicitly override the color to use for a block or a subtree. To do this, simply double click on the color column next to the block of interest. This will pop up the color chooser dialog. Explicitly overridden color for a block is indicated by a solid filled circle icon ● filled with the selected color. When an explicit color is set on a non-leaf node, all its children (and their children) inherit that color unless explicitly overridden. For such nodes that have a color inherited from their parent, we use the dotted-circle icon filled with a pattern icon ◍.

The second column reflects the opacity override for the blocks. Similar to color, the opacity could be simply using value from the display properties ◌, or explicitly set ◖, or inhertied from parent node ◌.

**Did you know?**

In most cases with multiblock datasets, ParaView uses the `vtkBlockColors` array for coloring. This is an array filled with random values so that each block can be colored using a different color. That makes it easier to visually see each of the blocks in the view. When in this mode, the `Multiblock Inspector` 's color column shows the color used for each of the blocks using the same icon as the one used for inherited colors i.e. ◍.

You can change colors and opacities for a specific block by double clicking on the corresponding icon. This allows setting values one at a time. For specifying color and opacity overrides for multiple elements, you can select the items and then right click to get the context menu. The context menu allows you to change these properties for all the selected items, as shown in Fig. 2.32.

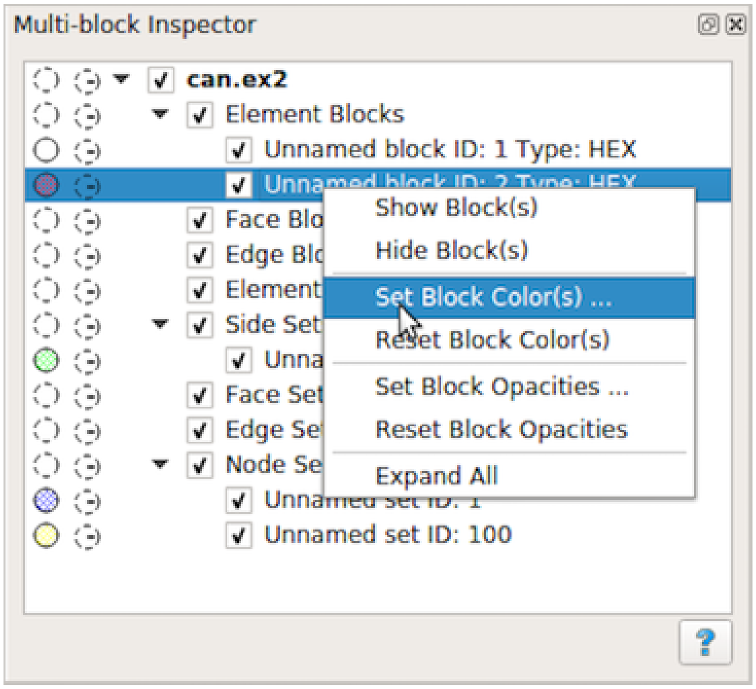


Fig. 2.32: The `Multiblock Inspector` context menu.

Besides using the `Multiblock Inspector` to set color and opacity overrides for blocks, you can also directly changes these parameters from the `Render View` itself. Simply right-click in the render view on the block of interest and you'll get a context menu, Fig. 2.33, that allows changing the block properties and are more.

Another useful feature with the `Multiblock Inspector` is selection. If you simply click on any row in the inspector, you will select the block (or subtree) and the active view will highlight the selected block(s). Conversely, if you make a block-based selection in the active `Render View` using , you will see the corresponding blocks highlighted in the `Multiblock Inspector` panel.

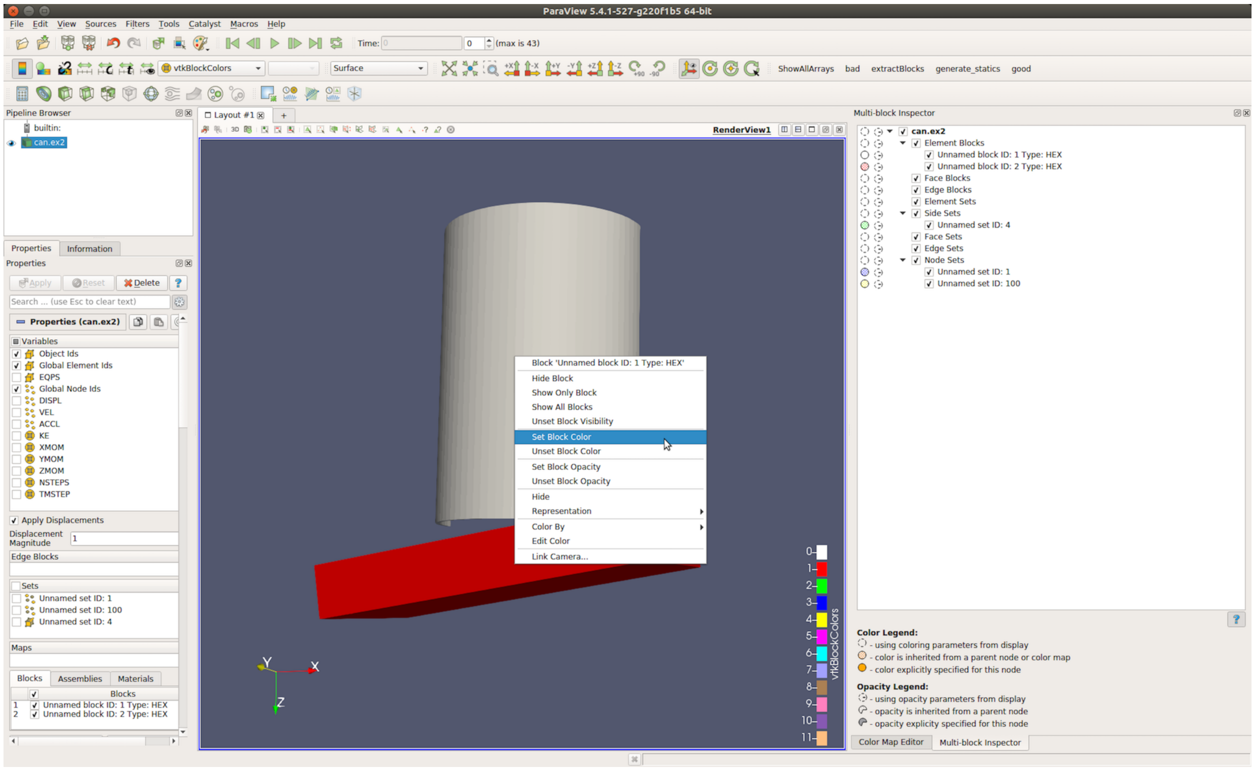


Fig. 2.33: Context menu in `Render View` can be used to change block display properties.

## 2.10 Annotations

Explicit labeling and annotation of particular data values is often an important element in data visualization and analysis. ParaView provides a variety of mechanisms to enable annotation in renderings ranging from free floating text rendered alongside other visual elements in the render view to data values associated with particular points or cells.

### 2.10.1 Annotation sources

Several types of text annotations can be added through the *Sources* > *Alphabetical* menu. Text from these sources is drawn on top of 3D elements in the render view. All annotation sources share some common properties under the `Display` section of the `Properties` panel. These include `Font Properties` such as the font to use, the size of the text, its color, opacity, and justification, as well as text effects to apply such as making it bold, italic, or shadowed.

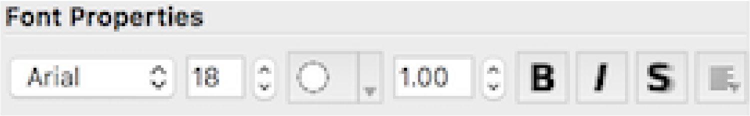


Fig. 2.34: Font property controls in annotation sources and filters.

There are three fonts available in **ParaView**: Arial, Courier, and Times. You can also supply an arbitrary TrueType font file (*.ttf) to use by selecting the `File` entry in the popup menu under `Font Properties` and clicking on the `...` button to the right of the font file text field. A file selection dialog will appear letting you choose a font file from the file system on which `paraview` (or `pvpython`) is running.

The remaining display properties control where the text is placed in the render view. There are two modes for placement, one that uses predefined positions relative to the render view, and one that enables arbitrary interactive placement in the render view. The first mode is active when the Use Window Location checkbox is selected. It enables the annotation to be placed in one of the four corners of the render view or centered horizontally at the top or bottom of the render view. Buttons with icons representing the location are shown in the Pipeline browser. These buttons correspond to locations in the render view as depicted in Fig. 2.35.

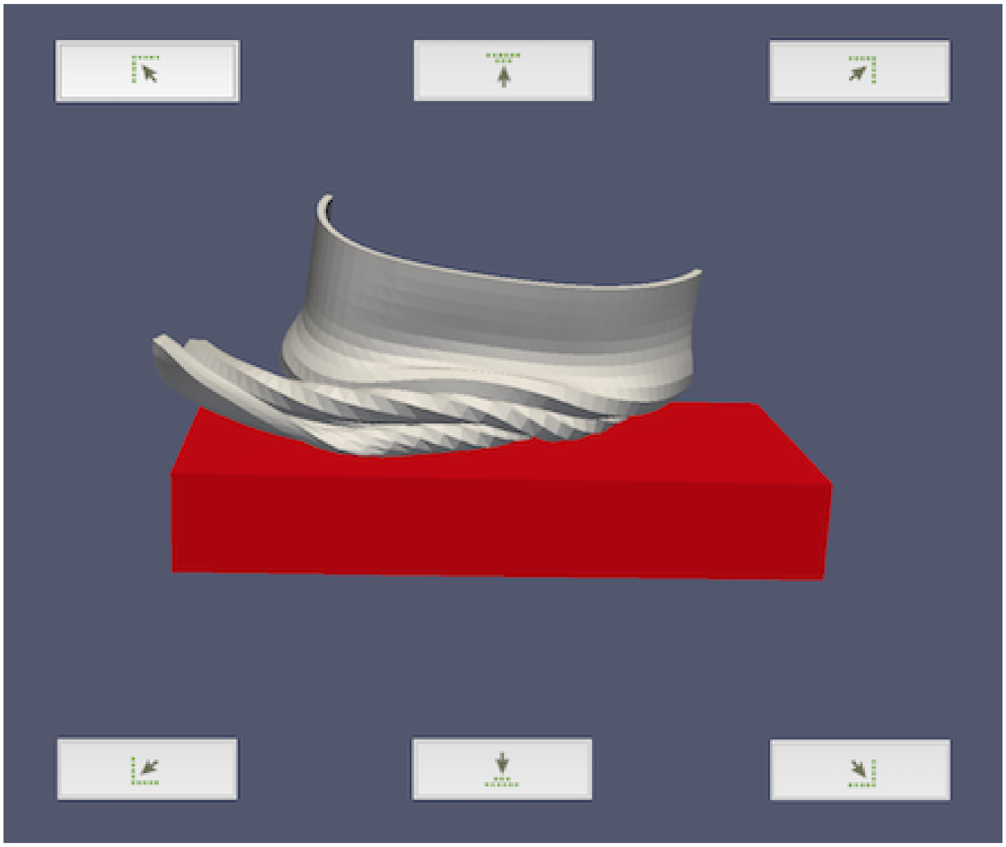


Fig. 2.35: Annotation placement buttons and where they place the annotation.

The second mode, activated by clicking the Lower Left Corner checkbox, lets you arbitrarily place the annotation. If the Interactivity property is enabled, you can click and drag the annotation in the render view to place it, or you can manually enter a location where the lower left corner of the annotation's bounding box should be placed. The coordinates are defined in terms of fractional coordinates that range from [0, 1] in the x and y dimensions. The coordinate system of the render view has a lower left origin, so a Lower Left Corner value of [0, 0] will place the annotation in the lower left corner of the render view.

### Text source

The Text source enables you to add a text annotation in the render view. It has one property defining what text is displayed. Text can be multiline, and it can contain numbers and unicode characters. Text may also contain Mathtex expressions between starting and ending dollar signs. Mathtex expressions are a subset of TeX math expressions [dt] . When Mathtext is used, the text can only be on a single line.

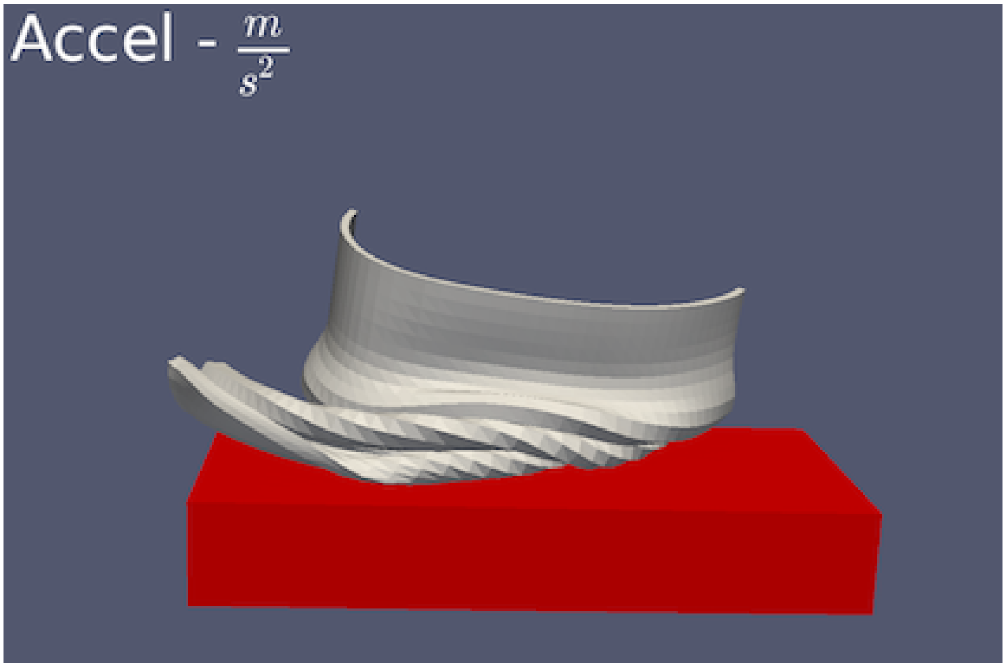


Fig. 2.36: An example of the Text source annotation in the upper left corner with a math expression rendered from a Mathtext [dt] expression.

### Annotate Time source

The Annotate Time source is nearly identical to the Text source, but it also offers access to the current time value set in ParaView. Control over the format of the time display is available through the Format property. This property takes a string with optional formatting sections understood by the fmt library. By default, the value is “Time: {time:f}” where the “time” term inside the curly braces is replaced with ParaView’s current time value, and the “:f” specifies that it should be formatted as a float with six decimal digits. For other formatting possibilities, please see the fmt syntax description at https://fmt.dev/latest/syntax.html. Examples are near the bottom of that page.

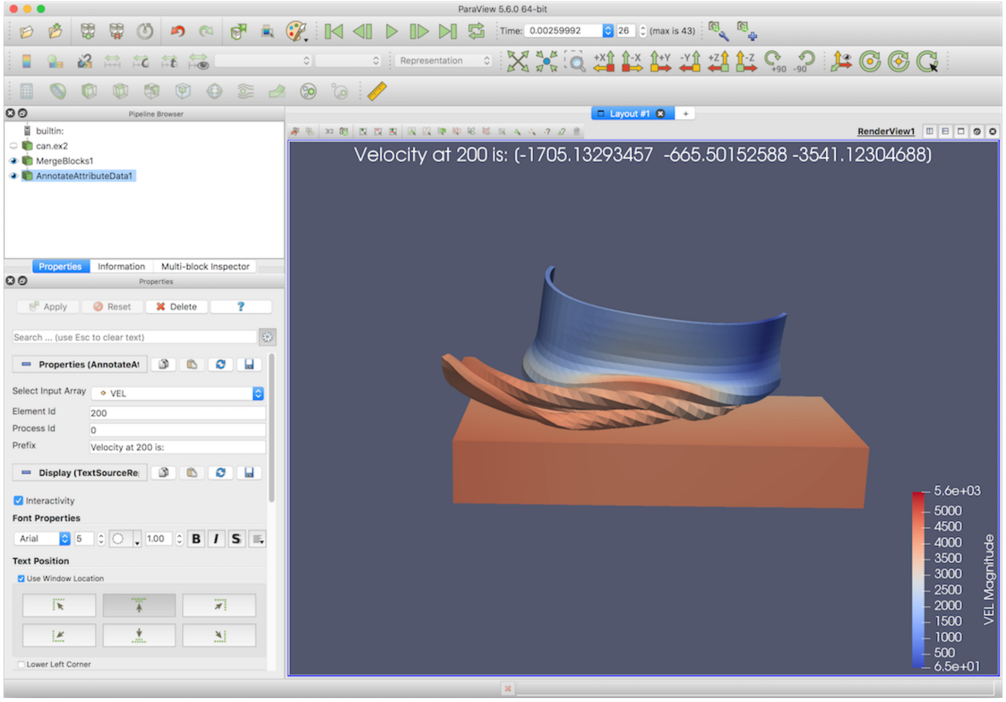
## 2.10.2 Annotation filters

The annotation sources described in the previous section are available for adding text annotations that do not depend on any loaded datasets. To create annotations that show values from an available data source in the Pipeline Browser , several annotation filters are available. The properties available to change the text font and annotation location are exactly the same as those available for the annotation sources described in the previous section.

### Annotate Attribute Data filter

The Annotate Attribute Data makes it possible to create an annotation with a data value from an array (or attribute) in a dataset. To use the filter, first select the data array with the data of interest in the Select Input Array . These arrays may be point, cell, or field data arrays. The Element Id property specifies the index of the point or cell whose value should be shown in the annotation. If the selected input array is a field array (not associated with points or cells), the Element Id specifies the tuple of the array to show. When running in parallel, the Process Id denotes the process that holds the array from which the value should be obtained.

The Prefix text property precedes the attribute value in the rendered annotation. There is no formatting string - the number is appended after the prefix. If the array value selected is a scalar value, the annotation will contain just the



Velocity at 200 is: (-1705.13293457 -665.50152588 -3541.12304688)

Select Input Array: VEL
Element Id: 200
Process Id: 0
Prefix: Velocity at 200 is:

☑ Interactivity

Font Properties: Arial 5 1.00

Text Position

☑ Use Window Location

☐ Lower Left Corner

Fig. 2.37: Properties of the `Annotate Attribute Data` filter.

number. On the other hand, if the array value is from a multicomponent array, the individual components will be added to the annotation label in a space-separated list that is surrounded by parentheses.

### `Annotate Global Data` filter

Some file formats include the concept of `global data`, a single data value stored in the data array for each time step. **ParaView** stores the set of such data values as a field data array associated with the dataset with the same number of values as timesteps. To display these global values in the render view, use the `Annotate Global Data` filter. The `Select Arrays` popup menu shows the available field data arrays. The `Prefix` and `Suffix` properties come before and after the data value in the annotation, respectively. The `Format` property is a C language number format specifier as you would use in a *printf* function call. The filter will provide a warning if the format is invalid for the `global data` type.

### `Annotate Time Filter`

A nice feature of **ParaView** is that it supports data sources that produce different data at different times. Examples include file readers that read in data for a requested time step and certain temporal filters. Each data source advertises to **ParaView** the time values for which it can produce data. The data produced and displayed in ParaView depends on the time you set in the **ParaView** `VCR Controls` or `Time Inspector` panel.

What is even nicer is that you can have several data sources that each advertise and respond to a possibly unique set of times. That is, available sources do not need to advertise that they support the same set of time points - in fact, they may define data at entirely different time points. Given a requested time, each data source will produce the data corresponding to the time it supports closest to the requested time. This features makes it possible to create animations from multiple datasets varying at different time resolutions, for instance.

While the `Annotate Time` source described earlier can be used to display **ParaView**'s currently requested time, it does not show the time value to which a particular data source is responding. For example **ParaView** may be requesting data for time 5.0, but if a source produces data for time values 10.0 and above, it will produce the data for time 10.0, even though time 5.0 was requested. To show the time for which a data source is producing data, you can instead use the Annotate Time Filter . Simply attach it to the source of interest. If several data sources are present, a separate instance of this filter may be attached to each one.

Control over the format of the time display is available through the Format property. The format string is a string supported by the fmt library and defaults to "Time: {time:f}" where the "time" string inside the curly braces is replaced by the currently loaded time value of the data source to which this filter is attached. This filter also includes Shift and Scale properties used to linearly transform the displayed time. The time value is first multiplied by the scale and then the shift is then added to it.

### Environment Annotation filter

If you want to display information about the environment in which a visualization was generated, use the Environment Annotation filter. By attaching this filter to a data source, you can have it automatically display your user name on the system running **ParaView**, show which operating system was used to generate it, present the date and time when the visualization was generated, and show the file name of the source data if applicable. Each of these items can be enabled or disabled by checkboxes in the Properties panel for this filter.

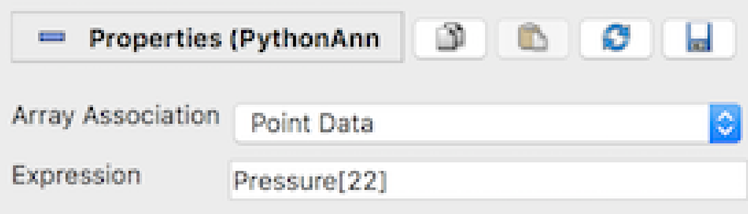
If the input source for this filter is a file reader, the File Name property is initialized to the name of the file. A checkbox labeled Display Full Path is available to show the full path of the file, but if unchecked, only the file name will be displayed. This default file path can be overridden by changing the text in the File Name property. If this filter is attached to a filter instead of a reader, the file path will be initialized to an empty string. It can be changed to the original file name manually, or an arbitrary string if so desired.

### Python Annotation filter

The most versatile annotation filter, the Python Annotation filter, offers the most general way of generating annotations that include information about the dataset. Values from point, cell, field, and row data arrays may be accessed and combined with mathematical operations in a short Python expression defined in the Expression property. The type of data arrays available for use in the Expression is set with the Array Association property.

Before going further, let's look at an example of how to use the Python Annotation filter. Assume you want to show a data value at from a point array named Pressure at point index 22. First, set the Array Association to Point Data to ensure point data arrays can be referenced in the Python annotation expression. To show the pressure value at point 22, set the Expression property to

```
Pressure[22]
```



Array Association: Point Data
Expression: Pressure[22]

Fig. 2.38: An example of a basic Python Annotation filter showing the value of the Pressure array at point 22.

You can augment the Python expression to give the annotation more meaning. To add a prefix, set the Expression to

```
'Pressure: %f' % (Pressure[22])
```

noindent All data arrays in the chosen association are provided as variables that can be referenced in the expression as long as their names are valid Python variables. Array names that are invalid Python variable names are available through a modified version of the array name. This sanitized version of the array name consists of the subset of characters in the array name that are letters, numbers, or underscore (_) joined together without spaces in the order in which they appear in the original array name. For example, an array named `Velocity X` will be made available in the variable `VelocityX`.

Point and cell data in composite datasets such as multiblock datasets is accessed somewhat differently than point or cell data in non-composite datasets. The expression

```
Pressure[22]
```

retrieves a single scalar value from a point array in a non-composite dataset, the same expression retrieves the 22nd element of the `Pressure` array in each block. These values are held in a VTKCompositeDataArray, which is a data structure that holds arrays associated with each block in the dataset. Hence, when the expression

```
Pressure[22]
```

is evaluated on a composite dataset, the value returned and displayed is actually an assemblage of array values from each block. To access the value from a single block, the array from that block must be selected from the `Arrays` member of the result VTKCompositeDataArray. To show the `Pressure` value associated with 22nd point of block 2, for example, set the expression to

```
Pressure[22].Arrays[2]
```

This expression yields a single data value in the rendered annotation, assuming that the `Pressure` array has a single component. To show a range of array values, use a Python range expression in the index into the `Pressure` field, e.g.,

```
Pressure[22:24].Arrays[2]
```

This will show the `Pressure` values for points 22 and 23 from block 2. You can also retrieve more than one array using an index range on the `Arrays` member, e.g.,

```
Pressure[22:24].Arrays[2:5]
```

This expression evaluates to `Pressure` for points 22 and 23 for blocks 2, 3, and 4.

The `Array Association` is really a convenience to make the set of data arrays of the given association available as variables that can be used in the `Expression`. The downside of using these array names is that arrays from only one array association are available at a time. That means annotations that require the combination of a cell data array and point data array, for example, cannot be expressed with these convenience Python variables alone.

Fortunately, you can access any array in the input to this filter with a slightly more verbose expression. For example, the following expression multiplies a cell data value by a point data value:

```
inputs[0].CellData['Volume'][0] * inputs[0].PointData['Pressure'][0]
```

Note that the arrays in the input are accessed in the above example using their original array names.

In the example above, the expression `inputs[0]` refers to the first input to the filter. While this filter can take only one input, it is based on the same code used by the `Python Calculator` (described in Section 1.5.9), which puts its several inputs into a Python list, hence the input to the `Python Annotation` filter is referenced as `inputs[0]`.

In addition to making variables for the current array association available in the expression, this filter provides some other variables that can be useful when computing an annotation value.

- `points` : Point locations (available for datasets with explicit points).
- `time_value` , `t_value` : The current time value set in **ParaView**.

- `time_steps` , `t_steps` : The number of timesteps available in the input.
- `time_range` , `t_range` : The range of timesteps in the input.
- `time_index` , `t_index` : The index of the current timestep in **ParaView**.

There are some situations where the variables above are not defined. If the input has no explicitly defined points, e.g., image data, the `points` variable is not defined. If the input does not define timesteps, the `time_*` and `t_*` variables are not defined.

**Finally, all the capabilities of the `Python Calculator`, documented in Section 1.5.9,**
are available, including the NumPy integration and access to the NumPy and SciPy methods.

**Common Errors**

The time-related variables are not needed to index into point or cell data arrays. Only the point and cell arrays loaded for the current timestep are available in the filter. You cannot access point or cell data from arbitrary timesteps from within this filter.

With the capabilities in this filter, it is possible to reproduce the other annotation sources and filters, as shown below.

- `Text` source: To produce the text "My annotation", write `"My annotation"`
- `Annotate Time` source: To produce the equivalent of `Time: {time:f}`, write `"Time: %f" % time_value`
- `Annotate Attribute Data` filter: To produce the equivalent of setting `Select Input Array` to `EQPS`, `Element Id` to `0` and `Process Id` to `0`, and `Prefix` to `Value is:`, write `'Value is: %.12f' % (inputs[0].CellData['EQPS'][0])` .
- `Annotate Global Data` filter: To produce the same annotation as setting `Select Arrays` to `KE`, `Prefix` to `Value is: `, `Format` to `%7.5g`, and empty suffix, write `"Value is: %7.5g" % (inputs[0].FieldData['KE'].Arrays[0][time_index])`
- `Annotate Time Filter` : To produce the equivalent of setting `Format` to `Time: %f`, `Shift` to `3`, and `Scale` to `2`, write `"Time: %f" % (2*time_value + 3)` .

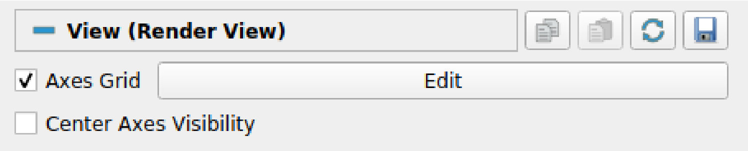
The examples above are meant to illustrate the versatility of the `Python Annotation` filter. Using the specialized annotation sources and filters are likely to be more convenient than entering the expressions in the examples.

## 2.11 Axes Grid

Oftentimes, you want to render a reference grid in the backgroud for a visualization – think axes in a chart view, except this time we are talking of the 3D `Render View` . Such a grid is useful to get an understanding for the data bounds and placement in 3D space. In such cases, you use the `Axes Grid` . `Axes Grid` renders a 3D grid with labels around the rendered scene. In this chapter, we will take a closer look at using and customizing the `Axes Grid` .

### 2.11.1 The basics

To turn on the `Axes Grid` for a `Render View` , you use the `Properties` panel. Under the `View` section, you check the `Axes Grid` checkbox to turn the `Axes Grid` on for the active view.



View (Render View)

☑ Axes Grid Edit

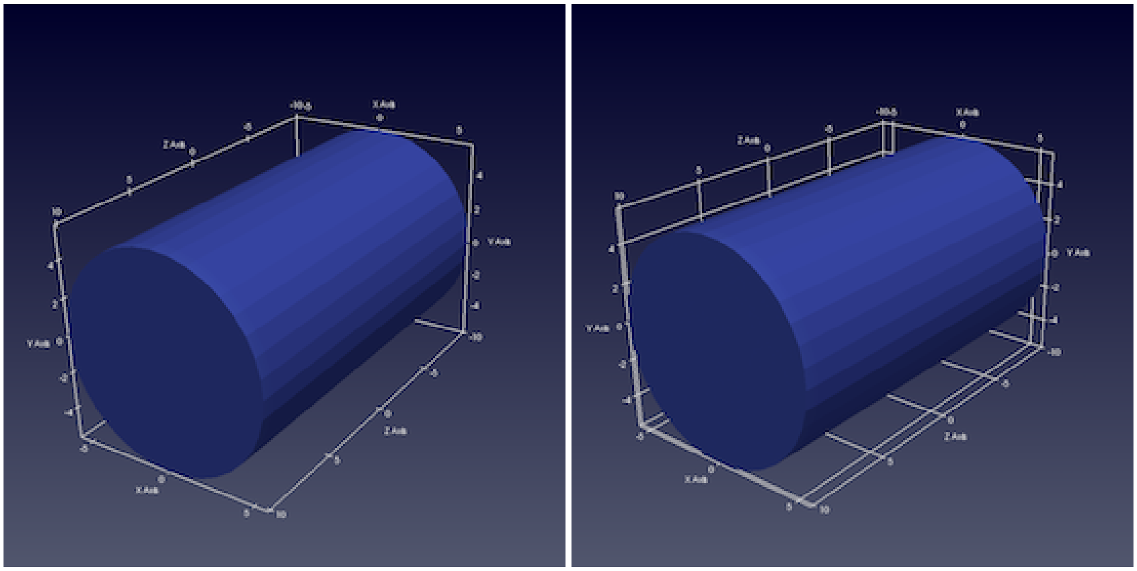
□ Center Axes Visibility

Clicking on the `Edit` button will pop up the `Axes Grid` properties dialog ( Fig. 2.39) that allows you to customize the `Axes Grid` . As with the `Properties` panel, this is a searchable dialog, hence you can use the `Search` box at the top of the dialog to search of properties of interest. At the same time, the  button can be used to toggle between default and advanced modes for the panel.

Using this dialog, you can change common properties like the titles ( `X Title` , `Y Title` , and `Z Title` ), title and label fonts using `Title Font Properties` and `Label Font Properties` for each of the axes directions, as well as the `Grid Color` . Besides labelling the axes, you can render a grid by checking `Show Grid` . Once you have the `Axes Grid` setup to your liking, you can use the  to save your selections so that they are automatically loaded next time you launch ParaView. You can always use the  to revert back to ParaView defaults.

### 2.11.2 Use cases

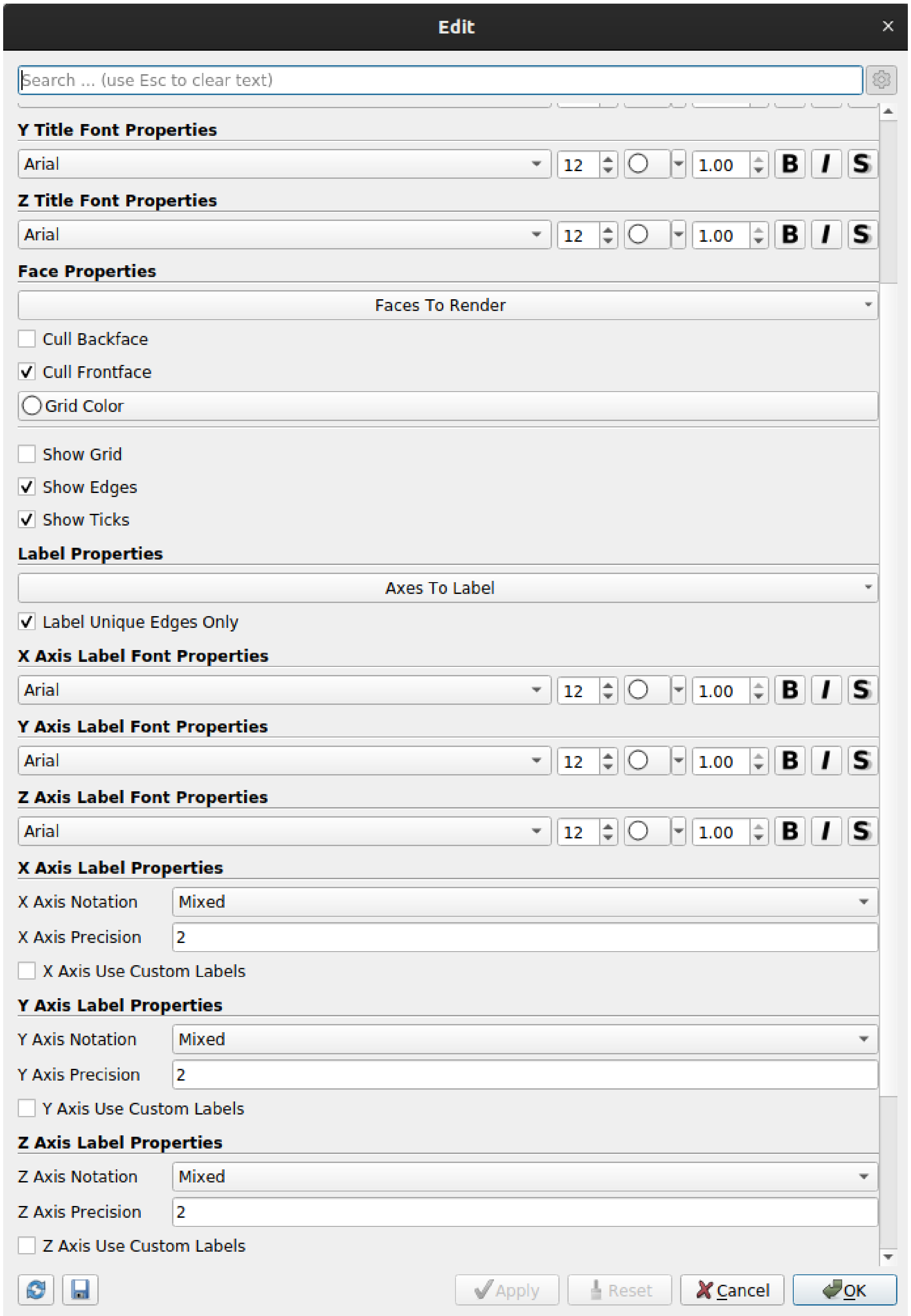
To get a better look at the available customizations, let's look at various visualizations possible and then see how you can set those up using the properties on the `Edit Axes Grid` dialog. In these examples, we use the `disk_out_ref.ex2` example dataset packaged with ParaView.



In the images above, on the left is the default `Axes Grid` . Simply turning on the visibility of the `Axes Grid` will generate such a visualization. The axes places always stay behind the rendered geometry even as you interact with the scene. As you zoom in and out, the labels and ticks will be updated based on visual cues.

To show a grid along the axes planes, aligned with the ticks and labels, turn on the `Show Grid` checkbox, resulting in a visualization on the right.

By default, the gridded faces are always the farthest faces i.e. they stay behind the rendered geometry and keep on updating as you rotate the scene. To fix which faces of the bounding-box are to be rendered, use the `Faces To Render`



Edit

Search ... (use Esc to clear text)

**Y Title Font Properties**

Arial | 12 | ○ | 1.00 | B I S

**Z Title Font Properties**

Arial | 12 | ○ | 1.00 | B I S

**Face Properties**

Faces To Render

☐ Cull Backface

☑ Cull Frontface

○ Grid Color

☐ Show Grid

☑ Show Edges

☑ Show Ticks

**Label Properties**

Axes To Label

☑ Label Unique Edges Only

**X Axis Label Font Properties**

Arial | 12 | ○ | 1.00 | B I S

**Y Axis Label Font Properties**

Arial | 12 | ○ | 1.00 | B I S

**Z Axis Label Font Properties**

Arial | 12 | ○ | 1.00 | B I S

**X Axis Label Properties**

X Axis Notation: Mixed

X Axis Precision: 2

☐ X Axis Use Custom Labels

**Y Axis Label Properties**

Y Axis Notation: Mixed

Y Axis Precision: 2

☐ Y Axis Use Custom Labels

**Z Axis Label Properties**

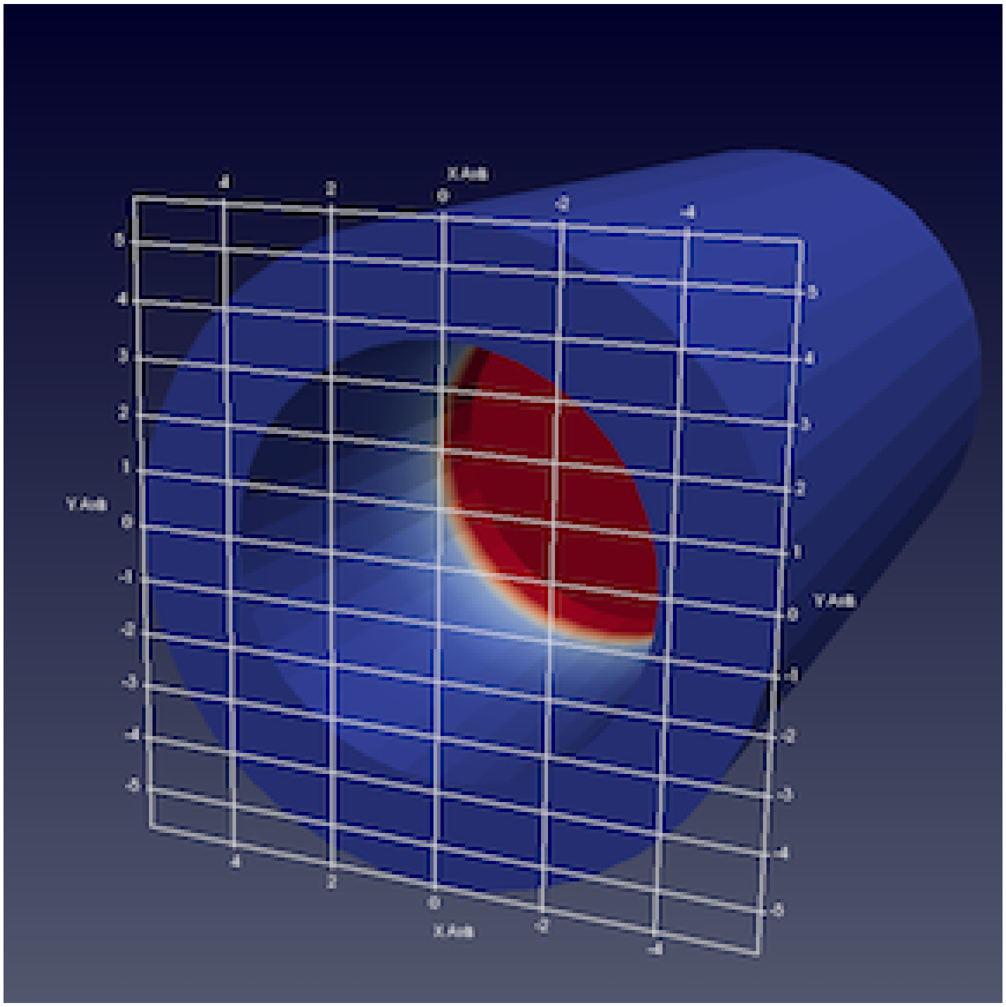
Z Axis Notation: Mixed

Z Axis Precision: 2

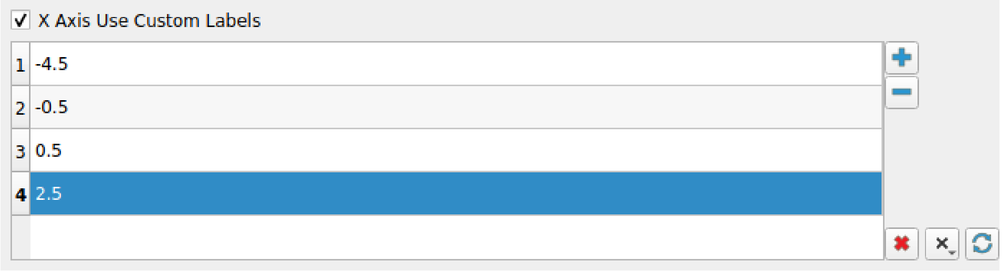
☐ Z Axis Use Custom Labels

Fig. 2.39: `Edit Axes Grid` dialog is used to customize the `Axes Grid` .

button (it's an advanced property, so you may have to search for it using the `Seach` box in the `Edit Axes Grid` dialog). Suppose, we want to label just one face, the lower XY face. In that case, uncheck all the other faces except `Min-XY` in menu popped up on clicking on the `Faces to Render` button. This will indeed just show the min-XY face, however as you rotate the scene, the face will get hidden as soon as the face gets closer to the camera than the dataset. This is because, by default, `Cull Frontfaces` is enabled. Uncheck `Cull Frontfaces` and ParaView will stop removing the face as it comes ahead of the geometry, enabling a visualization as follows.



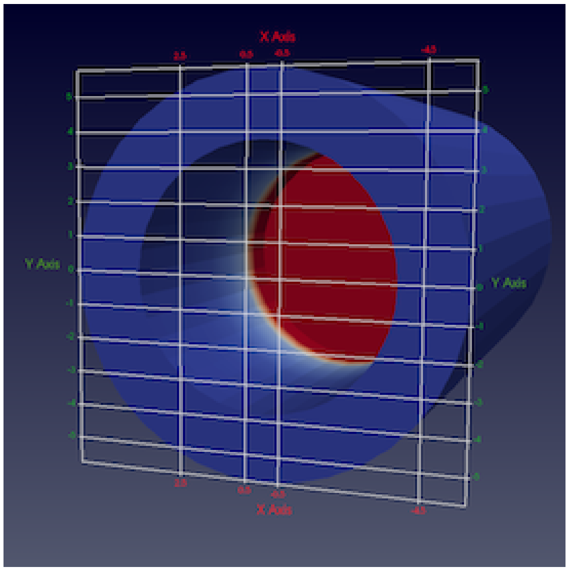
Besides controlling which faces to render, you can also control where the labels are placed. Let's say we want ParaView to decide how to place labels along the Y axis, however for the X axis, we want to explicitly label the values 2.5, 0.5, $-0.5$, and $-4.5$. To that, assuming we are the advanced mode for the `Edit Axes Grid` panel, check `X Axis Use Custom Labels`. That will show a table widget that allows you to add values as shown below.

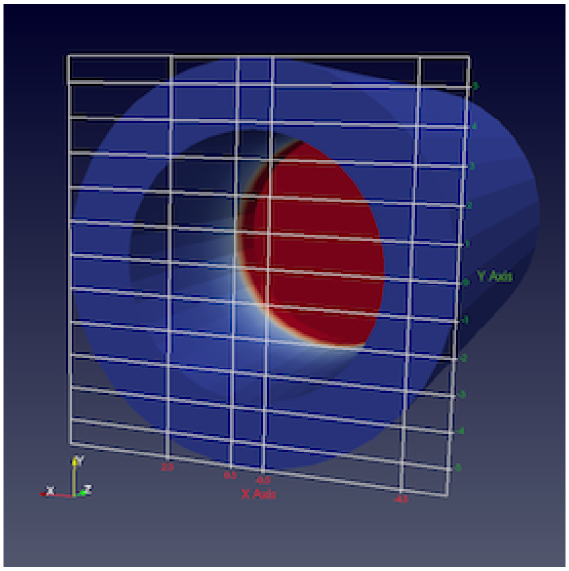


☑ X Axis Use Custom Labels

| | |
|---|---|
| 1 | -4.5 |
| 2 | -0.5 |
| 3 | 0.5 |
| 4 | 2.5 |

Using the + button, add the custom values. While at it, let's also change the `X Axis Label Font Properties` and `X Axis Title Font Properties` to change the color to red and similar for the Y axis, let's change the color to green. Increase the title font sizes to 18, to make them stand out and you will get a visualization as follows (below, left).





Here we see that both sides of the axis plane are labeled. Suppose you only want to label one of the sides, in that case use the `Axes To Label` property to uncheck all but `Min-X` and `Min-Y` . This will result in the visualization shown above, right.

### 2.11.3 `Axes Grid` in `pvpython`

In `pvpython`, `Axes Grid` is accessible as the `AxesGrid` property on the render view.

```
>>> renderView = GetActiveView()

# AxesGrid property provides access to the AxesGrid object.
>>> axesGrid = renderView.AxesGrid

# To toggle visibility of the axes grid,
>>> axesGrid.Visibility = 1
```

All properties on the `Axes Grid` that you set using the `Edit Axes Grid` dialog are available on this `axesGrid` object and can be changed as follows:

```
>>> axesGrid.XTitle = 'X Title'
>>> axesGrid.XTitleColor = [0.6, 0.6, 0.0]
>>> axesGrid.XAxisLabels = [-0.5, 0.5, 2.5, 3.5]
```

Note you can indeed use the tracing capabilities described in Section 1.1.6 to determine what Python API to use to change a specific property on the `Edit Axes Grid` dialog or use `help` .

```
>>> help(axesGrid)
Help on GridAxes3DActor in module paraview.servermanager object:

class GridAxes3DActor(Proxy)
```

(continues on next page)

(continued from previous page)

```
 |  GridAxes3DActor can be used to render a grid in a render view.
 |
 |  Method resolution order:
 |      GridAxes3DActor
 |      Proxy
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  Initialize = aInitialize(self, connection=None, update=True)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  AxesToLabel
 |      Set the mask to select the axes to label. The axes labelled will be a subset of␣
↪the
 |      axes selected depending on which faces are also being rendered.
 |
 |  CullBackface
 |      Set to true to hide faces of the grid facing away from the camera i.e. hide all
 |      back faces.
 |
 |  CullFrontface
 |      Set to true to hide faces of the grid facing towards from the camera i.e. hide␣
↪all
 |      front faces.
 |
 |  DataPosition
 |      If data is being translated, you can show the original data bounds for the axes
 |      instead of the translated bounds by setting the DataPosition to match the
 |      translation applied to the dataset.
 |
...
```

## 2.12 Customizing ParaView

**ParaView** can be customized in a number of ways to tailor it to your preferences and needs. Customization options include setting general application behavior, customizing default property values used for filters, representations, and views, and customizing aspects of the `paraview` client. This chapter describes the different ways to customize **ParaView**.

### 2.12.1 Settings

As with any large application, `paraview` provides mechanisms to customize some of its application behavior. These are referred to as **application settings** . or just **settings**. Such settings can be changed using the `Settings` dialog, which is accessed from the *Edit* > *Settings* menu (*ParaView* > *Preferences* on the Mac). We have seen parts of this dialog earlier, e.g., in Section 2.1.2, Section 2.7.12, and Section 2.7.12. In this section, we will take a closer look at some of the other options available in this dialog.

The `Settings` dialog is split into several tabs. The `General` tab consolidates most of the miscellaneous settings. The `Camera` tab enables you to change the mouse interaction mappings for the `Render View` and similar views. The `Render View` tab, which we saw earlier in Section 2.7.12 and Section 2.7.12, provides options in regards to rendering in `Render View` and similar views. The `Color Palette` tab is used to change the active color palette.

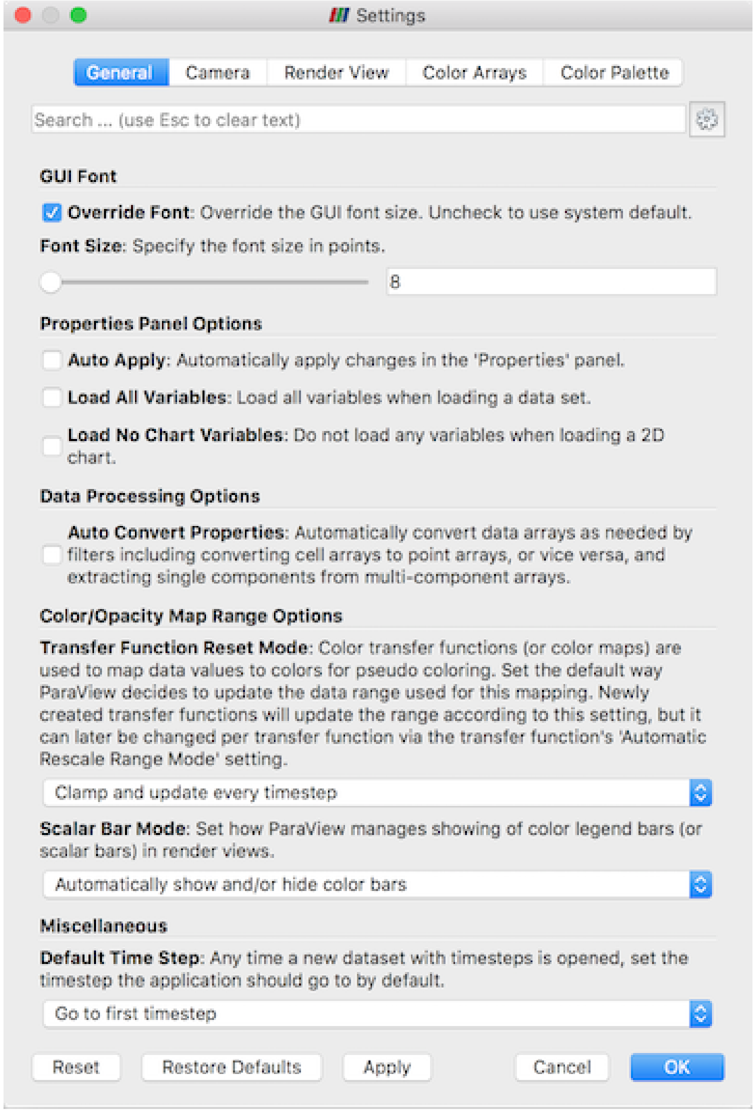
Using this dialog is not much different than the `Properties` panel. You have the `Search` box at the top, which allows you to search properties matching the input text ( Section 2.1.1). The  button can be used to toggle between default and advanced modes.

To apply the changes made to any of the settings, use the `Apply` or `OK` buttons. `OK` will apply the changes and close the dialog, while `Cancel` will reject any changes made and close the dialog. Any changes made to the options in this dialog are persistent across sessions. That is, the next time you launch `paraview`, you'll still be using the same settings chosen earlier. To revert to the default, use the `Restore Defaults` button. You can also manually edit the setting file as in Section 2.12.2. Furthermore, site maintainers can provide site-wide defaults for these, as is explained in Section 2.12.2.

Next, we will see some of the important options available. Those that are only available in the advanced mode are indicated as such using the  icon. You will either need to toggle on the advanced options with the  button or search for the option using the `Search` box.

#### General settings

- `General Options`
  - `Show Welcome Dialog` : Uncheck this to not show the welcome screen at application startup. You will need to restart `paraview` to see the effect.
  - `Show Save State On Exit` : When this is checked `paraview` will prompt you to save a state file when you exit the application.
  - `Crash Recovery` : When this is checked, `paraview` will intermittently save a backup state file as you make changes in the visualization pipeline. If `paraview` crashes for some reason, then when you relaunch `paraview`, it will provide you with a choice to load the backup state saved before the crash occurred. This is not 100% reliable, but some users may find it useful to avoid losing their visualization state due to a crash.
  - `Force Single Column Menus` : On platforms that support multicolumn menus, ensure all menu items are selectable on low-resolution screens.
- `GUI Font`
  - `Override Font` : When checked, use a custom font size for the user interface. This overrides the system default font size.
  - `Font Size` : The size of the font to use for UI elements.
- `View Options`
  - `Default View Type` : When `paraview` starts up, it creates `Render View` by default. You can use this option to change the type of the view that is created by default, instead of the `Render View` . You can even pick `None` if you don't want to create any view by default.



GUI Font

☑ **Override Font**: Override the GUI font size. Uncheck to use system default.

**Font Size**: Specify the font size in points.

8

Properties Panel Options

□ **Auto Apply**: Automatically apply changes in the 'Properties' panel.

□ **Load All Variables**: Load all variables when loading a data set.

□ **Load No Chart Variables**: Do not load any variables when loading a 2D chart.

Data Processing Options

□ **Auto Convert Properties**: Automatically convert data arrays as needed by filters including converting cell arrays to point arrays, or vice versa, and extracting single components from multi-component arrays.

Color/Opacity Map Range Options

**Transfer Function Reset Mode**: Color transfer functions (or color maps) are used to map data values to colors for pseudo coloring. Set the default way ParaView decides to update the data range used for this mapping. Newly created transfer functions will update the range according to this setting, but it can later be changed per transfer function via the transfer function's 'Automatic Rescale Range Mode' setting.

Clamp and update every timestep

**Scalar Bar Mode**: Set how ParaView manages showing of color legend bars (or scalar bars) in render views.

Automatically show and/or hide color bars

Miscellaneous

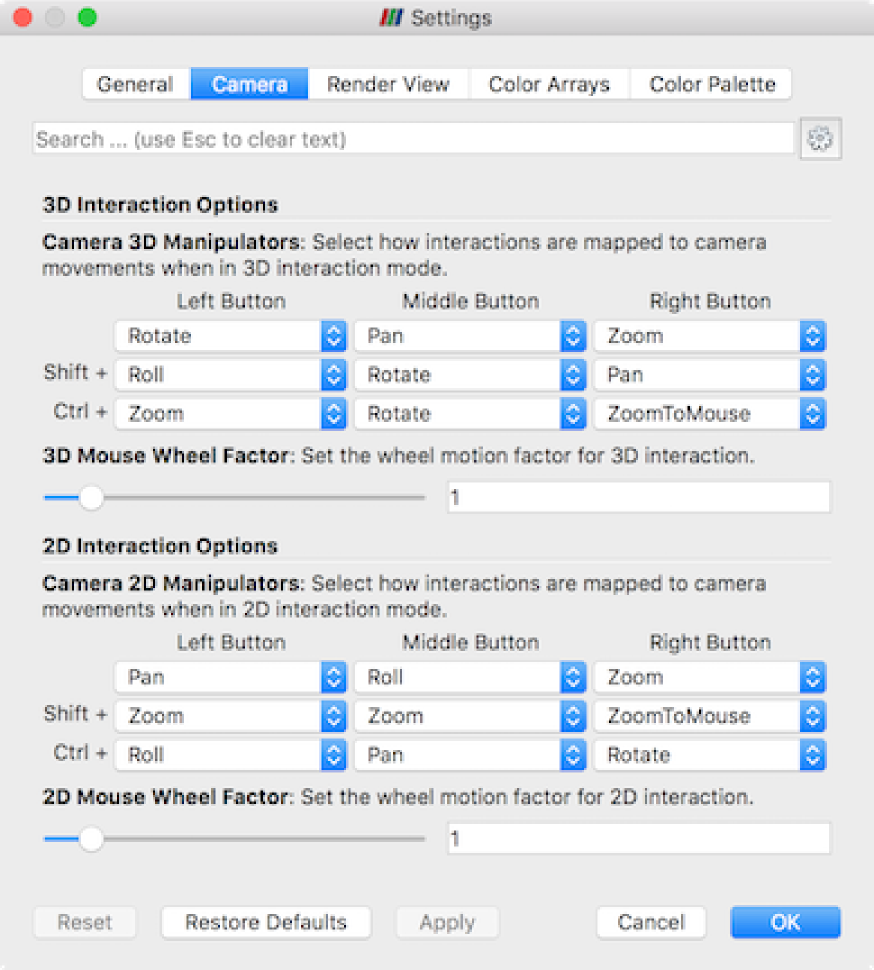
**Default Time Step**: Any time a new dataset with timesteps is opened, set the timestep the application should go to by default.

Go to first timestep

Fig. 2.40: `Settings` dialog in `paraview` showing the `General` settings tab.

- `Properties Panel Options`
  - `Auto Apply` : When checked, the `Properties` panel will automatically apply any changes you make to the properties without requiring you to click the `Apply` button. The same setting can also be toggled using the  button in the `Main Controls` toolbar.
  - `Auto Apply Active Only` : This limits the auto-applying to the properties on the active source alone.
  - `Properties Panel Mode` : This allows you to split the `Properties` panel into separate panels as described in Section 2.1.2.
- `Data Processing Options`
  - `Auto Convert Properties` : Several filters only work on one type of array, e.g., point data arrays or cell data arrays. Before using such filters, you are expected to apply the `Point Data To Cell Data` or `Cell Data To Point Data` filters. To avoid having to add these filters explicitly, you can check this checkbox. When checked, **ParaView** will automatically convert data arrays as needed by filters, including converting cell array to point arrays and vice-versa, as well as extracting a single component from a multi-component array.
- `Color/Opacity Map Range Options`
  - `Transfer Function Reset Mode` : This setting controls the initial settings for how **ParaView** will reset the ranges for color and opacity maps (or transfer functions). This sets the initial value of the `Automatic Rescale Range Mode` for newly created color/opacity maps ( Section 2.3.2). This setting can be changed on a per-color map basis after the color map has been created.
  - `Scalar Bar Mode` : This settings controls how `paraview` manages showing the color legend (or scalar bar) in `Render View` and similar views.
- `Default Time Step` Whenever a dataset with timesteps is opened, this setting controls how `paraview` will update the current time shown by the application. You can choose between `Leave current time unchanged, if possible`, `Go to first timestep`, and `Go to last timestep`.
- `Animation`
  - `Cache Geometry For Animation` : This enables caching of geometry when playing animations to attempt to speed up animation playback in a loop. When caching is enabled, data ranges reported by the `Information` panel and others can be incorrect, since the pipeline may not have updated.
  - `Animation Geometry Cache Limit` : When animation caching is enabled, this setting controls how much geometry (in kilobytes) can be cached by any rank. As soon as a rank's cache size reaches this limit, **ParaView** will no longer cache the remaining timesteps.
  - `Animation Time Notation` : Sets the display notation for the time in the annotation toolbar. Options are `Mixed` , `Scientific` , and `Fixed` .
  - `Animation Time Precision` : Sets the number of digits displayed in the time in the animation toolbar.
- `Maximum Number of Data Representation Labels` When a selection is labeled by data attributes this is the maximum number of labels to use. When the number of points/cells to label is above this value then a subset of this many will be labeled instead. Too many overlapping labels becomes illegible, so this is set to 100 by default.

### `Camera` settings



**3D Interaction Options**

**Camera 3D Manipulators:** Select how interactions are mapped to camera movements when in 3D interaction mode.

| | Left Button | Middle Button | Right Button |
|---|---|---|---|
| | Rotate | Pan | Zoom |
| Shift + | Roll | Rotate | Pan |
| Ctrl + | Zoom | Rotate | ZoomToMouse |

**3D Mouse Wheel Factor:** Set the wheel motion factor for 3D interaction.

1

**2D Interaction Options**

**Camera 2D Manipulators:** Select how interactions are mapped to camera movements when in 2D interaction mode.

| | Left Button | Middle Button | Right Button |
|---|---|---|---|
| | Pan | Roll | Zoom |
| Shift + | Zoom | Zoom | ZoomToMouse |
| Ctrl + | Roll | Pan | Rotate |

**2D Mouse Wheel Factor:** Set the wheel motion factor for 2D interaction.

1

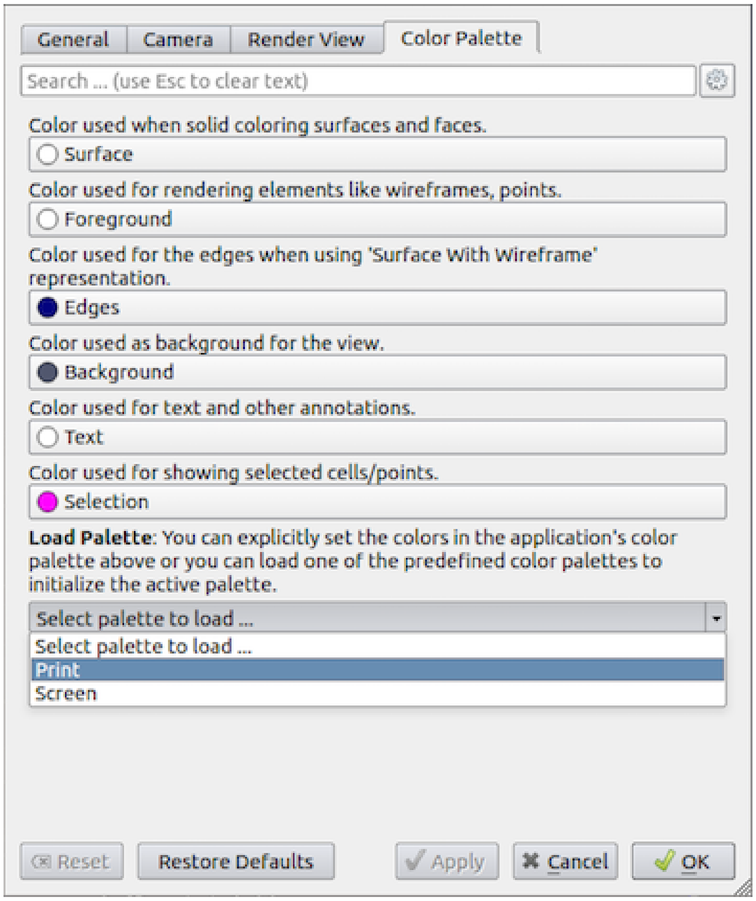
Fig. 2.41: `Settings` dialog in `paraview` showing the `Camera` settings tab.

This tab allows you to control how you can interact in `Render View` and similar views. Basically, you are setting up a mapping between each of the mouse buttons and keyboard modifiers, and the available interaction types including `Rotate`, `Pan`, `Zoom`, etc. The dialog allows you to set the interaction mapping separately for 3D and 2D interaction modes (see Section 1.4.4).

### `Render View` settings

Refer to Section 2.7.12 and Section 2.7.12 for various options available on the `Render View` tab.

### `Color Palette`



Color used when solid coloring surfaces and faces.

○ Surface

Color used for rendering elements like wireframes, points.

○ Foreground

Color used for the edges when using 'Surface With Wireframe' representation.

● Edges

Color used as background for the view.

● Background

Color used for text and other annotations.

○ Text

Color used for showing selected cells/points.

● Selection

**Load Palette**: You can explicitly set the colors in the application's color palette above or you can load one of the predefined color palettes to initialize the active palette.

Select palette to load ...

Select palette to load ...
Print
Screen

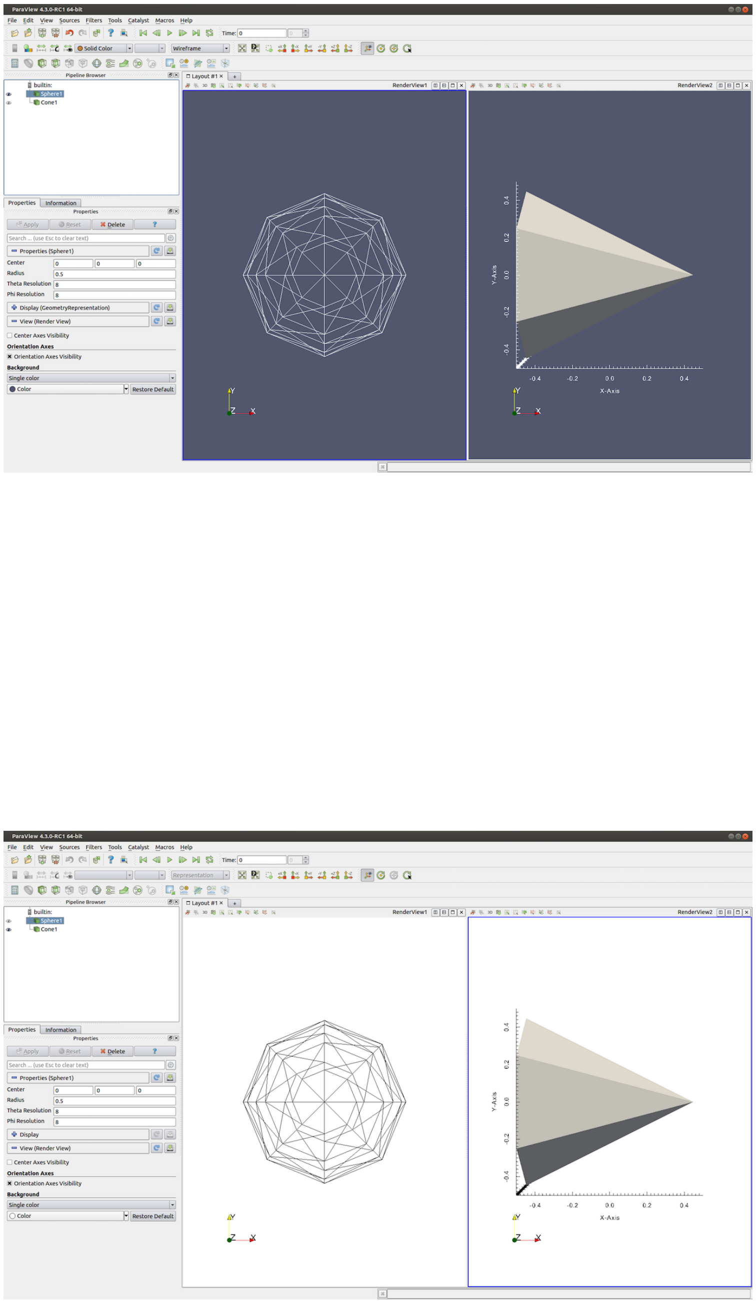
Fig. 2.42: `Settings` dialog in `paraview` showing the `Color Palette` settings tab.

The `Color Palette` tab ( Fig. 2.42) allows you to change the colors in the active color palette. The tab lists the available color categories `Surface` , `Foreground` , `Edges` , `Background` , `Text` , and `Selection` . You can manually set colors to use for each of these categories or load one of the predefined palettes using the `Load Palette` option. To understand **color palettes** , let's look at an example.

Let's start `paraview` and split the active view to create two `Render View` instances side by side. You may want to start `paraview` with the `-dr` command line argument to stop any of your current settings from interfering with this demo. Next, show `Sphere` as `Wireframe` in the view on the left, and show `Cone` as `Surface` in the view on the right. Also, turn on `Cube Axis` for `Cone` . You will see something like Fig. 2.43 (top).

Now let's say you want to generate an image for printing. Typically, for printing, you'd want the background color to be white, while the wireframes and annotations to be colored black. To do that, one way is to go change each of the colors for each each of the views, displays and cube-axes. You can imagine how tedious that will get especially with larger pipelines. Alternatively, using the `Settings` dialog, change the active color palette to `Print` as shown in Fig. 2.42 and then click `OK` or `Apply` . The visualization will immediately change to something like Fig. 2.43 (bottom).

Essentially, **ParaView** allows you to *link* any color property to one of the color categories. When the color palette is changed, any color property linked to a palette category will also be automatically updated to match the category



Center 0 0 0

Radius 0.5

Theta Resolution 8

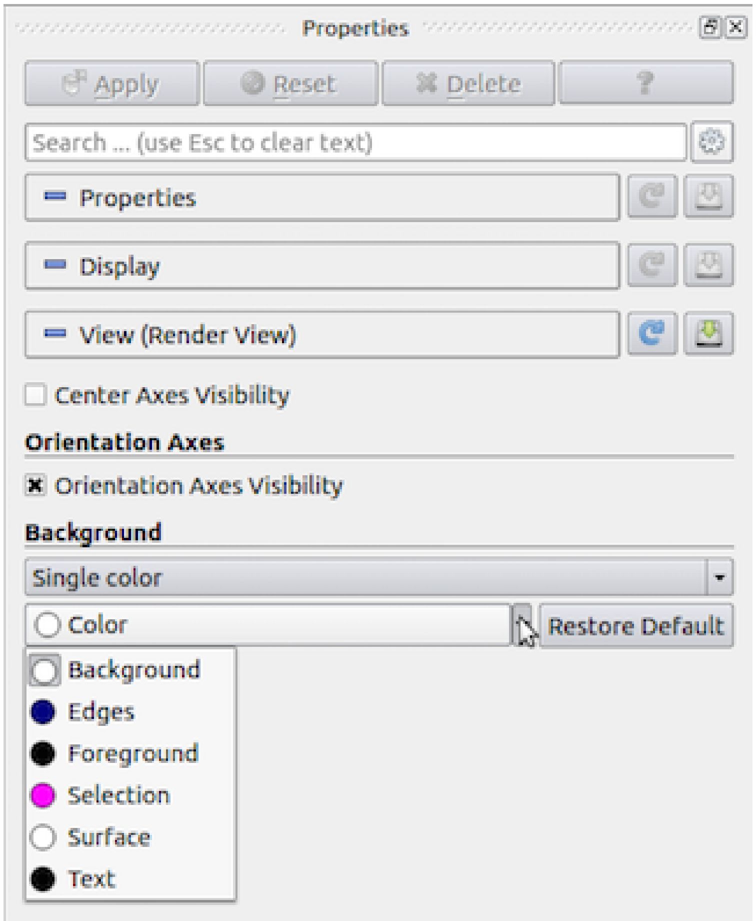
Phi Resolution 8

□ Center Axes Visibility

☒ Orientation Axes Visibility

Fig. 2.43: The effect of loading the `Print` color palette as the active palette. The top is the original visualization and the bottom shows the result after loading the `Print` palette.

color. Fig. 2.44 shows how to link a color property to a color palette category in the Properties panel. Use the tiny drop-down menu marker to make the menu pop up that shows the color palette categories. Select any one of them to link that property with the category. The link is automatically severed if you manually change the color by simply clicking on the button.



□ Center Axes Visibility

Orientation Axes

☒ Orientation Axes Visibility

Background

Single color

○ Color

Fig. 2.44: Popup menu allows you to link a color property to a color palette category in the Properties panel.}

### 2.12.2 Custom default settings

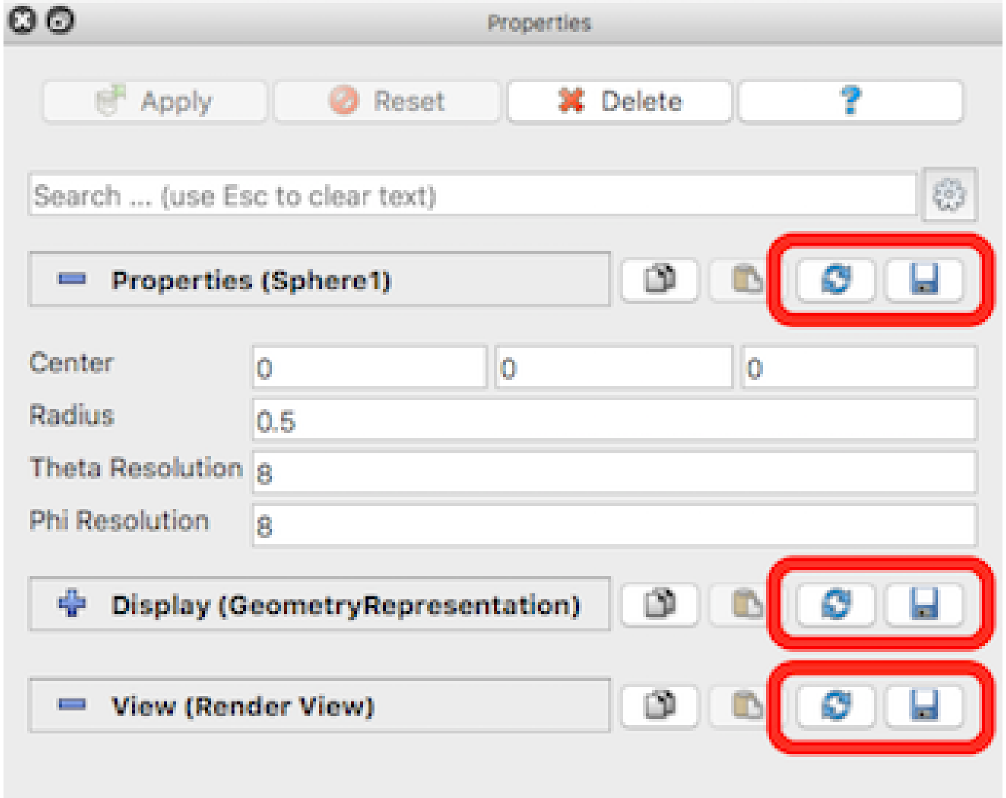
The section describes how to specify custom default settings for the properties of sources, readers, filters, representations, and views. This can be used to specify, for example, the default background color for new views, whether a gradient background should be used, the resolution of a sphere source, which data arrays to load from a particular file format, and the default setting for almost any other object property.

The same custom defaults are used across all the **ParaView** executables. This means that custom defaults specified in the paraview executable are also used as defaults in pvpython and pvbatch, which makes it easier to set up a visualization with paraview and use pvpython or pvbatch to generate an animation from time-series data, for example.

### Customizing defaults for properties

The Properties panel in `paraview` has three sections, `Properties` , `Display` , and `View` . Each section has two buttons. These buttons are circled in red in Fig. 2.45. The button with the disk icon is used to save the current property values in that section that have been applied with the `Apply` button. Property values that have been changed but not applied with the `Apply` button will not be saved as custom default settings.

The button with the circular arrow (or reload icon) is used to restore any custom property settings for the object to **ParaView**'s application defaults. Once you save the current property settings as defaults, those values will be treated as the defaults from then on until you change them to another value or reset them. The saved defaults are written to a configuration file so that they are available when you close and launch **ParaView** again.


Properties
Apply
Reset
Delete
?
Search ... (use Esc to clear text)
Properties (Sphere1)
Display (GeometryRepresentation)
View (Render View)

Center 0 0 0
Radius 0.5
Theta Resolution 8
Phi Resolution 8

Fig. 2.45: Buttons for saving and restoring default property values in the `Properties` panel.

You can undo your changes to the default property values by clicking on the reload button. This will reset the current view property values to `paraview`'s application defaults. To fully restore `paraview`'s default values, you need to click the save button again. If you don't, the restored default values will be applied only to the current object, and new instances of that object will have the custom default values that were saved the last time you clicked the save button.

### Example: specifying a custom background color

Suppose you want to change the default background color in the `Render View` . To do this, scroll down to the `View` section of the `Properties` panel and click on the combo box that shows the current background color. Select a new color, and click `OK` . Next, scroll up to the `View (Render View)` section header and click on the disk button to the right of the header. This will save the new background color as the default background color for new views. To see this, click on the + sign next to the tab above the 3D view to create a new layout. Click on the `Render View` button. A new render view will be created with the custom background color you just saved as default.

### Configuring default settings with JSON

Custom default settings are stored in a text file in the JSON (JavaScript Object Notation) format. We recommend to use the user interface in `paraview` to set most default values, but it is possible to set them by editing the JSON settings file directly. It is always a good idea to make a backup copy of a settings file prior to manual editing.

The **ParaView** executables read from and write to a file named `ParaView-UserSettings.json`, which is located in your home directory on your computer. On Windows, this file is located at `%APPDATA%/ParaView/ParaView-UserSettings.json`, where the `APPDATA` environment variable is usually something like `C:/Users/USERNAME/AppData/Roaming`, where `USERNAME` is your login name. On Unix-like systems, it is located under `~/.config/ParaView/ParaView-UserSettings.json`. This file will exist if you have made any default settings changes through the user interface in the `paraview` executable. Once set, these default settings will be available in subsequent versions of **ParaView**.

A simple example of a file that specifies custom default settings is shown below:

```
{
  "sources" : {
    "SphereSource" : {
      "Radius" : 3.5,
      "ThetaResolution" : 32
    },
    "CylinderSource" : {
      "Radius" : 2
    }
  },
  "views" : {
    "RenderView" : {
      "Background" : [0.0, 0.0, 0.0]
    }
  }
}
```

Note the hierarchical organization of the file. The first level of the hierarchy specifies the group to which the object whose settings are being specified refers ("sources" in this example). The second level names the object whose settings are being specified. Finally, the third level specifies the custom default settings themselves. Note that default values can be set to literal numbers, strings, or arrays (denoted by comma-separated literals in square brackets).

The names of groups and objects come from the XML proxy definition files in **ParaView**'s source code in the directory `ParaView/ParaViewCore/ServerManager/SMApplication/Resources` (`ParaView\ParaViewCore\ServerManager\SMApplication\Resources` on Windows systems) . The group name is defined by the `name` attribute in a `ProxyGroup` element. The object name comes from the `name` attribute in the `Proxy` element (or elements of `vtkSMProxy` subclasses). The property names come from the `name` attribute in the `*Property` XML elements for the object.

**Did you know?**

The application-wide settings available in `paraview` through the *Edit* > *Settings* menu are also saved to this user settings file. Hence, if you have changed the application settings, you will see some entries under a group named "settings".

### Configuring site-wide default settings

In addition to individual custom default settings, **ParaView** offers a way to specify site-wide custom default settings for a **ParaView** installation. These site-wide custom defaults must be defined in a JSON file with the same structure as the user settings file. In fact, one way to create a site settings file is to set the custom defaults desired in `paraview`, close the program, and then copy the user settings file to the site settings file. The site settings file must be named `ParaView-SiteSettings.json`.

The **ParaView** executables will search for the site settings file in several locations. If you installed **ParaView** in the directory `INSTALL`, then the **ParaView** executables will search for the site settings file in these directories in the specified order:

- *INSTALL/share/paraview-X.Y* ( *INSTALLshareparaview-X.Y* in Windows systems)
- *INSTALL/lib* (*INSTALLlib* in Windows systems)
- *INSTALL*
- *INSTALL/..* (*INSTALL/lib* in Windows systems)

where `X` is **ParaView**'s major version number and `Y` is the minor version number. **ParaView** executables will search these directories in the given order, reading in the first `ParaView-SiteSettings.json` file it finds. The conventional location for this kind of configuration file is in the `share` directory (the first directory searched), so we recommend placing the site settings file there.

Custom defaults in the user settings file take precedence over custom defaults in the site settings. If the same default is specified in both the `ParaView-SiteSettings.json` file and `ParaView-UserSettings.json` file in a user's directory, the default specified in the `ParaView-UserSettings.json` file will be used. This is true for both object property settings and application-settings set through the *Edit* > *Settings* menu.

To aid in debugging problems with the site settings file location, you can define an evironment variable named `PV_SETTINGS_DEBUG` to something other than an empty string. This will turn on verbose output showing where the **ParaView** executables are looking for the site settings file.

CHAPTER

THREE

# CATALYST

Catalyst is an **in-situ framework** created as part of ParaView a few years ago. With time and use, we saw the different drawbacks of our approach and then it was decided to create a **new architecture**, Catalyst API.

With this new implementation, it is:

- **easier to implement** in your simulation (less knowledge required)
- **easier to update** from a version to another (less dependencies, has binary compatibility)
- possible to activate **Steering** mode, where ParaView can modify simulation parameters at runtime

**Note:** This is documentation for the Paraview implementation of the Catalyst API (Catalyst 2) if you are looking for information regarding the previous version of catalyst check this manual.

## 3.1 Getting Started

This is a short introduction on how to use a C++ example from the ParaView code base. This code represents a simple simulation that makes use of the Catalyst API.

**Tip:** You can skip cloning the entrire ParaView source code by downloading just the example directory.

Contents of the `CxxFullExample` directory:

- `FEDriver.cxx`: the main loop of the simulation
- `FEDataStructure.[h,cxx]`: simulation files describing the data structures of the driver.
- `CatalystAdaptor.h`: the interface to the Catalyst library. Data descriptions and catalyst setup happens here. This is the main file to edit when you adapt the example for another simulation.
- `catalyst_pipeline.py`: a script called at runtime
- `catalyst_pipeline_live.py`: Alternative script to call at runtime that enables Live Visualization
- `CMakeLists.txt`: The CMake code to configure and build the project. It demonstrates how to find the Catalyst library and link against it.

### 3.1.1 Prerequisites

- MPI
- libcatalyst installed on your system.
- CMake
- ParaView binary (version 5.10.1+)

### 3.1.2 Building the example

```
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=<libcatalyst-install-dir> ../CxxFullExample
cmake --build .
```

### 3.1.3 Simple Run

Run the generated executable `./bin/CxxFullExampleV2` with `catalyst_script.py` as a parameter.

The executable needs to link to the ParaViewCatalyst library at runtime. To achieve this, we initialize `CATALYST_IMPLEMENTATION_PATHS` and `CATALYST_IMPLEMENTATION_NAME` to the path of the ParaViewCatalyst library and the name of the catalyst library implementation. Finally, we pass `catalyst_script.py` as script argument:

```
export CATALYST_IMPLEMENTATION_PATHS="<paraview-install-dir>/lib/catalyst"
export CATALYST_IMPLEMENTATION_NAME=paraview
./bin/CxxFullExample catalyst_pipeline.py
```

Expected output:

```
executing catalyst_pipeline
-----------------------------------
executing (cycle=0, time=0.0)
bounds: (0.0, 69.0, 0.0, 64.9, 0.0, 55.9)
velocity-magnitude-range: (0.0, 0.0)
pressure-range: (1.0, 1.0)
-----------------------------------
executing (cycle=1, time=0.1)
bounds: (0.0, 69.0, 0.0, 64.9, 0.0, 55.9)
velocity-magnitude-range: (0.0, 6.490000000000001)
pressure-range: (1.0, 1.0)
-----------------------------------
...
```

**Note:** Be careful that the MPI version used to build the simulation should be the same as the MPI used by ParaView. In case of errors, you may have to build ParaView by yourself, so ParaView and simulation share the same version of MPI. For this tutorial, you can also remove MPI-related lines in the example source code.

### 3.1.4 Generating Catalyst Scripts

To specify the pipelines executed each time Catalyst is triggered you can modify the `catalyst_script.py` directly or generate one interactively through the ParaView UI.

To generate a catalyst script, assemble your pipelines as usual and append extractor(s) at the end of each pipeline. Extractors can be created through the *Extractors* menu and allow to extract and save data as meshes, images or even tables. Once the pipeline is ready use *File > Save Catalyst State* to export the state as a Python script which you can supply to `./bin/CxxFullExampleV2`. For more details on extractors see section *Extractors* in the UserGuide section.

### 3.1.5 Live Visualization

When saving a Catalyst State you have the option to enable **Live Visualization** in the export wizard

To create your own script, start creating a pipeline in ParaView. Use *Extractors* to write your results (screenshots or meshes). Then use *File > Save Catalyst State* to export the Python script. Live Visualization is an option you can enable in the export wizard.

For an example try `catalyst_pipeline_live.py` to experiment with taking a screenshot and running with live visualization enabled. You can also load it as a State File to inspect its content.

Use *Catalyst > Connect* menu in ParaView so ParaView will wait for Catalyst input. Then run your simulation as explained above. The configured pipeline will appear in your ParaView session.

**Tip:** Use *Catalyst > Pause Simulation* before starting the simulation to pause the simulation on the first timestep. This allows inspection of the pipeline prior to running it. This is also useful for examples which iterate too fast to see the timestep updates.

**Note:** When the simulation ends, it breaks the connection with the ParaView application so it is expected to have some error message like the following:

```
ERROR: In <paraview_source_dir>/VTK/Parallel/Core/vtkSocketCommunicator.cxx, line 781
vtkSocketCommunicator (0x563b4cb9b5b0): Could not receive tag. 1

ERROR: In <paraview_source_dir>/Remoting/Core/vtkTCPNetworkAccessManager.cxx, line 296
vtkTCPNetworkAccessManager (0x563b457d9000): Some error in socket processing.
```

### 3.1.6 Using ParaViewCatalyst in your simulation

To enable the use of ParaViewCatalyst for your simulation code, look at the `CatalystAdaptor.h` file. It wraps the main catalyst functions `catalyst_*` while providing the expected arguments. For the protocols used in each of the calls see *here*.

## 3.2 Background

### 3.2.1 Introduction

Prior to ParaView version 5.9, for a simulation code to use ParaView for in situ processing required developing an adapter which had two parts: convert simulation data structures to VTK Data Object, and use classes provided by ParaView (collectively referred to as Catalyst) to initialize, and then execute the analysis pipelines on each simulation cycle.

Converting simulation data structures to VTK Data Objects is a non-trivial task and requires understanding of how VTK stores internal arrays and builds data objects. Simple mistakes could result in invalid memory accesses or costly data-copies impacting memory requirements and performance adversely.

Setting up and invoking ParaView via the Catalyst classes requires creating and using classes such as `vtkCPProcessor`, `vtkCPPipeline` and subclasses, `vtkCPInputDataDescription`, `vtkCPDataDescription`, and several others.

In other words, the adapter ended up with a lot of ParaView-specific C++ code that potentially changed between each version of ParaView and needed updates to accommodate newly added capabilities.

Since the simulation directly links against the custom Catalyst adapter developed specifically for the simulation, the simulation too is tightly coupled to a specific version of ParaView. Once built, it isn't possible to switch which version of ParaView is being used without rebuilding the adapter and the simulation.

To build an adapter, you need a ParaView SDK. Since official ParaView binaries do not provide headers and libraries that would comprise an SDK, you have to build ParaView from source. That itself can be a daunting task adding further to the complexity and learning curve.

To minimize several of these challenges, we revisited the design and implementation of the various components involved. To avoid confusion, all the Catalyst and in situ components described so far that are available prior to ParaView 5.9 are referred to as Legacy.

### 3.2.2 Catalyst and ParaView-Catalyst

The new design is built on the following key components:

- A stable API that simulation codes can use to describe data and invoke in situ processing pipelines.
- A lightweight implementation of this API that can be used to build simulations when using this API.
- An implementation of this API that uses ParaView for data processing that is ABI compatible with the lightweight implementation and hence can be dynamically replaced at load-time when launching the simulation.

The stable API is now called the Catalyst API. It is a C-only API (wrapped also in C++ and with bindings available for Python and Fortran) that includes mechanisms to describe data and other control parameters (using Conduit API) and trigger in situ processing. It is provided in a separate project together with a lightweight implementation called the **stub**.

The compatible ParaView-specific implementation of the Catalyst API is now called ParaView-Catalyst and is built and distributed as part of the ParaView distribution. *ParaView-Catalyst Blueprint* describes parameters supported by this Catalyst implementation for providing scripts to load, computational meshes etc.

A typical Catalyst adapter developed for a specific simulation, in this new approach, no longer directly builds VTK data objects. Instead the adapter simply describes its data structures using an implementation supported protocol. ParaView-Catalyst, the canonical implementation of the Catalyst API that uses ParaView, provides several ways of describing data and will continue to evolve to include a large set of data-structures and memory layouts used by codes.

If that's not adequate, developers can develop their own custom implementation for the Catalyst API. Such implementations are of course free to use whatever data processing and visualization libraries the developers choose. They can also use vtkInSituInitializationHelper, vtkInSituPipeline and subclasses, to use ParaView as the in situ processing engine.

## 3.3 ParaView-Catalyst Blueprint

### 3.3.1 Background

Starting with ParaView 5.9, ParaView distribution provides an implementation of the Catalyst API . This implementation is now referred to as **ParaView-Catalyst**. Thus, it is an implementation of the Catalyst In Situ API that uses ParaView for data processing and rendering.

The Catalyst in situ API comprises of 5 function calls that are used to pass data and control over to the Catalyst implementation from computational simulation codes: catalyst_initialize, catalyst_execute, catalyst_results , catalyst_finalize and catalyst_about. Each of these functions is passed a Conduit Node object. Conduit Node provides a flexible mechanism for describing hierarchical in-core scientific data. Since a Conduit Node is simply a light-weight container, we need to define the conventions used to communicate relevant data in each of the Catalyst API calls. Such conventions are collectively referred to as the Blueprint. Each Catalyst API implementation can develop its own blueprint. ParaView-Catalyst, the Catalyst API implementation provided by ParaView, also defines its blueprint called the **ParaView-Catalyst Blueprint**. This page documents this blueprint.

### 3.3.2 Protocol

A blueprint often includes several protocols, each defining the conventions for a specific use-case. Since there are three main Catalyst API functions, the blueprint currently defines three protocols, one for each of these Catalyst API functions.

- **protocol**: initialize: defines the options accepted by catalyst_initialize; these include things like adding ParaView Python scripts to load.
- **protocol**: execute: defines the protocol for catalyst_execute and includes information about Catalyst channels i.e. ports on which data is made available to in situ processing as well as the actual data from the simulation.
- **protocol**: finalize: defines the protocol for catalyst_finalize; currently, this is empty.

In each of the Catalyst API calls, ParaView looks for a top-level node named catalyst. The expected children vary based on the protocol described in the following sub-sections.

These top-level protocols use other internal protocols e.g. channel.

#### protocol: initialize

Currently, the initialize protocol defines how to pass scripts to load for analysis.

- catalyst/scripts: (optional) if present must either be a list or object node with child nodes that provides paths to the Python scripts to load for in situ analysis.
- catalyst/scripts/[name]: (optional) if present can be a string or object. If it is a string it is interpreted as path to the Python script. If it is an object it can have the following attributes.
  - catalyst/scripts/[name]/filename: path to the Python script
  - catalyst/scripts/[name]/args: (optional) if present must be of type list with each child node of type string. To retrieve these arguments from the script itself use the get_args() method of the paraview catalyst module.

Additionally, one can provide a list of pre-compiled pipelines to use.

- catalyst/pipelines: (optional) if present must be a list or object node with child nodes that are objects that provide type and parameters for hard-coded pipelines to use. Each object must be in accordance to the protocol: pipeline.

In MPI-enabled builds, ParaView is by default initialized to use MPI_COMM_WORLD as the global communicator. A specific MPI communicator can be provided as follows:

- catalyst/mpi_comm: (optional) if present, must be an integer representing the Fortran handle for the MPI communicator to use. The Fortran handle can be obtained from MPI_Comm using MPI_Comm_c2f().

### protocol: execute

Defines how to communicate data during each time-iteration.

time/timestep/cycle: this defines temporal information about the current invocation.

- catalyst/state/timestep: (optional) integral value for current timestep. if not specified, catalyst/cycle is used.
- catalyst/state/cycle: (optional) integral value for current cycle index, if not specified, 0 is assumed.
- catalyst/state/time: (optional) float64 value for current time, if not specified, 0.0 is assumed.
- catalyst/state/parameters: (optional) list of optional runtime parameters. If present, they must be of type list with each child node of type string.
- catalyst/state/multiblock: (optional) integral value. When present and set to 1, output will be a legacy vtkMultiBlockDataSet.

**channels**: channels are used to communicate simulation data. The **channels** node can have one or more children, each corresponding to a named channel. A channel represents a data-source in the analysis pipeline that is linked to the data being produced by the simulation code. Most simulation adapters will only have a single channel. Multi-physics simulation codes will likely have multiple channels, one channel for each type of physics being simulated (e.g. a channel for the fluid domain and a channel for the structured domain for a fluid-structure interaction simulation).

- catalyst/channels/[channel-name]: (protocol: channel): (optional) if present represents a named channel with the name channel-name. The node must be an object node satisfying the channel protocol.

The channel protocol is as follows:

- channel/type: (required) a string representing the channel type. Currently, the only supported values are mesh and multimesh.

  mesh is used to indicate that this channel is specified in accordance to the Conduit Mesh protocol.

  multimesh is an extension for multi-domain meshes (also called multiblocks) and is described later.

- channel/data: (required) an object node used to communicate the simulation data on this channel. This node must match the protocol requirements identified by the channel/type.
- channel/data/state/fields/[field-name]: (optional) defines extra fields associated to the current mesh. Each field is not associated to any topology and it could be a string, a numerical array or an array following the MCArray Blueprint protocol. In ParaView Catalyst, the each field will be added as a Field Data array to the generated VTK object.
- channel/state: (optional) fields to optionally override the catalyst/state temporal information.
- channel/state/timestep: (optional) if present, overrides catalyst/state/timestep for this channel.
- channel/state/cycle: (optional) if present, overrides catalyst/state/cycle for this channel.

- channel/state/time: (optional) if present, overrides catalyst/state/time for this channel. A channel will default to using the catalyst/state/ values for these parameters for each channel/state parameter not specified.
- channel/state/multiblock: (optional) if present, overrides catalyst/state/multiblock for this channel.

### protocol: finalize

Currently, this is empty.

### protocol: pipeline

Defines type and parameters for a hard-coded pipeline.

- **type: (required) a string identifying the type of the pipeline.**
  Currently supported value is io.

When type is io, the following attributes are supported.

- filename: (required) a string representing a filename. {timestep} may be used to replace with timestep or {time} may be used to replace with time value. You can use fmt style format specifiers e.g. {time:03f} etc. The filename may be used to determine which supported writer to use for saving the data.
- channel: (required) a string identifying the channel by its name.

### protocol: multimesh

This is the protocol used for the channel/data when the channel/type is set to "multimesh".

- channel/data/[block-name]: (protocol: mesh): (optional) if present, represents an individual mesh/block described using the Conduit Mesh protocol. This can be repeated for multiple blocks. block-name must be unique for each blocks.
- channel/assembly: (optional) if present, can be used to define arbitrary hierarchical relationships between individual meshes/blocks in the mutlimesh. For example, for two blocks named blockA and blockB, following is a possible hierarchy. Thus all nodes are either object, list or string, and if string, must refer to a valid block name.

```
channel/assembly/
                AllBlocks: ["blockA", "blockB"]
                BlockA: "blockA"
                BlockB: "blockB"
                SubGroup/
                        AnotherChild: "blockA"
```

## 3.4 Fides Reader

An alternative to *Mesh Blueprints* is using the Fides reader. Fides uses a different data model based around JSON and ADIOS2. More details about the Fides Data Model page. An example of using ParaViewCatalyst with Fides is located here.

## 3.5 Examples for Simulation Developers

These examples reside in the ParaView repository and are part of ParaView's CI process. The core differences are in the Adaptor class of each example which is where the simulation data are converted to a conduit node and passed to ParaView for processing. For a detailed example regarding the structure of a Catalyst Adaptor see here

To experiment with any example click on the download directory button in the GitLab interface and follow the instructions of *Getting Started* section.

- C driver that creates unstructured mesh in a single channel [code]
- C++ driver that creates unstructured mesh in a single channel [code]
- Fortran90 driver that creates uniform mesh in a single channel [code]
- Python driver that creates a uniform mesh in a single channel [code]
- C++ driver that creates uniform mesh and passes arguments to the catalyst script [code]
- C++ driver that creates 2 multiblock unstructured meshes each one on its dedicated channel [code]
- C++ driver that uses the *multimesh protocol* [code]
- C++ driver that utilizes a polygonal mesh [code]
- C++ driver that utilizes a polyhedral mesh [code]
- C++ dirver that utilizes an overlapping AMR dataset [code]
- C++ driver that utilizes steering mode, where one can modify simulation parameters at runtime [code]

## 3.6 Debugging Tips

- Set the environmental variable *PARAVIEW_LOG_CATALYST_VERBOSITY* to *INFO* to increase the verbosity of messages at the ParaViewCatalyst level.
- Dump the conduit nodes passed to Catalyst for inspection using Catalyst Replay,
- Use Catalyst Player to iterate through times-series files from disk to emulate a simulation.

CHAPTER

# FOUR

# PARAVIEW TUTORIALS

**Tutorials** are split in **Self-directed Tutorials** and **Classroom Tutorials**:

> **Self-directed Tutorial**'s Section 4.1.1 to Section 4.1.5 provide an introduction to the **ParaView** software and its history, and exercises on how to use **ParaView** that cover basic usage, batch Python scripting and visualizing large models.
>
> **Classroom Tutorials**'s Section 4.2.1 to Section 4.2.18 provide beginning, advanced, Python and batch, and targeted tutorial lessons on how to use **ParaView** that are presented as a 3-hour class internally within Sandia National Laboratories.

## 4.1 Self-directed Tutorial

**Self-directed Tutorial** provides an introduction to the **ParaView** software and its history, and exercises on how to use **ParaView** that cover basic usage, batch python scripting and visualizing large models. This **tutorial** was created by Kenneth Moreland at Sandia National Laboratories, has written guidance and background and can be followed independently.

Thanks to Amy Squillacote, David DeMarle, and W. Alan Scott for contributing material to this tutorial. And, of course, thanks to everyone at Kitware, Sandia National Laboratories, Los Alamos National Laboratory, and all other contributing organizations for their hard work in making **ParaView** what it is today.

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. 12-015215, through the Scientific Discovery through Advanced Computing (SciDAC) Institute of Scalable Data Management, Analysis and Visualization.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY

### 4.1.1 Introduction

**ParaView** is an open-source application for visualizing two- and three-dimensional datasets. The size of the datasets **ParaView** can handle varies widely depending on the architecture on which the application is run. The platforms supported by **ParaView** range from single-processor workstations to multiple-processor distributed-memory supercomputers or workstation clusters. Using a parallel machine, **ParaView** can process very large data sets in parallel and later collect the results. To date, **ParaView** has been demonstrated to process billions of unstructured cells and to process over a trillion structured cells. **ParaView**'s parallel framework has run on over 100,000 processing cores.

**ParaView**'s design contains many conceptual features that make it stand apart from other scientific visualization solutions.

- An open-source, scalable, multi-platform visualization application.
- Support for distributed computation models to process large datasets.
- An open, flexible, and intuitive user interface.
- An extensible, modular architecture based on open standards.
- A flexible BSD 3-clause license.
- Commercial maintenance and support.

**ParaView** is used by many academic, government, and commercial institutions all over the world. **ParaView**'s open license makes it impossible to track exactly how many users **ParaView** has, but it is thought to be many thousands large based on indirect evidence. For example, **ParaView** is downloaded roughly 100,000 times every year. **ParaView** also won the HPCwire Readers' Choice Award in 2010 and 2012 and HPCwire Editors' Choice Award in 2010 for Best HPC Visualization Product or Technology.

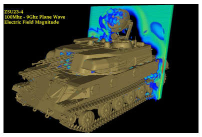


Fig. 4.1: ZSU23-4 Russian Anti-Aircraft vehicle being hit by a planar wave. Image courtesy of Jerry Clarke, US Army Research Laboratory.

As demonstrated in these visualizations, **ParaView** is a general-purpose tool with a wide breadth of applications. In addition to scaling from small to large data, **ParaView** provides many general-purpose visualization algorithms as well

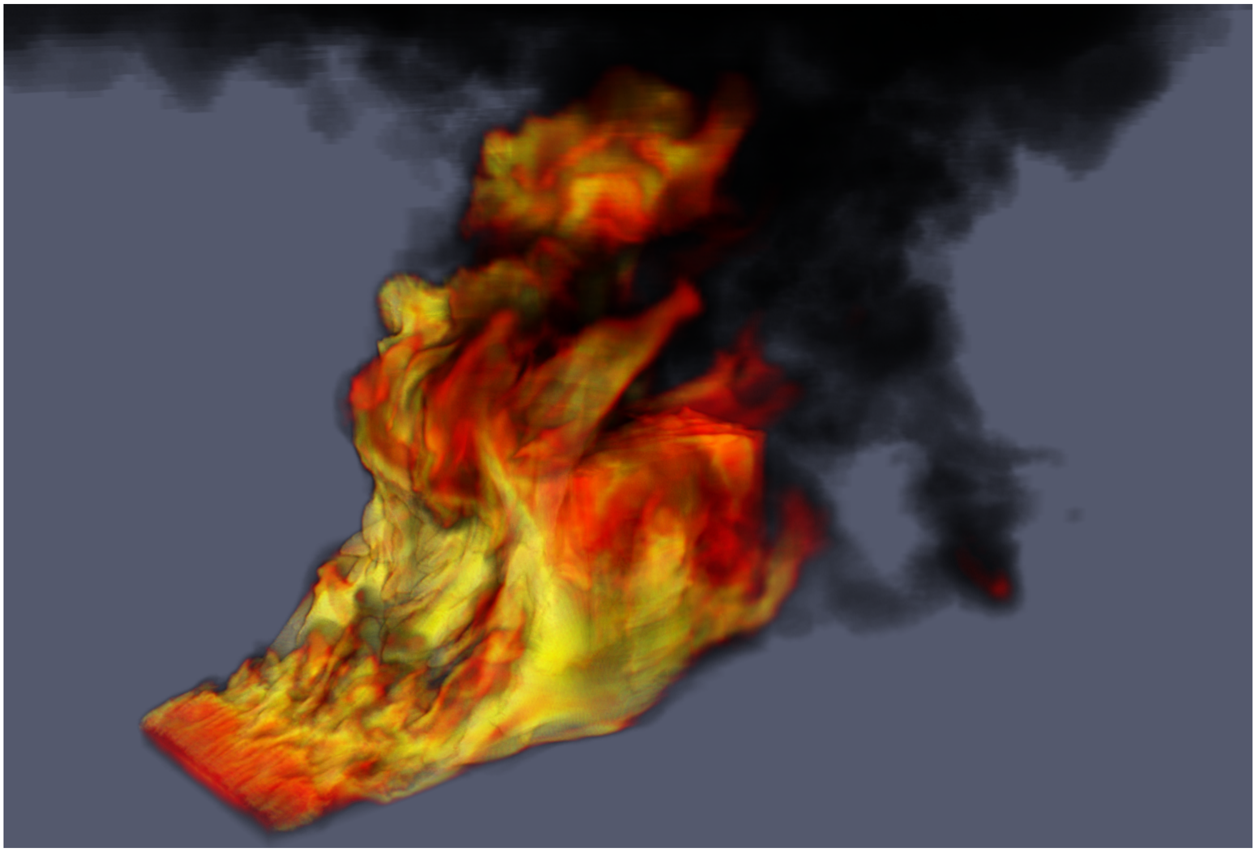


Fig. 4.2: A loosely coupled SIERRA-Fuego-Syrinx-Calore simulation with 10 million unstructured hexahedra cells of objects-in-crosswind fire.

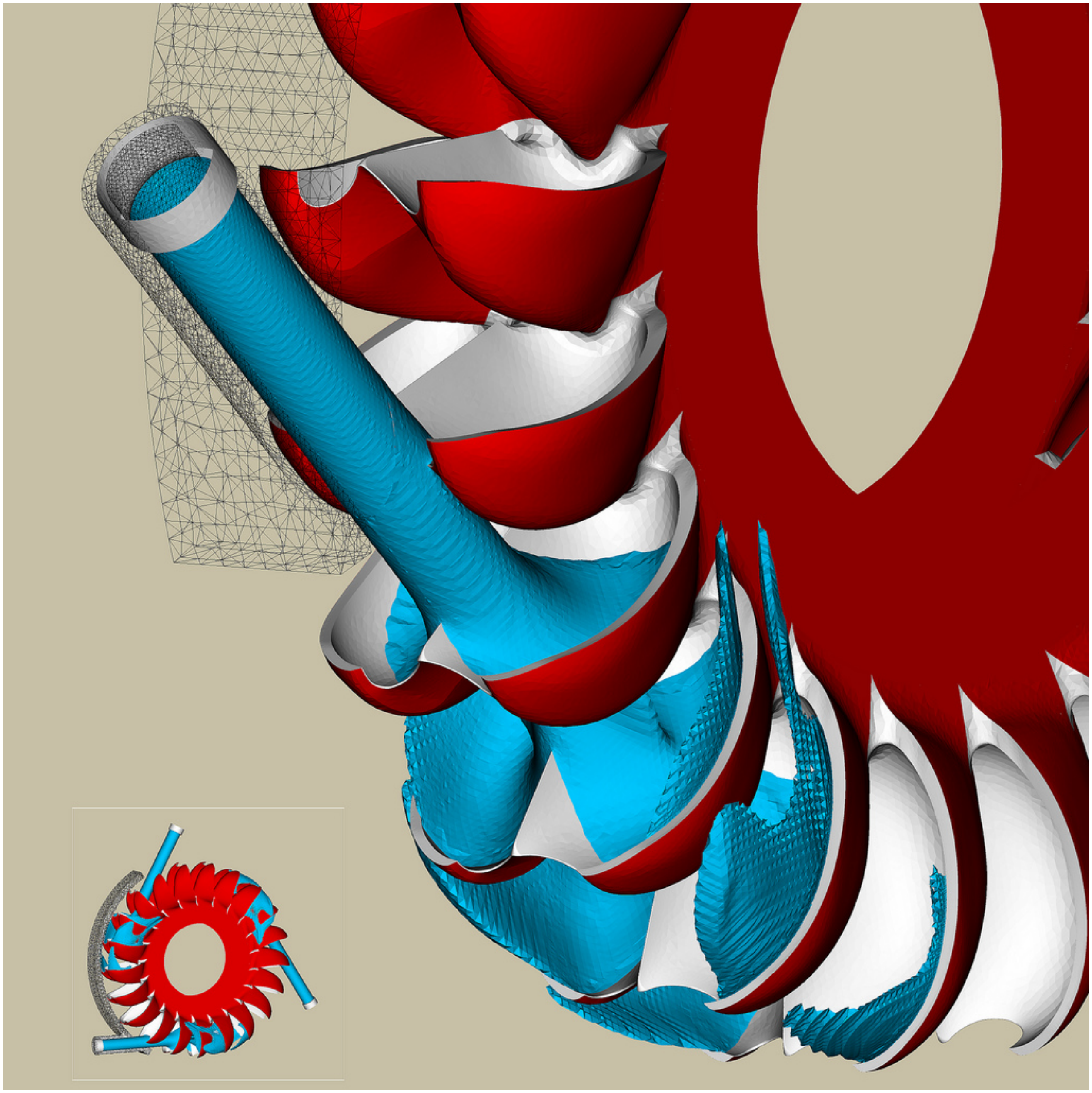


Fig. 4.3: Simulation of a Pelton turbine. Image courtesy of the Swiss National Supercomputing Centre.

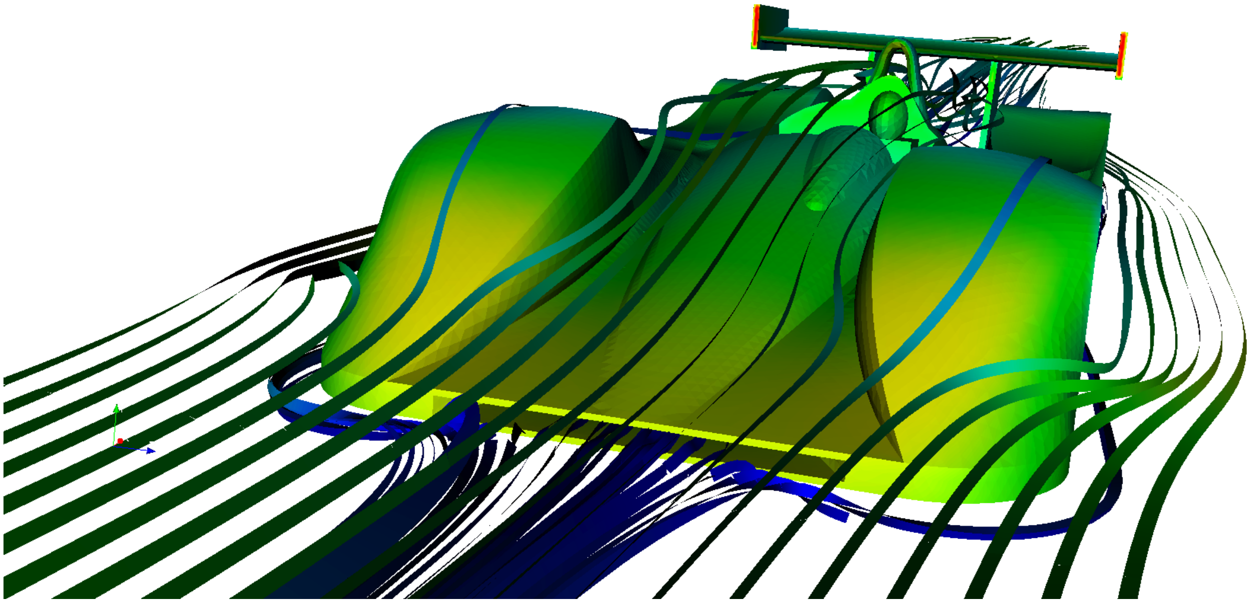
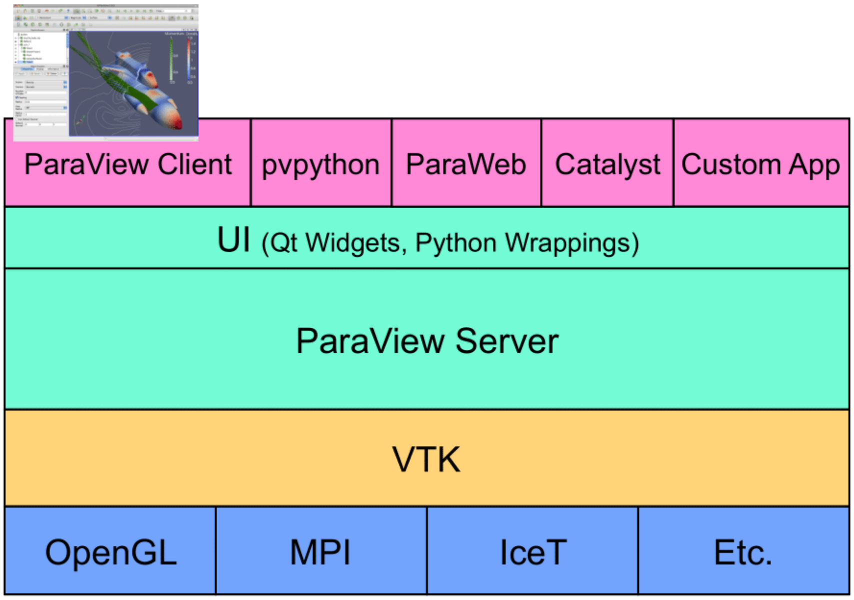


Fig. 4.4: Airflow around a Le Mans Race car. Image courtesy of Renato N. Elias, NACAD/COPPE/UFRJ, Rio de Janerio, Brazil.

as some specific to particular scientific disciplines. Furthermore, the **ParaView** system can be extended with custom visualization algorithms.



ParaView Client | pvpython | ParaWeb | Catalyst | Custom App

UI (Qt Widgets, Python Wrappings)

ParaView Server

VTK

OpenGL | MPI | IceT | Etc.

The application most people associate with **ParaView** is really just a small client application built on top of a tall stack of libraries that provide **ParaView** with its functionality. Because the vast majority of **ParaView** features are implemented in libraries, it is possible to completely replace the **ParaView** GUI with your own custom application. Furthermore, **ParaView** comes with a application that allows you to automate the visualization and post-processing with Python scripting.

Available to each **ParaView** application is a library of user interface components to maximize code sharing between them. A library provides the abstraction layer necessary for running parallel, interactive visualization. It relieves the client application from most of the issues concerning if and how **ParaView** is running in parallel. The **Visualization**

**Toolkit (VTK)** provides the basic visualization and rendering algorithms. VTK incorporates several other libraries to provide basic functionalities such as rendering, parallel processing, file I/O, and parallel rendering. Although this tutorial demonstrates using **ParaView** through the **ParaView** client application, be aware that the modular design of **ParaView** allows for a great deal of flexibility and customization.

### Development and Funding

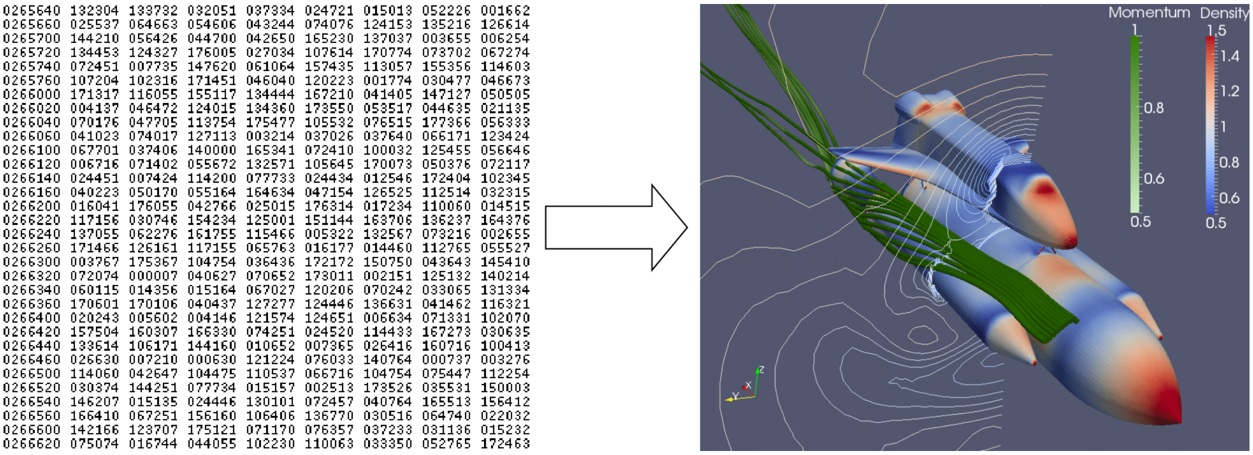
The **ParaView** project started in 2000 as a collaborative effort between Kitware Inc. and Los Alamos National Laboratory. The initial funding was provided by a three year contract with the US Department of Energy ASCI Views program. The first public release, **ParaView** 0.6, was announced in October 2002. Development of **ParaView** continued through collaboration of Kitware Inc. with Sandia National Laboratories, Los Alamos National Laboratories, the Army Research Laboratory, and various other academic and government institutions.

In September 2005, Kitware, Sandia National Labs and CSimSoft started the development of **ParaView** 3.0. This was a major effort focused on rewriting the user interface to be more user friendly and on developing a quantitative analysis framework. **ParaView** 3.0 was released in May 2007.

Since this time, **ParaView** development continues. **ParaView** 4.0 was released in June 2013 and introduced more cohesive GUI controls and better multiblock interaction. Subsequent releases also include the Catalyst library for in situ integration into simulation and other applications. **ParaView** 5.0 was released in January 2016 and provided a major update to the rendering system. The new rendering takes advantage of OpenGL 3.2 features to provide huge performance improvements. Subsequent releases also added support for ray cast rendering with the OSPRay library.

Development of **ParaView** continues today. Sandia National Laboratories continues to fund **ParaView** development through the ASC project. **ParaView** is part of the SciDAC Scalable Data Management, Analysis, and Visualization (SDAV) Institute Toolkit https://sdav-scidac.org. The US Department of Energy also funds **ParaView** through Los Alamos National Laboratories and various SBIR projects and other contracts. The US National Science Foundation also often funds **ParaView** through SBIR projects. Other institutions also have **ParaView** support contracts: Electricity de France, Mirarco, and oil industry customers. Also, because **ParaView** is an open source project, other institutions such as the Swiss National Supercomputing Centre contribute back their own development.

### Basics of Visualization

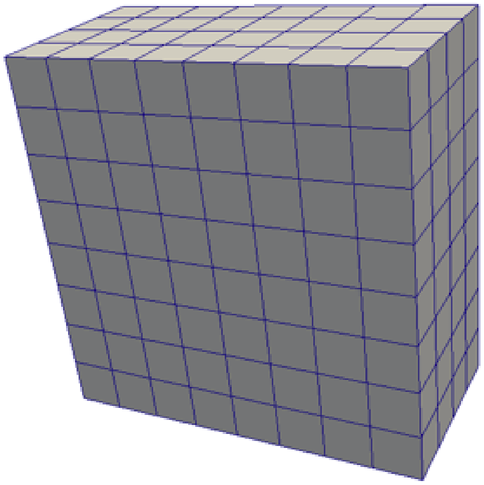


```
0265640 132304 133732 032051 037334 024721 015013 052226 001662
0265660 025537 064663 054606 043244 074076 124153 135216 126614
0265700 144210 056426 044700 042650 165230 137037 003655 006254
0265720 134453 124327 176005 027034 107614 170774 073702 067274
0265740 072451 007735 147620 061064 157435 113057 155356 114603
0265760 107204 102316 171451 046040 120223 001774 030477 046673
0266000 171317 116055 155117 134444 167210 041405 147127 050505
0266020 004137 046472 124015 134360 173550 053517 044635 021135
0266040 070176 047705 113754 175477 105532 076515 177366 056333
0266060 041023 074017 127113 003214 037026 037640 066171 123424
0266100 067701 037406 140000 165341 072410 100032 125455 056646
0266120 006716 071402 055672 132571 105645 170073 050376 072117
0266140 024451 007424 114200 077733 024434 012546 172404 102345
0266160 040223 050170 055164 164634 047154 126525 112514 032315
0266200 016041 176055 042766 025015 176314 017234 110060 014515
0266220 117156 030746 154234 125001 151144 163706 136237 164376
0266240 137055 062276 161755 115466 005322 132567 073216 002655
0266260 171466 126161 117155 065763 016177 014460 112765 055527
0266300 003767 175367 104754 036436 172172 150750 043643 145410
0266320 072074 000007 040627 070652 173011 002151 125132 140214
0266340 060115 014356 015164 067027 120206 070242 033065 131334
0266360 170601 170106 040437 127277 124446 136631 041462 116321
0266400 020243 005602 004146 121574 124651 006634 071331 102070
0266420 157504 160307 166330 074251 024520 114433 167273 030635
0266440 133614 106171 144160 010652 007365 026416 160716 100413
0266460 026630 007210 000630 121224 076033 140764 000737 003276
0266500 114060 042647 104475 110537 066716 104754 075447 112254
0266520 030374 144251 077734 015157 002513 173526 035531 150003
0266540 146207 015135 024446 130101 072457 040764 165513 156412
0266560 166410 067251 156160 106406 136770 030516 064740 022032
0266600 142166 123707 175121 071170 076357 037233 031136 015232
0266620 075074 016744 044055 102230 110063 033350 052765 172463
```

Put simply, the process of visualization is taking raw data and converting it to a form that is viewable and understandable to humans. This allows us to get a better cognitive understanding of our data. Scientific visualization is specifically concerned with the type of data that has a well defined representation in 2D or 3D space. Data that comes from simulation meshes and scanner data is well suited for this type of analysis.

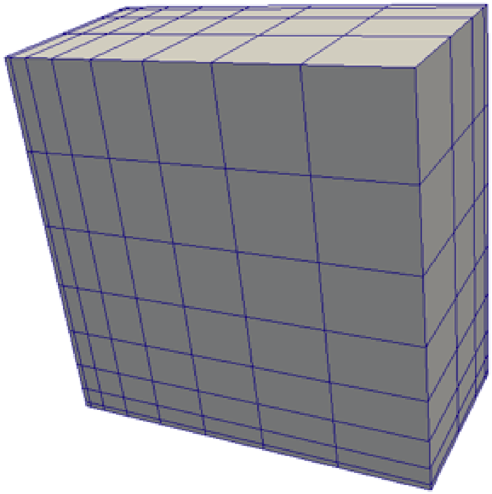
There are three basic steps to visualizing your data: reading, filtering, and rendering. First, your data must be read into **ParaView**. Next, you may apply any number of **filters** that process the data to generate, extract, or derive features from the data. Finally, a viewable image is rendered from the data.

**ParaView** was designed primarily to handle data with spatial representation. Thus the primary **data types** used in **ParaView** are meshes.



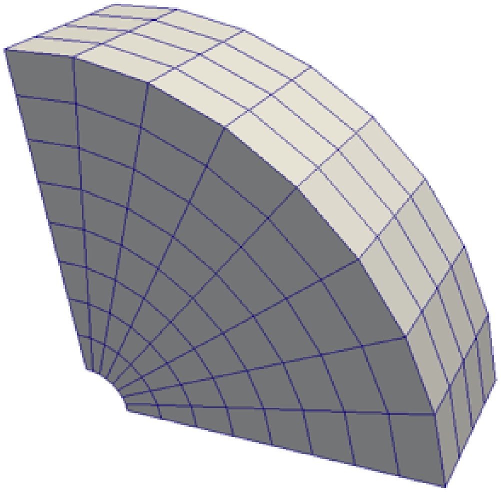
**Uniform Rectilinear (Image Data)**

A uniform rectilinear grid is a one- two- or three- dimensional array of data. The points are orthonormal to each other and are spaced regularly along each direction.



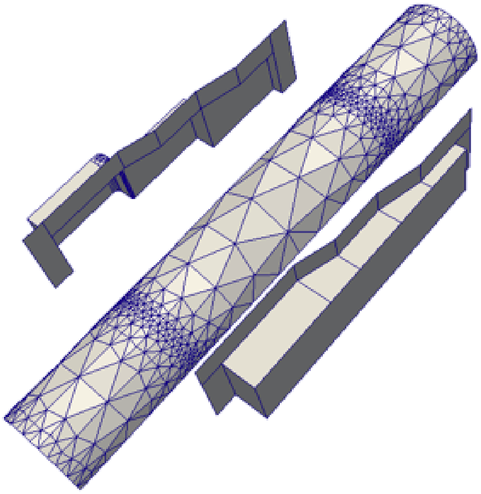
**Non-uniform Rectilinear (Rectilinear Grid)**

Similar to the uniform rectilinear grid except that the spacing between points may vary along each axis.



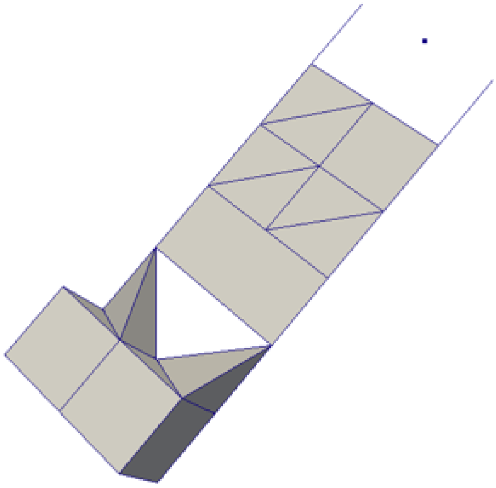
**Curvilinear (Structured Grid)**

Curvilinear grids have the same topology as rectilinear grids. However, each point in a curvilinear grid can be placed at an arbitrary coordinate (provided that it does not result in cells that overlap or self intersect). Curvilinear grids provide the more compact memory footprint and implicit topology of the rectilinear grids, but also allow for much more variation in the shape of the mesh.



**Polygonal (Poly Data)**

Polygonal datasets are composed of points, lines, and 2D polygons. Connections between cells can be arbitrary or nonexistent. Polygonal data represents the basic rendering primitives. Any data must be converted to polygonal data before being rendered (unless volume rendering is employed), although **ParaView** will automatically make this conversion.



**Unstructured Grid**

Unstructured datasets are composed of points, lines, 2D polygons, 3D tetrahedra, and nonlinear cells. They are similar to polygonal data except that they can also represent 3D tetrahedra and nonlinear cells, which cannot be directly rendered.

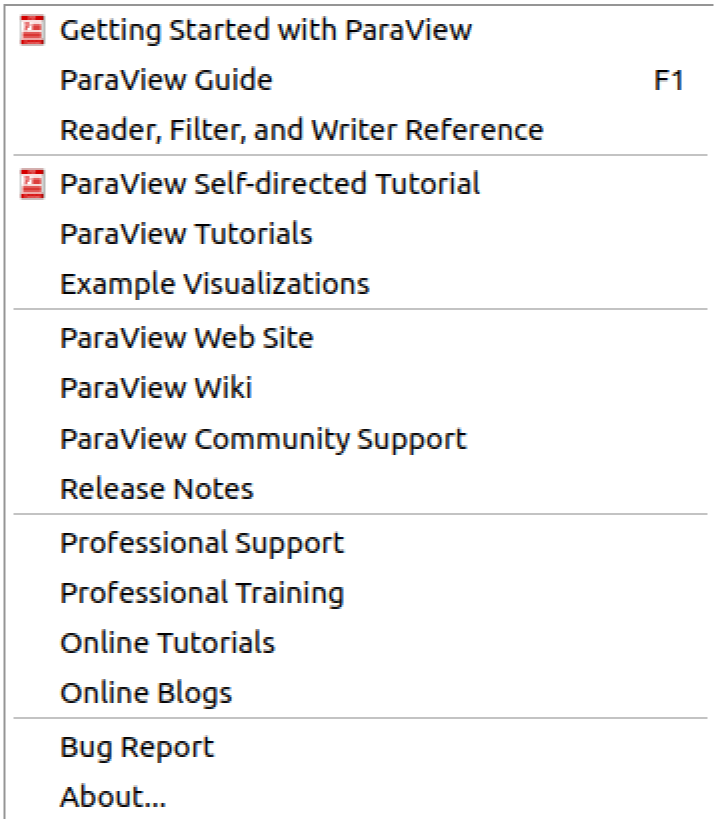
In addition to these basic data types, **ParaView** also supports **multiblock** data. A basic multiblock dataset is created whenever datasets are grouped together or whenever a file containing multiple blocks is read. **ParaView** also has some special data types for representing **Hierarchical Adaptive Mesh Refinement (AMR)**, **Hierarchical Uniform AMR**, **Octree**, **Tabular**, and **Graph** type datasets.

## More Information

There are many places to find more information about **ParaView**. The manual, titled *The ParaView Guide*, is available for purchase as a hard copy or can be downloaded for free from https://docs.paraview.org/.

The **ParaView** web page, https://www.paraview.org/, is also an excellent place to find more information about **ParaView**. From there can find helpful links to the **ParaView** Discourse forum, Wiki pages, and frequently asked questions as well as information about professional support services.

**ParaView**'s menu is a good place to start to find useful information and contains many links to documentation and training materials. The menu has entries that directly bring up the aforementioned *ParaView Guide* and web pages. The menu also provides several training materials including a quick *Getting Started with ParaView* guide, multiple tutorials (including this one), and some example visualizations.



### 4.1.2 Basic Usage

Let us get started using **ParaView**. In order to follow along, you will need your own installation of **ParaView**. If you do not already have **ParaView**, you can download a copy from https://www.paraview.org/Download/. **ParaView** launches like most other applications. On Windows, the launcher is located in the start menu. On Macintosh, open the application bundle that you installed. On Linux, execute `paraview` from a command prompt (you may need to set your path).

The examples in this tutorial rely on some data that is included with the binary distribution of **ParaView** starting with version 5.2. For earlier versions, the tutorial data are available at https://www.paraview.org/Wiki/The_ParaView_Tutorial. You may install this data into any directory that you like, but make sure that you can find that directory easily. Any time the tutorial asks you to load a file it will be from the directory you installed this data in.

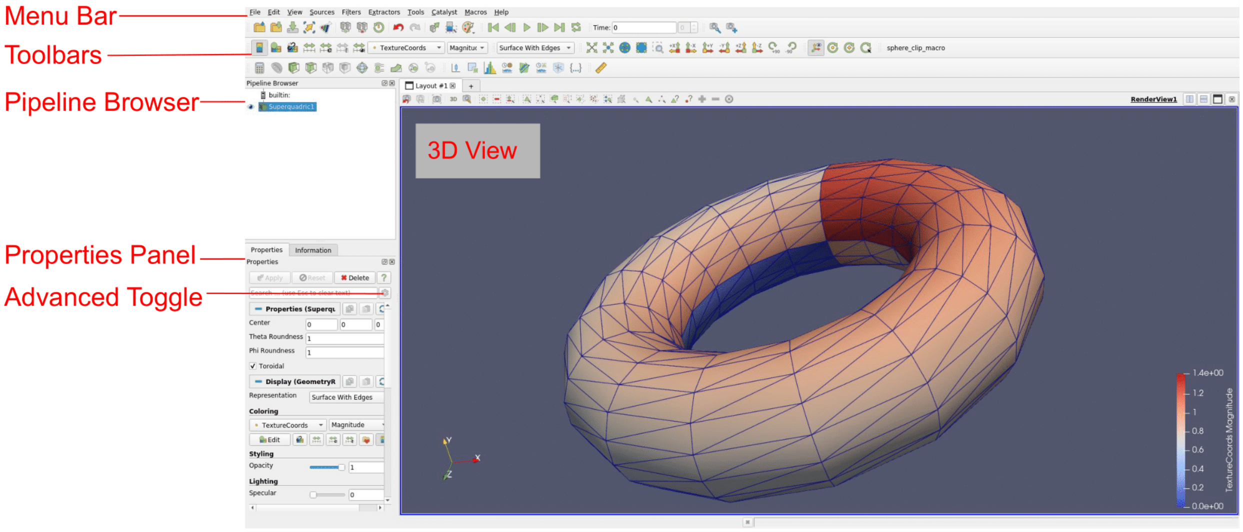
#### User Interface

The **ParaView** GUI conforms to the platform on which it is running, but on all platforms it behaves basically the same. The layout shown here is the default layout given when **ParaView** is first started. The GUI comprises the following components.

**Menu Bar**
As with just about any other program, the menu bar allows you to access the majority of features.

**Toolbars**
The toolbars provide quick access to the most commonly used features within **ParaView**.

**Pipeline Browser**
**ParaView** manages the reading and filtering of data with a pipeline. The pipeline browser allows you to view the



pipeline structure and select pipeline objects. The pipeline browser provides a convenient list of pipeline objects with an indentation style that shows the pipeline structure.

**Properties Panel**

The properties panel allows you to view and change the parameters of the current pipeline object. On the properties panel is an advanced properties toggle  that shows and hides advanced controls. The properties are by default coupled with an **Information** tab that shows a basic summary of the data produced by the pipeline object.

**3D View**

The remainder of the GUI is used to present data so that you may view, interact with, and explore your data. This area is initially populated with a 3D view that will provide a geometric representation of the data.

Note that the GUI layout is highly configurable, so that it is easy to change the look of the window. The toolbars can be moved around and even hidden from view. To toggle the use of a toolbar, use the View → Toolbars submenu. The pipeline browser and properties panel are both **dockable** windows. This means that these components can be moved around in the GUI, torn off as their own floating windows, or hidden altogether. These two windows are important to the operation of **ParaView**, so if you hide them and then need them again, you can get them back with the View menu.

## Sources

There are two ways to get data into **ParaView**: read data from a file or generate data with a **source** object. All sources are located in the Sources menu. Sources can be used to add annotation to a view, but they are also very handy when exploring **ParaView**'s features.

**Exercise 2.1 (Creating a Source)**

Let us start with a simple one. Go to the Sources menu, open the Geometric Shapes submenu, and select Cylinder. Once you select the Cylinder item you will notice that an item named Cylinder1 is added to and selected in the pipeline browser. You will also notice that the properties panel is filled with the properties for the cylinder source. Click the Apply button Apply to accept the default parameters.

Once you click Apply, the cylinder object will be displayed in the 3D view window on the right.

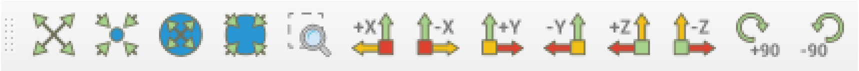
### Basic 3D Interaction

Now that we have created our first simple visualization, we want to interact with it. There are many ways to interact with a visualization in **ParaView**. We start by exploring the data in the 3D view.

**Exercise 2.2 (Interacting with a 3D View)**

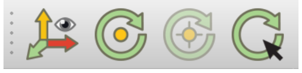
This exercise is a continuation of Exercise 2.1. You will need to finish that exercise before beginning this one.

You can manipulate the cylinder in the 3D view by dragging the mouse over the 3D view. Experiment with dragging different mouse buttons—left, middle, and right—to perform different rotate, pan, and zoom operations. Also try using the buttons in conjunction with the shift and ctrl modifier keys. Additionally you can hold down the x, y, or z key while you drag the mouse to constrain movement along the x, y, or z axis.



**ParaView** contains a couple of toolbars to help with camera manipulations. The first toolbar, the Camera Controls toolbar, shown here, provides quick access to particular camera views. The leftmost  button performs a **reset camera** such that it maintains the same view direction but repositions the camera such that the entire object can be seen. The second button  performs a **zoom to data**. It behaves very much like reset camera except that instead of positioning the camera to see all data, the camera is placed to look specifically at the data currently selected in the pipeline browser. The third button  performs a **reset camera closest** such that it maximizes the occupation, in the screen, of the whole scene bounding box. The fourth button  performs a **zoom closest to data**. It behaves very much like reset camera closest except that instead of positioning the camera to see all data, the camera is placed to look specifically at the data currently selected in the pipeline browser. You currently only have one object in the pipeline browser, so right now reset camera and zoom to data, and reset camera closest and zoom closest to data will perform the same operation.

The next button in the camera controls toolbar  allows you to select a rectangular region of the screen to zoom to (a **rubber-band zoom**). The following six buttons, starting with , reposition the camera to view the scene straight down one of the global coordinate's axes in either the positive or negative direction. The rightmost two buttons  rotate the view either clockwise or counterclockwise. Try playing with these controls now.



The second toolbar controls the location of the center of rotation and the visibility of the orientation axes. The rightmost button  allows you to pick the **center of rotation**. Try clicking that button then clicking somewhere on the cylinder. If you then drag the left button in the 3D view, you will notice that the cylinder now rotates around this new point. The next button to the left  replaces the center of rotation to the center of the object. The next button to the left  shows or hides axes drawn at the center of rotation. (You probably will not notice the effects when the center of rotation is at the center of the cylinder because the axes will be hidden by the cylinder. Use the pick center of rotation  again and you should be able to see the effects.) The final leftmost button  toggles showing the **orientation axes**, the always-viewable axes in the lower left corner of the 3D view.

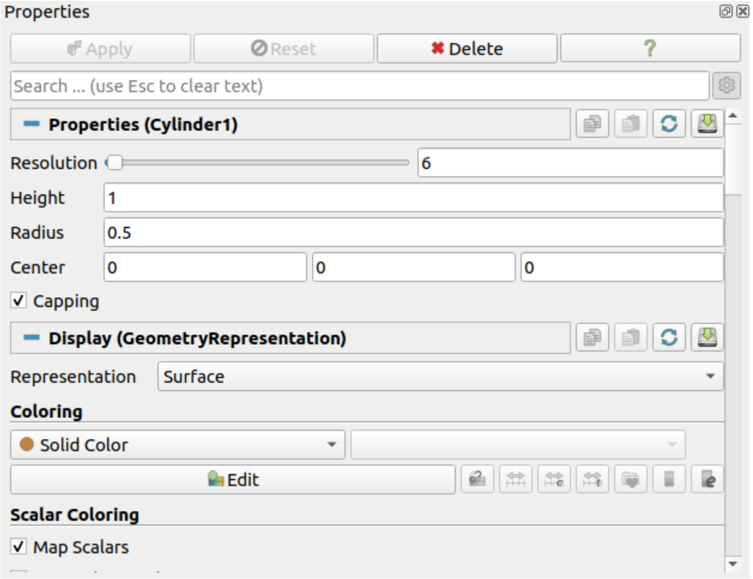
### Modifying Visualization Parameters

Although interactive 3D controls are a vital part of visualization, an equally important ability is to modify the parameters of the data processing and display. **ParaView** contains many GUI components for modifying visualization parameters, which we will begin to explore in the next exercise.

**Exercise 2.3 (Modifying Visualization Parameters)**

This exercise is a continuation of Exercise 2.2. You will need to finish that exercise before beginning this one.

You surely noticed that **ParaView** creates not a real cylinder but rather an approximation of a cylinder using polygonal **facets**. The default parameters for the cylinder source provide a very coarse approximation of only six facets. (In fact, this object looks more like a prism than a cylinder.) If we want a better representation of a cylinder, we can create one by increasing the Resolution parameter. The Resolution parameters, like all other parameters for the cylinder object, are located in the properties panel under the Apply button when the cylinder object is selected in the pipeline browser.



Resolution 6

Height 1

Radius 0.5

Center 0 0 0

☑ Capping

Representation Surface

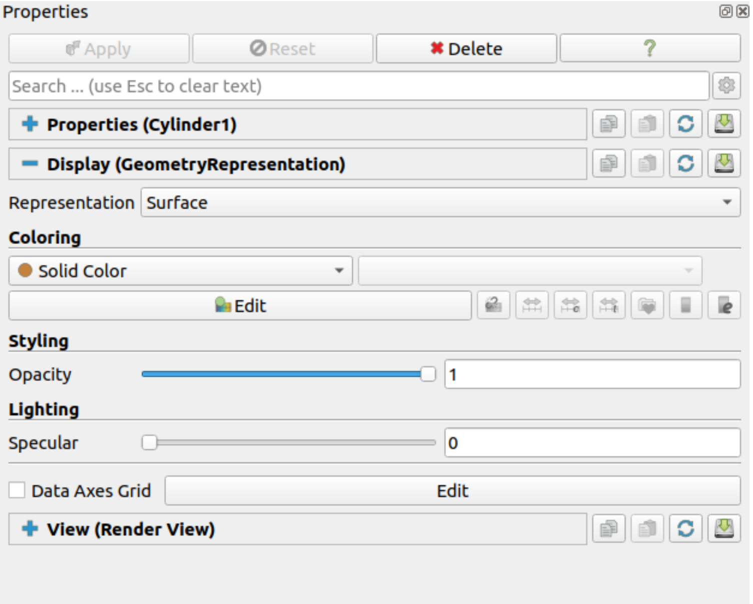
● Solid Color

☑ Map Scalars

Using either the slider or text edit, increase the resolution to 50 or more. Notice that the Apply button Apply become colored again. This is because changes you make to the object properties are not immediately enacted. The highlighted button is a reminder that the parameters of one or more pipeline objects are "out of sync" with the data that you are viewing. Hitting the Apply button will accept these changes whereas hitting the Reset button Reset will revert the options back to the last time they were applied. Hit the Apply button now. The resolution is changed so that it is virtually indistinguishable from a true cylinder.

If your work has you creating cylinder sources frequently and you find yourself modifying Resolution or other parameters to some value other than the default each time, you can save your preferred default parameters by hitting the save parameters button . Once you hit the button, **ParaView** will remember your preferences for objects of that type and use those parameters when you create future objects. Conversely, if you have changed the parameters and want to reset them to the "factory default," you can click the restore parameters button . As we will see in future exercises, we can have multiple visualization objects open at once. To copy parameters from one object to another, use the copy  and paste  parameters buttons.

If you scroll down the properties panel, you will notice a set of Display properties. Try these options now by clicking on the Edit  button under Coloring to select a new color for the cylinder. (This button is also replicated in the toolbar.) You may notice that you do not need to hit Apply for display properties.



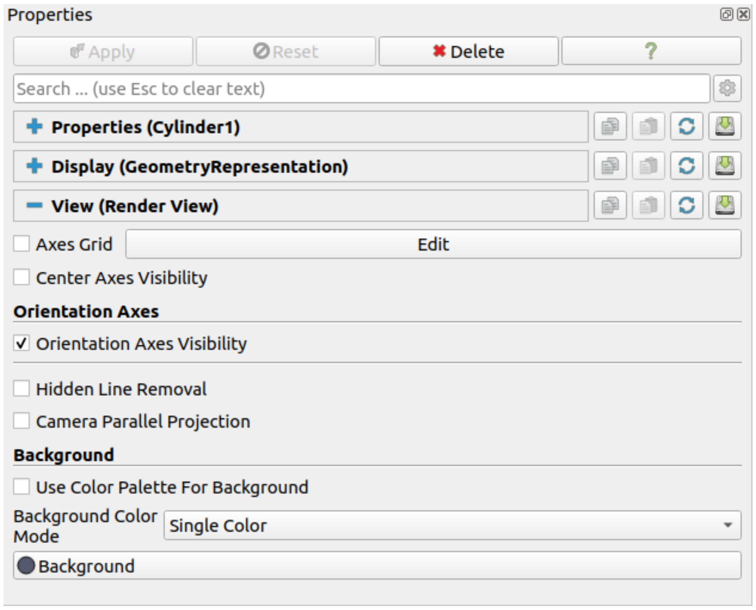
Representation Surface

● Solid Color

Opacity 1

Specular 0

□ Data Axes Grid



□ Axes Grid

□ Center Axes Visibility

☑ Orientation Axes Visibility

□ Hidden Line Removal

□ Camera Parallel Projection

□ Use Color Palette For Background

Background Color Mode Single Color

● Background

If you scroll down further yet to the bottom of the properties panel, you will notice a set of View properties. Use the view properties to turn on the Axes Grid.

By default many of the lesser used display properties are hidden. The **advanced properties** toggle  can be used to show or hide these extra parameters. There is also a search box at the top of the properties panel that can be used to quickly find a property. Try typing specular into this search box now. Under the display properties you should see an option named Specular. This controls the intensity of the specular highlight seen on shiny objects. Set this parameter to 1 to make the cylinder shiny.

Most objects have similar display and view properties. Here are some other common tricks you can do with most objects using parameters available in the properties panel and that you can try now.

- Show 3D axes at the borders of the object containing rulers showing the physical distance in each direction by clicking the Axes Grid checkbox under the View options.
- Make objects transparent by changing their Opacity parameter. An opacity parameter of 1 is completely opaque, a parameter of 0 is completely invisible, and values in between are varying degrees of see through.

From the previous exercises you have noted that some visualization operations (but not all) require pressing the Apply button before seeing the effect of the change. This apply button serves an important function. When visualizing large data, which **ParaView** is designed to do, simple actions like creating an object or changing a parameter can take a long time. Thus this two phased approach allows you to establish all the visualization parameters for a particular action before enacting an operation (by hitting Apply). However, when dealing with small data, operations complete near instantaneously, so the process of hitting Apply becomes redundant. In these cases, you may wish to turn on auto apply.

**Exercise 2.4 (Toggle Auto Apply)**

Find the auto apply  button in the top toolbar. This is a toggle button. Click it now and note that it stays depressed.

While auto apply is on, it is no longer necessary to hit the Apply button. Try changing the Resolution of the cylinder source as you did in Exercise 2.3 (or create a new source if your cylinder is no longer available). Note that as soon as you make the change, the visualization is updated.

You can turn off auto apply by clicking the toolbar button  again. You can complete the rest of these exercises with auto apply either on or off. The instructions will assume that auto apply is off and prompt you to hit the Apply button. If you have auto apply on, ignore these instructions.

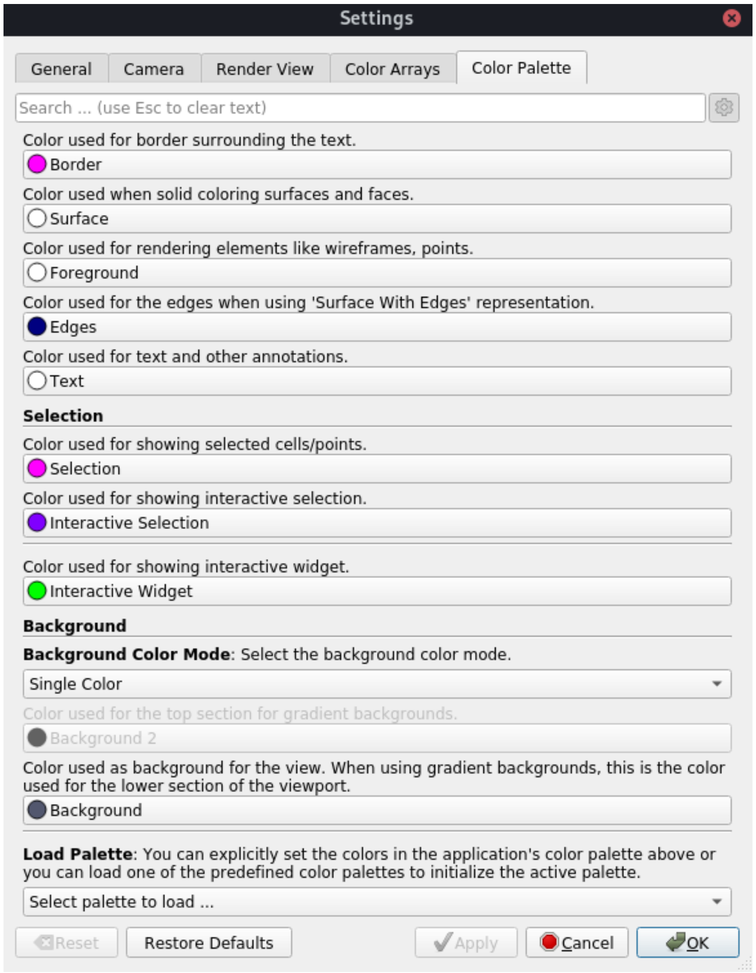
As you would expect, **ParaView** allows you to control the color of many elements. In many cases the changing the color of one element necessitates the changing of another. For example, if changing the background to a light color, it is important to change text on that background to a dark color. Otherwise the text will be unreadable. To help manage sets of interdependent colors, **ParaView** supports the idea of color palettes. You can easily change the view's color palette using the load color palette button  in the toolbar.

**Exercise 2.5 (Changing the Color Palette)**

Make sure the orientation axes is shown in the lower left corner. This is toggled with the  button as described in Exercise 2.2. Note that the orientation axis has the labels "X," "Y," and "Z."

Find the load color palette button  in the top toolbar. Click that button to get a pull down menu of available palettes. Experiment with different palettes. Observe that both the background color and the labels in the orientation axes change.

The colors used for the color palettes are part of **ParaView**'s settings. You can see and set all of these colors in the Edit → Settings (**ParaView** → Preferences on the Mac) under the Color Palette tab. You can also get to the color palette settings by clicking on the color palette button  and selecting the Edit Current Palette… button.



Color used for border surrounding the text.

● Border

Color used when solid coloring surfaces and faces.

○ Surface

Color used for rendering elements like wireframes, points.

○ Foreground

Color used for the edges when using 'Surface With Edges' representation.

● Edges

Color used for text and other annotations.

○ Text

**Selection**

Color used for showing selected cells/points.

● Selection

Color used for showing interactive selection.

● Interactive Selection

Color used for showing interactive widget.

● Interactive Widget

**Background**

**Background Color Mode**: Select the background color mode.

Single Color

Color used for the top section for gradient backgrounds.

● Background 2

Color used as background for the view. When using gradient backgrounds, this is the color used for the lower section of the viewport.

● Background

**Load Palette**: You can explicitly set the colors in the application's color palette above or you can load one of the predefined color palettes to initialize the active palette.

Select palette to load ...

Now is a good time to note the undo and redo buttons in the toolbar. Visualizing your data is often an exploratory process, and it is often helpful to revert back to a previous state. You can even undo back to the point before your data were created and redo again.

**Exercise 2.6 (Undo and Redo)**

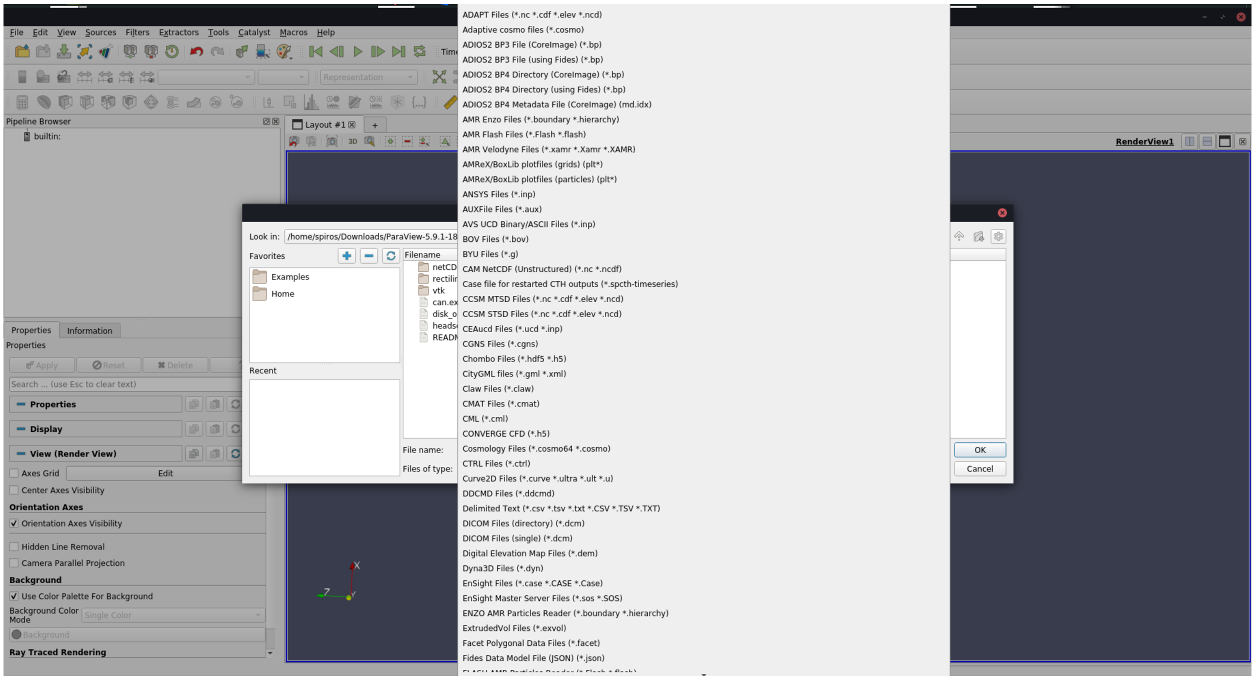
Experiment with the undo and redo buttons. If you have not done so, create and modify a pipeline object like what is done in Exercise 2.1. Watch how parameter changes can be reverted and restored. Also notice how whole pipeline objects can be destroyed and recreated.

There are also undo camera and redo camera buttons at the view's toolbar. These allow you to go back and forth between camera angles that you have made so that you do not have to worry about errant mouse movements ruining that perfect view. Move the camera around and then use these buttons to revert and restore the camera angle.

We are done with the cylinder source now. We can delete the pipeline object by making sure the cylinder is selected in the pipeline browser and hitting delete ✖ Delete in the properties panel.

## Loading Data

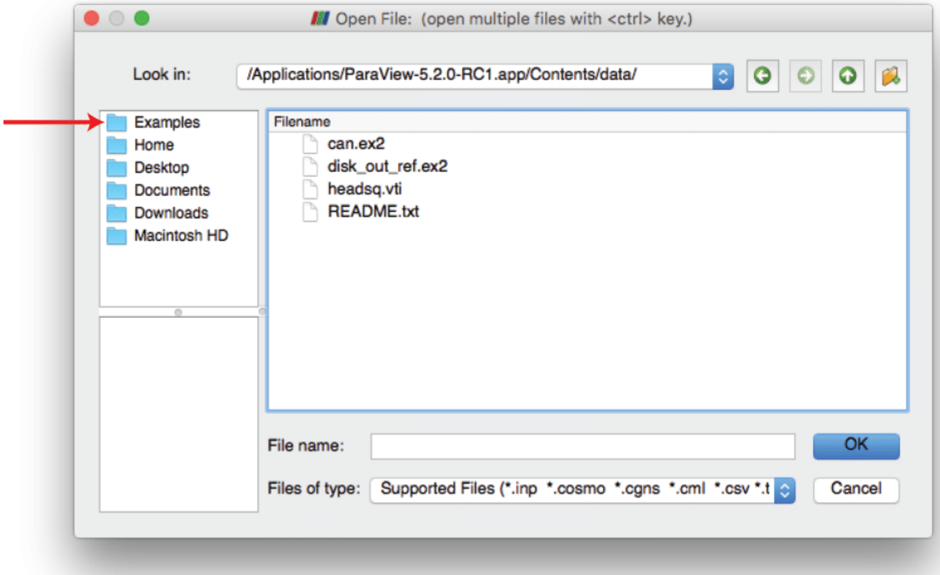
Now that we have had some practice using the **ParaView** GUI, let us load in some real data. As you would expect, the Open command is the first one off of the File menu, and there is also toolbar button for opening a file. **ParaView** currently supports about 220 distinct file formats, and the list grows as more types get added. To see the current list of supported files, invoke the Open command and look at the list of files in the Files of type chooser box.


File Edit View Sources Filters Extractors Tools Catalyst Macros Help
Pipeline Browser
builtin:
Layout #1
RenderView1
Properties
Information
Apply
Reset
Delete
Search ... (use Esc to clear text)
Properties
Display
View (Render View)
Axes Grid
Edit
Center Axes Visibility
Orientation Axes
Orientation Axes Visibility
Hidden Line Removal
Camera Parallel Projection
Background
Use Color Palette For Background
Background Color Mode
Single Color
Background
Ray Traced Rendering
Look in:
/home/spiros/Downloads/ParaView-5.9.1-18
Favorites
Examples
Home
Recent
Filename
netCD
rectilir
vtk
can.ex
disk_o
heads
READM
File name:
Files of type:
OK
Cancel
ADAPT Files (*.nc *.cdf *.elev *.ncd)
Adaptive cosmo files (*.cosmo)
ADIOS2 BP3 File (CoreImage) (*.bp)
ADIOS2 BP3 File (using Fides) (*.bp)
ADIOS2 BP4 Directory (CoreImage) (*.bp)
ADIOS2 BP4 Directory (using Fides) (*.bp)
ADIOS2 BP4 Metadata File (CoreImage) (md.idx)
AMR Enzo Files (*.boundary *.hierarchy)
AMR Flash Files (*.Flash *.flash)
AMR Velodyne Files (*.xamr *.Xamr *.XAMR)
AMReX/BoxLib plotfiles (grids) (plt*)
AMReX/BoxLib plotfiles (particles) (plt*)
ANSYS Files (*.inp)
AUXFile Files (*.aux)
AVS UCD Binary/ASCII Files (*.inp)
BOV Files (*.bov)
BYU Files (*.g)
CAM NetCDF (Unstructured) (*.nc *.ncdf)
Case file for restarted CTH outputs (*.spcth-timeseries)
CCSM MTSD Files (*.nc *.cdf *.elev *.ncd)
CCSM STSD Files (*.nc *.cdf *.elev *.ncd)
CEAucd Files (*.ucd *.inp)
CGNS Files (*.cgns)
Chombo Files (*.hdf5 *.h5)
CityGML files (*.gml *.xml)
Claw Files (*.claw)
CMAT Files (*.cmat)
CML (*.cml)
CONVERGE CFD (*.h5)
Cosmology Files (*.cosmo64 *.cosmo)
CTRL Files (*.ctrl)
Curve2D Files (*.curve *.ultra *.ult *.u)
DDCMD Files (*.ddcmd)
Delimited Text (*.csv *.tsv *.txt *.CSV *.TSV *.TXT)
DICOM Files (directory) (*.dcm)
DICOM Files (single) (*.dcm)
Digital Elevation Map Files (*.dem)
Dyna3D Files (*.dyn)
EnSight Files (*.case *.CASE *.Case)
EnSight Master Server Files (*.sos *.SOS)
ENZO AMR Particles Reader (*.boundary *.hierarchy)
ExtrudedVol Files (*.exvol)
Facet Polygonal Data Files (*.facet)
Fides Data Model File (JSON) (*.json)
Z
X

**ParaView**'s modular design allows for easy integration of new VTK readers into **ParaView**. Thus, check back often for new file formats. If you are looking for a file reader that does not seem to be included with **ParaView**, check in with the **ParaView** mailing list (paraview@paraview.org). There are many file readers included with VTK but not exposed within **ParaView** that could easily be added. There are also many readers created that can plug into the VTK framework but have not been committed back to VTK; someone may have a reader readily available that you can use.

**Exercise 2.7 (Opening a File)**

Let us open our first file now. Click the Open toolbar button (or menu item) . Note that **ParaView** uses a custom file browser, which provides several convenience features. On the left side of the file browser dialog are a pair of boxes containing lists of directories, which provide quick access to files in common directories. The top left list contains a list of common data directories on your system. The bottom left list, which is initially empty, is filled with directories from which you have recently loaded files. Double click on the Examples directory listed in the top left box. This is a directory created by the **ParaView** installation that contains the files we use in this tutorial.


Open File: (open multiple files with <ctrl> key.)

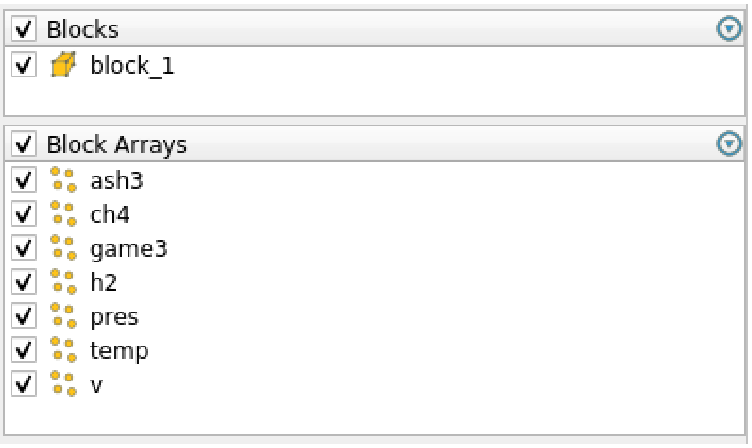
Look in: /Applications/ParaView-5.2.0-RC1.app/Contents/data/

Filename
- can.ex2
- disk_out_ref.ex2
- headsq.vti
- README.txt

File name:

Files of type: Supported Files (*.inp *.cosmo *.cgns *.cml *.csv *.t

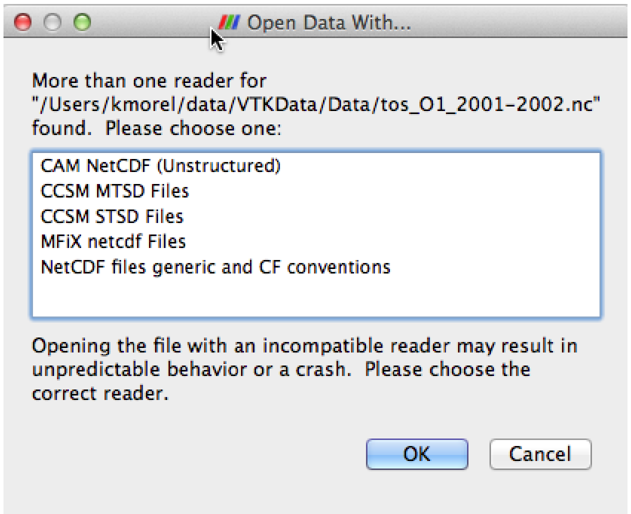
Open the file disk_out_ref.ex2. Note that opening a file is a two step process, so you do not see any data yet. Instead, you see that the properties panel is populated with several options about how we want to read the data.


☑ Blocks
- ☑ block_1

☑ Block Arrays
- ☑ ash3
- ☑ ch4
- ☑ game3
- ☑ h2
- ☑ pres
- ☑ temp
- ☑ v

Click the checkbox in the header of the block arrays list to turn on the loading of all the block arrays. This is a small dataset, so we do not have to worry about loading too much into memory. Once all of the variables are selected, click Apply to load all of the data. When the data are loaded you will see that the geometry looks like a cylinder with a hollowed out portion in one end. These data are the output of a simulation for the flow of air around a heated and spinning disk. The mesh you are seeing is the air around the disk (with the cylinder shape being the boundary of the simulation). The hollow area in the middle is where the heated disk would be were it meshed for the simulation.

Most of the time **ParaView** will be able to determine the appropriate method to read your file based on the file extension and underlying data, as was the case in Exercise 2.7. However, with so many file formats supported by **ParaView** there are some files that cannot be fully determined. In this case, **ParaView** will present a dialog box asking what type of file is being loaded. The following image is an example from opening a netCDF file, which is a generic file format for which **ParaView** has many readers for different conventions.



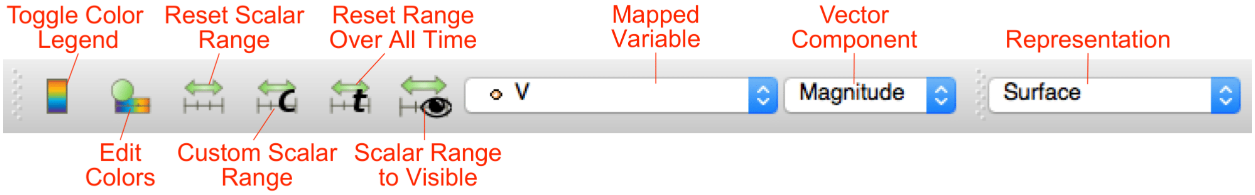
Open Data With...

More than one reader for "/Users/kmorel/data/VTKData/Data/tos_O1_2001-2002.nc" found. Please choose one:

CAM NetCDF (Unstructured)
CCSM MTSD Files
CCSM STSD Files
MFiX netcdf Files
NetCDF files generic and CF conventions

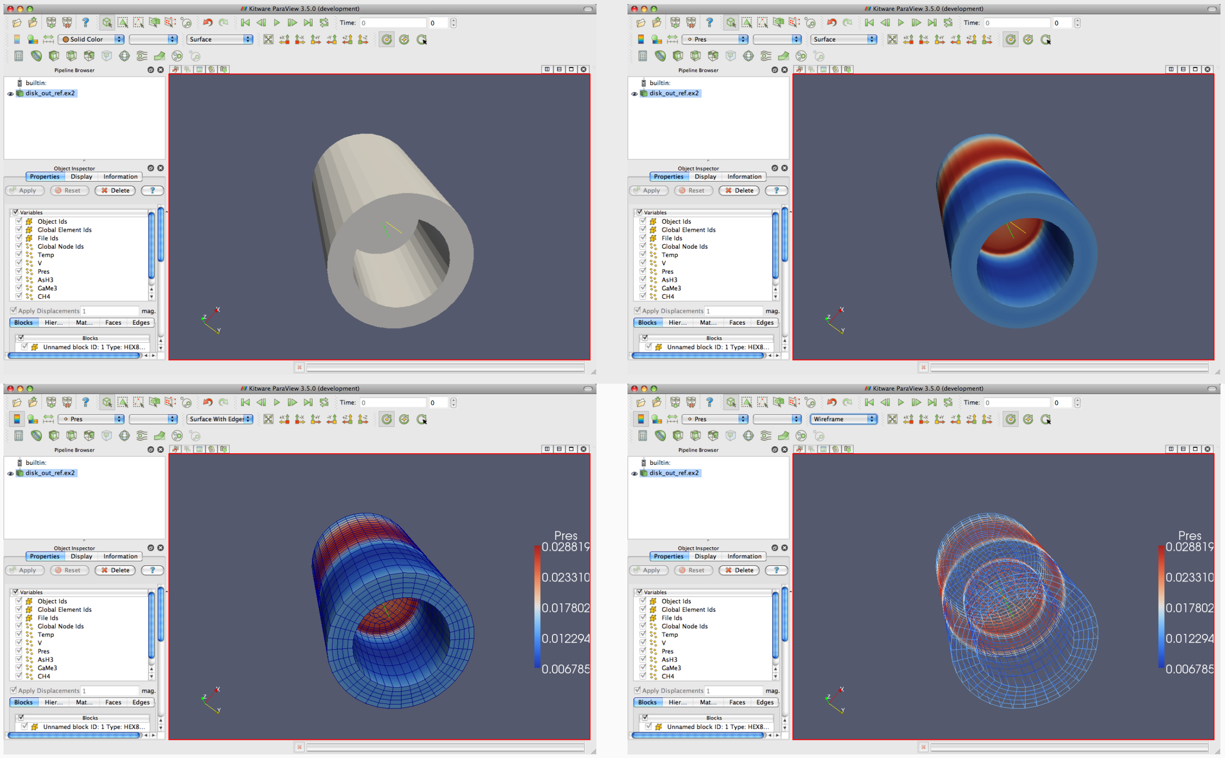
Opening the file with an incompatible reader may result in unpredictable behavior or a crash. Please choose the correct reader.

Before we continue on to filtering the data, let us take a quick look at some of the ways to represent the data. The most common parameters for representing data are located in a pair of toolbars. (They can also be found in the Display group of the properties panel.)



**Exercise 2.8 (Representation and Field Coloring)**

Play with the data representation a bit. Make sure disk_out_ref.ex2 is selected in the pipeline browser. (If you do not have the data loaded, repeat Exercise 2.7.) Use the variable chooser to color the surface by the pres variable. To see the structure of the mesh, change the representation to Surface With Edges. You can view both the cell structure and the interior of the mesh with the Wireframe representation.



## Filters

We have now successfully read in some data and gleaned some information about it. We can see the basic structure of the mesh and map some data onto the surface of the mesh. However, as we will soon see, there are many interesting features about these data that we cannot determine by simply looking at the surface of these data. There are many variables associated with the mesh of different types (scalars and vectors). And remember that the mesh is a solid model. Most of the interesting information is on the inside.

We can discover much more about our data by applying **filters**. Filters are functional units that process the data to generate, extract, or derive features from the data. Filters are attached to readers, sources, or other filters to modify its data in some way. These filter connections form a **visualization pipeline**. There is a great amount of filters available in **ParaView**. Here are the most common, which are all available by clicking on the respective icon in the filters toolbar.

**Calculator**
Evaluates a user=defined expression on a per=point or per-cell basis.

**Contour**
Extracts the points, curves, or surfaces where a scalar field is equal to a user-defined value. This surface is often also called an **isosurface**.

**Clip**
Intersects the geometry with a half space. The effect is to remove all the geometry on one side of a user-defined plane.

**Slice**
Intersects the geometry with a plane. The effect is similar to clipping except that all that remains is the geometry where the plane is located.

**Threshold**
Extracts cells that lie within a specified range of a scalar field.

**Extract Subset**
Extracts a subset of a grid by defining either a volume of interest or a sampling rate.

**Glyph**
Places a **glyph**, a simple shape, on each point in a mesh. The glyphs may be oriented by a vector and scaled by a vector or scalar.

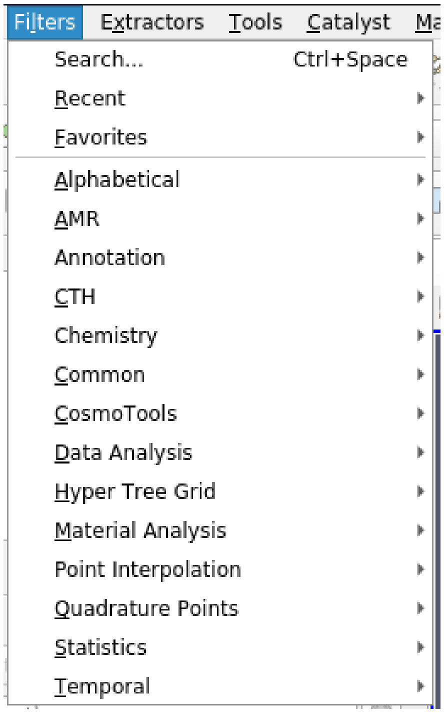
**Stream Tracer**
Seeds a vector field with points and then traces those seed points through the (steady state) vector field.

**Warp (vector)**
Displaces each point in a mesh by a given vector field.

**Group Datasets**
Combines the output of several pipeline objects into a single multi block dataset.

**Extract Level**
Extract one or more items from a multiblock dataset.

These eleven filters are a small sampling of what is available in **ParaView**. In the Filters menu are a great many more filters that you can use to process your data. **ParaView** currently exposes more than one hundred filters, so to make them easier to find the Filters menu is organized into submenus.



These submenus are organized as follows.

**Recent**
The list of most recently used filters sorted with the most recently used filters on top.

**Favorites**
The list includes your favorites filters.

**Alphabetical**
An alphabetical listing of all the filters available. If you are not sure where to find a particular filter, this list is

guaranteed to have it. There are also many filters that are not listed anywhere but in this list.

**AMR**
A set of filters designed specifically for data in an adaptive mesh refinement (AMR) structure.

**Annotation**
Filters that add annotation (such as text information) to the visualization.

**CTH**
Filters used to process results from a CTH simulation.

**Chemistry**
This contains filters for chemistry related datasets.

**Common**
The most common filters. This is the same list of filters available in the filters toolbar and listed previously.

**CosmoTools**
This contains filters developed at LANL for cosmology research.

**Data Analysis**
The filters designed to retrieve quantitative values from the data. These filters compute data on the mesh, extract elements from the mesh, or plot data.

**Hyper Tree Grid**
This contains filters for hyper-tree grid datasets.

**Material Analysis**
Filters for processing data from volume fractions of materials.

**Point Interpolation**
Filters that take an unstructured collection of points in space without cells connecting them and estimate the field interpolation between them.

**Quadrature Points**
Filters to support simulation data given as integration points that can be used for numerical integration with Gaussian quadrature.

**Statistics**
This contains filters that provide descriptive statistics of data, primarily in tabular form.

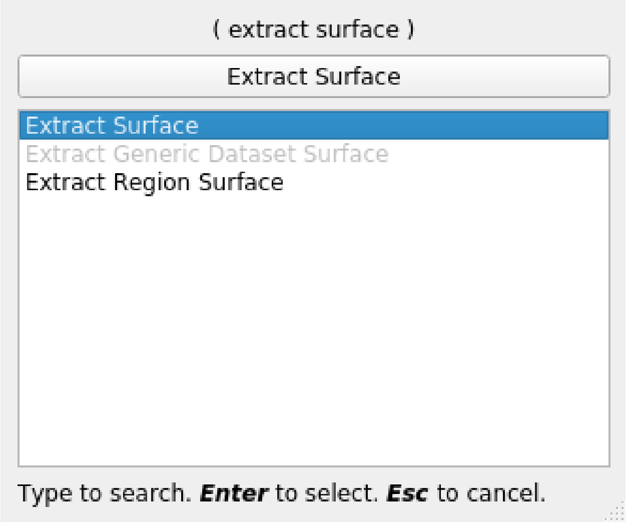
**Temporal**
Filters that analyze or modify data that changes over time. All filters can work on data that changes over time because they are executed on each time snapshot. However, filters in this category will retrieve the available time extents and examine how data changes over time.

Searching through these lists of filters, particularly the full alphabetical list, can be cumbersome. To speed up the selection of filters, you should use the **quick launch** dialog. Pressing the ctrl and space keys together on Windows or Linux or the alt and space keys together on Macintosh, **ParaView** brings up a small, lightweight dialog box like the one shown here.

Type in words or word fragments that are contained in the filter name, and the box will list only those sources and filters that match the terms. Hit enter to add the object to the pipeline browser. Press Esc a couple of times to cancel the dialog.

You have probably noticed that some of the filters are grayed out. Many filters only work on a specific types of data and therefore cannot always be used. **ParaView** disables these filters from the menu and toolbars to indicate (and enforce) that you cannot use these filters.

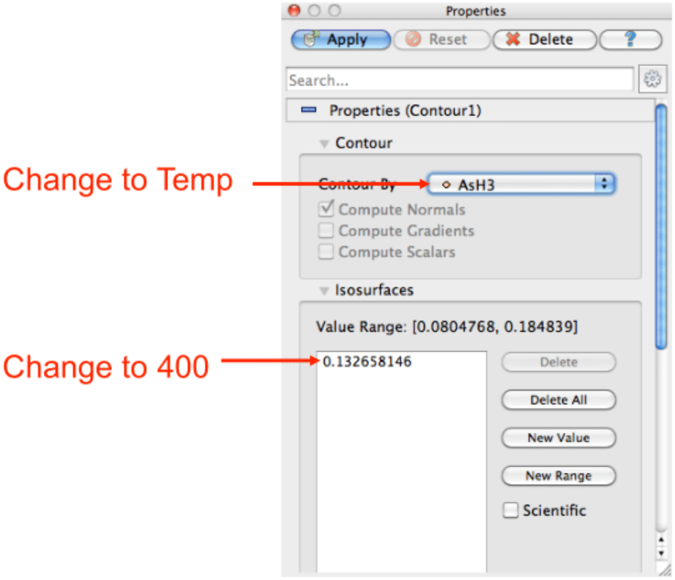
Throughout this tutorial we will explore many filters. However, we cannot explore all the filters in this forum. Consult the Filters Menu chapter of **ParaView**'s on-line or built-in help for more information on each filter.



Type to search. ***Enter*** to select. ***Esc*** to cancel.

**Exercise 2.9 (Apply a Filter)**

Let us apply our first filter. If you do not have the disk_out_ref.ex2 data loaded, do so now (Exercise 2.7). Make sure that disk_out_ref.ex2 is selected in the pipeline browser and then select the contour filter  from the filter toolbar or Filters menu. Notice that a new item is added to the pipeline filter underneath the reader and that the properties panel updates to the parameters of the new filter. As with reading a file, applying a filter is a two step process (unless auto apply is enabled). After creating the filter you get a chance to modify the parameters (which you will almost always do) before applying the filter.



Contour By: AsH3

☑ Compute Normals
□ Compute Gradients
□ Compute Scalars

Value Range: [0.0804768, 0.184839]

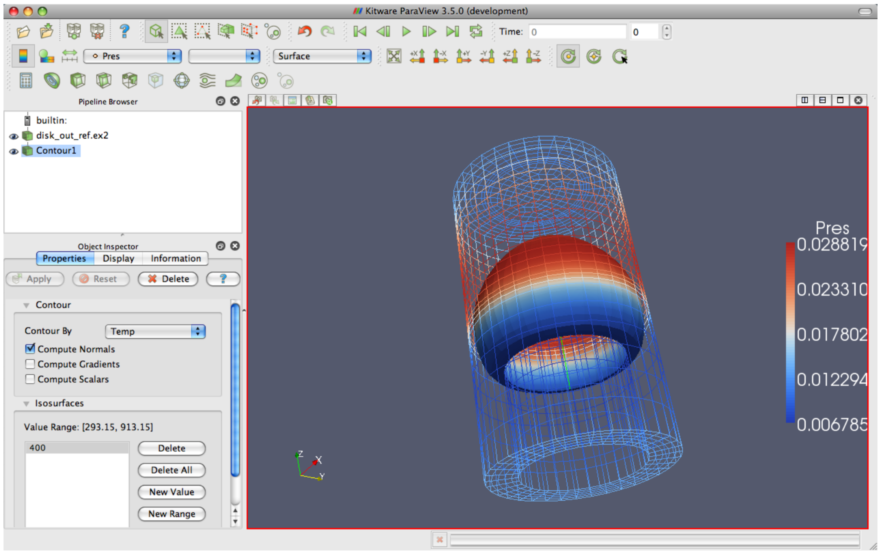
0.132658146

Change to Temp

Change to 400

We will use the contour filter to create an isosurface where the temperature is equal to 400 K. First, change the Contour By parameter to the temp variable. Then, change the isosurface value to 400. Finally, hit Apply. You will see the isosurface appear inside of the volume. If disk_out_ref.ex2 was still colored by pressure from Exercise 2.8, then the surface is colored by pressure to match.

In the preceding exercise, we applied a filter that processed the data and gave us the results we needed. For most common operations, a single filter operation is sufficient to get the information we need. However, filters are of the



Contour By Temp
☑ Compute Normals
□ Compute Gradients
□ Compute Scalars
Value Range: [293.15, 913.15]
400

same class as readers. That is, the general operations we apply to readers can also be applied to filters. Thus, you can apply one filter to the data that is generated by another filter. These readers and filters connected together form what we call a **visualization pipeline**. The ability to form visualization pipelines provides a powerful mechanism for customizing the visualization to your needs.

Let us play with some more filters. Rather than show the mesh surface in wireframe, which often interferes with the view of what is inside it, we will replace it with a cutaway of the surface. We need two filters to perform this task. The first filter will extract the surface, and the second filter will cut some away.

**Exercise 2.10 (Creating a Visualization Pipeline)**

The images and some of the discussion in this exercise assume you are starting with the state right after finishing Exercise 2.9. Finish that exercise before beginning this one.

Start by adding a filter that will extract the surfaces. We do that with the following steps.

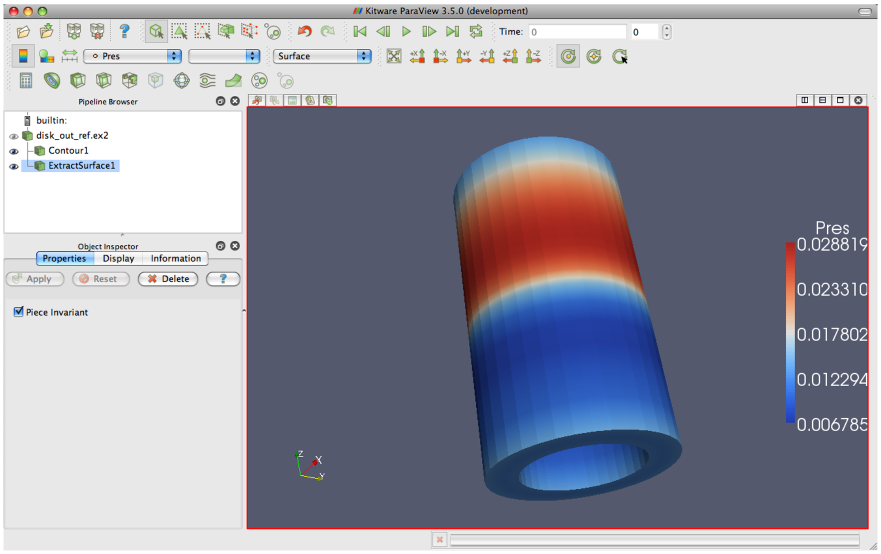
1. Select disk_out_ref.ex2 in the pipeline browser.
2. From the menu bar, select Filters → Alphabetical → Extract Surface. Or bring up the quick launch (ctrl+space Win/Linux, alt+space Mac), type extract surface, and select that filter.
3. Hit the Apply button.

When you apply the Extract Surface filter, you will once again see the surface of the mesh. Although it looks like the original mesh, it is different in that this mesh is hollow whereas the original mesh was solid throughout.

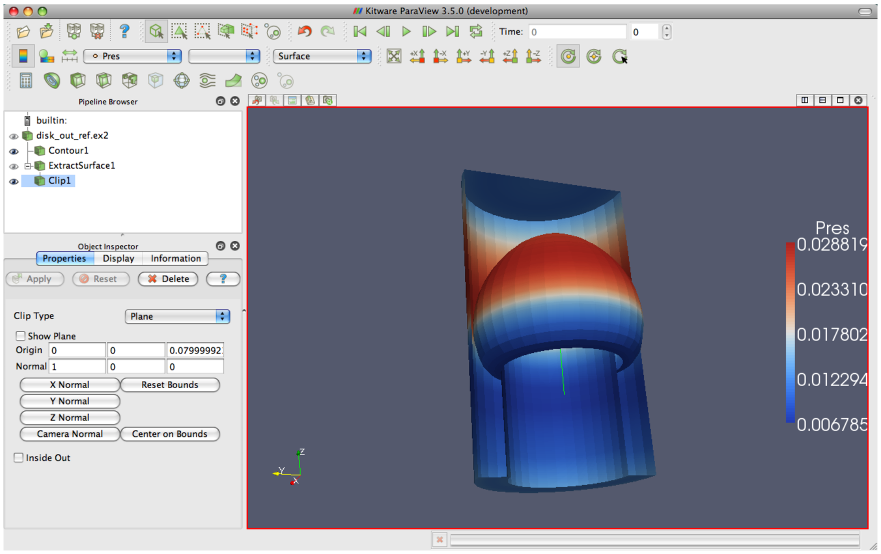
If you were showing the results of the contour filter, you cannot see the contour anymore, but do not worry. It is still in there hidden by the surface. If you are showing the contour but you did not see any effect after applying the filter, you may have forgotten step one and applied the filter to the wrong object. If the ExtractSurface1 object is not connected directly to the disk_out_ref.ex2, then this is what went wrong. If so, you can delete the filter and try again.

Now we will cut away the external surface to expose the internal structure and isosurface underneath (if you have one).

4. Verify that ExtractSurface1 is selected in the pipeline browser.
5. Create a clip filter  from the toolbar or Filters menu.
6. Uncheck the Show Plane checkbox ☑ Show Plane in the properties panel.
7. Click the Apply button.



☑ Piece Invariant



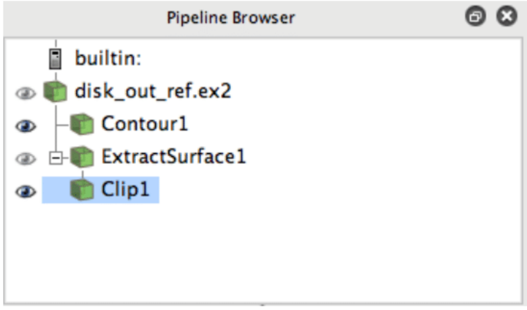
Clip Type: Plane

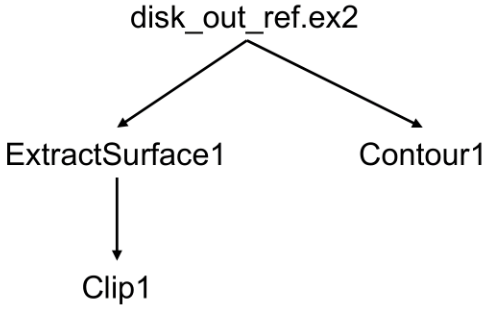
□ Show Plane

| | | | |
|---|---|---|---|
| Origin | 0 | 0 | 0.07999992 |
| Normal | 1 | 0 | 0 |

□ Inside Out

If you have a contour, you should now see the isosurface contour within a cutaway of the mesh surface. You will probably have to rotate the mesh to see the contour clearly.





---

Now that we have added several filters to our pipeline, let us take a look at the layout of these filters in the pipeline browser. The pipeline browser provides a convenient list of pipeline objects that we have created. This makes it easy to select pipeline objects and change their **visibility** by clicking on the eyeball icons  next to them. But also notice the indentation of the entries in the list and the connecting lines toward the left. These features reveal the **connectivity** of the pipeline. It shows the same information as the traditional graph layout on the right, but in a much more compact space. The trouble with the traditional layout of pipeline objects is that it takes a lot of space, and even moderately sized pipelines require a significant portion of the GUI to see fully. The pipeline browser, however, is complete and compact.

## Multiview

Occasionally in the pursuit of science we can narrow our focus down to one variable. However, most interesting physical phenomena rely on not one but many variables interacting in certain ways. It can be very challenging to present many variables in the same view. To help you explore complicated visualization data, **ParaView** contains the ability to present multiple views of data and correlate them together.

So far in our visualization we are looking at two variables: We are coloring with pressure and have extracted an isosurface with temperature. Although we are starting to get the feel for the layout of these variables, it is still difficult to make correlations between them. To make this correlation easier, we can use multiple views. Each view can show an independent aspect of the data and together they may yield a more complete understanding.

On top of each view is a small toolbar, and the buttons controlling the creation and deletion of views are located on the right side of this tool bar. There are four buttons in all. You can create a new view by splitting an existing view horizontally or vertically with the  and  buttons, respectively. The ✕ button deletes a view, whose space is consumed by an adjacent view. The □ temporarily fills view space with the selected view until 🗗 is pressed.

---

**Exercise 2.11 (Using Multiple Views)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select Edit → Reset Session from the menu or hit  on the toolbar. This option deletes all of your current work and resets **ParaView** back to its initial state. It is roughly the equivalent of restarting **ParaView**.

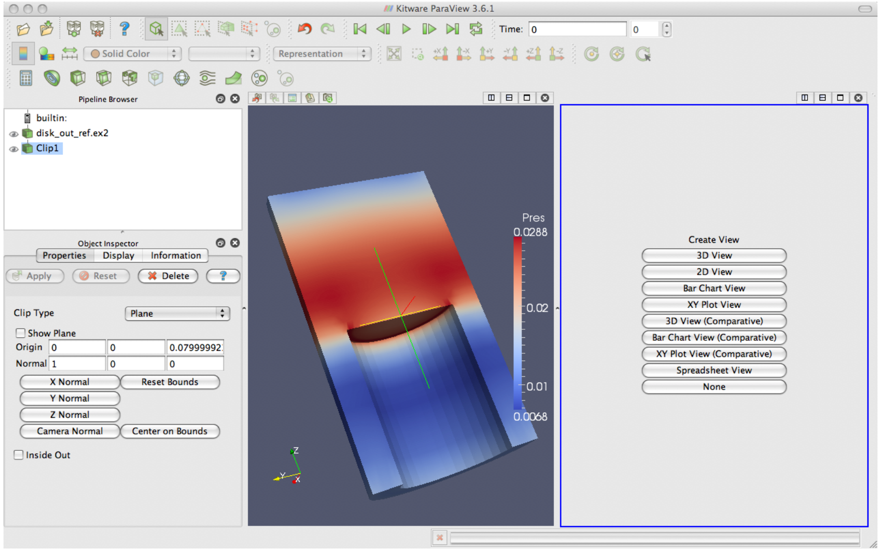
First, we will look at one variable. We need to see the variable through the middle of the mesh, so we are going to clip the mesh in half.

1. Open the file disk_out_ref.ex2, load all variables, Apply (see Exercise 2.7).
2. Add the Clip filter  to disk_out_ref.ex2.

3. Uncheck the Show Plane checkbox ☑ Show Plane in the properties panel.
4. Click the Apply button.
5. Color the surface by pressure by changing the variable chooser (in the toolbar) from vtkBlockColors to pres.

Now we can see the pressure in a plane through the middle of the mesh. We want to compare that to the temperature on the same plane. To do that, we create a new view to build another visualization.

6. Press the  button.



Clip Type: Plane
□ Show Plane
Origin: 0 | 0 | 0.07999992
Normal: 1 | 0 | 0
□ Inside Out

The current view is split in half and the right side is blank, ready to be filled with a new visualization. Notice that the view in the right has a blue border around it. This means that it is the **active view**. Widgets that give information about and controls for a single view, including the pipeline browser and properties panel, follow the active view. In this new view we will visualize the temperature of the mesh.

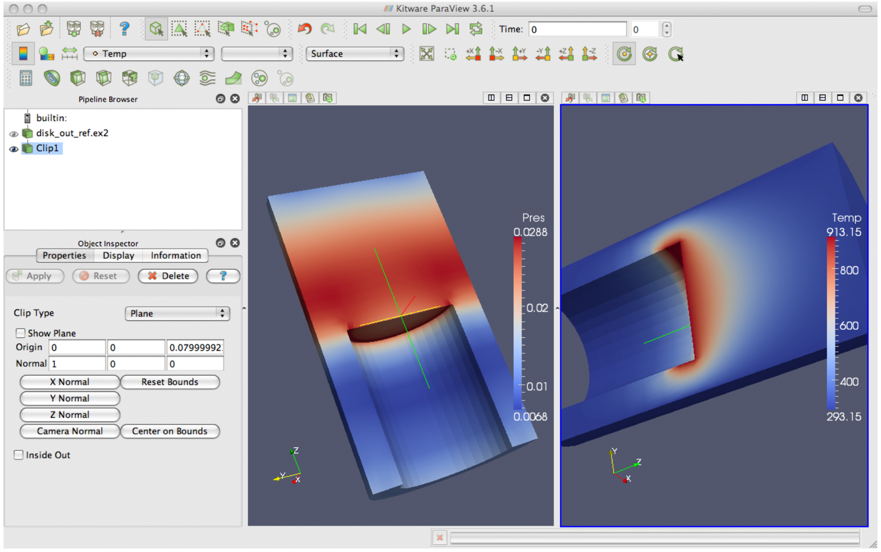
7. Make sure the blue border is still around the new, blank view (to the right). You can make any view the active view by simply clicking on it.
8. Turn on the visibility of the clipped data by clicking the eyeball  next to Clip1 in the pipeline browser.
9. Color the surface by temperature by selecting Clip1 in the pipeline browser and changing the variable chooser (in the toolbar) from Solid Color to temp.

We now have two views: one showing information about pressure and the other information about temperature. We would like to compare these, but it is difficult to do because the orientations are different. How are we to know how a location in one correlates to a location in the other? We can solve this problem by adding a **camera link** so that the two views will always be drawn from the same viewpoint. Linking cameras is easy.

10. Right click on one of the views and select Link Camera… from the pop up menu. (If you are on a Mac with no right mouse button, you can perform the same operation with the menu option Tools → Add Camera Link….)
11. Click in the second view.
12. Try moving the camera in each view.

*Voilà*! The two cameras are linked; each will follow the other. With the cameras linked, we can make some comparisons between the two views. Click the  button to get a straight-on view of the cross section and zoom in a bit.

Notice that the temperature is highest at the interface with the heated disk. That alone is not surprising. We expect the air temperature to be greatest near the heat source and drop off away from it. But notice that at the same position the



Kitware ParaView 3.6.1

Time: 0

Pipeline Browser

builtin:
disk_out_ref.ex2
Clip1

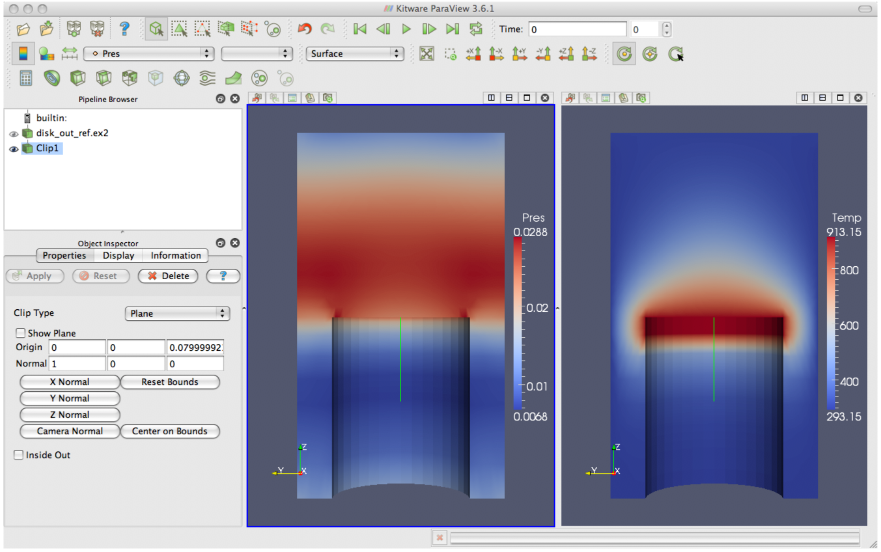
Clip Type Plane

□ Show Plane

Origin 0 0 0.07999992

Normal 1 0 0

□ Inside Out



Kitware ParaView 3.6.1

Time: 0

Pipeline Browser

builtin:
disk_out_ref.ex2
Clip1

Clip Type Plane

□ Show Plane

Origin 0 0 0.07999992

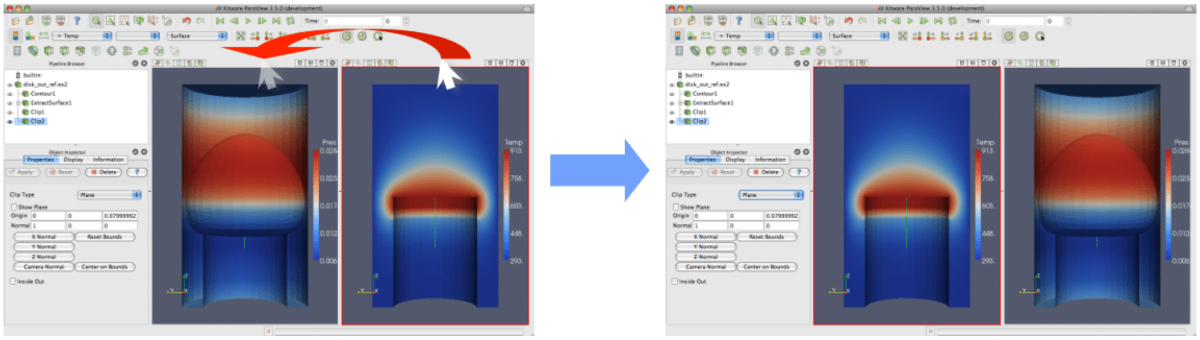
Normal 1 0 0

□ Inside Out

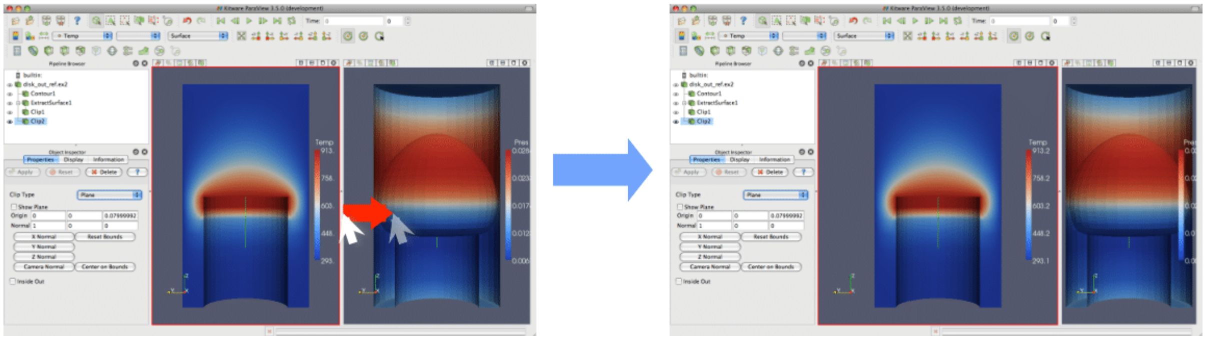
pressure is not maximal. The air pressure is maximal at a position above the disk. Based on this information we can draw some interesting hypotheses about the physical phenomenon. We can expect that there are two forces contributing to air pressure. The first force is that of gravity causing the upper air to press down on the lower air. The second force is that of the heated air becoming less dense and therefore rising. We can see based on the maximal pressure where these two forces are equal. Such an observation cannot be drawn without looking at both the temperature and pressure in this way.

Multiview in **ParaView** is of course not limited to simply two windows. Note that each of the views has its own set of multiview buttons. You can create more views by using the split view buttons to arbitrarily divide up the working space. And you can delete views ✕ at any time.

The location of each view is also not fixed. You are also able to swap two views by clicking on one of the view toolbars (somewhere outside of where the buttons are), holding down the mouse button, and dragging onto one of the other view toolbars. This will immediately swap the two views.



You can also change the size of the views by clicking on the space in between views, holding down the mouse button, and dragging in the direction of either one of the views. The divider will follow the mouse and adjust the size of the views as it moves.



### Vector Visualization

Let us see what else we can learn about this simulation. The simulation has also outputted a velocity field describing the movement of the air over the heated rotating disk. We will use **ParaView** to determine the currents in the air.

A common and effective way to characterize a vector field is with **streamlines**. A streamline is a curve through space that at every point is tangent to the vector field. It represents the path a weightless particle will take through the vector field (assuming steady-state flow). Streamlines are generated by providing a set of **seed points**.

**Exercise 2.12 (Streamlines)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to select reset session from the toolbar.

1. Open the file disk_out_ref.ex2, load all variables, Apply (see Exercise 2.7).
2. Add the stream tracer filter to disk_out_ref.ex2.
3. Change the Seed Type parameter in the properties panel to Point Cloud.
4. Uncheck the Show Sphere checkbox ☑ Show Sphere in the properties panel under the seed type.
5. Click the Apply button to accept these parameters.

The surface of the mesh is replaced with some swirling lines. These lines represent the flow through the volume. Notice that there is a spinning motion around the center line of the cylinder. There is also a vertical motion in the center and near the edges.

The new geometry is off-center from the previous geometry. We can quickly center the view on the new geometry with the **reset camera** command. This command centers and fits the visible geometry within the current view and also resets the center of rotation to the middle of the visible geometry.

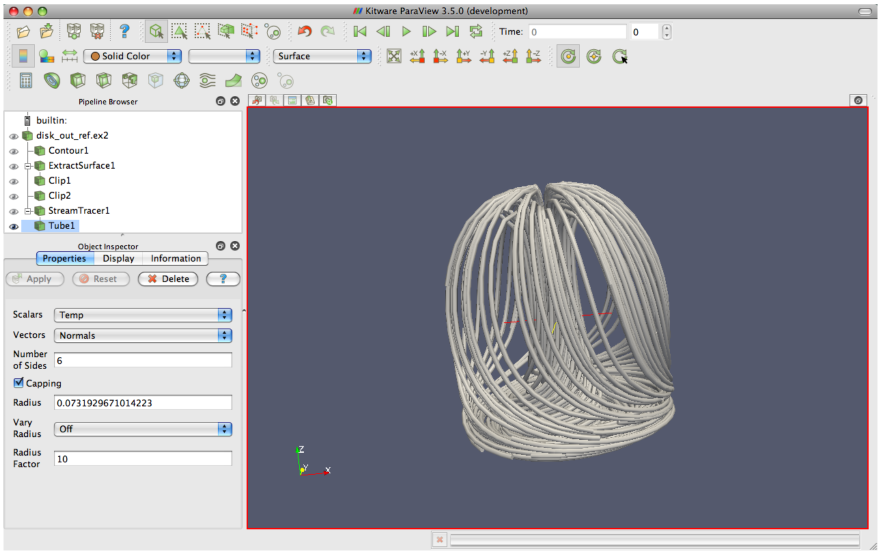
One issue with the streamlines as they stand now is that the lines are difficult to distinguish because there are many close together and they have no shading. Lines are a 1D structure and shading requires a 2D surface. Another issue with the streamlines is that we cannot be sure in which direction the flow is.

In the next exercise, we will modify the streamlines we created in Exercise 2.12 to correct these problems. We can create a 2D surface around our stream traces with the tube filter. This surface adds shading and depth cues to the lines. We can also add glyphs to the lines that point in the direction of the flow.

**Exercise 2.13 (Making Streamlines Fancy)**

This exercise is a continuation of Exercise 2.12. You will need to finish that exercise before beginning this one.

1. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to add the Tube filter to the streamlines.
2. Hit the Apply button.



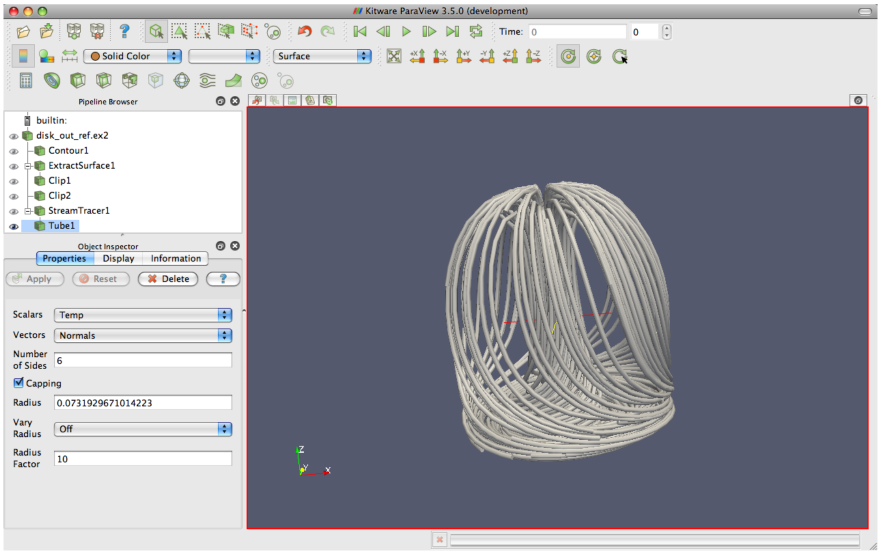
Scalars: Temp
Vectors: Normals
Number of Sides: 6
☑ Capping
Radius: 0.0731929671014223
Vary Radius: Off
Radius Factor: 10

You can now see the streamlines much more clearly. As you look at the streamlines from the side, you should be able to see circular convection as air heats, rises, cools, and falls. If you rotate the streams to look down the Z axis at the bottom near where the heated plate should be, you will also see that the air is moving in a circular pattern due to the friction of the rotating disk.

(Note that as an alternative to adding the Tube filter, you can instead directly render lines as tubes. This applies a special rendering mode that renders wide lines with tube-like shading. To do this, *instead* of creating the Tube filter set Line Width display parameter to some value larger than 1 (say 5) and click on the Render Lines As Tubes display parameter.)

Now we can get a little fancier. We can add glyphs to the streamlines to show the orientation and magnitude.

1. Select StreamTracer1 in the pipeline browser.
2. Add the glyph filter  to StreamTracer1.
3. In the properties panel, change the Glyph Type option to Cone.
4. In the properties panel, change the Orientation Array to v.
5. In the properties panel, change the Scale Array to v.
6. Click the reset  button to the right of Scale Factor.
7. Hit the Apply button.
8. Color the glyphs with the temp variable.



Scalars: Temp
Vectors: Normals
Number of Sides: 6
☑ Capping
Radius: 0.0731929671014223
Vary Radius: Off
Radius Factor: 10

Now the streamlines are augmented with little pointers. The pointers face in the direction of the velocity, and their size is proportional to the magnitude of the velocity. Try using this new information to answer the following questions.

- Where is the air moving the fastest? Near the disk or away from it? At the center of the disk or near its edges?
- Which way is the plate spinning?
- At the surface of the disk, is air moving toward the center or away from it?

## Plotting

**ParaView**'s plotting capabilities provide a mechanism to drill down into your data to allow quantitative analysis. Plots are usually created with filters, and all of the plotting filters can be found in the Data Analysis submenu of Filters. There is also a data analysis toolbar containing the most common data analysis filters, some of which are used to generate plots.

**Compute Quartiles**
Computes the quartiles for each field in the dataset and then shows a box chart depicting the quartiles.

**Extract Selection**
Extracts any data selected into its own object. Selections are described in Section 4.1.2.

**Histogram**
Allows you to generate the histogram of a data array from a dataset.

**Plot Global Variables Over Time**
Datasets sometimes capture information in "global" variables that apply to an entire dataset rather than a single point or cell. This filter plots the global information over time. **ParaView**'s handling of time is described in Section 4.1.2.

**Plot Over Line**
Allows you to define a line segment in 3D space and then plot field information over this line.

**Plot Selection Over Time**
Takes the fields in selected points or cells and plots their values over time. Selections are described in Section 4.1.2 and time is described in Section 4.1.2.

**Probe**
Provides the field values in a particular location in space.

**Programmable Filter**
Allows you to program a user-defined filter. To use the filter you define a function that retrieves input of the correct type, creates data, and then manipulates the output of the filter.

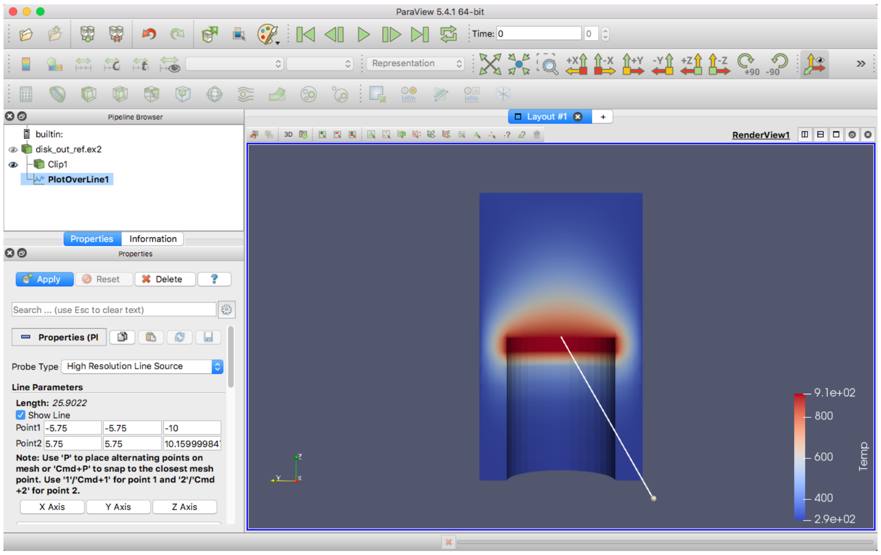
In the next exercise, we create a filter that will plot the values of the mesh's fields over a line in space.

**Exercise 2.14 (Plot Over a Line in Space)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Open the file disk_out_ref.ex2, load all variables, Apply (see Exercise 2.7).
2. Add the Clip filter  to disk_out_ref.ex2, Uncheck the Show Plane ☑ Show Plane and the Invert checkboxes in the properties panel, and click (like in Exercise 2.11). This will make it easier to see and manipulate the line we are plotting over.
3. Click on disk_out_ref.ex2 in the pipeline browser to make that the active object.
4. From the toolbars, select the plot over line  filter.

In the active view you will see a line through your data with a ball at each end. If you move your mouse over either of these balls, you can drag the balls through the 3D view to place them. If you hold down the x, y, or z key while you drag one of these balls, the movement will be constrained to that axis. Notice that each time you move the balls some of the fields in the properties panel also change. You can also place the balls by hovering your mouse over the target location and hitting the 1, 2, or p key. The 1 key will place the first ball at the surface underneath the mouse cursor. The 2 key will likewise place the second ball. The p key will alternate between placing the first and second balls. If you hold down the Ctrl modifier while hitting any of these keys, then the ball will be placed at the nearest point of the



Probe Type High Resolution Line Source

Line Parameters

Length: *25.9022*

☑ Show Line

Point1 -5.75 -5.75 -10

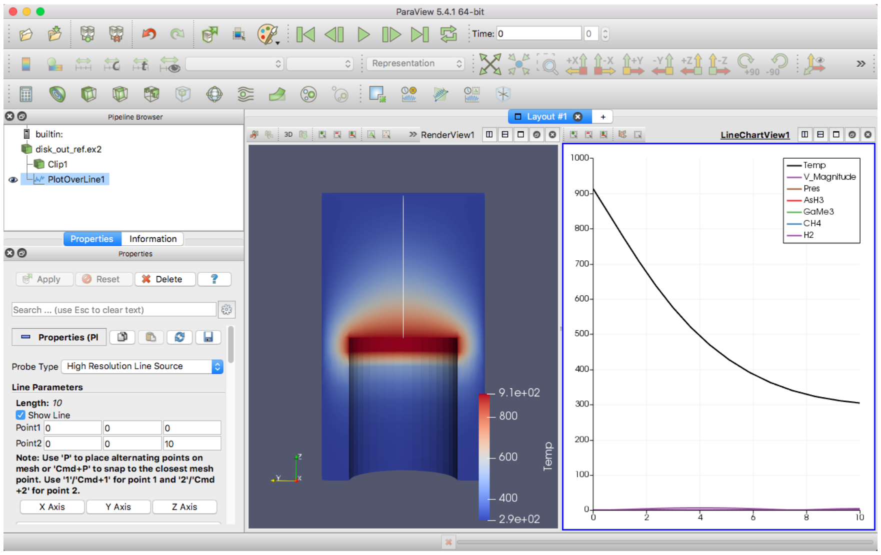
Point2 5.75 5.75 10.15999984

Note: Use 'P' to place alternating points on mesh or 'Cmd+P' to snap to the closest mesh point. Use '1'/'Cmd+1' for point 1 and '2'/'Cmd +2' for point 2.

underlying mesh rather than directly under the mouse. This was the purpose of adding the clip filter: It allows us to easily add the endpoints to this plane. Note that placing the endpoints in this manner only works when rendering solid surfaces. It will not work with a volume rendered image or transparent surfaces.

This representation is called a **3D widget** because it is a GUI component that is manipulated in 3D space. There are many examples of 3D widgets in **ParaView**. This particular widget, the line widget, allows you to specify a line segment in space. Other widgets allow you to specify points or planes.

5. Adjust the line so that it goes from the base of the disk straight up to the top of the mesh using the 3D widget manipulators, the p key shortcut, or the properties panel parameters. The plot works best with Point 1 around $(0, 0, 0)$ and Point 2 around $(0, 0, 10)$.
6. Once you have your line satisfactorily located, click the button.



Probe Type High Resolution Line Source

Line Parameters

Length: *10*

☑ Show Line

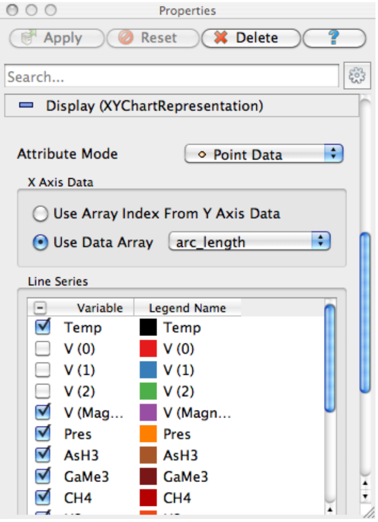
Point1 0 0 0

Point2 0 0 10

Note: Use 'P' to place alternating points on mesh or 'Cmd+P' to snap to the closest mesh point. Use '1'/'Cmd+1' for point 1 and '2'/'Cmd +2' for point 2.

There are several interactions you can do with the plot. Roll the mouse wheel up and down to zoom in and out. Drag with the middle button to do a rubber band zoom. Drag with the left button to scroll the plot around. You can also use the reset camera command  to restore the view to the full domain and range of the plot.

Plots, like 3D renderings, are considered views. Both provide a representation for your data; they just do it in different ways. Because plots are views, you interact with them in much the same ways as with a 3D view. If you look in the Display section of the properties panel, you will see many options on the representation for each line of the plot including colors, line styles, vector components, and legend names.



Attribute Mode: Point Data

X Axis Data

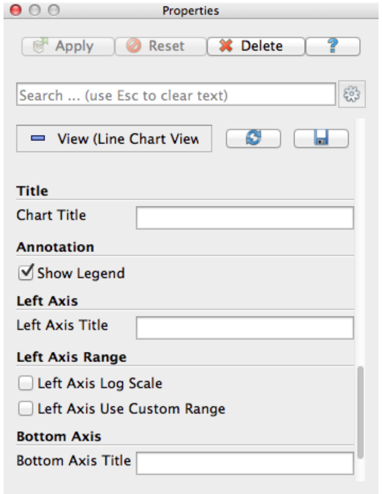
○ Use Array Index From Y Axis Data

● Use Data Array: arc_length

Line Series

| | Variable | | Legend Name |
|---|---|---|---|
| ☑ | Temp | ■ | Temp |
| □ | V (0) | 🟥 | V (0) |
| □ | V (1) | 🟦 | V (1) |
| □ | V (2) | 🟩 | V (2) |
| ☑ | V (Mag… | 🟪 | V (Magn… |
| ☑ | Pres | 🟧 | Pres |
| ☑ | AsH3 | 🟫 | AsH3 |
| ☑ | GaMe3 | 🟫 | GaMe3 |
| ☑ | CH4 | 🟥 | CH4 |

If you scroll down further to the View section of the properties panel, you to change plot-wide options such as labels, legends, and axes ranges.



Title

Chart Title:

Annotation

☑ Show Legend

Left Axis

Left Axis Title:

Left Axis Range

□ Left Axis Log Scale

□ Left Axis Use Custom Range

Bottom Axis

Bottom Axis Title:

Like any other views, you can capture the plot with the File → Save Screenshot. Additionally, if you choose File → Export Scene… you can export a file with vector graphics that will scale properly for paper-quality images. We will discuss these image capture features later in Section 4.1.2. You can also resize and swap plots in the GUI like you can other views.

In the next exercise, we modify the display to get more information out of our plot. Specifically, we use the plot to compare the pressure and temperature variables.

**Exercise 2.15 (Plot Series Display Options)**

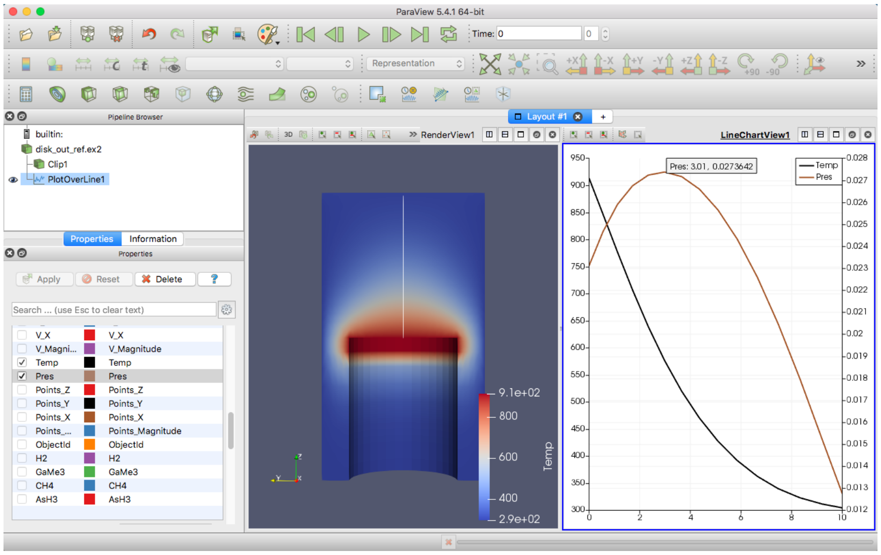
This exercise is a continuation of Exercise 2.14. You will need to finish that exercise before beginning this one.

1. Choose a place in your GUI that you would like the plot to go and try using the split, delete, resize, and swap view features to move it there.

2. Make the plot view active, go to the Display section of the properties panel, and turn off all variables except temp and pres.

The temp and pres variables have different units. Putting them on the same scale is not useful. We can still compare them in the same plot by placing each variable on its own scale. The line plot in **ParaView** allows for a different scale on the left and right axis, and you can scale each variable individually on each axis.

3. Select the pres variable in the Display options.
4. Change the Chart Axis to Bottom - Right

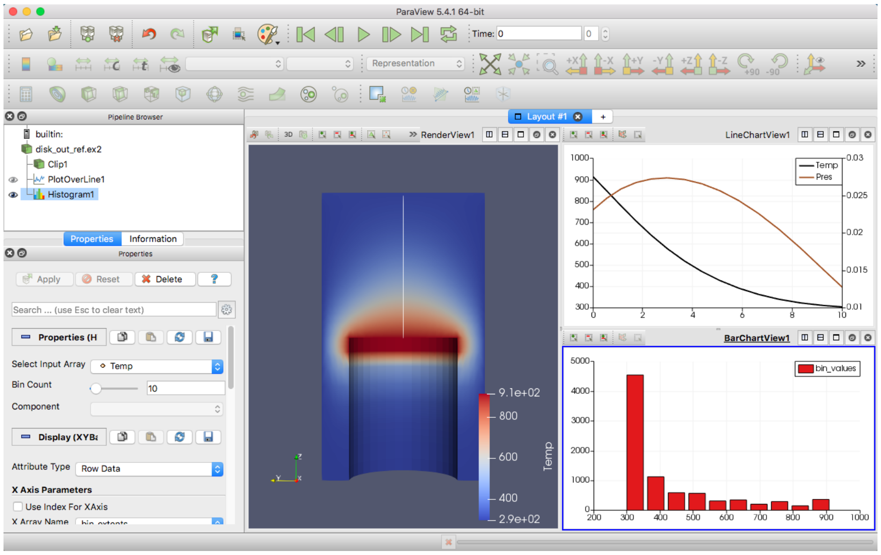


☐ V_X 🟥 V_X
☐ V_Magni... 🟪 V_Magnitude
☑ Temp ⬛ Temp
☑ Pres 🟫 Pres
☐ Points_Z 🟥 Points_Z
☐ Points_Y ⬛ Points_Y
☐ Points_X 🟫 Points_X
☐ Points_... 🟦 Points_Magnitude
☐ ObjectId 🟧 ObjectId
☐ H2 🟪 H2
☐ GaMe3 🟩 GaMe3
☐ CH4 🟦 CH4
☐ AsH3 🟥 AsH3

From this plot we can verify some of the observations we made in Section 4.1.2. We can see that the temperature is maximal at the plate surface and falls as we move away from the plate, but the pressure goes up and then back down. In addition, we can observe that the maximal pressure (and hence the location where the forces on the air are equalized) is about 3 units away from the disk.

---

The **ParaView** framework is designed to accommodate any number of different types of views. This is to provide researchers and developers a way to deliver new ways of looking at data. To see another example of view, select disk_out_ref.ex2 in the pipeline browser, and then select Filters → Data Analysis → Histogram . Make the histogram for the temp variable, and then hit Apply the button.



Select Input Array: Temp

Bin Count: 10

Component:

Attribute Type: Row Data

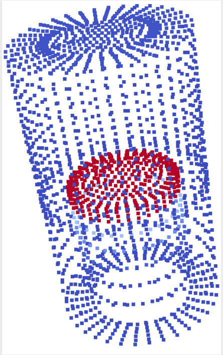
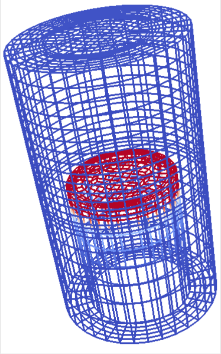
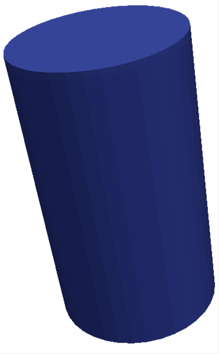
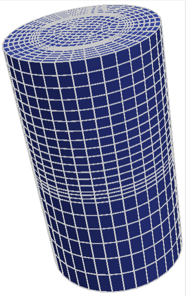
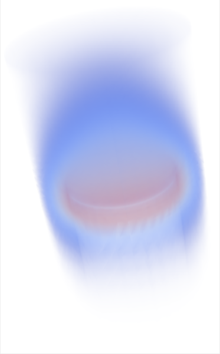
☐ Use Index For XAxis

X Array Name: bin_extents

## Volume Rendering

**ParaView** has several ways to represent data. We have already seen some examples: surfaces, wireframe, and a combination of both. **ParaView** can also render the points on the surface or simply draw a bounding box of the data.

Table 4.1: Representations

| | | | | |
|---|---|---|---|---|
|  |  |  |  |  |
| Points | Wireframe | Surface | Surface With Edges | Volume |

A powerful way that **ParaView** lets you represent your data is with a technique called **volume rendering**. With volume rendering, a solid mesh is rendered as a translucent cloud with the scalar field determining the color and density at every point in the cloud. Unlike with surface rendering, **volume rendering** allows you to see features all the way through a volume.

Volume rendering is enabled by simply changing the representation of the object. Let us try an example of that now.

**Exercise 2.16 (Turning On Volume Rendering)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Open the file disk_out_ref.ex2, load all variables, Apply (see Exercise 2.7.
2. Make sure disk_out_ref.ex2 is selected in the pipeline browser. Change the variable viewed to temp and change the representation to Volume.

3. If you get an Are you sure? dialog box warning you about the change to the volume representation, click Yes to enact the change.
4. Show Center  of rotation.

The solid opaque mesh is replaced with a translucent volume. You may notice that when rotating your object that the rendering is temporarily replaced with a simpler transparent surface for performance reasons. We discuss this behavior in more detail later in Section 4.1.4.

A useful feature of **ParaView**'s volume rendering is that it can be mixed with the surface rendering of other objects. This allows you to add context to the volume rendering or to mix visualizations for a more information-rich view. For example, we can combine this volume rendering with a streamline vector visualization like we did in Exercise 2.12.

**Exercise 2.17 (Combining Volume Rendering and Surface-Based Visualization)**

This exercise is a continuation of Exercise 2.16. You will need to finish that exercise before beginning this one.

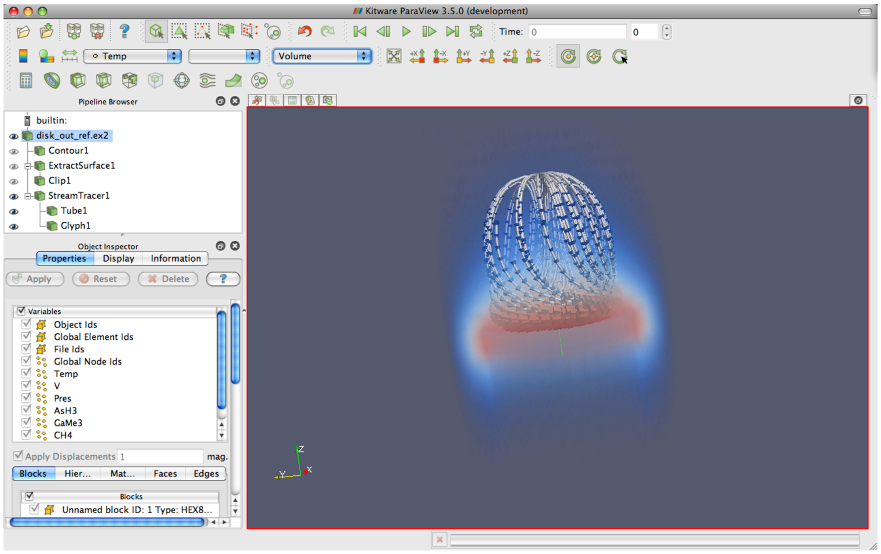
1. Add the stream tracer filter  to disk_out_ref.ex2.
2. Change the Seed Type parameter in the properties panel to Point Cloud.
3. Uncheck the Show Sphere checkbox ☑ Show Sphere in the properties panel under the seed type.
4. Click the Apply button to accept these parameters.

You should now be seeing the streamlines embedded within the volume rendering. The following additional steps add geometry to make the streamlines easier to see much like in Exercise 2.13. They are optional, so you can skip them if you wish.

5. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to apply the Tube filter and hit Apply.
6. If the streamlines are colored by temp, change that to Solid Color.
7. Select StreamTracer1 in the pipeline browser.
8. Add the glyph filter  to StreamTracer1.
9. In the properties panel, change the Glyph Type option to Cone.
10. In the properties panel, change the Orientation Array to v.
11. In the properties panel, change the Scale Array to v.
12. Click the reset  button to the right of Scale Factor.
13. Hit the Apply button.
14. Color the glyphs with the temp variable.

The streamlines are now shown in context with the temperature throughout the volume.

By default, **ParaView** will render the volume with the same colors as used on the surface with the transparency set to 0 for the low end of the range and 1 for the high end of the range. **ParaView** also provides an easy way to change the **transfer function**, how scalar values are mapped to color and transparency. You can access the transfer function editor by selecting the volume rendered-pipeline object (in this case disk_out_ref.ex2) and clicking on the edit color map  button.



The first time you bring up the color map editor, it should appear at the right side of the **ParaView** GUI window. Like most of the panels in **ParaView**, this is a dockable window that you can move around the GUI or pull off and place elsewhere on your desktop. Like the properties panel, some of the advanced options are hidden to simplify the interface. To access these hidden features, toggle the  button in the upper right or type a search string.

The two colorful boxes at the top represent the transfer function. The first box with a function plot with colors underneath represents the transparency whereas the long box at the bottom represents the colors. The dots on the transfer functions represent the **control points**. The control points are the specific color and opacity you set at particular scalar values, and the colors and transparency are interpolated between them. Clicking on a blank spot in either bar will create a new control point. Clicking on an existing control point will select it. The selected control point can be dragged throughout the box to change its scalar value and transparency (if applicable). Double clicking on a color control point will allow you to change the color. The selected control point will be deleted when you hit the backspace or delete key.

**Did you know?**

For surface rendering, the transparency controls have no effect unless “Enable opacity mapping for surfaces” is enabled.

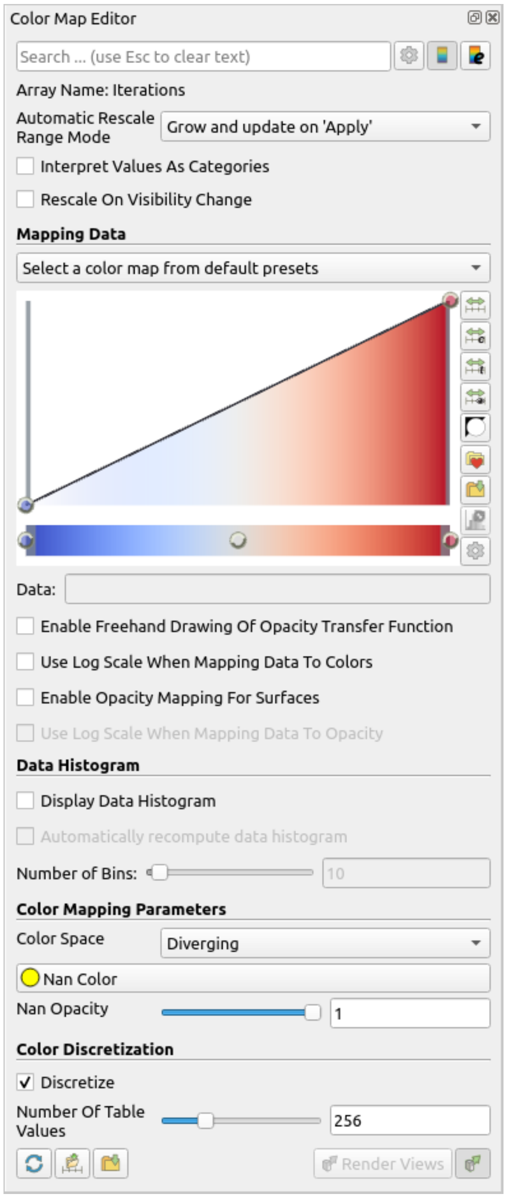
Directly below the color and transparency bars is a text entry widget to numerically specify the Data Value of the selected control point. Below this are checkbox options to Use log scale when mapping data to colors, to Enable opacity mapping for surfaces, and to Automatically rescale transfer functions to fit data. (Note that this last option causes the data range to be resized under most operations that change data, but not when the time value changes. See Section 4.1.2 for more details.)

The following Color Space parameter changes how colors are interpolated. This parameter has no effect on the color at the control points, but can drastically affect the colors between the control points. Finally, the Nan Color allows you to select a color for “invalid” values. A **NaN** is a special floating point value used to represent something that is not a number (such as the result of $0/0$).

Setting up a transfer function can be tedious, so you can save it by clicking the Save to preset  button. The Choose preset  button brings up a dialog that allows you to manage and apply the color maps that you have created as well as many provided by **ParaView**.

**Exercise 2.18 (Modifying Volume Rendering Transfer Functions)**

This exercise is a continuation of Exercise 2.17. You will need to finish that exercise (or minimally Exercise 2.16) before beginning this one.



Color Map Editor

Search ... (use Esc to clear text)

Array Name: Iterations

Automatic Rescale Range Mode: Grow and update on 'Apply'

□ Interpret Values As Categories

□ Rescale On Visibility Change

**Mapping Data**

Select a color map from default presets

Data:

□ Enable Freehand Drawing Of Opacity Transfer Function

□ Use Log Scale When Mapping Data To Colors

□ Enable Opacity Mapping For Surfaces

□ Use Log Scale When Mapping Data To Opacity

**Data Histogram**

□ Display Data Histogram

□ Automatically recompute data histogram

Number of Bins: 10

**Color Mapping Parameters**

Color Space: Diverging

Nan Color

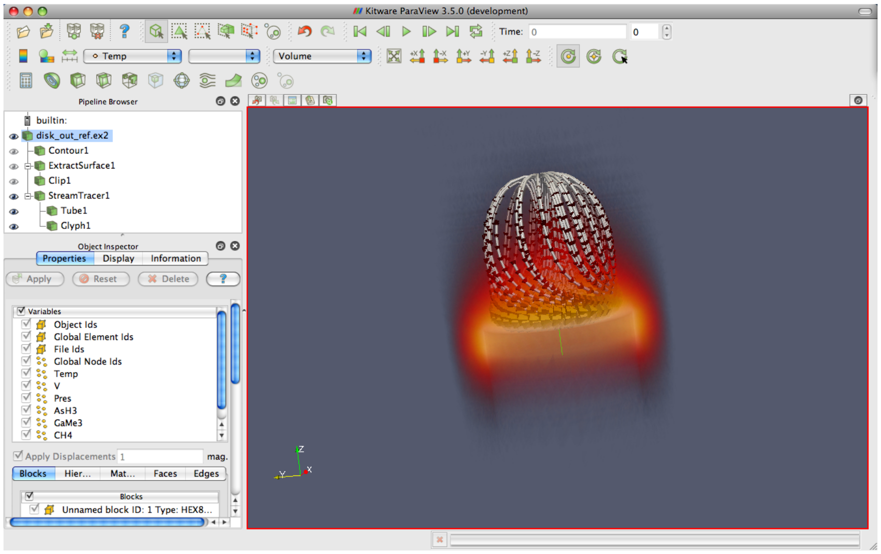
Nan Opacity: 1

**Color Discretization**

☑ Discretize

Number Of Table Values: 256

1. Click on disk_out_ref.ex2 in the pipeline browser to make that the active object.
2. Click on the edit color map  button.
3. Change the volume rendering to be more representative of heat. Press Choose preset  , select Black-Body Radiation in the dialog box, and then click Apply followed by Close.
4. Try adding and changing control points and observe their effect on the volume rendering. By default as you make changes the render views update. If this interactive update is too slow, you can turn off this feature by toggling the  button. When automatic updates are off, transfer function changes are not applied until the  Render Views is clicked.



Notice that not only did the color mapping in the volume rendering change, but all the color mapping for temp changed including the cone glyphs if you created them. This ensures consistency between the views and avoids any confusion from mapping the same variable with different colors or different ranges.

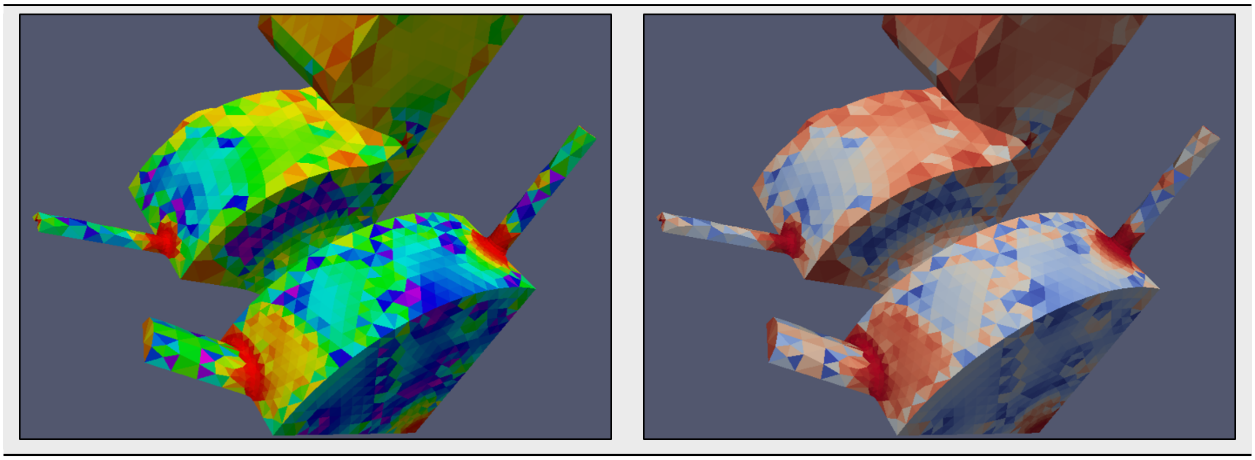
---

While looking through the color map presents  , you probably noticed one or more entries with **Rainbow** as part of the title that incorporate the colors of the a rainbow into the color map. You may also recognize this set of colors from other visualizations you have seen in the past.

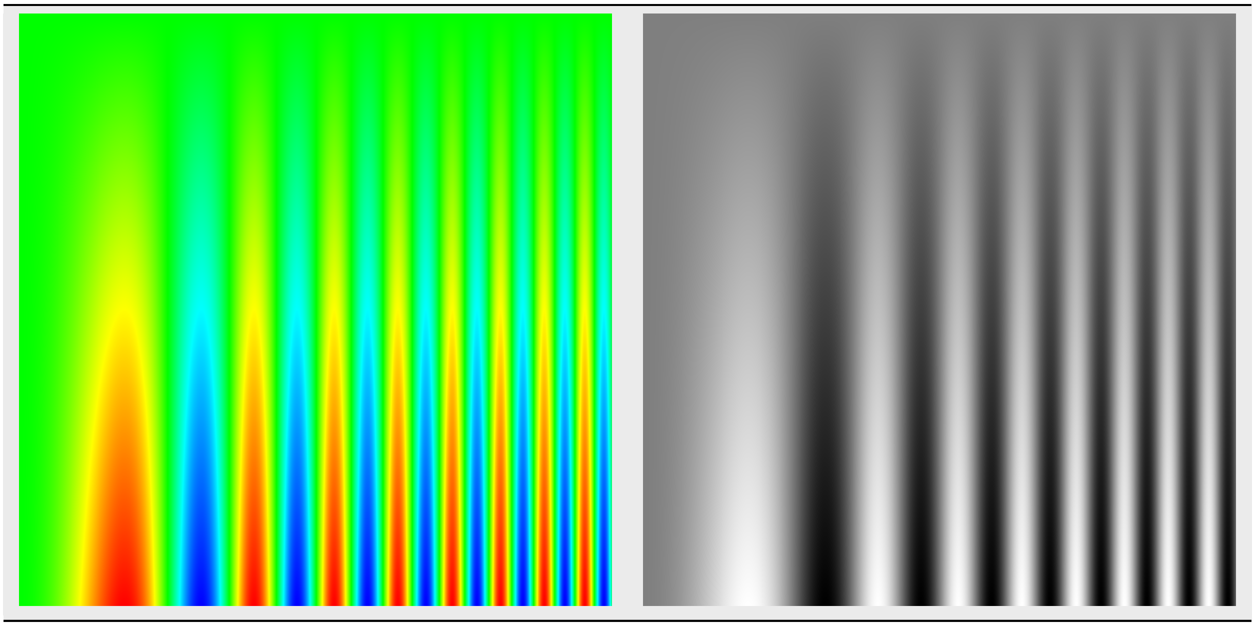
Rainbow colors are certainly a popular choice. However, we recommend that you **never use rainbow color maps** in your visualizations.

The problem with rainbow colors is that they have numerous perceptual properties that serve to obfuscate the data. Although we will not go into detail into the many perceptual studies that provide evidence that rainbow colors are bad for data display, we provide a brief synopsis of the problems.

The first problem is that the colors do not follow any natural perceived ordering. Some color groups lead to a natural perception of order (with relative brightness being the strongest perceptual cue). The hues of the rainbow, however, have no real ordered meaning in our visual system. Rather, we have to learn an ordering, which can lead to visual confusion. Consider the following two example images. In the left image, the rainbow hues make it difficult to ascertain the relative high and low values that are more clear in the right image.



The second problem with the perception of rainbow colors is that the perceptual changes in the colors are not uniform. The colors appear to change faster in the cyan and yellow regions, which can introduce artifacts in those regions that do not exist in the data. The colors appear to change more slowly in the blue, green, and red regions, which creates larger bands of color that hide artifacts in the data. We can see this effect in the following two images of a spatial contrast sensitivity function. The grayscale on the right faithfully reproduces the function. However, the rainbow colors on the left hides the variation in low contrast regions and appears less smooth in the high-contrast regions.



A third problem with the rainbow color map is that it is sensitive to deficiencies in vision. Roughly 5% of the population cannot distinguish between the red and green colors. Viewers with color deficiencies cannot distinguish many colors considered “far apart” in the rainbow color map.

We provide this description of the problems with rainbow colors in the hopes that you do not use these color maps. Despite the well know problems with rainbow colors, they remain a popular choice. Relying on rainbow hues obfuscates data, which circumvents the entire process of data analysis that **ParaView** provides. Multiple perceptual studies have shown that subjects overestimate the effectiveness of rainbow colors. That is, people think they do better with rainbow colors when in fact they do worse. Don’t fall into this trap.

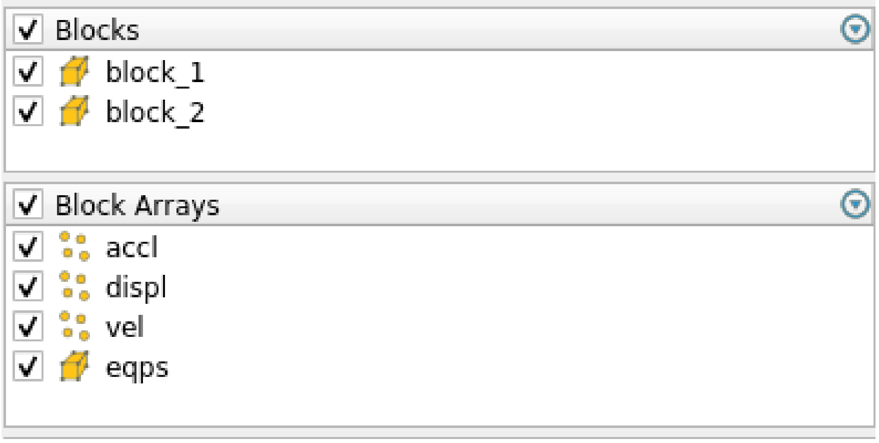
## Time

Now that we have thoroughly analyzed the disk_out_ref simulation, we will move to a new simulation to see how **ParaView** handles time. In this section we will use a new dataset from another simple simulation, this time with data that changes over time.

**Exercise 2.19 (Loading Temporal Data)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

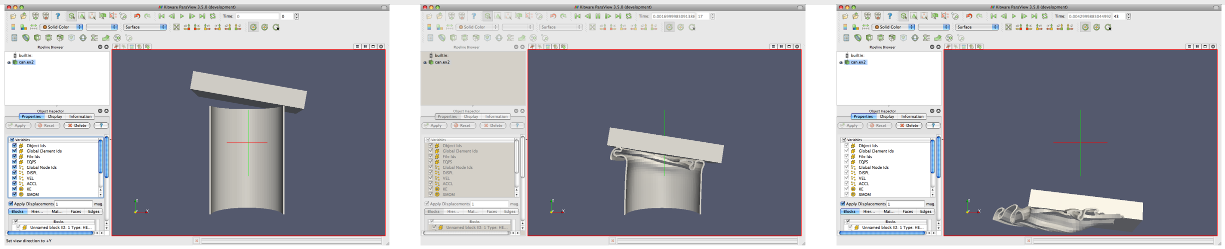
1. Open the file can.ex2.



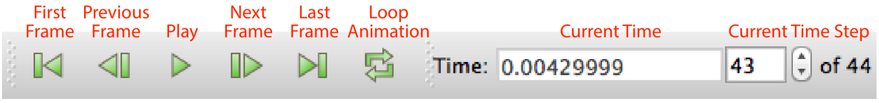
☑ Blocks
☑ block_1
☑ block_2

☑ Block Arrays
☑ accl
☑ displ
☑ vel
☑ eqps

2. As before, click the checkbox in the header of the block arrays list to turn on the loading of all the block arrays and hit the Apply button.
3. Press the  button to orient the camera to the mesh.
4. Press the play button  in the toolbars and watch **ParaView** animate the mesh to crush the can with the falling brick.



That is really all there is to dealing with data that is defined over time. **ParaView** has an internal concept of time and automatically links in the time defined by your data. Become familiar with the toolbars that can be used to control time.



Time: 0.00429999 43 of 44

Saving an animation is equally as easy. From the menu, select File → Save Animation. **ParaView** provides dialogs specifying how you want to save the animation, and then automatically iterates and saves the animation.

**Exercise 2.20 (Temporal Data Pitfall)**

The biggest pitfall users run into is that with mapping a set of colors whose range changes over time. To demonstrate this, do the following.

1. If you are not continuing from Exercise 2.19, open the file can.ex2, load all variables, Apply.
2. Go to the first time step .
3. Color by the EQPS variable.
4. Play  through the animation (or skip to the last time step ).

The coloring is not very useful. To quickly fix the problem:

5. While at the last time step, click the Rescale to Data Range  button.
6. Play  the animation again.

The colors are more useful now.

Although this behavior seems like a bug, it is not. It is the consequence of two unavoidable behaviors. First, when you turn on the visibility of a scalar field, the range of the field is set to the range of values in the current time step. Ideally, the range would be set to the max and min over all time steps in the data.

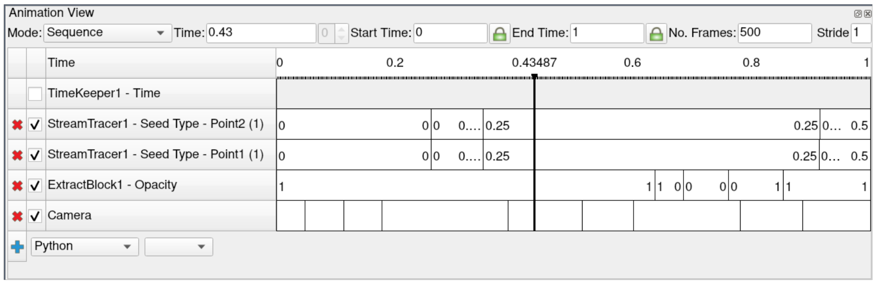
However, this requires **ParaView** to load in all of the data on the initial read, and that is prohibitively slow for large data. Second, when you animate over time, it is important to hold the color range fixed even if the range in the data changes. Changing the scale of the data as an animation plays causes a misrepresentation of the data. It is far better to let the scalars go out of the original color map's range than to imply that they have not. There are several workarounds to this problem:

- If for whatever reason your animation gets stuck on a poor color range, simply go to a representative time step and hit . This is what we did in the previous exercise.
- Open the settings dialog box accessed in the menu from Edit → Settings (**ParaView** → Preferences on the Mac). Under the General tab, find the option labeled Default Time Step and change it to Go to last timestep. (If you have trouble finding this option, try typing timestep into the setting's search box.) When this is selected, **ParaView** will automatically go to the last time step when loading any dataset with time. For many data (such as in can), the field ranges are more representative at the last time step than at the beginning. Thus, as long as you color by a field before changing the time, the color range will be adequate.
- Click the Rescale to Custom Data Range  toolbar button. This is a good choice if you cannot find, or do not know, a "representative" time step or if you already know a good range to use.
- If you are willing to wait or have small data, you can use the Rescale to data range over all timesteps  toolbar button and **ParaView** will compute this overall temporal range automatically. Keep in mind that this option will require **ParaView** to load your entire dataset over all time steps. Although **ParaView** will not hold more than one time step in memory at a time, it will take a long time to pull all that memory off of disk for large datasets.

**ParaView** has many powerful options for controlling time and animation. The majority of these are accessed through the **animation view**. From the menu, click on View → Animation View.

**Notice**

This part mention the animation view. This was replaced by the Time Manager. While the interface are quite similar, some differences may exist. Please see the ParaView User's Guide 'Animation' section for more.



Animation View

Mode: Sequence Time: 0.43 0 Start Time: 0 End Time: 1 No. Frames: 500 Stride 1

Time

TimeKeeper1 - Time

StreamTracer1 - Seed Type - Point2 (1)

StreamTracer1 - Seed Type - Point1 (1)

ExtractBlock1 - Opacity

Camera

Python

For now we will examine the controls at the top of the animation view. (We will examine the use of the tracks in the rest of the animation view later in Section 4.1.2.) The **animation mode** parameter determines how **ParaView** will step through time during playback. There are three modes available.

**Sequence**
Given a start and end time, break the animation into a specified number of frames spaced equally apart.

**Real Time**
**ParaView** will play back the animation such that it lasts the specified number of seconds. The actual number of frames created depends on the update time between frames.

**Snap To TimeSteps**
**ParaView** will play back exactly those time steps that are defined by your data.

Whenever you load a file that contains time, **ParaView** will automatically change the animation mode to Snap To TimeSteps. Thus, by default you can load in your data, hit play , and see each time step as defined in your data. This is by far the most common use case.

A counter use case can occur when a simulation writes data at variable time intervals. Perhaps you would like the animation to play back relative to the simulation time rather than the time index. No problem. We can switch to one of the other two animation modes. Another use case is the desire to change the playback rate. Perhaps you would like to speed up or slow down the animation. The other two animation modes allow us to do that.

---

**Exercise 2.21 (Slowing Down an Animation with the Animation Mode)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Open the file can.ex2, load all variables, Apply (see Exercise 2.19).
2. Press the  button to orient the camera to the mesh.
3. Press the play button  in the toolbars.

During this animation, **ParaView** is visiting each time step in the original data file exactly once. Note the speed at which the animation plays.

4. If you have not done so yet, make the animation view visible: View → Animation View.
5. Change the animation mode to Real Time. By default the animation is set up with the time range specified by the data and a duration of 10 seconds.
6. Play  the animation again.

The result looks similar to the previous Snap To TimeSteps animation, but the animation is now a linear scaling of the simulation time and will complete in 10 seconds.

7. Change the Duration to 60 seconds.
8. Play  the animation again.

The animation is clearly playing back more slowly. Unless your computer is updating slowly, you will also notice that the animation appears jerkier than before. This is because we have exceeded the temporal resolution of the dataset.

Often showing the jerky time steps from the original data is the desired behavior; it is showing you exactly what is present in the data. However, if you wanted to make an animation for a presentation, you may want a smoother animation.

There is a filter in **ParaView** designed for this purpose. It is called the **temporal interpolator**. This filter will interpolate the positional and field data in between the time steps defined in the original dataset.

**Exercise 2.22 (Temporal Interpolation)**

This exercise is a continuation of Exercise 2.21. You will need to finish that exercise before beginning this one.

1. Make sure can.ex2 is highlighted in the pipeline browser.
2. Select Filters → Temporal → Temporal Interpolator or apply the Temporal Interpolator filter using the quick launch (ctrl+space Win/Linux, alt+space Mac).
3. Apply.
4. Split the view , show the TemporalInterpolator1 in one, show can.ex2 in the other, and link the cameras.
5. Play  the animation.

You should notice that the output from the temporal interpolator animates much more smoothly than the original data.

It is worth noting that the temporal interpolator can (and often does) introduce artifacts in the data. It is because of this that **ParaView** will never apply this type of interpolation automatically; you will have to explicitly add the Temporal Interpolator. In general, mesh deformations often interpolate well but moving fields through a static mesh do not. Also be aware that the Temporal Interpolator only works if the topology remains consistent. If you have an adaptive mesh that changes from one time step to the next, the Temporal Interpolator will give errors.

### Text Annotation

When using **ParaView** as a communication tool it is often helpful to annotate the images you create with text. With **ParaView** it is very easy to create text annotation wherever you want in a 3D view. There is a special **text source** that simply places some text in the view.

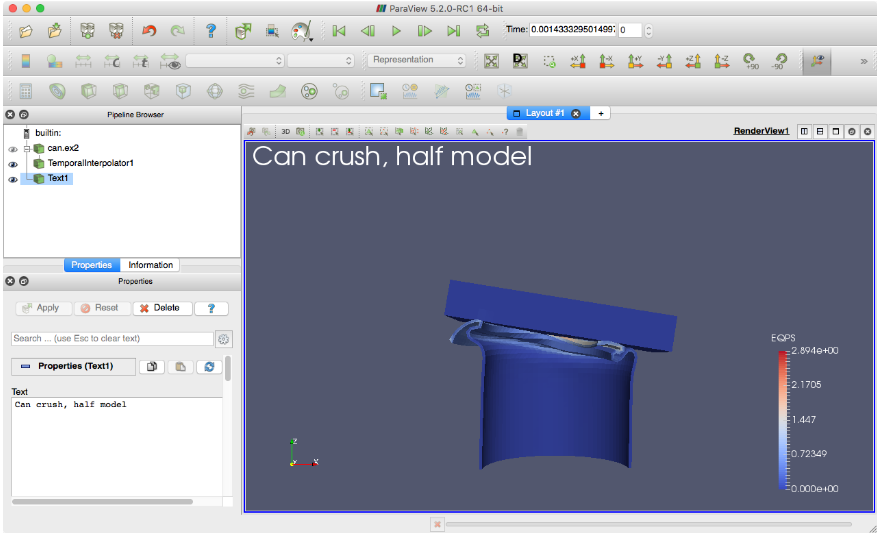
**Exercise 2.23 (: Adding Text Annotation)**

**If you are continuing this exercise after finishing Exercise 2.22,**

feel free to simply continue. If you have had to restart **ParaView** since or your state does not match up well enough, it is also fine to start with a fresh state using .

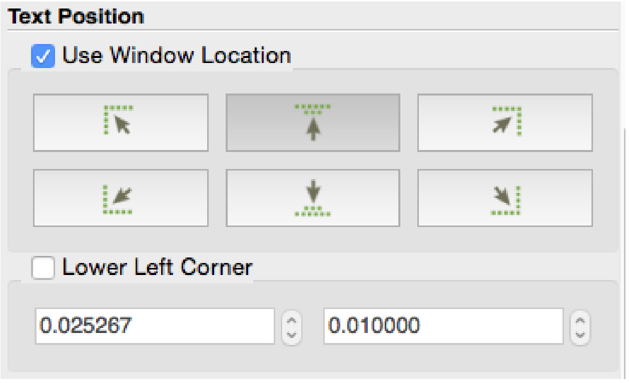
1. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to create the Text source (or from the menu bar Sources → Annotation → Text) and hit Apply.
2. In the text edit box of the properties panel, type a message.
3. Hit the Apply button.

The text you entered appears in the 3D view. If you scroll down to the Display options in the properties panel, you will see six buttons that allow you to quickly place the text in each of the four corners of the view as well as centered at the top and bottom.



Text

```
Can crush, half model
```



Text Position

☑ Use Window Location

☐ Lower Left Corner

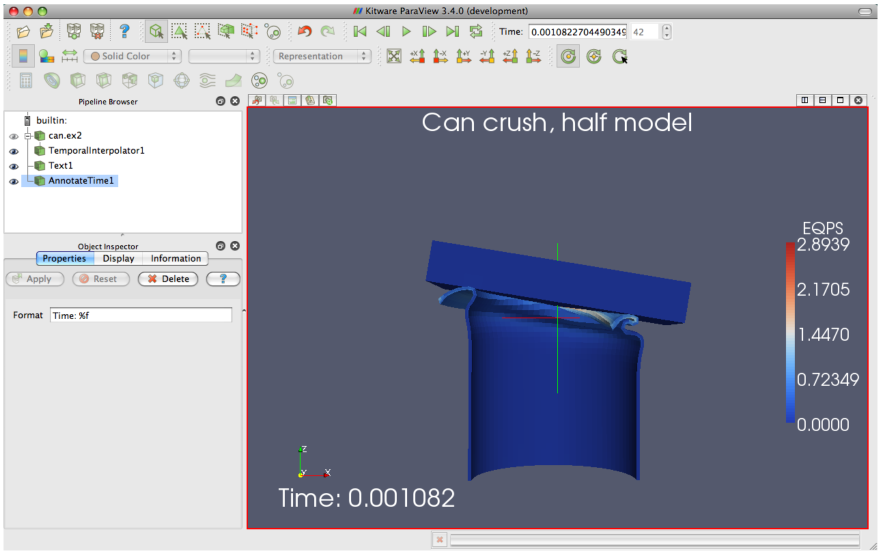
0.025267 0.010000

You can place this text at an arbitrary position by clicking the Lower Left Corner checkbox. With the Lower Left Corner option checked, you can use the mouse to drag the text to any position within the view.

Often times you will need to put the current time value into the text annotation. Typing the correct time value can be tedious and error prone with the standard text source and impossible when making an animation. Therefore, there is a special **annotate time source** that will insert the current animation time into the string.

**Exercise 2.24 (Adding Time Annotation)**

1. If you do not already have it loaded from a previous exercise, open the file can.ex2, Apply.
2. Add an Annotate Time source (Sources → Annotate → Annotate Time or use the quick launch: ctrl+space Win/Linux, alt+space Mac). Apply.
3. Move the annotation around as necessary.
4. Play  and observe how the time annotation changes.

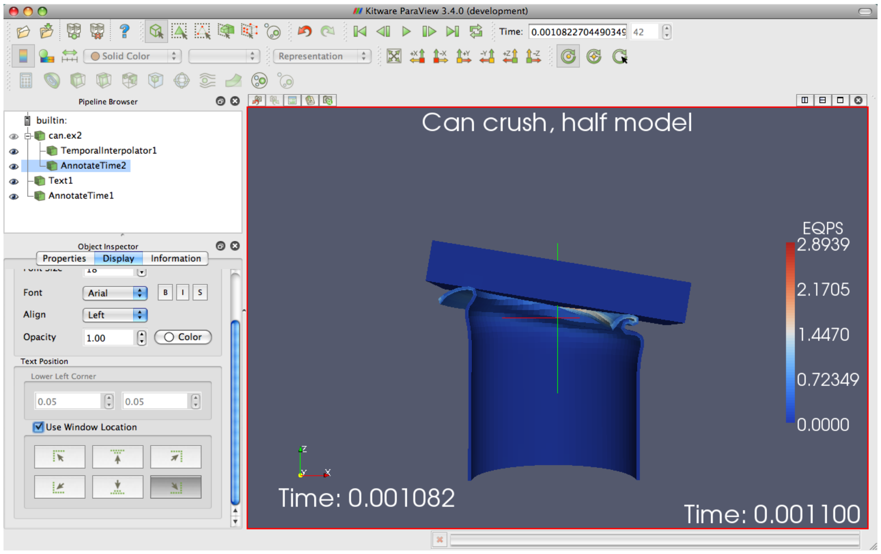


Format: Time: %f

There are instances when the current animation time is not the same as the time step read from a data file. Often it is important to know what the time stored in the data file is, and there is a special version of annotate time that acts as a filter.

5. Select can.ex2 in the pipeline browser.
6. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to apply the Annotate Time Filter.
7. Apply.
8. Move the annotation around as necessary.
9. Check the animation mode in the Animation View. If it is set to Snap to TimeSteps, change it to Real Time.
10. Play  and observe how the time annotation changes.

You can close the animation view. We are done with it for now, but we will revisit it again in Section 4.1.2.



Time: 0.0010822704490349 42

Font Arial
Align Left
Opacity 1.00
Text Position
Lower Left Corner 0.05 0.05
☑ Use Window Location

Time: 0.001082

Time: 0.001100

## Save Screenshot and Save Animation

One of the most important products of any visualization is screenshots and movies that can be used in presentations and reports. In this section we save a screenshot (picture) and animation (movie). Once again, we will use the can.ex2 dataset.

---

**Exercise 2.25 (Save Screenshot)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Open the file can.ex2, load all variables, Apply (see Exercise 2.19).
2. Press the  button to orient the camera to the mesh.
3. Color by points' ids. We use points' ids so that the 3D object has some color.
4. Select File → Save Screenshot .

This brings us to the file selection screen. If you pull down the menu Files of type: at the bottom of the dialog box, you will see several file types supported including portable network graphics (PNG), and joint photographic experts group (JPEG).

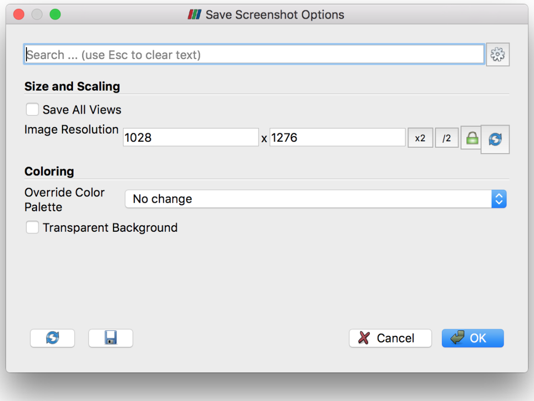
Select a File name for your file, and place it somewhere you can later find and delete. We usually recommend saving images as PNG files. The lossy compression of JPEG often creates noticeable artifacts in the images generated by **ParaView**, and the compression of PNG is better than most other raster formats.

5. Press the OK button.

After you select a file, you will be presented with a dialog box for more screenshot options.

The Save Screenshot Options dialog box includes numerous important controls.

At the top of the dialog box are Size and Scaling options. The Image Resolution options allow you to create an image that is larger (or smaller) than the current size of the 3D view. You can directly enter a resolution or use the x2 and /2 buttons to double or halve the resolution. The  button toggles locking the aspect ratio when selecting an image resolution. The  button restores the resolution to that of the view in the GUI. If you have multiple views open, there is also a Save All Views checkbox that toggles between saving only the active view and saving all views in one image.



Save Screenshot Options

Search ... (use Esc to clear text)

**Size and Scaling**

□ Save All Views

Image Resolution 1028 x 1276

**Coloring**

Override Color Palette: No change

□ Transparent Background

The Coloring controls modify how some elements, such as the background, get colored. The Override Color Palette pulldown menu allows a user to use the default color scheme, one with a white color motif for printing, or other defined palettes. These are the same palettes described in Exercise *[ex:ChangingTheColorPalette]* on page . There is also a Transparent Background checkbox that adds an alpha channel to the written image (in formats that support it).

There are also many advanced options that, as always are accessible via the  button or the search bar.

6. Press the OK button.

Using your favorite image viewer, find and load the image you created. If you have no image viewer, **ParaView** itself is capable of loading PNG files.

---

Next, we will save an animation.

---

**Exercise 2.26 (Save Animation)**

1. If you do not already have it loaded from the previous exercise, open the file can.ex2, load all variables, and Apply (see Exercise 2.19).
2. Select File → Save Animation .

This brings us to the file selection screen. If you pull down the menu Files of type: at the bottom of the dialog box, you will see the several file types including Ogg/Theora, AVI, JPEG, and PNG.

Select a File name for your file, and place it somewhere you can later find and delete. AVI will create a movie format that can be used on windows, and with some open source viewers. Ogg/Theora is used in many open source viewers. Otherwise, you can create a **flipbook**, or series of images. These images can be stitched together to form a movie using numerous open source tools. For now, try creating an AVI.

3. Press the OK button.

The Save Animation Options dialog box looks essentially the same as the Save Screenshot Options dialog for screenshots. If you look at the advanced options , you may note that there are some added Animation Options that allow you to control the frame rate and narrow the time steps that will be animated.

1. Press the OK button.

Using your favorite movie viewer, find and load the image you created.

---

### Selection

The goal of visualization is often to find the important details within a large body of information. **ParaView**'s selection abstraction is an important simplification of this process. Selection is the act of identifying a subset of some dataset. There are a variety of ways that this selection can be made, most of which are intuitive to end users, and a variety of ways to display and process the specific qualities of the subset once it is identified.

More specifically the subset identifies particular select points, cells, or blocks within any single dataset. There are multiple ways of specifying which elements to include in the selection including indentifier lists of multiple varieties, spatial locations, and scalar values and scalar ranges.

In **ParaView**, selection can take place at any time, and the program maintains a current selected set that is linked between all views. That is, if you select something in one view, that selection is also shown in all other views that display the same object.

The most direct means to create a selection is via the Find Data  panel. Launch this dialog from the toolbar or the Edit menu. From this dialog you can enter characteristics of the data that you are searching for. For example, you could look for points whose velocity magnitude is near terminal velocity. Or you could look for cells whose strain exceeds the failure of the material. The following exercise provides a quick example of using the Find Data  panel box.

---

**Exercise 2.27 (Performing Query-Based Selections)**

In this exercise we will find all cells with a large equivalent plastic strain (EQPS).

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Open the file can.ex2, load all variables, Apply (see Exercise 2.19).
2. Go to the last time step .
3. Open the find data panel  .
4. From the Data Producer combo box, choose can.ex2.
5. From the Element Type combo box, choose Cell.
6. In the third row of widgets, choose EQPS from the first combo box, is >= from the second combo box, and enter 1.5 in the final text box.
7. Click the Find Data button.

Observe the spreadsheet below the Find Data button that gets populated with the results of your query. Each row represents a cell and each column represents a field value or property (such as an identifier).

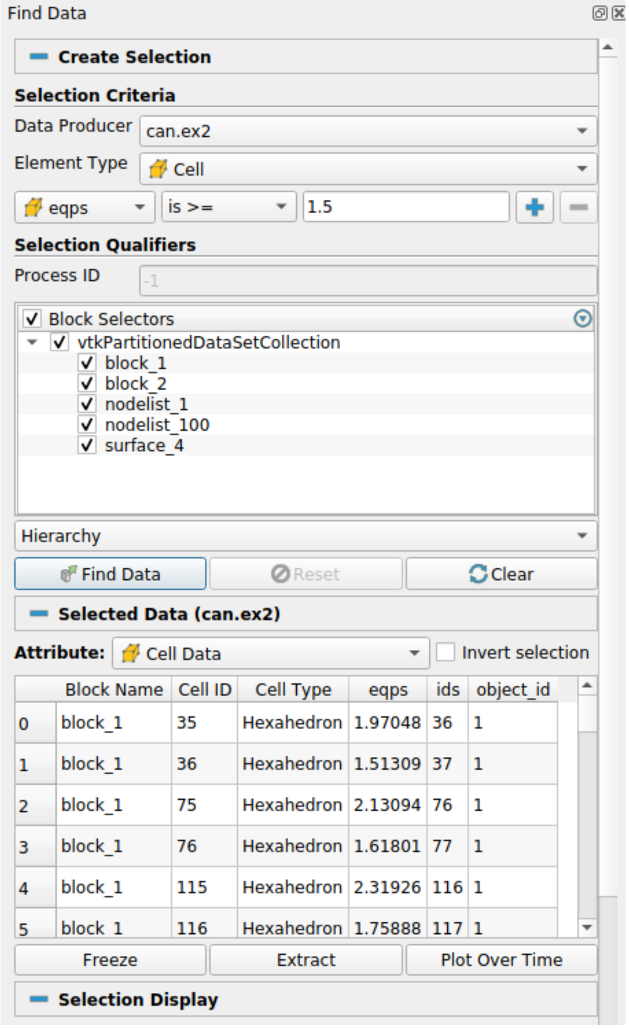
You may also notice that several cells are highlighted in the 3D view of the main **ParaView** window. These highlights represent the selection that your query created. Close the Find Data panel and note that the selection remains.

---

Another way of creating a selection is to pick elements right inside the 3D view. Most of the 3D view selections are performed with a **rubber-band selection**. That is, by clicking and dragging the mouse in the 3D view, you will create a boxed region that will select elements underneath it. There are also some 3D view selections that allow you to select within a polygonal region drawn on the screen. There are several types of interactive selections that can be performed, and you initiate one by selecting one of the icons in the small toolbar over the 3D view or using one of the shortcut keys. The following 3D selections are possible.

**Select Cells On (Surface)**
Selects cells that are visible in the view under a rubber band. (Shortcut: s)

**Select Points On (Surface)**
Selects points that are visible in the view under a rubber band. (Shortcut: d)



Data Producer: can.ex2

Element Type: Cell

eqps is >= 1.5

Process ID: -1

☑ Block Selectors

- ☑ vtkPartitionedDataSetCollection
  - ☑ block_1
  - ☑ block_2
  - ☑ nodelist_1
  - ☑ nodelist_100
  - ☑ surface_4

Attribute: Cell Data ☐ Invert selection

| | Block Name | Cell ID | Cell Type | eqps | ids | object_id |
|---|---|---|---|---|---|---|
| 0 | block_1 | 35 | Hexahedron | 1.97048 | 36 | 1 |
| 1 | block_1 | 36 | Hexahedron | 1.51309 | 37 | 1 |
| 2 | block_1 | 75 | Hexahedron | 2.13094 | 76 | 1 |
| 3 | block_1 | 76 | Hexahedron | 1.61801 | 77 | 1 |
| 4 | block_1 | 115 | Hexahedron | 2.31926 | 116 | 1 |
| 5 | block_1 | 116 | Hexahedron | 1.75888 | 117 | 1 |

**Select Cells Through (Frustum)**
Selects all cells that exist under a rubber band. (Shortcut: f)

**Select Points Through (Frustum)**
Selects all points that exist under a rubber band. (Shortcut: g)

**Select Cells With Polygon**
Like Select Cells On except that you draw a polygon by dragging the mouse rather than making a rubber-band selection.

**Select Points With Polygon**
Like Select Points On except that you draw a polygon by dragging the mouse rather than making a rubber-band selection.

**Select Blocks**
Selects blocks in a multiblock dataset. (Shortcut: b)

**Interactively Select Cells**
Enter an interactive selection mode where cells are highlighted as the mouse hovers over them and selected when clicked.

**Interactively Select Points**
Enter an interactive selection mode where points are highlighted as the mouse hovers near them and selected when clicked.

In addition to these selection modes, there are a hover point query mode and a hover cell query mode that allow you to quickly inspect the field values at a visible point by hovering the mouse over it. There is also a clear selection button that will remove the current selection.

Several of the selection modes have shortcut keys that allow you to make selections more quickly. Use them by placing the mouse cursor somewhere in the currently selected 3D view and hitting the appropriate key. Then click on the cell or block you want selected (or drag a rubber band over multiple elements).

Feel free to experiment with the selections now.

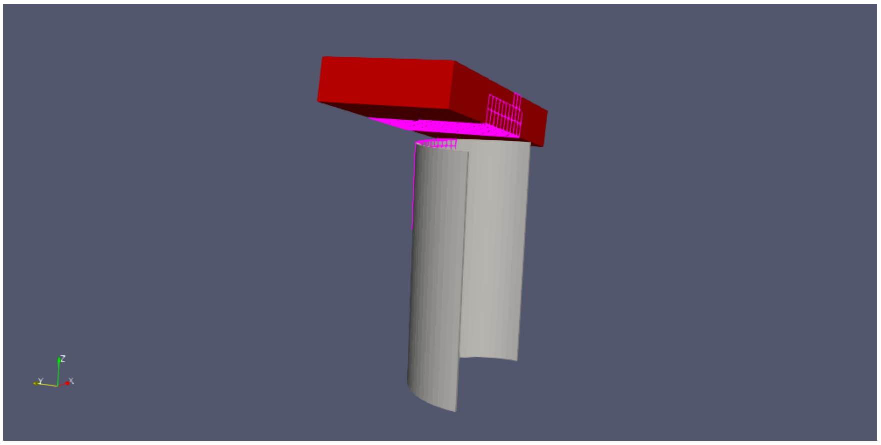
You can manage your selection with the Find Data panel even if the selection was created with one of these 3D interactions rather than directly with a find data query. The find data panel allows you to view all the points and cells in the selection as well as perform simple operations on the selection. These include inverting the selection (a checkbox just over the spreadsheet), adding labels (Exercise 2.29), freezing selections (Exercise 2.28), and shortcuts for the Plot Selection Over Time and Extract Selection filters (Exercise 2.30 and Exercise 2.31, respectively).

Experiment with selections in Find Data a bit. Open the Find Data panel. Then make selections using the rubber-band selection and see the results in the Find Data panel. Also experiment with altering the selection by inverting selections with the Invert selection checkbox.

It should be noted that selections can be internally represented in different ways. Some are recorded as a list of data element ids. Others are specified as a region in space or by query parameters. Although the selections all look the same, they can behave differently, especially with respect to changes in time. The following exercise demonstrates how these different selection mechanisms can behave differently.

**Exercise 2.28 (Data Element Selections vs. Spatial Selections)**

1. If you do not already have it loaded from the previous exercise, open the file can.ex2, load all variables, Apply (see Exercise 2.19).
2. Make a selection using the Select Cells Through tool.
3. If it is not already visible, show the Find Data panel.
4. Play the animation a bit. Notice that the region remains fixed and the selection changes based on what cells move in or out of the region.



5. Go to a timestep where some data are selected. In the Find Data panel, click the Freeze Selection button.
6. Play  again. Notice that the cells selected are fixed regardless of position.

In summary, a spatial selection (created with one of the select through tools) will re-perform the selection at each time step as elements move in and out of the selected region. Likewise, other queries such as field range queries will also re-execute as the data changes. However, when you select the Freeze Selection button, **ParaView** captures the identifiers of the currently selected elements so that they will remain the same throughout the animation.

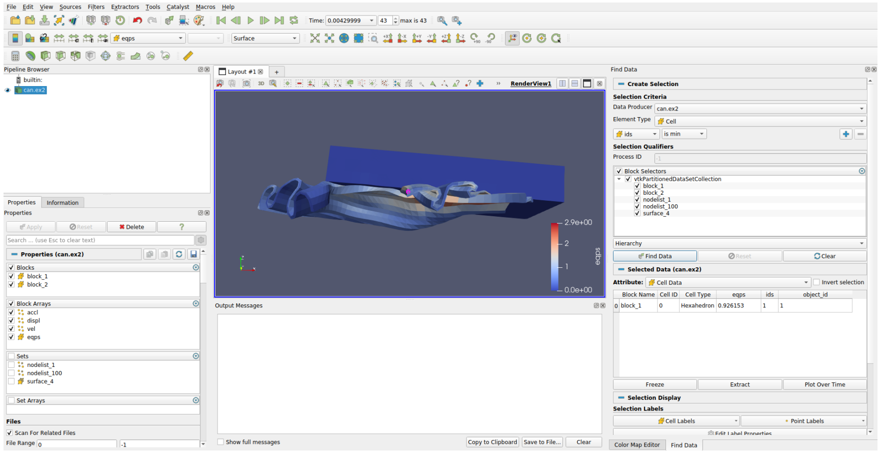
---

The spreadsheet in the find panel provides a readable way to inspect field data. However, sometimes it is helpful to place the field data directly in the 3D view. The next exercise describes how we can do that.

---

**Exercise 2.29 (Labeling Selections)**

1. If you do not already have it loaded from the previous exercise, open the file can.ex2, load all variables, Apply (see Exercise 2.19).
2. Go to the last time step .
3. If it is not already visible, show the Find Data  panel.
4. Using the controls at the top of the find data panel, create a selection where cells' ids is min. Click Find Data.
5. In the Cell Labels chooser, select EQPS.



Time: 0.00429999 | 43 | max is 43

eqps | Surface

Properties (can.ex2)

☑ Blocks
☑ block_1
☑ block_2

☑ Block Arrays
☑ accl
☑ displ
☑ vel
☑ eqps

□ Sets
□ nodelist_1
□ nodelist_100
□ surface_4

□ Set Arrays

Files
☑ Scan For Related Files
File Range 0 -1

Create Selection

Selection Criteria
Data Producer can.ex2
Element Type Cell
ids | is min

Selection Qualifiers
Process ID -1

☑ Block Selectors
☑ vtkPartitionedDataSetCollection
☑ block_1
☑ block_2
☑ nodelist_1
☑ nodelist_100
☑ surface_4

Hierarchy

Selected Data (can.ex2)

Attribute: Cell Data □ Invert selection

| | Block Name | Cell ID | Cell Type | eqps | ids | object_id |
|---|---|---|---|---|---|---|
| 0 | block_1 | 0 | Hexahedron | 0.926153 | 1 | 1 |

Selection Display

Selection Labels
Cell Labels | Point Labels

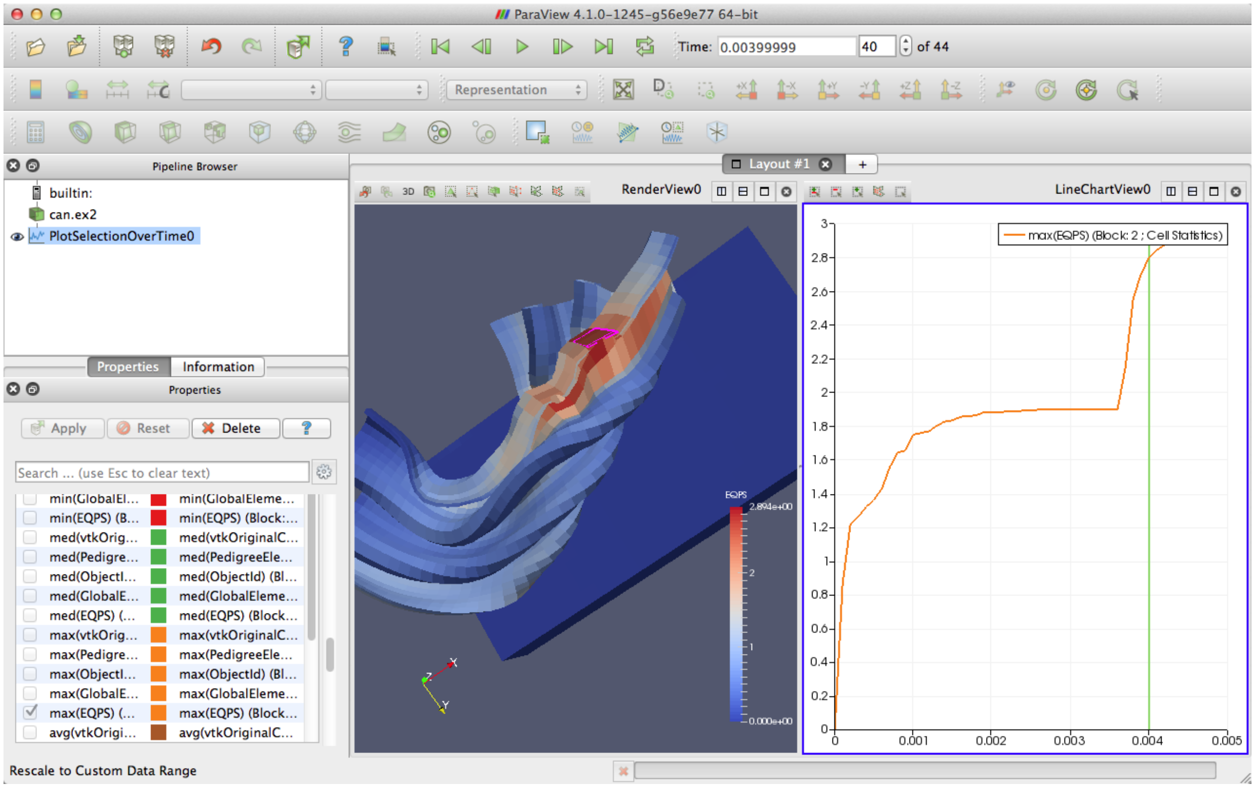
**ParaView** places the values for the EQPS field near the selected cell that contains that value. You should be able to see it in the 3D view. It is also possible to change the look of the font with respect to type, size, and color by clicking the  button to the right of the label choosers.

6. When you are done, turn off the labels by unchecking the entry in the Cell Labels chooser.

**ParaView** provides the ability to plot field data over time. These plots can work on a selection, and to make this easier the Find Data  panel contains convenience controls to create them.

**Exercise 2.30 (Plot Over Time)**

1. If you do not already have it loaded from the previous exercise, open the file can.ex2, load all variables, Apply (see Exercise 2.19).
2. If it is not already visible, show the Find Data  panel.
3. Using the controls at the top of the find data panel, create a selection where EQPS is max. Click Find Data.
4. Click the Plot Selection Over Time button at the bottom of the find data panel to add that filter. This filter is also easily accessible by the  toolbar button in the main **ParaView** window.
5. In the properties panel in the main **ParaView** window, click Apply.


ParaView 4.1.0-1245-g56e9e77 64-bit
Time: 0.00399999
40
of 44
Representation
Pipeline Browser
builtin:
can.ex2
PlotSelectionOverTime0
Properties
Information
Apply
Reset
Delete
Search ... (use Esc to clear text)
min(GlobalEl... min(GlobalEleme...
min(EQPS) (B... min(EQPS) (Block:...
med(vtkOrig... med(vtkOriginalC...
med(Pedigre... med(PedigreeEle...
med(ObjectI... med(ObjectId) (Bl...
med(GlobalE... med(GlobalEleme...
med(EQPS) (... med(EQPS) (Block...
max(vtkOrig... max(vtkOriginalC...
max(Pedigre... max(PedigreeEle...
max(ObjectI... max(ObjectId) (Bl...
max(GlobalE... max(GlobalEleme...
☑ max(EQPS) (... max(EQPS) (Block...
avg(vtkOrigi... avg(vtkOriginalC...
Layout #1
RenderView0
LineChartView0
EQPS
2.894e+00
2
1
0.000e+00
max(EQPS) (Block: 2 ; Cell Statistics)
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
0.001
0.002
0.003
0.004
0.005
Rescale to Custom Data Range

Note that the selection you had was automatically added as the selection to use in the Properties panel when the Plot Selection Over Time  filter was created. If you want to change the selection, simply make a new one and click Copy Active Selection in the Properties panel.

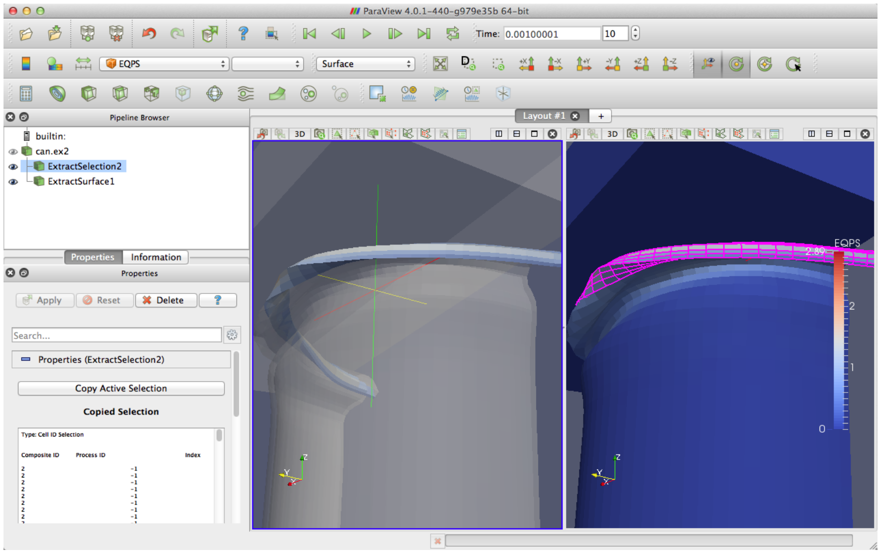
Also note that the plot was for the maximum EQPS value at each timestep, which could potentially be from a different cell at every timestep. If the desire is to identify a cell with a maximum value at some timestep and then plot that cell's value at every timestep, then use the Freeze Selection feature demonstrated in Exercise 2.28.

You can also extract a selection in order to view the selected points or cells separately or perform some independent processing on them. This is done through the Extract Selection  filter.

**Exercise 2.31 (Extracting a Selection)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Open the file can.ex2, load all variables, Apply (see Exercise 2.19).
2. Make a sizable cell selection for example, with Select Cells Through .
3. Create an Extract Selection  filter (available on the toolbar and from the find data panel).
4. Apply.



The object in the view is replaced with the cells that you just selected. (Note that in this image I added a translucent surface and a second view with the original selection to show the extracted cells in relation to the full data.) You can perform computations on the extracted cells by simply adding filters to the extract selection pipeline object.

As we described before, Views and Find Data  panel can be used to create different types of selections. To save and combine the created selections, you can use the `Selection Editor` panel. This panel can be accessed from View → Selection Editor.

**Exercise 2.32 (Selection Editor)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Open the file can.ex2, load all variables, Apply (see Exercise 2.19) and click .
2. Enable View → Selection Editor.

3. Use  to create a frustum cell selection at the bottom-left part of the can, and press ✚ in the Selection Editor to save the created selection. As you can see the saved selection has a name *s0*, and it's described as a frustum selection.

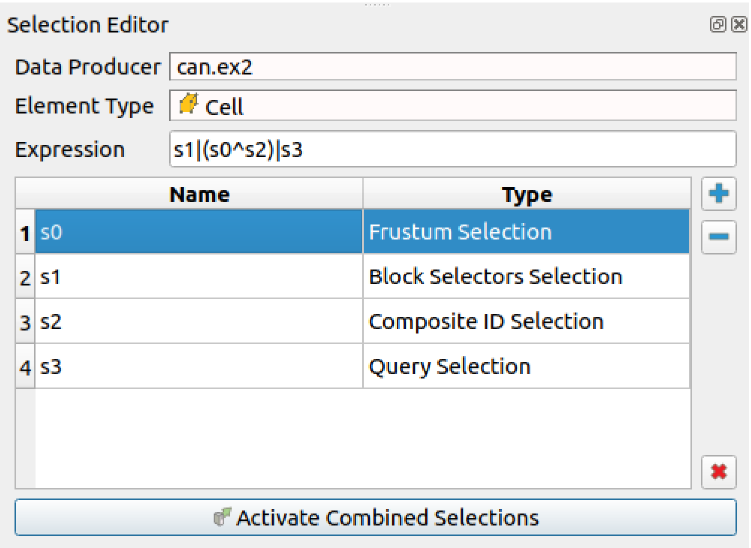
4. Use  to select the top block of the can and press ✚ in the Selection Editor to save the created selection.

5. On top of the frustum selection that you created previously, use  to create a slightly smaller polygon cell selection, and press ✚ in the Selection Editor to save the created selection.

   1. You can click on the previous Frustum selection in the Selection Editor to interactively visualize it.

6. Open the Find data panel , select can.ex2 as the Data Producer, select Cell as the Element Type, and set ID $>=$ 3600, press ✚ in the Find data panel and set ID $>=$ 3900. Finally, press ✚ in the Selection Editor to save the created selection.

7. The Selection Editor should now have 4 different selections, which can be combined using a boolean expression. Set Expression to $s1|(s0$ ^ $s2)|s3$.



Selection Editor

Data Producer: can.ex2

Element Type: Cell

Expression: s1|(s0^s2)|s3

| | Name | Type |
|---|---|---|
| 1 | s0 | Frustum Selection |
| 2 | s1 | Block Selectors Selection |
| 3 | s2 | Composite ID Selection |
| 4 | s3 | Query Selection |

8. Press *Activate Combined Selections*.

More information regarding the Selection Editor can be found in Section 1.6.8 of the User's Guide.

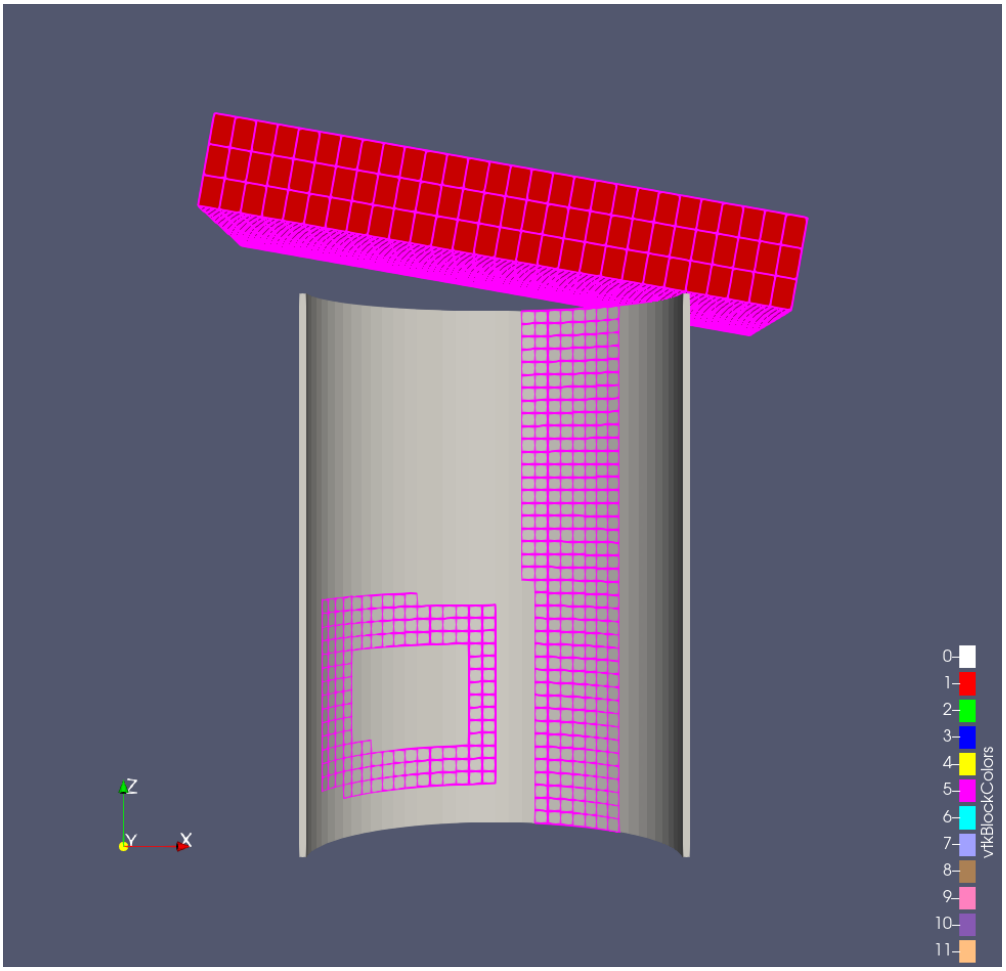
## Animations

**Notice**

This part mention the animation view. This was replaced by the time manager. While the interfaces are quite similar, some differences may exist. Please see the ParaView User's Guide 'Animation' section for more.

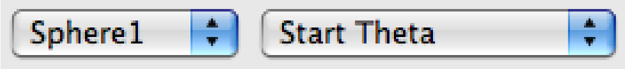
We have already seen how to animate a dataset with time in it (hit  ) and other ways to manipulate temporal data in Section 4.1.2. However, **ParaView**'s animation capabilities go far beyond that. With **ParaView** you can animate nearly any property of any pipeline object.

**Exercise 2.33 (Animating Properties)**

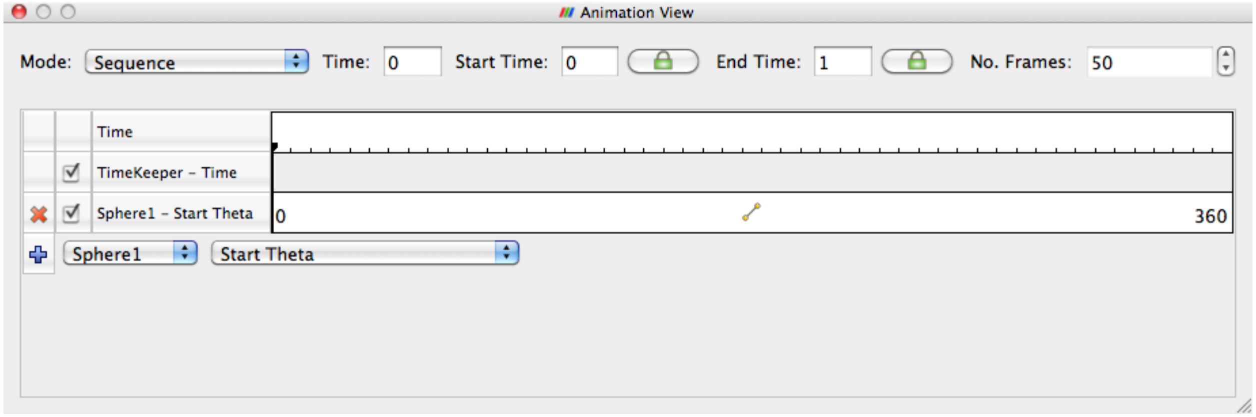

Z
Y
X
0
1
2
3
4
5
6
7
8
9
10
11
vtkBlockColors

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to create the Sphere source and Apply it.
2. If the animation view is not visible, make it so: View → Animation View.
3. Change the No. Frames option to 50 (10 will go far too quickly).
4. Find the property selection widgets at the bottom of the animation view and select Sphere1 in the first box and Start Theta in the second box.



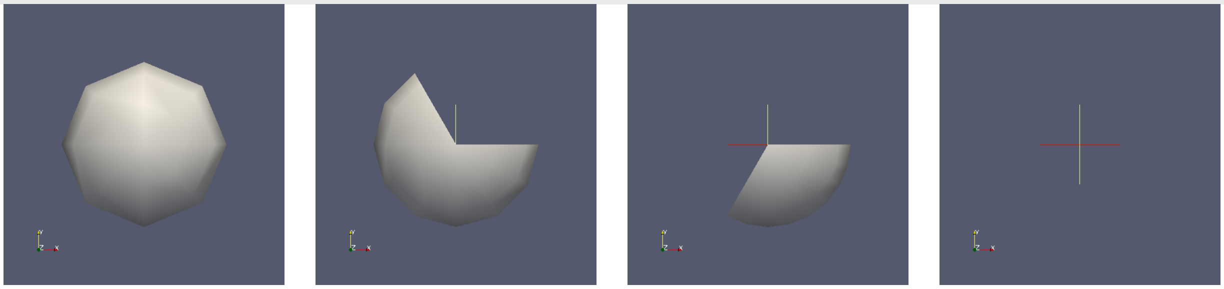
Hit the ✚ button.



Mode: Sequence Time: 0 Start Time: 0 End Time: 1 No. Frames: 50

| | | | |
|---|---|---|---|
| | | Time | |
| | ☑ | TimeKeeper – Time | |
| ✖ | ☑ | Sphere1 – Start Theta | 0 360 |
| ✚ | Sphere1 | Start Theta | |

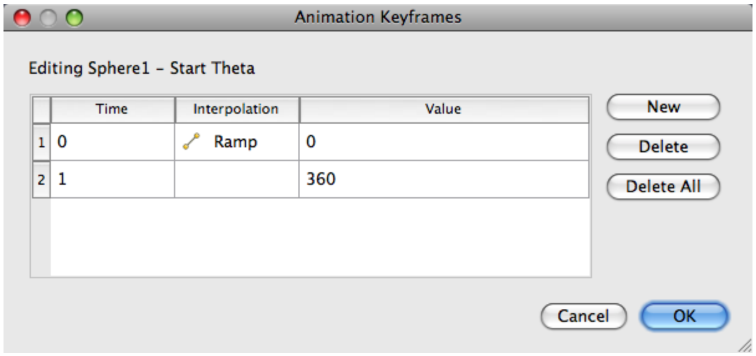
If you play  the animation, you will see the sphere open up then eventually wrap around itself and disappear.



What you have done is created a **track** for the Start Theta property of the Sphere1 object. A track is represented as horizontal bars in the animation view. They hold **key frames** that specify values for the property at a specific time instance. The value for the property is interpolated between the key frames. When you created a track two key frames were created automatically: a key frame at the start time with the minimal value and a key frame at the end time with the maximal value. The property you set here defines the start range of the sphere.

You can modify a track by double clicking on it. That will bring up a dialog box that you can use to add, delete, and modify key frames.

We use this feature to create a new key frame in the animation in the next exercise.



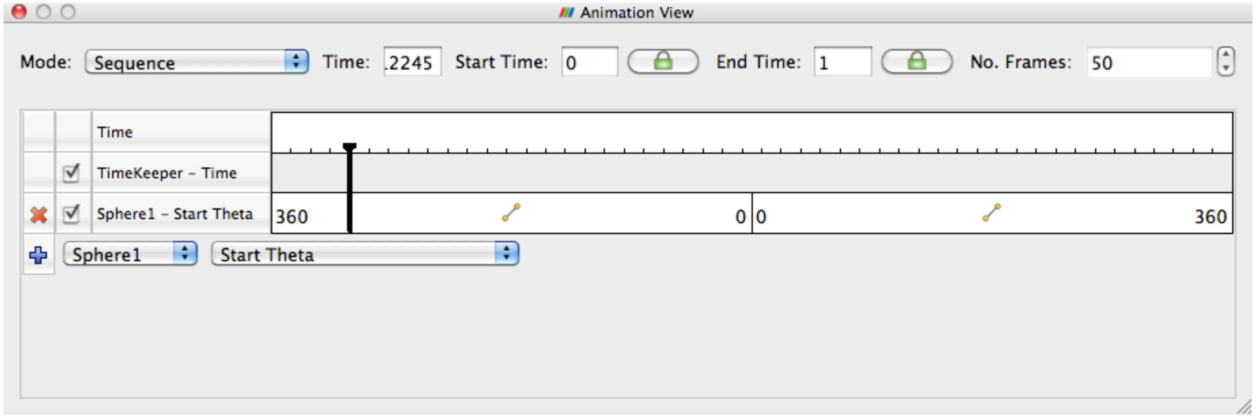
Editing Sphere1 – Start Theta

|  | Time | Interpolation | Value |
|---|---|---|---|
| 1 | 0 | Ramp | 0 |
| 2 | 1 |  | 360 |

**Exercise 2.34 (Modifying Animation Track Keyframes)**

This exercise is a continuation of Exercise 2.33. You will need to finish that exercise before beginning this one.

1. Double-click on the Sphere1 – Start Theta track.
2. In the Animation Keyframes dialog, click the New button. This will create a new key frame.
3. Modify the first key frame value to be 360.
4. Modify the second key frame time to be 0.5 and value to be 0.
5. Click OK.



Mode: Sequence  Time: .2245  Start Time: 0  End Time: 1  No. Frames: 50

Sphere1  Start Theta

When you play the animation, the sphere will first get bigger and then get smaller again.

You are not limited to animating just one property. You can animate any number of properties you wish. Now we will create an animation that depends on modifying two properties.

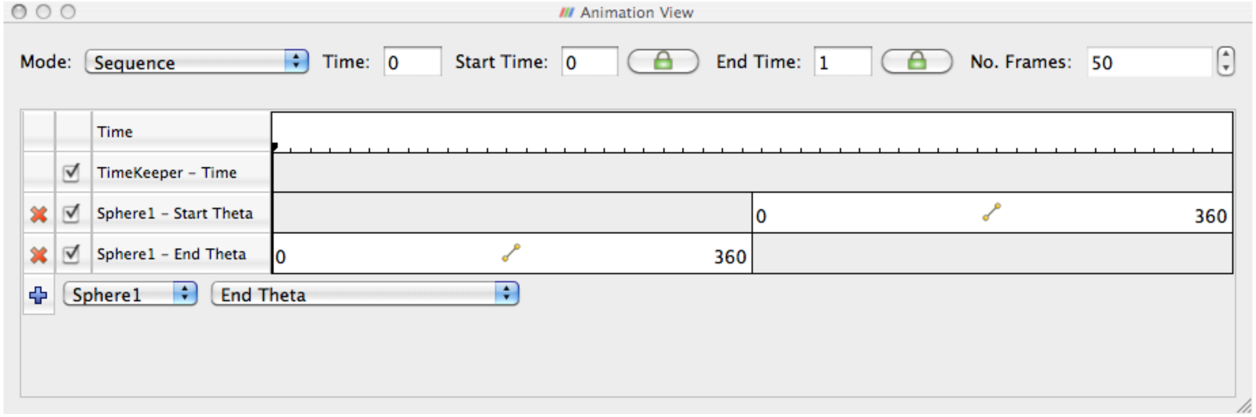
**Exercise 2.35 (Multiple Animation Tracks)**

This exercise is a continuation of Exercise 2.33 and Exercise 2.34. You will need to finish those exercises before beginning this one.

1. Double-click on the Sphere1 – Start Theta track.
2. In the Animation Keyframes dialog, Delete the first track (at time step 0).
3. Click OK.
4. In the animation view, create a track for the Sphere1 object, End Theta property.

5. Double-click on the Sphere1 – End Theta track.

6. Change the time for the second key frame to be 0.5.



Animation View

Mode: Sequence Time: 0 Start Time: 0 End Time: 1 No. Frames: 50

| | | | |
|---|---|---|---|
| | | Time | |
| | ☑ | TimeKeeper – Time | |
| ✖ | ☑ | Sphere1 – Start Theta | 0 … 360 (from 0.5 to 1) |
| ✖ | ☑ | Sphere1 – End Theta | 0 … 360 (from 0 to 0.5) |
| ✚ | Sphere1 | End Theta | |

The animation will show the sphere creating and destroying itself, but this time the range front rotates in the same direction. It makes for a very satisfying animation when you loop the animation.

In addition to animating properties for pipeline objects, you can animate the camera. **ParaView** provides methods for animating the camera along curves that you specify. The most common animation is to rotate the camera around an object, always facing the object, and **ParaView** provides a means to automatically generate such an animation.

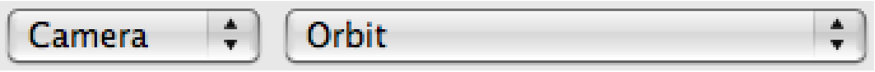
**Exercise 2.36 (Camera Orbit Animations)**

For this exercise, we will orbit the camera around whatever data you have loaded. If you are continuing from the previous exercise, you are set up. If not, simply load or create some data. To see the effects, it is best to have asymmetry in the geometry you load. can.ex2 is a good dataset to load for this exercise.

1. Place the camera where you want the orbit to start. The camera will move to the right around the viewpoint.

2. Make sure the animation view panel is visible (View → Animation View if it is not).

3. In the property selection widgets, select Camera in the first box and Orbit in the second box.



Camera Orbit

Hit the ✚ button.

Before the new track is created, you will be presented with a dialog box that specifies the parameters of the orbit. The default values come from the current camera position, which is usually what you want.

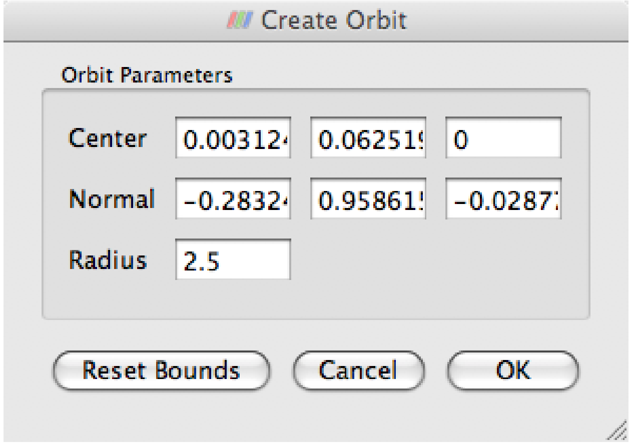
4. Click OK.

5. Play .

The camera will now animate around the object.

Another common camera annotation is to follow an object as it moves through space. Imagine a simulation of a traveling bullet or vehicle. If we hold the camera steady, then the object will quickly move out of view. To help with this situation, **ParaView** provides a special track that allows the camera to follow the data in the scene.

**Exercise 2.37 (Following Data in an Animation)**


Create Orbit

Orbit Parameters

Center 0.00312 0.06251 0

Normal -0.2832 0.95861 -0.0287

Radius 2.5

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

1. Open the file can.ex2, load all variables, Apply (see Exercise 2.19).
2. Press the  button to orient the camera to the mesh.
3. Make sure the animation view panel is visible (View → Animation View if it is not).
4. In the property selection widgets, select Camera in the first box and Follow Data in the second box.

Hit the ✚ button.

5. Play .

Note that instead of the can crushing to the bottom of the view, the animation shows the can lifted up to be continually centered in the image. This is because the camera is following the can down as it is crushed.

### 4.1.3 Batch Python Scripting

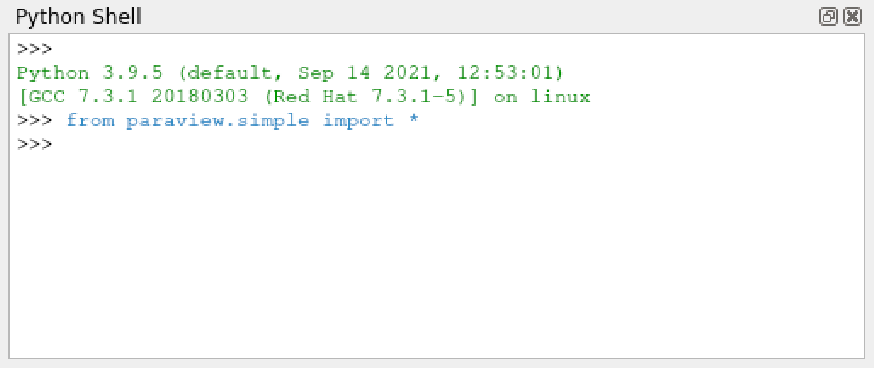
Python scripting can be leveraged in two ways within **ParaView**. First, Python scripts can automate the setup and execution of visualizations by performing the same actions as a user at the GUI. Second, Python scripts can be run inside pipeline objects, thereby performing parallel visualization algorithms. This chapter describes the first mode, batch scripting for automating the visualization.

Batch scripting is a good way to automate mundane or repetitive tasks, but it is also a critical component when using **ParaView** in situations where the GUI is undesired or unavailable. The automation of Python scripts allows you to leverage **ParaView** as a scalable parallel post-processing framework. We are also leveraging Python scripting to establish *in situ* computation within simulation code. (**ParaView** supports an *in situ* library called **Catalyst**, which is documented in Section 4.1.4 of this tutorial and more fully online at https://www.paraview.org/in-situ/).

This tutorial gives only a brief introduction to Python scripting. More comprehensive documentation on scripting is given in the **ParaView** User's Guide. There are also further links on **ParaView**'s documentation web page (https://www.paraview.org/documentation) including a complete reference to the **ParaView** Python API.

### Starting the Python Interpreter

There are many ways to invoke the Python interpreter. The method you use depends on how you are using the scripting. The easiest way to get a python interpreter, and the method we use in this tutorial, is to select View → Python Shell from the menu. This will bring up a dialog box containing controls for **ParaView**'s Python shell. This is the Python interpreter, where you directly control **ParaView** via the commands described later. For convenience in typing, **ParaView**'s Python shell supports tab completion and history browsing with the up and down keys.



```
>>>
Python 3.9.5 (default, Sep 14 2021, 12:53:01)
[GCC 7.3.1 20180303 (Red Hat 7.3.1-5)] on linux
>>> from paraview.simple import *
>>>
```

If you are most interested in getting started on writing scripts, feel free to skip to the next section past the discussion of the other ways to invoke scripting.

**ParaView** comes with two command line programs that execute Python scripts: `pvpython` and `pvbatch`. They are similar to the `python` executable that comes with Python distributions in that they accept Python scripts either from the command line or from a file and they feed the scripts to the Python interpreter.

The difference between `pvpython` and `pvbatch` is subtle and has to do with the way they establish the visualization service. `pvpython` is roughly equivalent to the `paraview` client GUI with the GUI replaced with the Python interpreter. It is a serial application that connects to a **ParaView** server (which can be either builtin or remote). `pvbatch` is roughly equivalent to `pvserver` except that commands are taken from a Python script rather than from a socket connection to a **ParaView** client. It is a parallel application that can be launched with `mpirun` (assuming it was compiled with MPI), but it cannot connect to another server; it is its own server. In general, you should use `pvpython` if you will be using the interpreter interactively and `pvbatch` if you are running in parallel.

It is also possible to use the **ParaView** Python modules from programs outside of **ParaView**. This can be done by pointing the `PYTHONPATH` environment variable to the location of the **ParaView** libraries and Python modules and pointing the `LD_LIBRARY_PATH` (on Unix/Linux), `DYLD_LIBRARY_PATH` (on Mac), or `PATH` (on Windows) environment variable to the **ParaView** libraries. Running the Python script this way allows you to take advantage of third-party applications such as Python's Integrated Development and Learning Environment (IDLE). For more information on setting up your environment, consult the **ParaView** Wiki.

### Tracing ParaView State

Before diving into the depths of the Python scripting features, let us take a moment to explore the automated facilities for creating Python scripts. The **ParaView** GUI's Python Trace feature allows one to very easily create Python scripts for many common tasks. To use Trace, one simply begins a trace recording via Start Trace, found in the Tools menu, and ends a trace recording via Stop Trace, also found in the Tools menu. This produces a Python script that reconstructs the actions taken in the GUI. That script contains the same set of operations that we are about to describe. As such, Trace recordings are a good resource when you are trying to figure out how to do some action via the Python interface, and conversely the following descriptions will help in understanding the contents of any given Trace script.

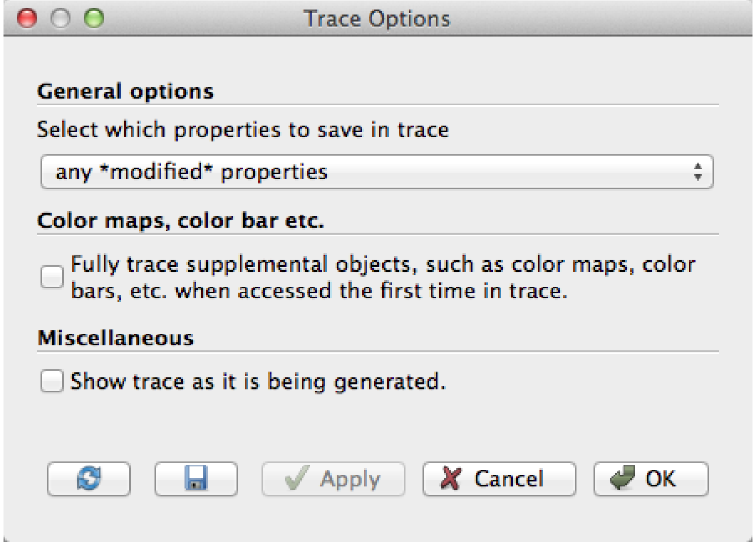
**Exercise 3.1 (Creating a Python Script Trace)**

If you have been following an exercise in a previous section, now is a good time to reset **ParaView**. The easiest way to do this is to select Edit → Reset Session from the menu or hit  on the toolbar.

1. Click the Start Trace option in the Tools menu.
2. A dialog box with options for the trace is presented. We will discuss the meaning of these options later. For now, just click OK.
3. Build a simple pipeline in the main **ParaView** GUI. For example, create a sphere source and then clip it.
4. Click Stop Trace in the Tools menu.
5. An editing window will open populated with a Python script that replicates the operations you just made.

Even if you have not been exposed to **ParaView**'s Python bindings, the commands being performed in the traced script should be familiar. Once saved to your hard drive, you can of course edit the script with your favorite editor. The final script can be interpreted by the `pvpython` or `pvbatch` program for totally automated visualization. It is also possible to run this script in the GUI. The Python Shell dialog has a Run Script button that invokes a saved script.

It should be noted that there is also a way to capture the current **ParaView** state as a Python script without tracing actions. Simply select Save State… from the **ParaView** File menu and choose to save as a Python .py state file (as opposed to a **ParaView** .pvsm state file). We will not have an exercise on state Python scripts, but suffice it to say they can be used in much the same way as traced Python scripts. You are welcome to experiment with this feature as you like.



Trace Options

General options

Select which properties to save in trace

any *modified* properties

Color maps, color bar etc.

□ Fully trace supplemental objects, such as color maps, color bars, etc. when accessed the first time in trace.

Miscellaneous

□ Show trace as it is being generated.

As noted earlier in the exercise, Python tracing has some options that are presented in a dialog box before the tracing starts. The first option selections what properties are saved to the trace. Some properties you explicitly set through the GUI, such as a value entered in a GUI widget. Some properties are set internally by the **ParaView** application, such as the initial position of a clip plane based on the bounds of the object it is being applied to. Many other properties are left at some default value. You can choose one of the following classes of properties to save:

**all properties**
: Traces the values of all properties even if they remain at the default. This can be helpful to introspect all possible properties or to ensure a consistent state regardless of the settings for other users. This also yields a very verbose output that can be hard to read.

**any *modified* properties**
: Ignores any properties that do not change from their defaults. This is a good option for most use cases.

**only *user-modified* properties**

Ignores any properties that are not explicitly set by the user. Traces of this nature rely on any internally set properties being reapplied when the script is run.

The next option has to do with supplemental objects that are managed by the **ParaView** GUI (or client) rather than in the server's state. Check this box to capture all of the state associated with these objects, which includes color maps, color bars, and other annotation.

Finally, **ParaView** provides the option to show the trace file as it is being generated. This can be a helpful option to use when learning what Python commands can be used to replicate particular actions in the **ParaView** GUI.

### Macros

A simple but powerful way to customize the behavior of **ParaView** is to add your Python script as a **macro**. A macro is simply an automated script that can be invoked through its button in a toolbar or its entry in the menu bar. Any Python script can be assigned to a macro.

**Exercise 3.2 (Adding a Macro)**

This exercise is a continuation of Exercise 3.1. You will need to finish that exercise before beginning this one. You should have the editing window containing the Python script created in Exercise 3.1 open.

1. In the menu bar (of the editing window), select File → Save As Macro. . . .
2. Choose a descriptive name for the macro file and save it in the default directory provided by the browser. You should now see your macro on the Macro toolbar at the top of the **ParaView** GUI.

At this point, you should now see your macro added to the toolbars. By default, macro toolbar buttons are placed in the middle row all the way to the right. If you are short on space in your GUI, you may need to move toolbars around to see it. You will also see that your macro has been added to the Macros menu.

3. Close the Python editor window.
4. Delete the pipeline you have created by either selecting Edit → Delete All from the menu or selecting  from the toolbar.
5. Activate your macro by clicking on the toolbar button or selecting it in the Macros menu.

In this example our macro created something from scratch. This is helpful if you often load some data in the same way every time. You can also trace the creation of filters that are applied to existing data. A macro from a trace of this nature allows you to automate the same visualization on different data.

### Creating a Pipeline

As described in the previous two sections, the **ParaView** GUI's Python Trace feature provides a simple mechanism to create scripts. In this section we will begin to describe the basic bindings for **ParaView** scripting. This is important information in building Python scripts, but you can always fall back on producing traces with the GUI.

The first thing any **ParaView** Python script must do is load the `paraview.simple` module. This is done by invoking

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

In general, this command needs to be invoked at the beginning of any **ParaView** batch Python script. This command is automatically invoked for you when you bring up the scripting dialog in **ParaView**, but you must add it yourself when using the Python interpreter in other programs (including `pvpython` and `pvbatch`).

The paraview.simple module defines a function for every source, reader, filter, and writer defined in **ParaView**. The function will be the same name as shown in the GUI menus with spaces and special characters removed. For example, the Sphere function corresponds to Sources → Sphere in the GUI and the PlotOverLine function corresponds to Filters → Data Analysis → Plot Over Line. Each function creates a pipeline object, which will show up in the pipeline browser (with the exception of writers), and returns an object that is a **proxy** that can be used to query and manipulate the properties of that pipeline object.

There are also several other functions in the paraview.simple module that perform other manipulations. For example, the pair of functions Show and Hide turn on and off, respectively, the visibility of a pipeline object in a view. The Render function causes a view to be redrawn.

To obtain a concise list of the functions available in paraview.simple, invoke dir(paraview.simple). Alternatively, as explained in Section 4.1.3 you can get a verbose listing via help(paraview.simple).

**Exercise 3.3 (Creating and Showing a Source)**

If you have been following an exercise in a previous section, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar.

If you have not already done so, open the Python shell in the **ParaView** GUI by selecting View → Python Shell from the menu. You will notice that the following has been added for you already.

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

Create and show a Sphere source by typing the following in the Python shell.

```python
sphere = Sphere()
Show()
Render()
ResetCamera()
```

The Sphere command creates a sphere pipeline object. Once it is executed you will see an item in the pipeline browser created. We save a proxy to the pipeline object in the variable sphere. We are not using this variable (yet), but it is good practice to save references to your pipeline objects.

The subsequent Show command turns on visibility of this object in the view, and the subsequent Render causes the results to be seen. Finally, although the **ParaView** GUI automatically adjusts the camera to data shown for the first time, the Python scripting does not. The call to ResetCamera performs this automatic camera adjustment if necessary.

At this point you can interact directly with the GUI again. Try changing the camera angle in the view with the mouse.

**Exercise 3.4 (Creating and Showing a Filter)**

Creating filters is almost identical to creating sources. By default, the last created pipeline object will be set as the input to the newly created filter, much like when creating filters in the GUI.

This exercise is a continuation of Exercise 3.3. You will need to finish that exercise before beginning this one.

Type in the following script in the Python shell that hides the sphere and then adds the shrink filter to the sphere and shows that.

```python
Hide()
shrink = Shrink()
Show()
Render()
```

The sphere should be replaced with the output of the Shrink filter, which makes all of the polygons smaller to give the mesh an exploded type of appearance.

So far as we have built pipelines, we have accepted the default parameters for the pipeline objects. As we have seen in the exercises of Section 4.1.2, it is common to have to modify the parameters of the objects using the properties panel.

In Python scripting, we use the **proxy** returned from the creation functions to manipulate the pipeline objects. These proxies are in fact Python objects with class attributes that correspond to the same properties you set in the properties panel. They have the same names as those in the properties panel with spaces and other illegal characters removed. Use `dir(variable)` or `help(variable)` to get a list of all attributes on any variable that you have access to. In most cases, simply assign values to an object's attributes in order to change them.

**Exercise 3.5 (Changing Pipeline Object Properties)**

This exercise is a continuation of Exercise 3.3 and Exercise 3.4. You will need to finish those exercises before beginning this one.

Recall that we have so far created two Python variables, `sphere` and `shrink`, that are proxies to the corresponding pipeline objects. First, enter the following command into the Python shell to get a concise listing of all attributes of the sphere.

```
dir(sphere)
```

Next, enter the following command into the Python shell to get the current value of the Theta Resolution property of the sphere.

```
print(sphere.ThetaResolution)
```

The Python interpreter should respond with the result `8`. (Note that using the `print` function, which instructs Python to output the arguments to standard out, is superfluous here as the Python shell will output the result of any command anyway.) Let us double the number of polygons around the equator of the sphere by changing this property.

```
sphere.ThetaResolution = 16
Render()
```

The shrink filter has only one property, Shrink Factor. We can adjust this factor to make the size of the polygons larger or smaller. Let us change the factor to make the polygons smaller.

```
shrink.ShrinkFactor = 0.25
Render()
```

You may have noticed that as you type in Python commands to change the pipeline object properties, the GUI in the properties panel updates accordingly.

So far we have created only non-branching pipelines. This is a simple and common case and, like many other things in the `paraview.simple` module, is designed to minimize the amount of work for the simple and common case but also provide a clear path to the more complicated cases. As we have built the non-branching pipeline, **ParaView** has automatically connected the filter input to the previously created object so that the script reads like the sequence of operations it is. However, if the pipeline has branching, we need to be more specific about the filter inputs.

**Exercise 3.6 (Branching Pipelines)**

This exercise is a continuation of Exercise 3.3 through Exercise 3.5. You will need to finish Exercise 3.3 and Exercise 3.4 before beginning this one (Exercise 3.5 is optional).

Recall that we have so far created two Python variables, `sphere` and `shrink`, that are proxies to the corresponding pipeline objects. We will now add a second filter to the sphere source that will extract the wireframe from it. Enter the following in the Python shell.

```
wireframe = ExtractEdges(Input=sphere)
Show()
Render()
```

An Extract Edges filter is added to the sphere source. You should now see both the wireframe of the original sphere and the shrunken polygons.

Notice that we explicitly set the input for the Extract Edges filter by providing `Input=sphere` as an argument to the `ExtractEdges` function. What we are really doing is setting the `Input` property upon construction of the object. Although it would be possible to create the object with the default input, and then set the input later, it is not recommended. The problem is that not all filters accept all input. If you initially create a filter with the wrong input, you could get error messages before you get a chance to change the `Input` property to the correct input.

The sphere source having two filters connected to its output is an example of **fan out** in the pipeline. It is always possible to have multiple filters attached to a single output. Some filters, but not all, also support having multiple filters connected to their input. Multiple filters are attached to an input is known as **fan in**. In **ParaView**'s Python scripting, fan in is handled much like fan out, by explicitly defining a filter's inputs. When setting multiple inputs (on a single port), simply set the input to a list of pipeline objects rather than a single one.

**Did you know?**

Filters that have multiple input ports, like `ResampleWithDataset`, use different names to distinguish amongst the input properties instead. The ports are typically called `Input` and `Source` but consult `Trace` or `help` to be sure.

For example, let us group the results of the shrink and extract edges filters using the Group Datasets filter. Type the following line in the Python shell.

```
group = GroupDatasets(Input=[shrink, wireframe])
Show()
```

There is now no longer any reason for showing the shrink and extract edges filters, so let us hide them. By default, the `Show` and `Hide` functions operate on the last pipeline object created (much like the default input when creating a filter), but you can explicitly choose the object by giving it as an argument. To hide the shrink and extract edges filters, type the following in the Python shell.

```
Hide(shrink)
Hide(wireframe)
Render()
```

In the previous exercise, we saw that we could set the `Input` property by placing `Input=({input object})` in the arguments of the creator function. In general we can set any of the properties at object construction by specifying `{property name}={property value}`. For example, we can set both the `Theta Resolution` and `Phi Resolution` when we create a sphere with a line like this.

```
sphere = Sphere(ThetaResolution=360, PhiResolution=180)
```

### Active Objects

If you have any experience with the **ParaView** GUI, then you should already be familiar with the concept of an active object. As you build and manipulate visualizations within the GUI, you first have to select an object in the pipeline browser. Other GUI panels such as the properties panel will change based on what the active object is. The active object is also used as the default object to use for some operations such as adding a filter.

The batch Python scripting also understands the concept of the active object. In fact, when running together, the GUI and the Python interpreter share the same active object. When you created filters in the previous section, the default input they were given was actually the active object. When you created a new pipeline object, that new object became the active one (just like when you create an object in the GUI).

You can get and set the active object with the `GetActiveSource` and `SetActiveSource` functions, respectively. You can also get a list of all pipeline objects with the `GetSources` function. When you click on a new object in the GUI pipeline browser, the active object in Python will change. Likewise, if you call `SetActiveSource` in python, you will see the corresponding entry become highlighted in the pipeline browser.

**Exercise 3.7 (Experiment with Active Pipeline Objects)**

This exercise is a continuation of the exercises in the previous section. However, if you prefer you can create any pipeline you want and follow along.

Play with active objects by trying the following.

- Get a list of objects by calling `GetSources()`. Find the sources and filters you created in that list.
- Get the active object by calling `GetActiveSource()`. Compare that to what is selected in the pipeline browser.
- Select something new in the pipeline browser and call `GetActiveSource()` again.
- Change the active object with the `SetActiveSource()` function. You can use one of the proxy objects you created earlier as an argument to `SetActiveSource`. Observe the change in the pipeline browser.

In addition to maintaining an active pipeline object, **ParaView** Python scripting also maintains an active view. As a **ParaView** user, you should also already be familiar with multiple views and the active view. The active view is marked in the GUI with a blue border. The Python functions `GetActiveView` and `SetActiveView` allow you to query and change the active view. As with pipeline objects, the active view is synchronized between the GUI and the Python interpreter.

### Online Help

This tutorial, as well as similar instructions in the **ParaView** book and Wiki, is designed to give the key concepts necessary to understand and create batch Python scripts. The detailed documentation including complete lists of functions, classes, and properties available is maintained by the **ParaView** build process and provided as online help from within the **ParaView** application. In this way we can ensure that the documentation is up to date for whatever version of **ParaView** you are using and that it is easily accessible.

The **ParaView** Python bindings make use of the `help` built-in function. This function takes as an argument any Python object and returns some documentation on it. For example, typing

```
help(paraview.simple)
```

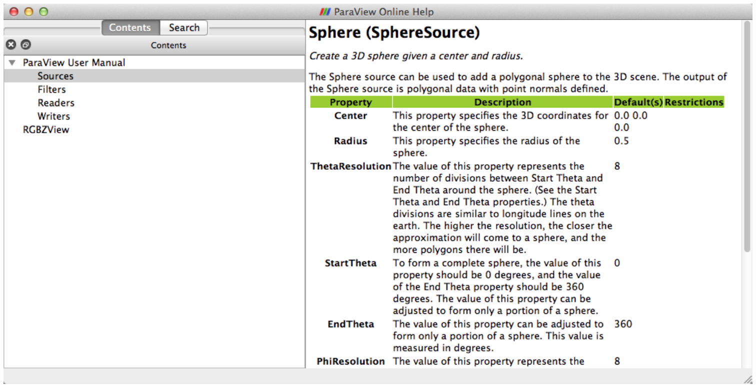
returns a brief description of the module and then a list of all the functions included in the module with a brief synopsis of what each one does. For example

```
help(Sphere)
sphere = Sphere()
help(sphere)
```

will first give help on the Sphere function, then use it to create an object, and then give help on the object that was returned (including a list of all the properties for the proxy).

Most of the widgets displayed in the properties panel's Properties group are automatically generated from the same introspection that builds the Python classes. (There are a small number of exceptions where a custom panel was created for better usability.) Thus, if you see a labeled widget in the properties panel, there is a good chance that there is a corresponding property in the Python object with the same name.

Regardless of whether the GUI contains a custom panel for a pipeline object, you can still get information about that object's properties from the GUI's online help. As always, bring up the help with the  toolbar button. You can find documentation for all the available pipeline objects under the Sources, Filters, Readers, and Writers entries in the help Contents. Each entry gives a list of objects of that type. Clicking on any one of the objects gives a list of the properties you can set from within Python.



**Sphere (SphereSource)**

*Create a 3D sphere given a center and radius.*

The Sphere source can be used to add a polygonal sphere to the 3D scene. The output of the Sphere source is polygonal data with point normals defined.

| Property | Description | Default(s) | Restrictions |
|---|---|---|---|
| **Center** | This property specifies the 3D coordinates for the center of the sphere. | 0.0 0.0 0.0 | |
| **Radius** | This property specifies the radius of the sphere. | 0.5 | |
| **ThetaResolution** | The value of this property represents the number of divisions between Start Theta and End Theta around the sphere. (See the Start Theta and End Theta properties.) The theta divisions are similar to longitude lines on the earth. The higher the resolution, the closer the approximation will come to a sphere, and the more polygons there will be. | 8 | |
| **StartTheta** | To form a complete sphere, the value of this property should be 0 degrees, and the value of the End Theta property should be 360 degrees. The value of this property can be adjusted to form only a portion of a sphere. | 0 | |
| **EndTheta** | The value of this property can be adjusted to form only a portion of a sphere. This value is measured in degrees. | 360 | |
| **PhiResolution** | The value of this property represents the | 8 | |

## Reading from Files

The equivalent to opening a file in the **ParaView** GUI is to create a reader in Python scripting. Reader objects are created in much the same way as sources and filters; paraview.simple has a function for each reader type that creates the pipeline object and returns a proxy object. One can instantiate any given reader directly as described below, or more simply call reader = OpenDataFile({filename})

All reader objects have at least one property (hidden in the GUI) that specifies the file name. This property is conventionally called either FileName or FileNames. You should always specify a valid file name when creating a reader by placing something like FileName={full path} in the arguments of the construction object. Readers often do not initialize correctly if not given a valid file name.

**Exercise 3.8 (Creating a Reader)**

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset **ParaView**. The easiest way to do this is to select  from the toolbar. You will also need the Python shell. If you have not already done so, open it with View → Python Shell from the menu.

In this exercise we are loading the disk_out_ref.ex2 file from the Python shell. Locate this file on your computer and be ready to type or copy it into the Python shell. We will reference it as {path}/disk_out_ref.ex2. You can use the file browser to help you locate this file. (Click on the Examples quick access directory and observe where the file browser takes you.) Common paths for the file are as follows:

**Mac**

```
/Applications/ParaView-x.x.x.app/Contents/examples
```

Windows

```
C:/Program Files/ParaView x.x.x/examples
```

Linux

```
/usr/local/lib/paraview-x.x.x/examples
```

Create the reader while specifying the file name by entering the following in the Python shell.

```
reader = OpenDataFile('{path}/disk_out_ref.ex2')
Show()
Render()
ResetCamera()
```

### Querying Field Attributes

In addition to having properties specific to the class, all proxies for pipeline objects share a set of common properties and methods. Two very important such properties are the `PointData` and `CellData` properties. These properties act like **dictionaries**, an associative array type in Python, that maps variable names (in strings) to `ArrayInformation` objects that hold some characteristics of the fields. Of particular note are the `ArrayInformation` methods `GetName`, which returns the name of the field, `GetNumberOfComponents`, which returns the size of each field value (1 for scalars, more for vectors), and `GetRange`, which returns the minimum and maximum values for a particular component.

**Exercise 3.9 (Getting Field Information)**

This exercise is a continuation of Exercise 3.8. You will need to finish that exercise before beginning this one.

To start with, get a handle to the point data and print out all of the point fields available.

```
pd = reader.PointData
print(pd.keys())
```

Get some information about the “pres” and “v” fields.

```
print(pd["pres"].GetNumberOfComponents())
print(pd["pres"].GetRange())
print(pd["v"].GetNumberOfComponents())
```

Now let us get fancy. Use the Python `for` construct to iterate over all of the arrays and print the ranges for all the components.

```
for ai in pd.values():
   print(ai.GetName(), ai.GetNumberOfComponents(), end=" ")
   for i in range(ai.GetNumberOfComponents()):
      print(ai.GetRange(i), end=" ")
   print()
```

### Representations

Representations are the "glue" between the data in a pipeline object and a view. The representation is responsible for managing how a dataset is drawn in the view. The representation defines and manages the underlying rendering objects used to draw the data as well as other rendering properties such as coloring and lighting. Parameters made available in the Display group of the GUI are managed by representations. There is a separate representation object instance for every pipeline-object–view pair. This is so that each view can display the data differently.

Representations are created automatically by the GUI. In python scripting they are created with the `Show` function instead. In fact `Show` returns a proxy to the representation. Therefore you can save `Show`'s return value in a variable as we've done above for sources, filters and readers. If you neglect to save it, you can always get it back with the `GetRepresentation` function. With no arguments, this function will return the representation for the active pipeline object and the active view. You can also specify a pipeline object or view or both.

**Exercise 3.10 (Coloring Data)**

This exercise is a continuation of Exercise 3.8 (and optionally Exercise 3.9). If you do not have the exodus file open, you will need to finish Exercise 3.8 before beginning this one.

Let us change the color of the geometry to blue and give it a very pronounced specular highlight (that is, make it really shiny). Type in the following into the Python shell to get the representation and change the material properties.

```
readerRep = GetRepresentation()
readerRep.DiffuseColor = [0, 0, 1]
readerRep.SpecularColor = [1, 1, 1]
readerRep.SpecularPower = 128
readerRep.Specular = 1
Render()
```

Now rotate the camera with the mouse in the GUI to see the effect of the specular highlighting.

We can also use the representation to color by a field variable. There is actually quite a bit of state that must be set to change field variables. The `ColorBy` function provides a simple interface to color a mesh by a field value. A subsequent call to `UpdateScalarBars` also updates the color bar annotation. Enter the following into the Python shell to color the mesh by the "pres" field variable.

```
ColorBy(readerRep, ("POINTS", "pres"))
UpdateScalarBars()
Render()
```

### Views

Drawing areas or windows are called Views in **ParaView**. As with readers, sources, filters, and representations, views are wrapped into python objects and these can be created, obtained and controlled via scripts.

Views are usually created for you by the GUI, but in python you have to create views more intentionally. The most convenient way to do so is to rely on the way that `Render` returns a view, creating one first if necessary. If you prefer, you can create specific view types via `CreateView('{viewname}')` or `CreateRenderView`, `CreateXYPlotView` and the like. However you make them, call `GetRenderViews` to get a list of all Views, or `GetActiveView` get access to the currently active view. There is also a function named `GetRenderView` (no 's' on the end) that gets the active view if there is one or creates a new one if there is no active view.

Once you have a view you have access to all of the properties that you see on the View group of the GUI. For instance you can easily turn on and off the orientation widget, change the background color, alter the lighting and more. Besides these first level properties, the view also gives you access to other scene wide controls such as the camera, animation time, and when not running alongside the GUI, the view's size.

**Exercise 3.11 (Controlling the View)**

This exercise is a continuation of Exercise 3.8 (and optionally Exercise 3.9, and Exercise 3.10. If you do not have the exodus file open, you will need to finish Exercise 3.8 before beginning this one.

Let us change the background color of the scene from **ParaView**'s default gray to a nice gradient instead. Type the following into the Python shell to get a hold of the View and change it.

```
view = GetActiveView()
view.Background = [0, 0, 0]
view.Background2 = [0, 0, 0.6]
view.UseGradientBackground = True
Render()
```

Next, let us ask the view what position the camera is sitting at, and then move it within a for loop to create a short animation.

```
x, y, z = view.CameraPosition
print(x, y, z)
for iter in range(0,10):
   x = x + 1
   y = y + 1
   z = z + 1
   view.CameraPosition = [x, y, z]
   print(x, y, z)
   Render()
```

### Saving Results

Within a script it is easy to save out results, and by saving your data and your scripts it becomes easy to create reproducible visualization with **ParaView**.

As within the GUI, there are several products that you might like to save out when you are working with **ParaView**.

- To save out the data produced by a filter, add a writer to the filter with the desired filename using the `CreateWriter` function and then update the writer proxy with the `UpdatePipeline` method. This is analogous to clicking on a pipeline element and selecting File → Save Data.
- Saving images is as simple as typing `SaveScreenshot('{path}/filename.still_extension')`.
- Assuming your **ParaView** is linked to an encoder and codecs, saving compressed animations is as simple as typing `WriteAnimation('{path}/filename.animation_extension')`.

In all cases **ParaView** uses the file name extension to determine the specific file type to create.

**Exercise 3.12 (Save Results)**

This exercise is a continuation of Exercise 3.8 (and optionally Exercise 3.9 through Exercise 3.11). If you do not have the exodus file open, you will need to finish Exercise 3.8 before beginning this one.

Let us first probe the data to get something compact out of it. Then we will save out the result of the probe in the form of a comma separated values file so that we can look at it in a text editor and import it into any other tool we choose.

```
plot = PlotOverLine()
plot.Source.Point1 = [0, 0, 0]
plot.Source.Point2 = [0, 0, 10]
writer = CreateWriter("{path}/plot.csv")
writer.UpdatePipeline()
```

Next, lets create a LineChartView to show the plot in and then save out a screenshot of our results.

```
plotView = FindViewOrCreate("MyView", viewtype="XYChartView")
Show(plot)
Render()
SaveScreenshot("{path}/plot.png")
```

As you can see, **ParaView**'s scripting interface is quite powerful, and once you know the fundamentals and are familiar with Python's syntax, it is fairly easy to get up and running with it. We have just touched on the higher level aspects of **ParaView** scriptability in this tutorial. More details, including how to run python scripted filters, how to work with numpy and other tools, and how to package your scripts for execution under batch schedulers can be found online.

### 4.1.4 Visualizing Large Models

**ParaView** is used frequently at Sandia National Laboratories and other institutions for visualizing data from large-scale simulations run on the world's largest supercomputers including the examples shown here.

In this section we discuss visualizing large meshes like these using the parallel visualization capabilities of **ParaView**. This section has some exercises, but is less "hands-on" than the previous section. Primarily you will learn the conceptual knowledge needed to perform large parallel visualization. The exercises demonstrate the basic techniques needed to run **ParaView** on parallel machines.

The most fundamental idea to grasp is that when run on a large machine every node processes different regions of the entire dataset simultaneously. Thus the workable data resolution is limited by the aggregate memory space of the machine. We now present the basic **ParaView** architecture and parallel algorithms and demonstrate how to apply this knowledge.

#### Parallel Visualization Algorithms

We are fortunate in visualization in that many operations are straightforward to parallelize. The data we deal with is contained in a mesh, which means the data is already broken into little pieces by the cells. We can do visualization on a distributed parallel machine by first dividing the cells among the processes. For demonstrative purposes, consider this very simplified mesh.

Now let us say we want to perform visualizations on this mesh using three processes. We can divide the cells of the mesh as shown below with the blue, yellow, and pink regions.

Once partitioned, many visualization algorithms will work by simply allowing each process to independently run the algorithm on its local collection of cells. For example, take clipping (which is demonstrated in multiple exercises including Exercise **??**). Let us say that we define a clipping plane and give that same plane to each of the processes.

Each process can independently clip its cells with this plane. The end result is the same as if we had done the clipping serially. If we were to bring the cells together (which we would never actually do for large data for obvious reasons) we would see that the clipping operation took place correctly.

Many, but not all operations are thus trivially parallelizeable. Other operations are straightforward to parallelize if ghost layers as described in Section 4.1.4 are available. Other operations require more extensive data sharing among nodes. In these filters **ParaView** resorts to MPI to communicate amongst the nodes of the machine.

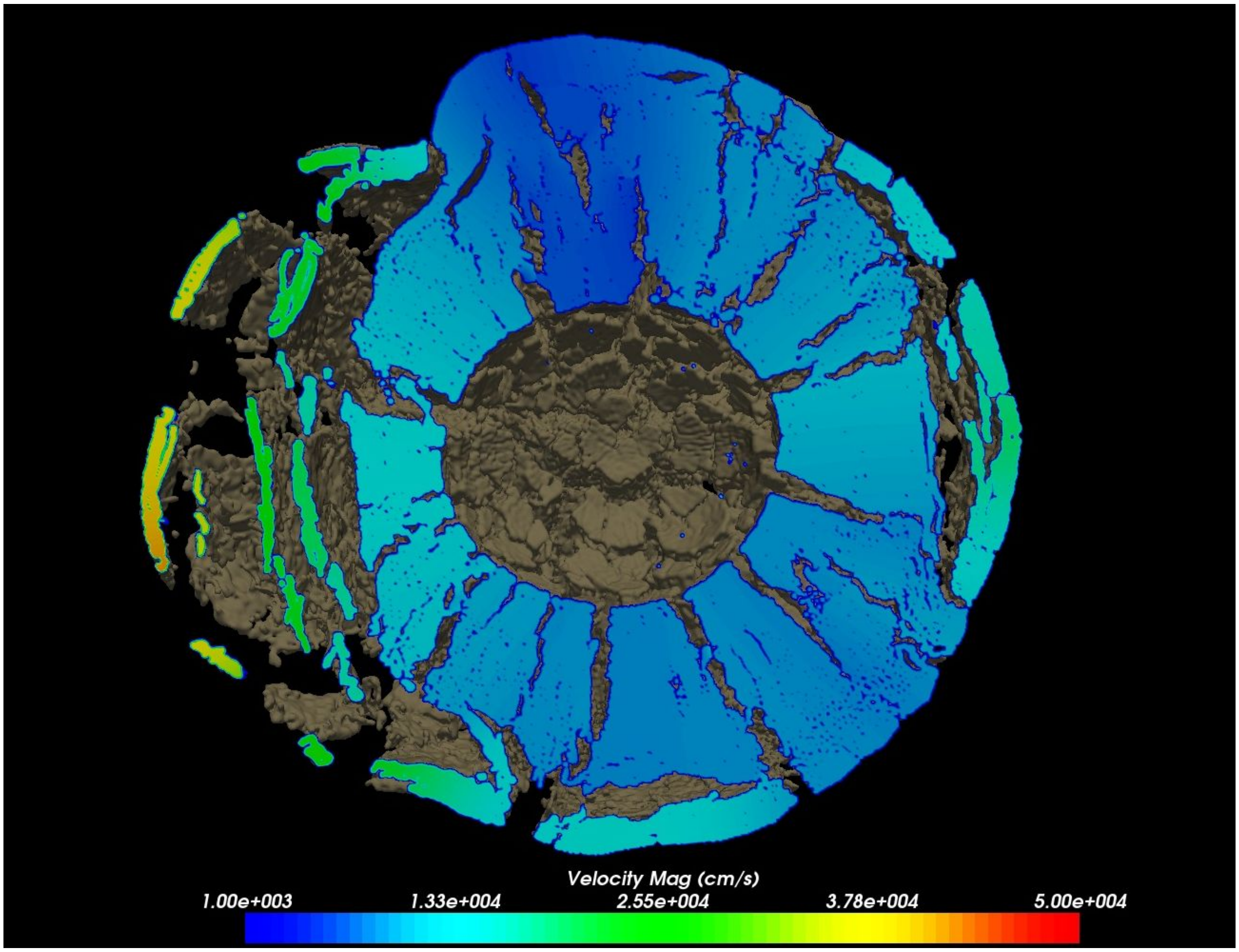


Fig. 4.5: CTH shock physics simulation with over 1 billion cells of a 10 megaton explosion detonated at the center of the Golevka asteroid.

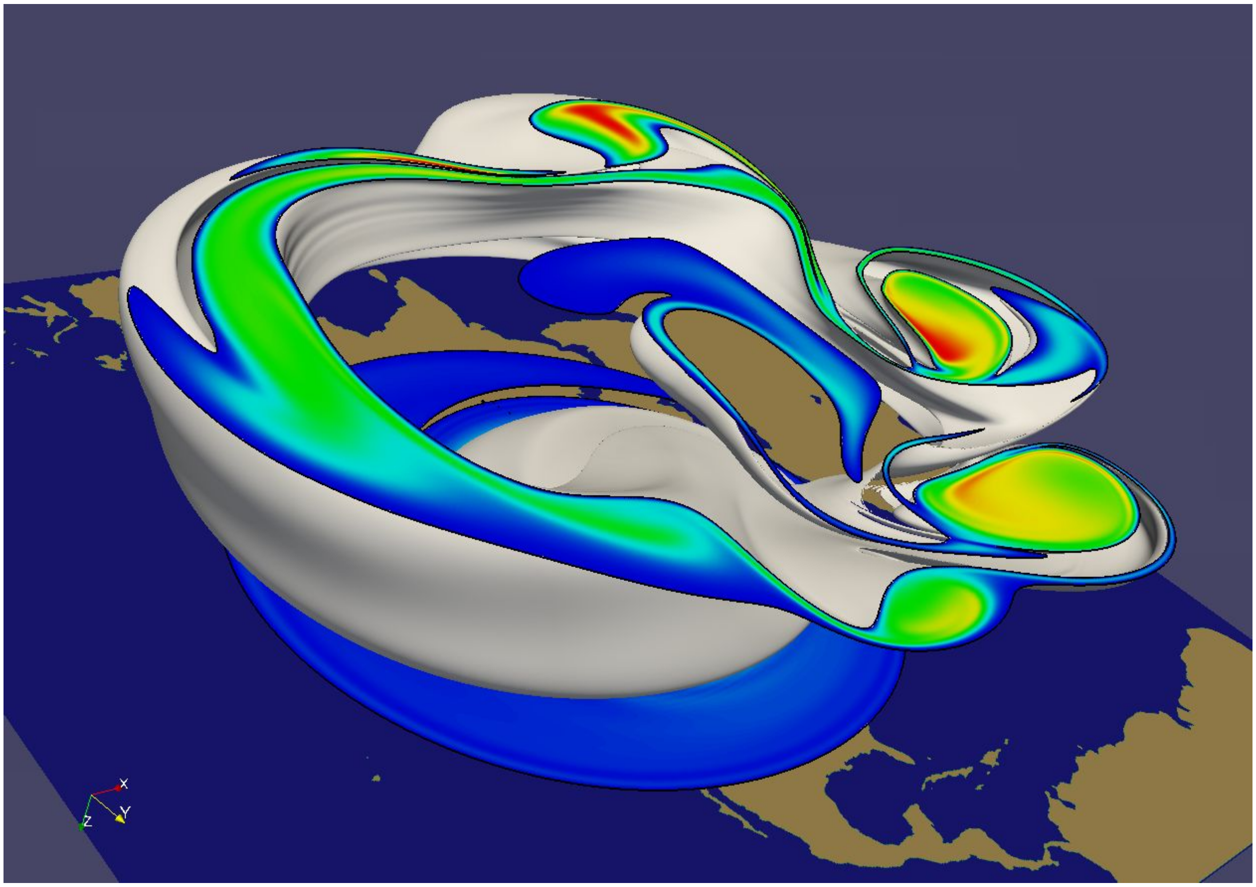


Fig. 4.6: SEAM Climate Modeling simulation with 1 billion cells modeling the breakdown of the polar vortex, a circumpolar jet that traps polar air at high latitudes.

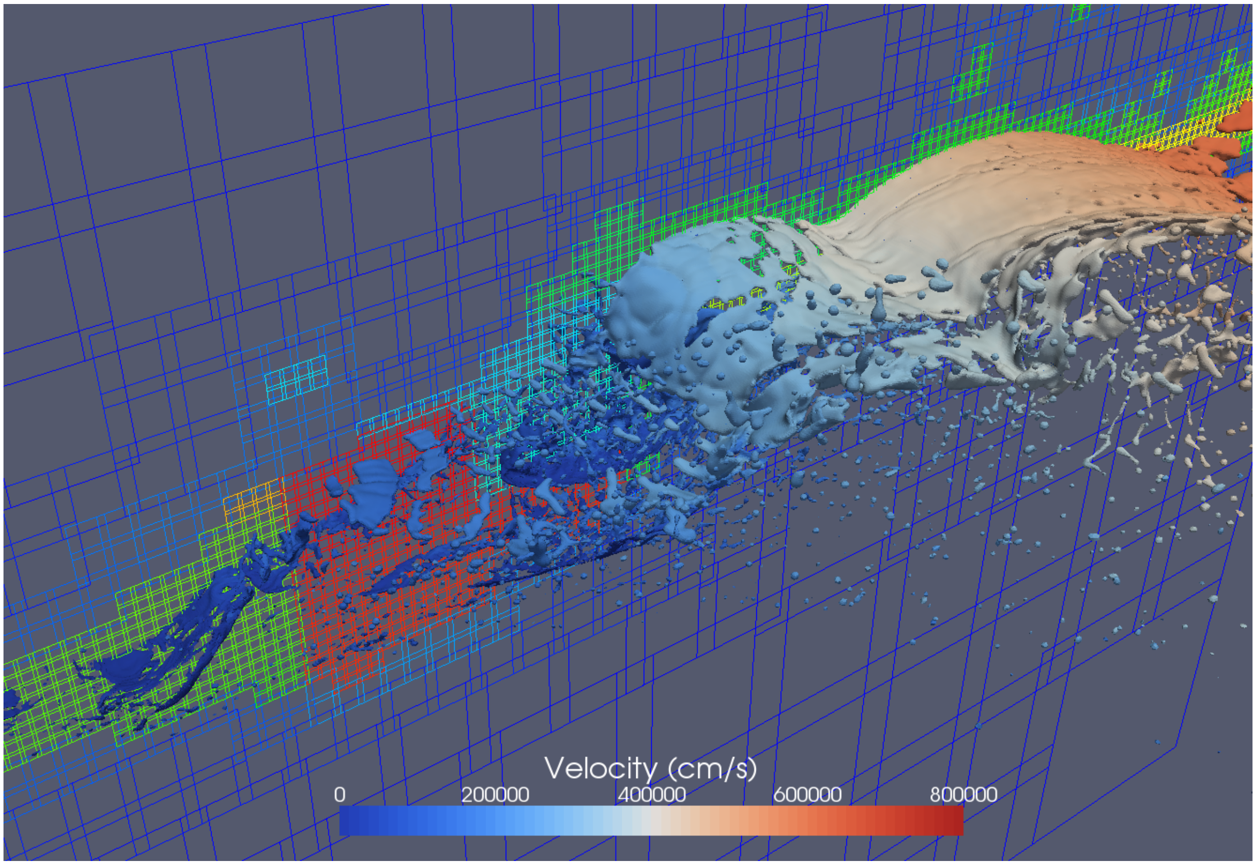


Fig. 4.7: A CTH simulation that generates AMR data. **ParaView** has been used to visualize CTH simulation AMR data comprising billions of cells, 100's of thousands of blocks, and eleven levels of hierarchy (not shown).

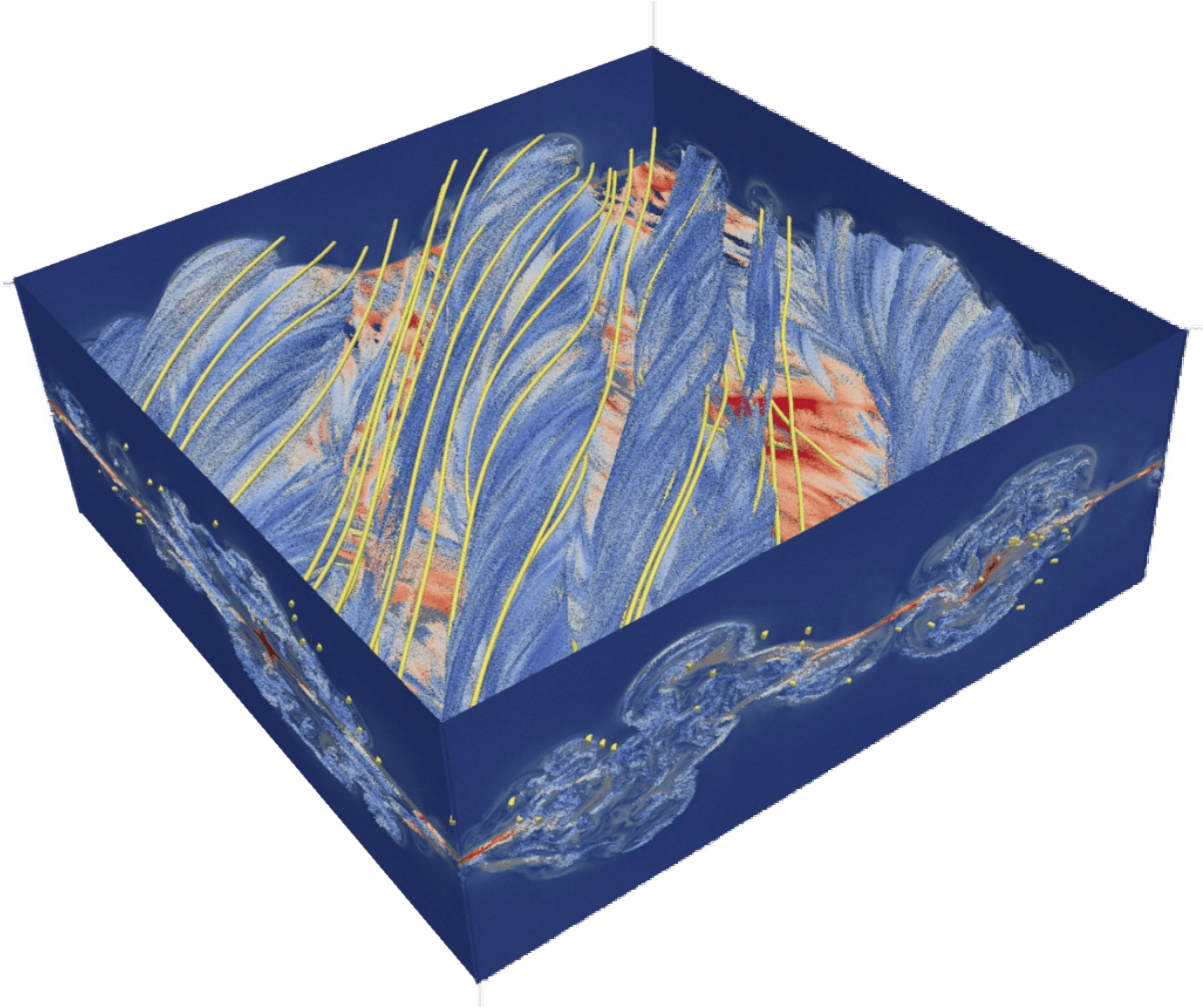


Fig. 4.8: A VPIC simulation of magnetic reconnection with 3.3 billion structured cells. Image courtesy of Bill Daughton, Los Alamos National Laboratory.

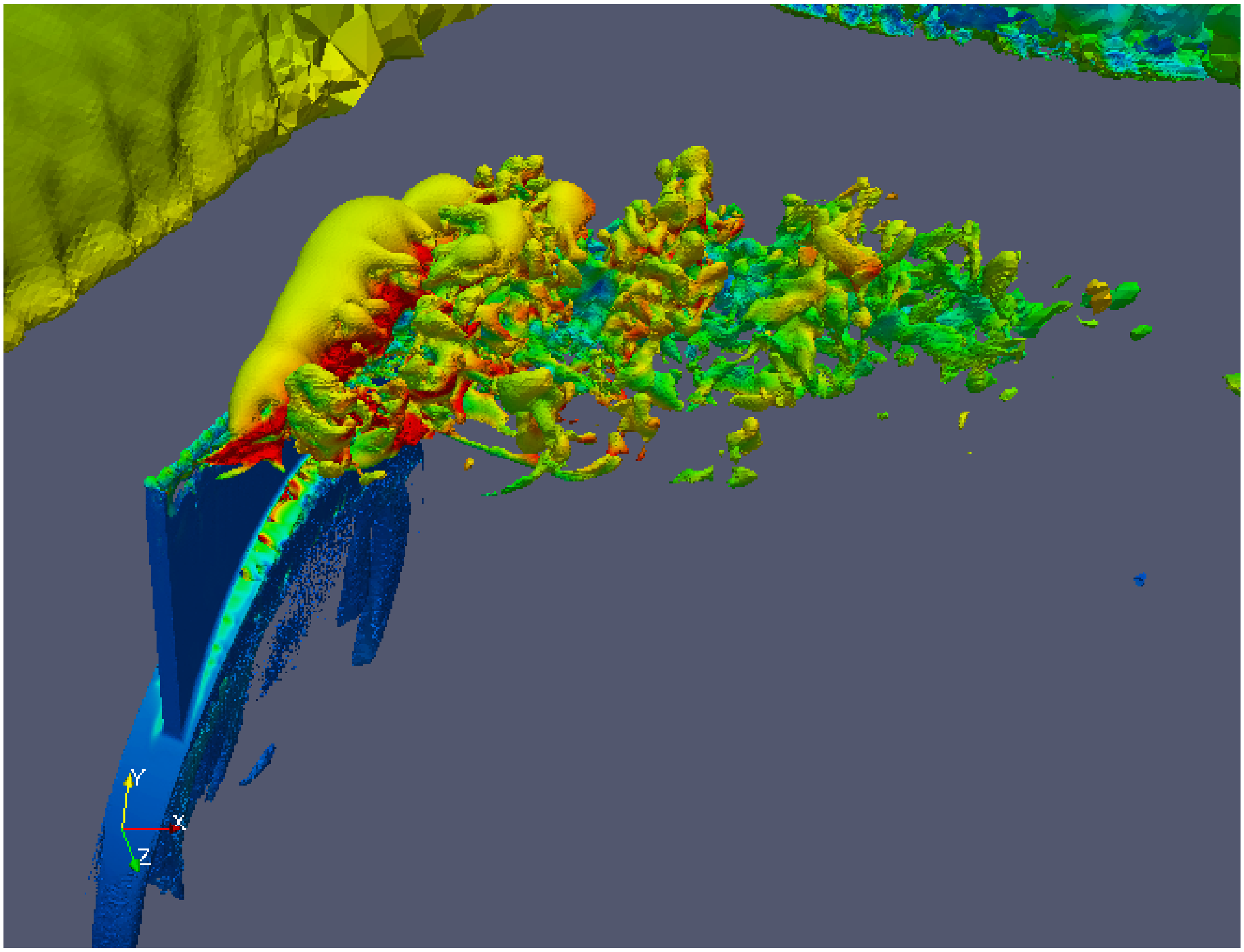


Fig. 4.9: A large scale in-situ PHASTA simulation that generated a 3.3 billion tetrahedral mesh simulating the flow over a full wing where a synthetic jet issues an unsteady crossflow jet (run on 160 thousand MPI processes).

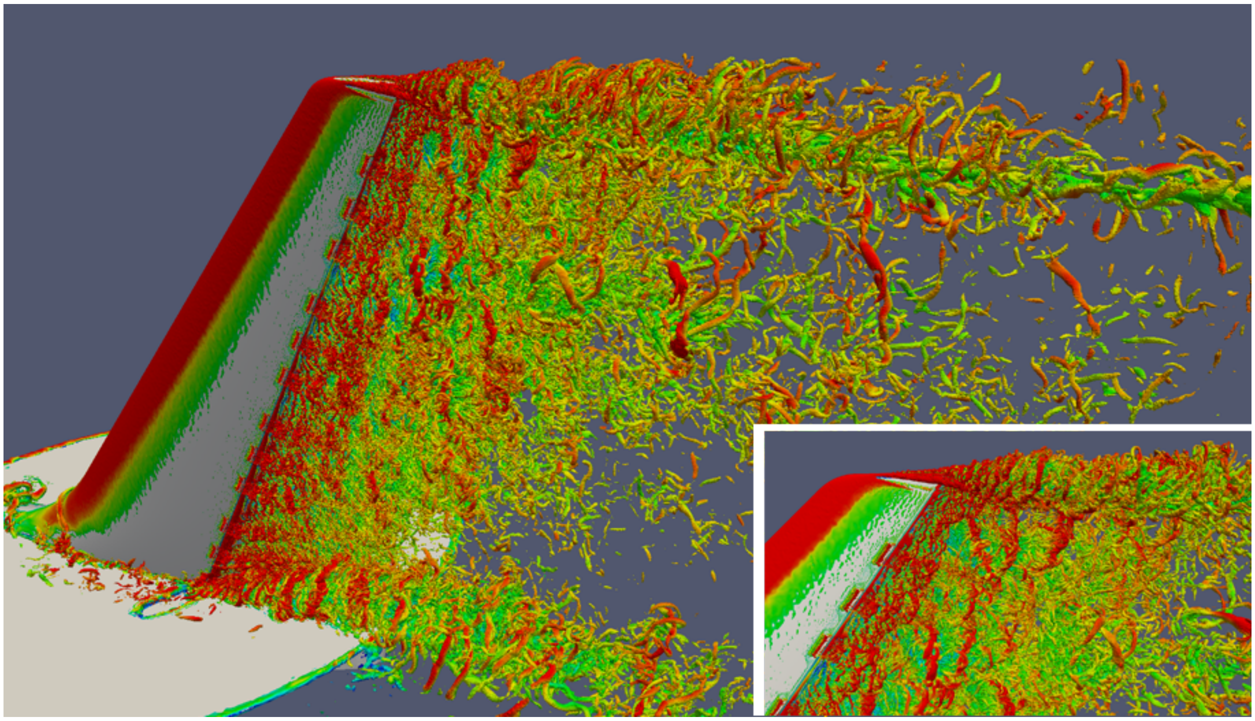
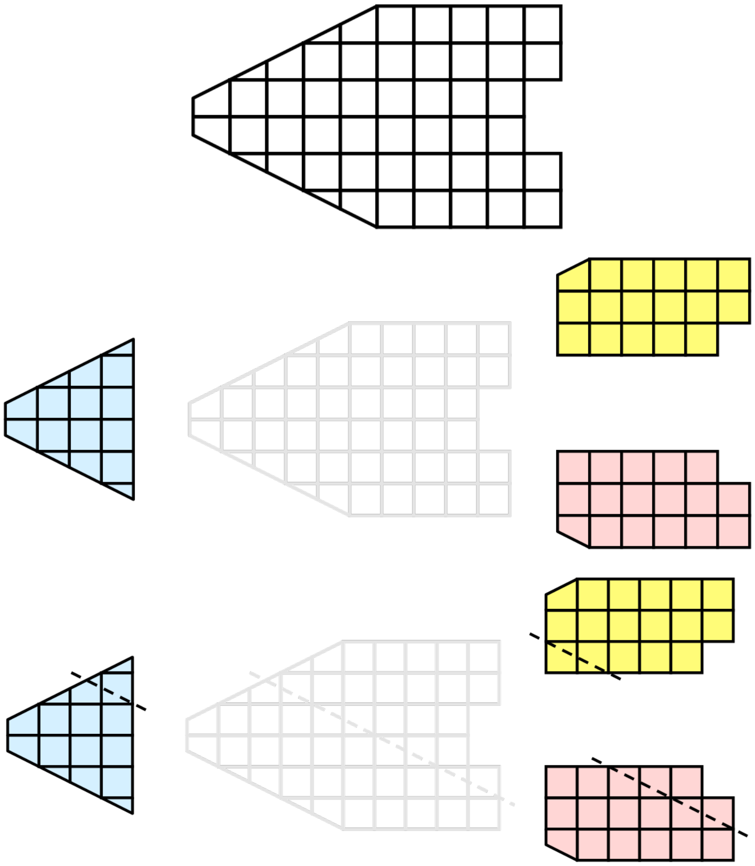
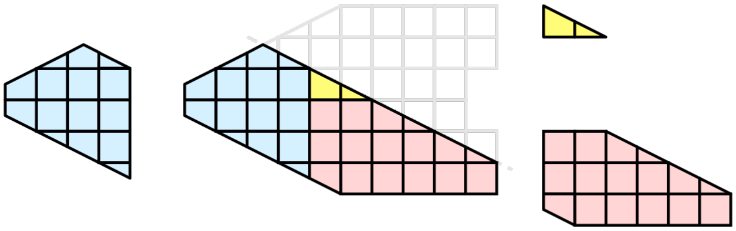


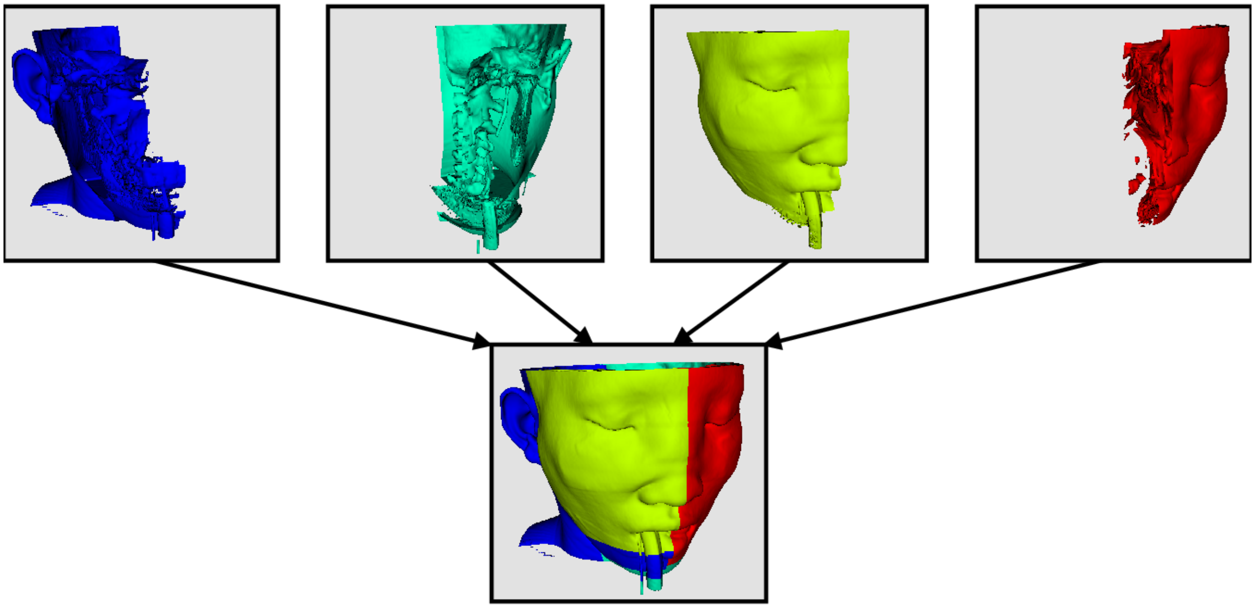
Fig. 4.10: A large scale in-situ PHASTA simulation that generated a 1.3 billion element mesh simulating the wake of a deflected wing flap (run on 256 thousand MPI processes). Images courtesy of Michel Rasquin, Argonne National Laboratory.





### Basic Parallel Rendering

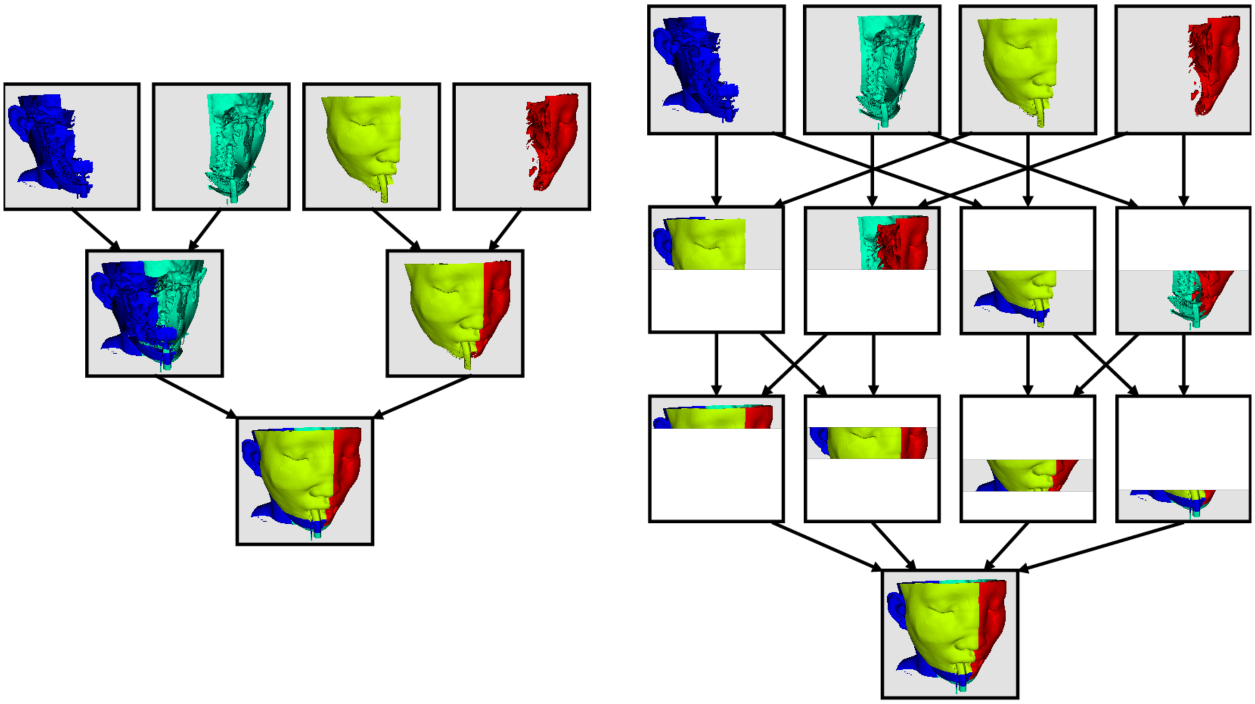
When performing parallel visualization, we are careful to ensure that the data remains partitioned among all of the processes up to and including the rendering processes. **ParaView** uses a parallel rendering library called **IceT**. IceT uses a **sort-last** algorithm for parallel rendering. This parallel rendering algorithm has each process independently render its partition of the geometry and then **composites** the partial images together to form the final image.



The preceding diagram is an oversimplification. IceT contains multiple parallel image compositing algorithms such as **binary tree**, **binary swap**, and **radix-k** that efficiently divide work among processes using multiple phases.

The wonderful thing about sort-last parallel rendering is that its efficiency is completely insensitive to the amount of data being rendered. This makes it a very scalable algorithm and well suited to large data. However, the parallel rendering overhead does increase linearly with the number of pixels in the image. This can be a problem for example when driving tile display walls with **ParaView**. Consequently, some of the rendering parameters described later in this chapter deal with limiting image size.

Because there is an overhead associated with parallel rendering, **ParaView** can also render serially and will do so automatically when the visible data is small enough. When the visible meshes are smaller than a user defined threshold preference, or when parallel rendering is turned off or unavailable, the geometry is shipped to the display node, which then rasterizes it locally. Obviously, this should only happen when the data being rendered is small or the rendering process will be overwhelmed.



### ParaView Architecture

With this introduction to parallel visualization, it is useful to know something about how **ParaView** is structured and how it orchestrates the parallel tasks described above.

**ParaView** is designed as a three-tier client-server architecture. The three logical units of **ParaView** are as follows.

**Data Server**
: The unit responsible for data reading, filtering, and writing. All of the pipeline objects seen in the pipeline browser are contained in the data server. The data server can be parallel.

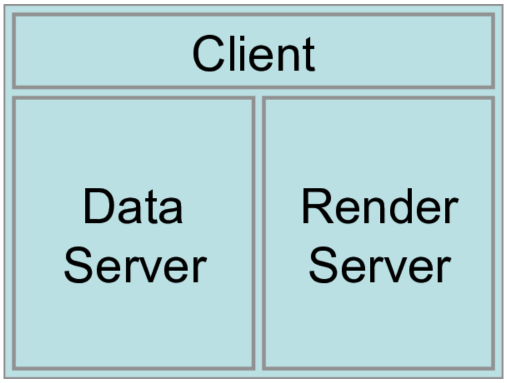
**Render Server**
: The unit responsible for rendering. The render server can also be parallel, in which case built in parallel rendering is also enabled.

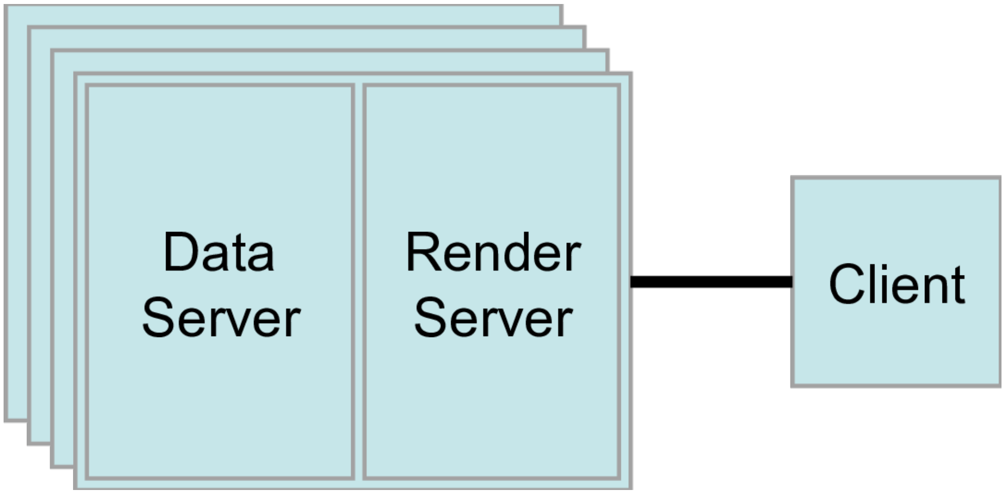
**Client**
: The unit responsible for establishing visualization. The client controls the object creation, execution, and destruction in the servers, but does not contain any of the data (thus allowing the servers to scale without bottlenecking on the client). If there is a GUI, that is also in the client. The client is always a serial application.

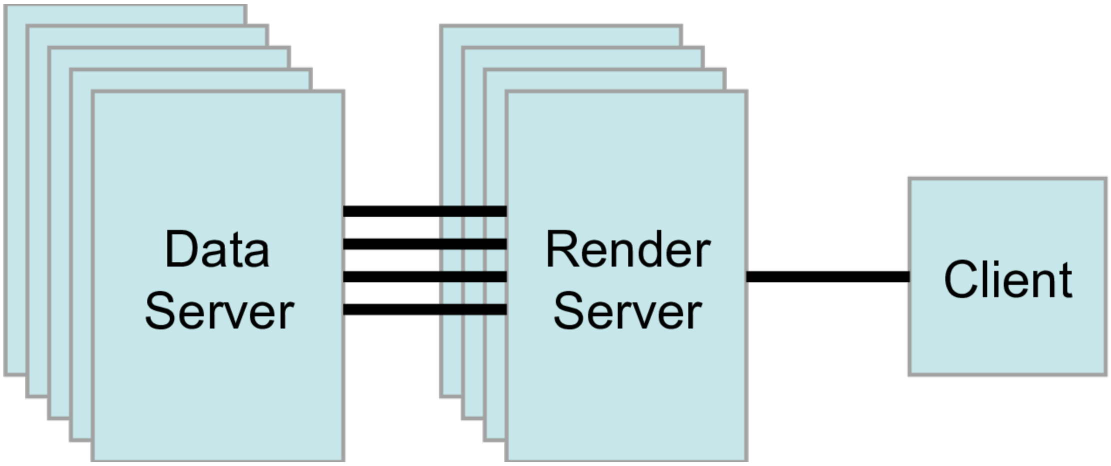
These logical units need not be physically separated. Logical units are often embedded in the same application, removing the need for any communication between them. There are three modes in which you can run **ParaView**. Note that no matter what mode **ParaView** runs in the user interface and scripting API that you learned in Section 4.1.2 and Section 4.1.3 undergoes very little change.

The first mode, which you are already familiar with, is **standalone** mode. In standalone mode, the client, data server, and render server are all combined into a single serial application. When you run the `paraview` application, you are automatically connected to a **builtin** server so that you are ready to use the full features of **ParaView**.

The second mode is **client-server** mode. In client-server mode, you execute the `pvserver` program on a parallel machine and connect to it with the `paraview` client application. The `pvserver` program has both the data server and render server embedded in it, so both data processing and rendering take place there. The client and server are connected via a socket, which is assumed to be a relatively slow mode of communication, so data transfer over this socket is minimized.


Client
Data Server
Render Server


Data Server
Render Server
Client


Data Server
Render Server
Client

The third mode is **client–render-server–data-server** mode. In this mode, all three logical units are running in separate programs. As before, the client is connected to the render server via a single socket connection. The render server and data server are connected by many socket connections, one for each process in the render server. Data transfer over the sockets is minimized.

Although the client-render server-data server mode is supported, we almost never recommend using it. The original intent of this mode was to take advantage of heterogeneous environments where one might have a large, powerful computational platform and a second smaller parallel machine with graphics hardware in it. However, in practice we find any benefit is almost always outstripped by the time it takes to move geometry from the data server to the render server. If the computational platform is much bigger than the graphics cluster, then use software rendering on the large computational platform. If the two platforms are about the same size just perform all the computation on the graphics cluster.

### Accessing a Parallel ParaView Server

Accessing a standalone installation of **ParaView** is trivial. Download and install a pre-compiled binary or get it from your package manager, and go. To visualize extreme scale results, you need to access an installation of **ParaView**, built with the MPI components enabled, on a machine with sufficient aggregate memory to hold the entire dataset and derived data products. Accessing a parallel enabled installation of **ParaView**'s server is intrinsically harder than acessing a standalone version.

Recent binary distributions of **ParaView** provided by Kitware do include MPI for Mac, Linux and optionally Windows. The Windows MPI enabled binaries depend upon Microsoft's MPI, which must be installed separately. We will not cover those in the rest of this tutorial, but feel free to ask questions on the **ParaView** mailing list.

Note that the specific MPI library version bundled into the binaries were chosen for widest possible compatibility. For production use **ParaView** will be much more effective when compiled with an MPI version that is tuned for your machine's networking fabric. Your system administrators can help you decide what MPI version to use.

To compile **ParaView** on a parallel machine you or your system administrators will need the following.

- CMake cross-platform build setup tool https://www.cmake.org
- MPI
- OpenGL, either using a GPU in on- (X11) or off- screen (EGL) modes, or in software via Mesa.
- Python +NumPy +Matplotlib (all optional but strongly recommended)
- Qt $\geq$ 4.7 (optional)

Compiling without one of the optional libraries means a feature will not be available. Compiling without Qt means that you will not have the GUI application and compiling without Python means that you will not have scripting available. NumPy is almost as important as Python itself, and Matplotlib is helpful for numeric text and some types of views and color lookup tables.

To compile **ParaView**, you first run CMake, which will allow you to set up compilation parameters and point to libraries on your system. This will create the make files that you then use to build **ParaView**. For more details on building a **ParaView** server, see the **ParaView** Wiki.

https://www.paraview.org/Wiki/Setting_up_a_ParaView_Server#Compiling

Compiling **ParaView** on exotic machines, for example HPC class machines that require cross compilation and some cloud-based on demand systems, can be an arduous task. You are welcome to seek free advice on the **ParaView** mailing list or contracted assistance from Kitware.

Fortunately there are a number of large scale systems on which **ParaView** has already been installed. If you are fortunate enough to have an account on one of these systems, you shouldn't need to compile anything. The **ParaView** community maintains an opt in list of these with pointers to system specific documentation on the **ParaView** Wiki. We invite system maintainers to add to the list if they are permitted to.

https://www.paraview.org/Wiki/ParaView/HPC_Installations

Running **ParaView** in parallel is also intrinsically more difficult than running the standalone client. It typically involves a number of steps that change depending on the hardware you are running on: logging in to remote computers, allocating parallel nodes, launching a parallel program, and sometimes tunneling through firewalls to establish interactive connections to the compute nodes. These steps are discussed in more detail in the next two sections in which we finally return from theory to practical exercises.

### Batch Processing

**ParaView**'s Python scripted interface, introduced in Section 4.1.3 has the important qualities of being runnable offline, with the human removed from the loop, and being inherently reproducible.

On large scale systems there are two steps that you need to master. The first is to run **ParaView** in parallel through MPI. The second is to submit a job through the system's queuing system. For both steps the syntax varies from machine to machine and you should consult your system's documentation for the details.

We demonstrate how to run **ParaView** in parallel with the simplest case of running Kitware's binary in parallel on the PC class system that you have at hand.

**Exercise 4.1 (Running a visualization script in parallel)**

You will need three things, the `mpiexec` program to spawn MPI parallel programs, the `pvbatch` executable and a Python script.

As for the Python script, let us use an example in which each node draws a portion of a simple polygonal sphere with a color map that varies over the number of processes in the job.

```python
from paraview.simple import *
sphere = Sphere()
rep = Show()
ColorBy(rep, ("POINTS", "vtkProcessId"))
Render()
rep.RescaleTransferFunctionToDataRange(True)
Render()
WriteImage("parasphere.png")
```

Use an editor to type this script into a file called parasphere.py and save it somewhere. Note this is a stripped down version of a script recorded from a parallel **ParaView** session in which we created a Sources → Sphere and chose the vtkProcessId array to color by.

As for the executables, let us use the ones that come with **ParaView**. After installing Kitware's **ParaView** binaries, you will find `mpiexec` and `pvbatch` at:

- On Mac:

  ```
  /Applications/ParaView-x.x.x.app/Contents/MacOS/mpiexec
  ```

  and

  ```
  /Applications/ParaView-x.x.x.app/Contents/bin/pvbatch
  ```

- On Linux and assuming you have extracted the binary into /usr/local:

  ```
  /usr/local/lib/paraview-x.x.x/mpiexec
  ```

  and

```
/usr/local/bin/pvbatch
```

Note that you will need to add **ParaView**'s lib directory (/usr/local/lib/paraview-x.x.x) to your LD_LIBRARY_PATH to use the mpi that comes with **ParaView**.

- On Windows and assuming you installed the "MPI" version of **ParaView** as well as the MS-MPI software (available separately from Microsoft for free):

```
C:/Program Files/Microsoft MPI/Bin/mpiexec
```

and

```
C:/Program Files/ParaView x.x.x/bin/pvbatch
```

Now we run the script in parallel by issuing this from the command line.

- On Mac:

```
/Applications/ParaView-x.x.x.app/Contents/MacOS/mpiexec -np 4 /Applications/
↪ParaView-x.x.x.app/Contents/bin/pvbatch parasphere.py
```

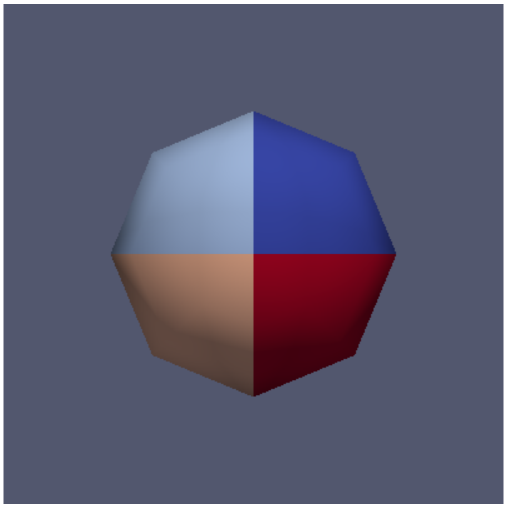
- On Linux:

```
/usr/local/lib/paraview-x.x.x/mpiexec -np 4 /usr/local/bin/pvbatch parasphere.py
```

- On Windows:

```
mpiexec -np 4 "C:/Program Files/ParaView x.x.x/bin/pvbatch" parasphere.py
```

**ParaView** will run momentarily and it may or may not briefly display a window on your desktop. In any case you will now find a file named parasphere.png that looks like the following:



---

Submitting a job through your system's queuing system typically involves issuing a command like the following on the command line.

```
qsub -A <project name to charge to compute time to> \
 -N <number of nodes>  \
```

(continues on next page)

(continued from previous page)

```
-n <number of processors on each node> \
mpiexec -np <N*n> \
  pvbatch <arguments for pvbatch> \
  script_to_execute.py <arguments for Python script>
```

The command reserves the requested number of compute nodes on the machine and, at some time in the future when the nodes become ready for your use, spawns an MPI job. The MPI job runs **ParaView**'s Python scripted server in parallel, telling it to process the supplied script.

There are many queuing systems and mpi implementations and site specific policies. Thus it it impossible to provide an exact syntax that will work on every system here. Ask your system administrators for guidance. Once you have the syntax, you should be able to run the sphere example above to determine if you have a working system.

### Interactive Parallel Processing

For day to day work with large datasets it is feasible to do interactive visualization in parallel too. In this mode you can dynamically inspect the data and modify the visualization pipeline while you work with the **ParaView** GUI on a workstation far from the HPC resource where the data is being processed.

In interactive configurations the data processing and rendering portions work in parallel like above but they are controlled from the **ParaView** GUI instead of a Python script.

Starting the server process is similar to the above, but instead of using the `pvbatch` executable we use the `pvserver` executable. The former is limited to taking commands from Python scripts, the later is built to take commands from a remote **ParaView** GUI program.

Client-server connections are established through the `paraview` client application. You connect to servers and disconnect from servers with the  and  buttons. When **ParaView** starts, it automatically connects to the builtin server. It also connects to builtin whenever it disconnects  from a server.

When you hit the  button, **ParaView** presents you with a dialog box containing a list of known servers you may connect to. This list of servers can be both site- and user-specific.

You can specify how to connect to a server either through the GUI by pressing the Add Server button or through an XML definition file. There are several options for specifying server connections, but ultimately you are giving **ParaView** a command to launch the server and a host to connect to after it is launched.

Once more we will demonstrate using the Kitware binaries and then outline the steps required on large scale systems where the syntax varies widely.

**Exercise 4.2 (Interactive Parallel Visualization)**

The `pvserver` executable can be found in the same directory as the `pvbatch` exectuable. Let us begin by running it in parallel.

- On Mac:

```
/Applications/ParaView-x.x.x.app/Contents/MacOS/mpiexec -np 4 /Applications/
↪ParaView-x.x.x.app/Contents/bin/pvserver &
```

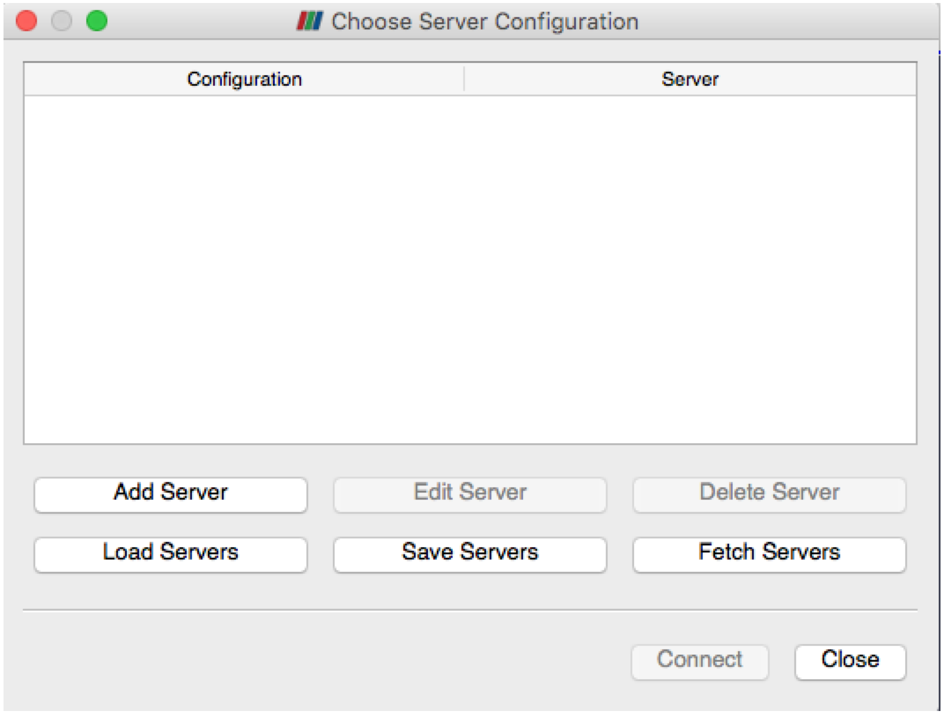
- On Linux:

```
/usr/local/lib/paraview-x.x.x/mpiexec -np 4 /usr/local/bin/pvserver &
```

- On Windows:

```
mpiexec -np 4 ^ "C:/Program Files/ParaView x.x.x/bin/pvserver"
```

Now start up the **ParaView** GUI and click the  button. This will bring the Choose Server Configuration dialog box where you can download or create server connection paths, or connect to predefined paths.



Now click the Add Server button to create a new path, which we will construct to connect to the waiting `pvserver` running on your local machine.

In the Edit Server Configuration Dialog, change the Name of the connection path from "unknown" to "my computer". In actual use one would enter in a nickname for and the IP address of the remote machine we want to connect to. Now hit the Configure button.

On the resulting Edit Server Launch Configuration dialog, leave the Startup Type as Manual since we have already started a waiting `pvserver`. In actual use one would typically use Command and in the dialog enter in a script that starts up the server on the remote machine. Click Save on the dialog to finish defining the connection.

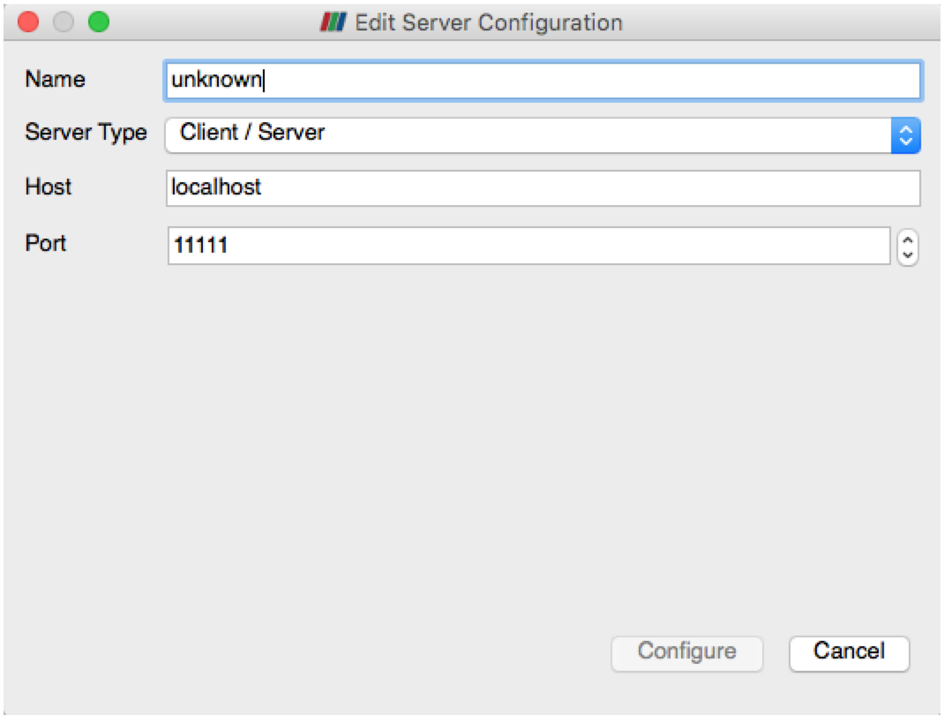
Now that we have defined a connection, it will show up under the Choose Server Configuration dialog whenever we hit . Click on the name of the connection, "my computer" in this case, and click Connect to establish the connection between the GUI and `pvserver`.

Once connected, simply File → Open files on the remote system and work with them as before. To complete the exercise, open or create a dataset, and choose the vtkProcessId array to examine how it is partioned across the distributed memory of the server.

---

In practice, the syntax of starting up a parallel server on a remote machine and connecting to it will be more involved. Typically the steps include ssh'ing to a remote machine's login node, submitting a job that runs a script that requests some number of nodes and runs `pvserver` under MPI on them with `pvserver` made to connect back to the waiting client (instead of the reverse as above) over at least one ssh tunnel hops. System administrators and other adventurous folk can find details online here:

- https://www.paraview.org/Wiki/Setting_up_a_ParaView_Server#Running_the_Server
- https://www.paraview.org/Wiki/Reverse_connection_and_port_forwarding

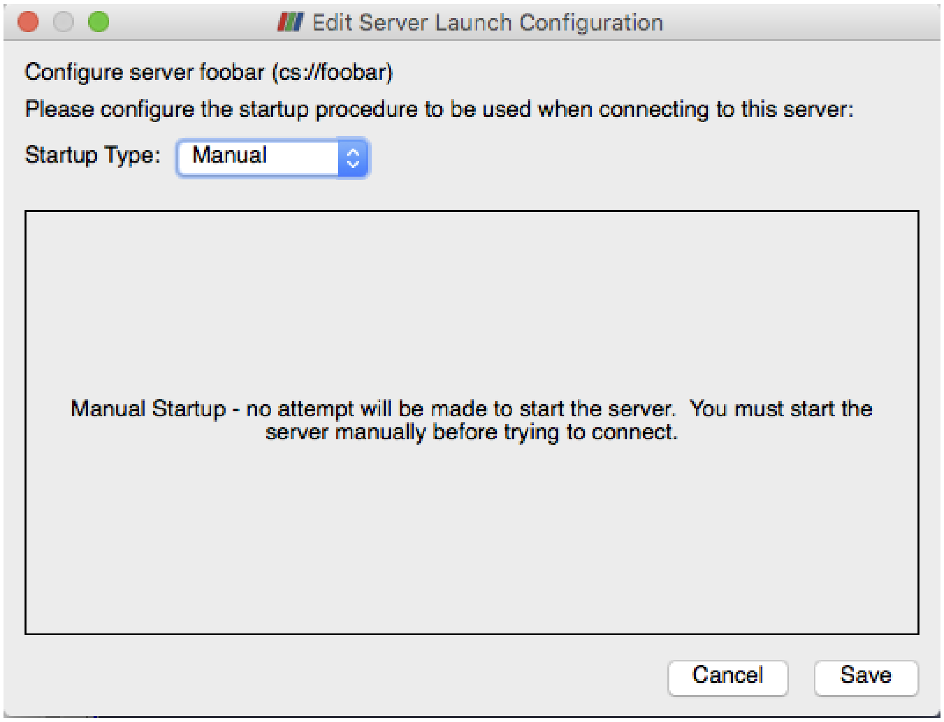


Name: unknown

Server Type: Client / Server

Host: localhost

Port: 11111



Configure server foobar (cs://foobar)

Please configure the startup procedure to be used when connecting to this server:

Startup Type: Manual

Manual Startup - no attempt will be made to start the server.  You must start the server manually before trying to connect.

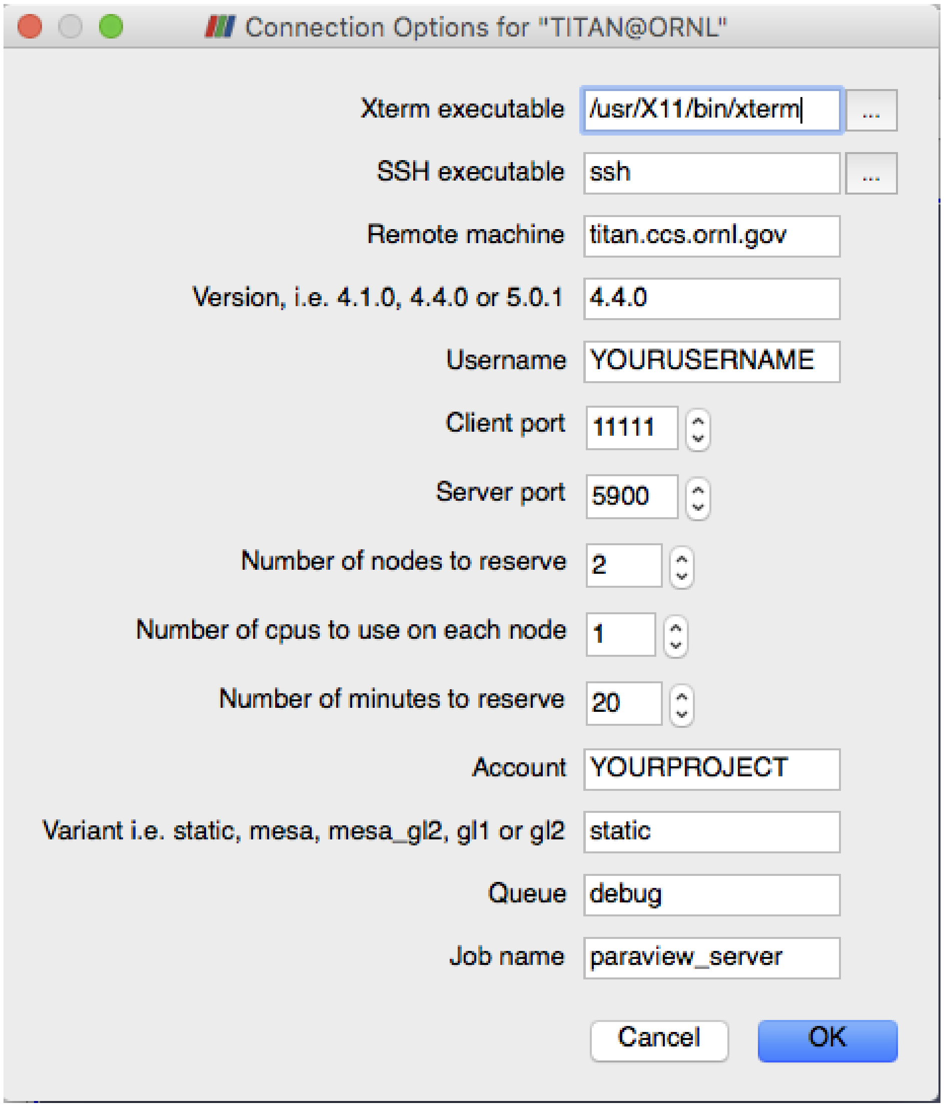
On the local machine you may need helper utilities as well. XQuartz (for X11) on Mac and Putty (for ssh) on Windows are useful for starting up remote connections. These are required to complete the next exercise.

Once a connection is established the steps should be saved in scripts and configuration files. System administrators can publish these so that authorized users can simply download and use them. Several of these configuration files are hosted on the **ParaView** website where the **ParaView** client can download them from.

**Exercise 4.3 (Fetching and using Connections)**

Once again, click on  to bring up the Choose Server Configuration dialog. This time, hit the Fetch Servers button. This will bring up the Fetch Server Configurations dialog box populated with a list of server definitions from the webite. Choose one and click the Import Selected button to download it into your **ParaView** preferences. The new connection is now available just as the connection to "my computer" is.

To use the remote machine, once more click . This time choose the new machine instead of "my machine" and click Connect. Doing so will bring up a Connection Options for … dialog box that lets where you enter in the parameters for your parallel session.



Connection Options for "TITAN@ORNL"

| | |
|---|---|
| Xterm executable | /usr/X11/bin/xterm |
| SSH executable | ssh |
| Remote machine | titan.ccs.ornl.gov |
| Version, i.e. 4.1.0, 4.4.0 or 5.0.1 | 4.4.0 |
| Username | YOURUSERNAME |
| Client port | 11111 |
| Server port | 5900 |
| Number of nodes to reserve | 2 |
| Number of cpus to use on each node | 1 |
| Number of minutes to reserve | 20 |
| Account | YOURPROJECT |
| Variant i.e. static, mesa, mesa_gl2, gl1 or gl2 | static |
| Queue | debug |
| Job name | paraview_server |

The parameters you will want to set include your username on the remote machine, the number of nodes and processes to reserve and the amount of time you want to reserve them for. Each site may have additional choices that are passed into the launch scripts on the server. Once set, click OK to try to establish a connection.

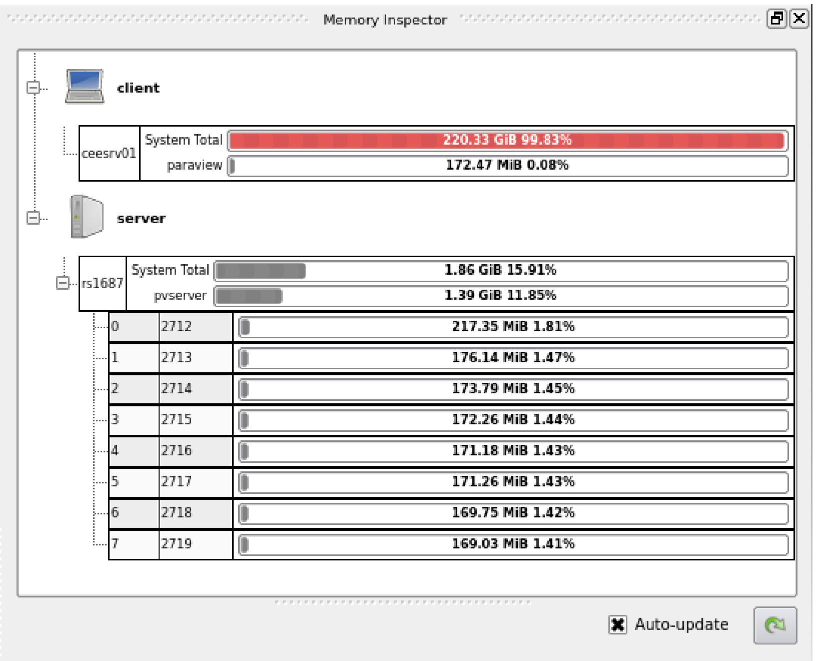
When you click OK **ParaView** typically spawns a terminal window (xterm on Mac and Linux, cmd.exe on Windows). Here you will have to enter in your credentials to actually access the remote machine. Once logged on, the script to reserve the nodes automatically runs and in most cases the terminal session has a menu with options for looking at the remote system's queue. When your job reaches the top of the queue, the **ParaView** 3D View window will return and you can begin doing remote visualization, this time with the capacity of the remote machine at your fingertips.

## Parallel Data Processing Practicalities

### Keeping Track of Memory

When working with very large models, it is important to keep track of memory usage on the computer. One of the most common and frustrating problems encountered with large models is running out of memory. This in turn will lead to thrashing in the virtual memory system or an outright program fault.

Section 4.1.4 and Section 4.1.4 provide suggestions to reduce your memory usage. Even so, it is wise to keep an eye on the memory available in your system. **ParaView** provides a tool called the **memory inspector** designed to do just that.



**client**

| | | |
|---|---|---|
| ceesrv01 | System Total | 220.33 GiB 99.83% |
| | paraview | 172.47 MiB 0.08% |

**server**

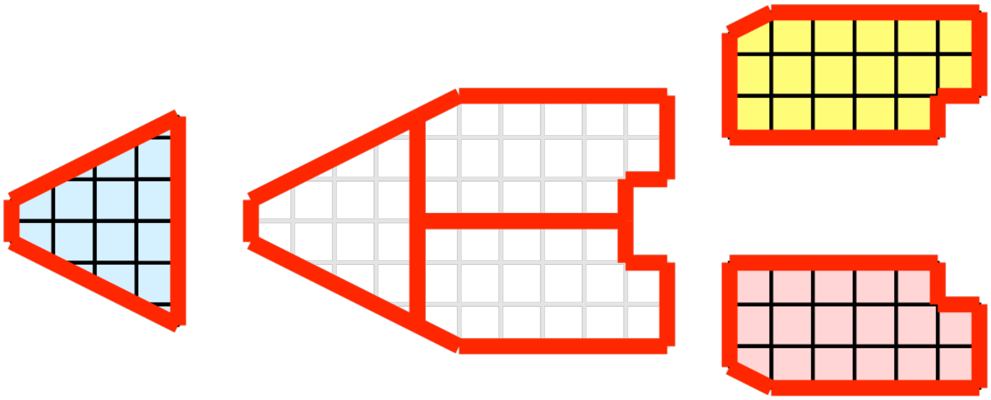
| | | |
|---|---|---|
| rs1687 | System Total | 1.86 GiB 15.91% |
| | pvserver | 1.39 GiB 11.85% |
| 0 | 2712 | 217.35 MiB 1.81% |
| 1 | 2713 | 176.14 MiB 1.47% |
| 2 | 2714 | 173.79 MiB 1.45% |
| 3 | 2715 | 172.26 MiB 1.44% |
| 4 | 2716 | 171.18 MiB 1.43% |
| 5 | 2717 | 171.26 MiB 1.43% |
| 6 | 2718 | 169.75 MiB 1.42% |
| 7 | 2719 | 169.03 MiB 1.41% |

☒ Auto-update

To access the memory inspector, select in the menu bar View → Memory Inspector. The memory inspector provides information for both the client you are running on and any server you might be connected to. It will tell you the total amount of memory used on the system and the amount of memory **ParaView** is using. For servers containing multiple nodes, information both for the conglomerate job and for each individual node are given. Note that a memory issue in any single node can cause a problem for the entire **ParaView** job.

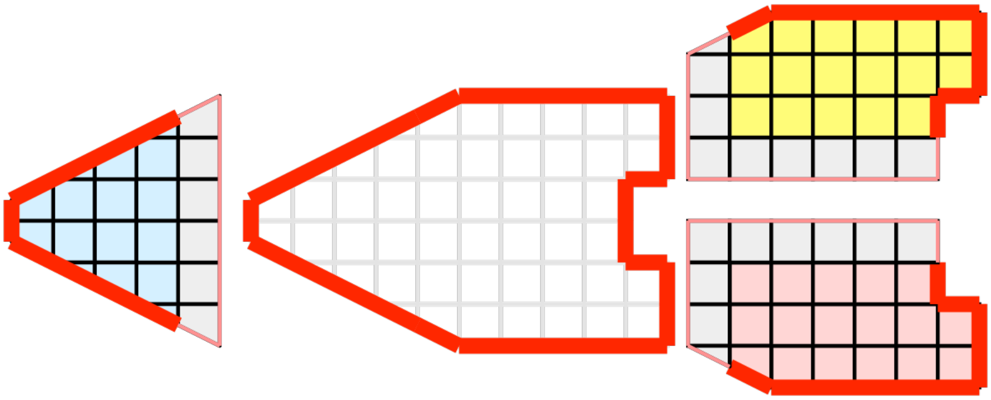
### Ghost Levels

Unfortunately, blindly running visualization algorithms on partitions of cells does not always result in the correct answer. As a simple example, consider the **external faces** algorithm. The external faces algorithm finds all cell faces that belong to only one cell, thereby identifying the boundaries of the mesh. What happens when we run external faces independently on our partitions?



Oops. We see that when all the processes ran the external faces algorithm independently, many internal faces were incorrectly identified as being external. This happens where a cell in one partition has a neighbor in another partition. A process has no access to cells in other partitions, so there is no way of knowing that these neighboring cells exist.

The solution employed by **ParaView** and other parallel visualization systems is to use **ghost cells** (sometimes also called **halo regions**). Ghost cells are cells that are held in one process but actually belong to another. To use ghost cells, we first have to identify all the neighboring cells in each partition. We then copy these neighboring cells to the partition and mark them as ghost cells, as indicated with the gray colored cells in the following example.



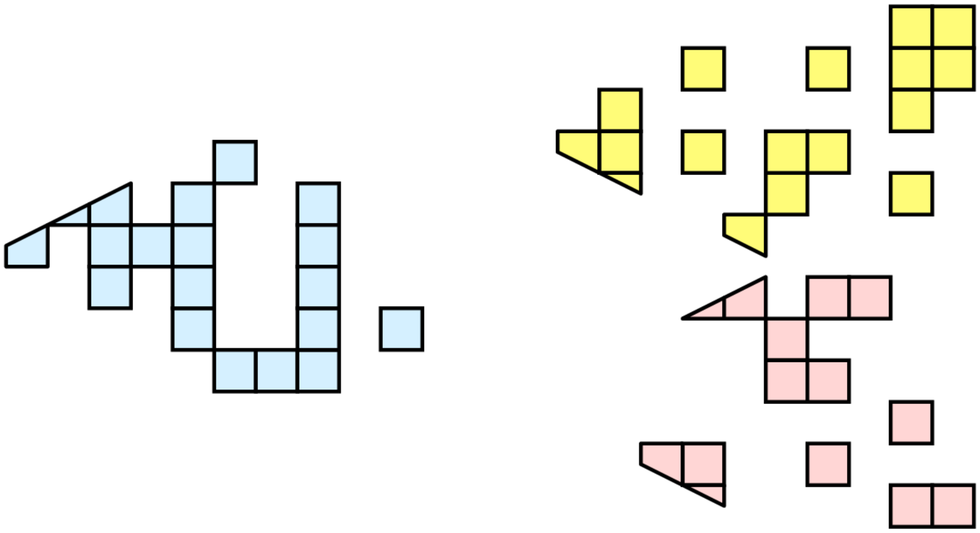
When we run the external faces algorithm with the ghost cells, we see that we are still incorrectly identifying some internal faces as external. However, all of these misclassified faces are on ghost cells, and the faces inherit the ghost status of the cell it came from. **ParaView** then strips off the ghost faces and we are left with the correct answer.

In this example we have shown one layer of ghost cells: only those cells that are direct neighbors of the partition's cells. **ParaView** also has the ability to retrieve multiple layers of ghost cells, where each layer contains the neighbors of the previous layer not already contained in a lower ghost layer or the original data itself. This is useful when we have cascading filters that each require their own layer of ghost cells. They each request an additional layer of ghost cells from upstream, and then remove a layer from the data before sending it downstream.

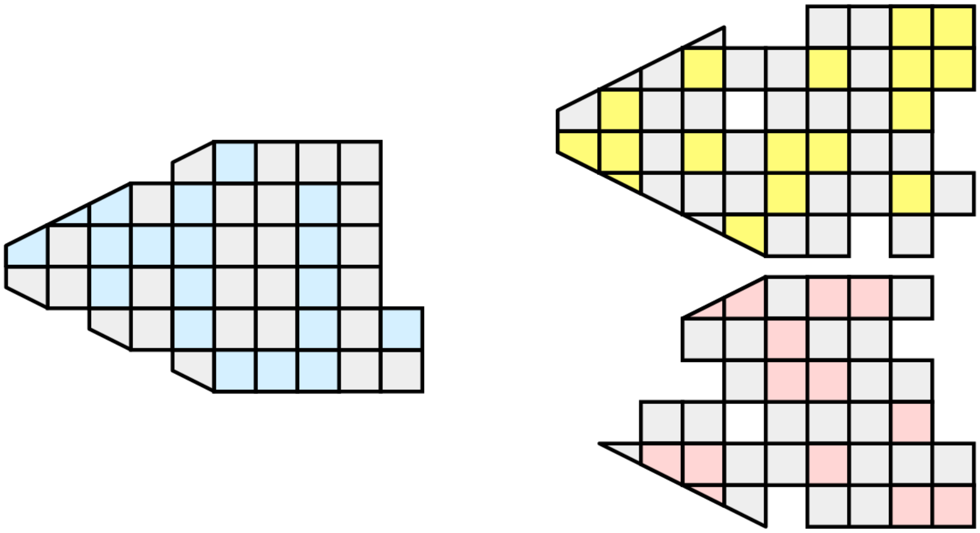
### Data Partitioning

For the most part, **ParaView** leaves the task of breaking up or partitioning the data to the simulation code that produced the data. It is then the responsibility of **ParaView**'s specific reader module for the file format in question to work efficiently, typically reading different files from different nodes independently but simultaneously. In ideal cases the rest of the parallel pipeline also operates independently and simultaneously.

Still, in the general case, since we are breaking up and distributing our data, it is prudent to address the ramifications of how we partition the data. The data shown in Section 4.1.4 has a **spatially coherent** partitioning. That is, all the cells of each partition are located in a compact region of space. There are other ways to partition data. For example, you could have a random partitioning.



Random partitioning has some nice features. It is easy to create and is friendly to load balancing. However, a serious problem exists with respect to ghost cells.



In this example, we see that a single level of ghost cells nearly replicates the entire dataset on all processes. We have thus removed any advantage we had with parallel processing. Because ghost cells are used so frequently, random partitioning is not used in **ParaView**.

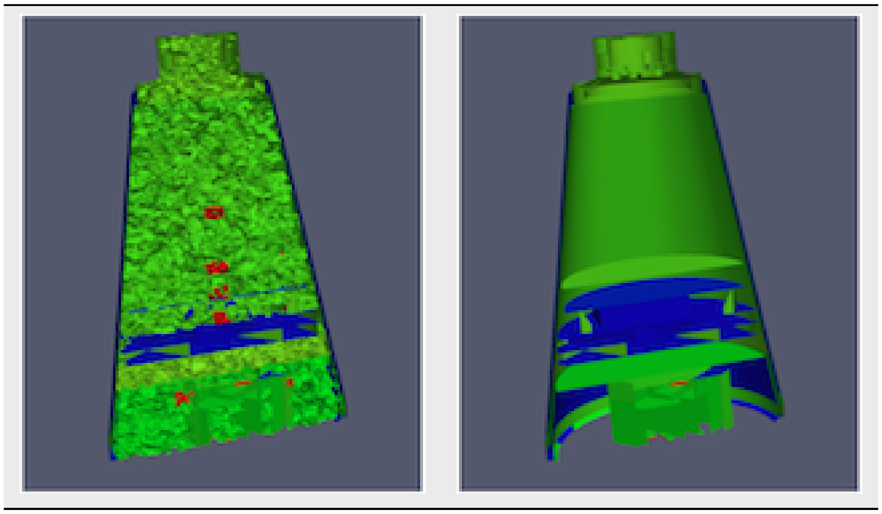
### D3 Filter

The previous sections described the importance of load balancing and ghost levels for parallel visualization. This section describes how to achieve that.

Load balancing and ghost cells are handled automatically by **ParaView** when you are reading structured data (image data, rectilinear grid, and structured grid). The implicit topology makes it easy to break the data into spatially coherent chunks and identify where neighboring cells are located.

It is an entirely different matter when you are reading in unstructured data (poly data and unstructured grid). There is no implicit topology and no neighborhood information available. **ParaView** is at the mercy of how the data was written to disk. Thus, when you read in unstructured data there is no guarantee about how well load balanced your data will be. It is also unlikely that the data will have ghost cells available, which means that the output of some filters may be incorrect.

Fortunately, **ParaView** has a filter that will both balance your unstructured data and create ghost cells. This filter is called D3, which is short for distributed data decomposition. Using D3 is easy; simply attach the filter (located in Filters → Alphabetical → D3) to whatever data you wish to repartition.



The most common use case for D3 is to attach it directly to your unstructured grid reader. Regardless of how well load balanced the incoming data might be, it is important to be able to retrieve ghost cells so that subsequent filters will generate the correct data. The example above shows a cutaway of the extract surface filter on an unstructured grid. On the left we see that there are many faces improperly extracted because we are missing ghost cells. On the right the problem is fixed by first using the D3 filter.

### Ghost Cells Generator Filter

In many cases a better alternative to D3 filter is the Ghost Level Generator. This filter is more efficient than the D3 filter because it makes the assumption that the input unstructured grid data is already partitioned into spatially coherent regions. This is generally a safe assumption as simulations benefit from coherency too. Because of this, it concerns itself only with cells at and near the external shell of the mesh, and does not consider or transfer the bulk of the data at the interior.

### Advice

#### Matching Job Size to Data Size

*How many cores should I have in my |ParaView| server?* This is a common question with many important ramifications. It is also an enormously difficult question. The answer depends on a wide variety of factors including what hardware each processor has, how much data is being processed, what type of data is being processed, what type of visualization operations are being done, and your own patience.

Consequently, we have no hard answer. We do however have several rules of thumb.

**If you are loading structured data** (image data, rectilinear grid, structured grid), try to have a minimum of one core per 20 million cells. If you can spare the cores, one core for every 5 to 10 million cells is usually plenty.

**If you are loading unstructured data** (poly data, unstructured grid), try to have a minimum of one core per 1 million cells. If you can spare the cores, one core for every 250 to 500 thousand cells is usually plenty.

As stated before, these are just rules of thumb, not absolutes. You should always try to experiment to gage what your core to data size should be. And, of course, there will always be times when the data you want to load will stretch the limit of the resources you have available. When this happens, you will want to make sure that you avoid data explosion and that you cull your data quickly.

#### Avoiding Data Explosion

The pipeline model that **ParaView** presents is very convenient for exploratory visualization. The loose coupling between components provides a very flexible framework for building unique visualizations, and the pipeline structure allows you to tweak parameters quickly and easily.

The downside of this coupling is that it can have a larger memory footprint. Each stage of this pipeline maintains its own copy of the data. Whenever possible, **ParaView** performs **shallow copies** of the data so that different stages of the pipeline point to the same block of data in memory. However, any filter that creates new data or changes the values or topology of the data must allocate new memory for the result. If **ParaView** is filtering a very large mesh, inappropriate use of filters can quickly deplete all available memory. Therefore, when visualizing large datasets, it is important to understand the memory requirements of filters.

Please keep in mind that the following advice is intended *only for when dealing with very large amounts of data and the remaining available memory is low*. When you are not in danger of running out of memory, ignore all of the following advice.

When dealing with structured data, it is absolutely important to know what filters will change the data to unstructured. Unstructured data has a much higher memory footprint, per cell, than structured data because the topology must be explicitly written out. There are many filters in **ParaView** that will change the topology in some way, and these filters will write out the data as an unstructured grid, because that is the only dataset that will handle any type of topology that is generated. The following list of filters will write out a new unstructured topology in its output that is roughly equivalent to the input. These filters should *never* be used with structured data and should be used with caution on unstructured data.

- Append Datasets
- Append Geometry
- Clean
- Clean to Grid
- Connectivity
- D3
- Delaunay 2D/3D

- Extract Edges
- Linear Extrusion
- Loop Subdivision
- Reflect
- Rotational Extrusion
- Shrink
- Smooth
- Subdivide
- Tessellate
- Tetrahedralize
- Triangle Strips
- Triangulate

Technically, the Ribbon and Tube filters should fall into this list. However, as they only work on 1D cells in poly data, the input data is usually small and of little concern.

This similar set of filters also output unstructured grids, but they also tend to reduce some of this data. Be aware though that this data reduction is often smaller than the overhead of converting to unstructured data. Also note that the reduction is often not well balanced. It is possible (often likely) that a single process may not lose any cells. Thus, these filters should be used with caution on unstructured data and extreme caution on structured data.

- Clip
- Decimate
- Extract Cells by Region
- Extract Selection
- Quadric Clustering
- Threshold

Similar to the items in the preceding list, Extract Subset  performs data reduction on a structured dataset, but also outputs a structured dataset. So the warning about creating new data still applies, but you do not have to worry about converting to an unstructured grid.

This next set of filters also outputs unstructured data, but it also performs a reduction on the dimension of the data (for example 3D to 2D), which results in a much smaller output. Thus, these filters are usually safe to use with unstructured data and require only mild caution with structured data.

- Cell Centers
- Contour
- Extract CTH Parts
- Extract Surface
- Feature Edges
- Mask Points
- Outline (curvilinear)
- Slice
- Stream Tracer

These filters do not change the connectivity of the data at all. Instead, they only add field arrays to the data. All the existing data is shallow copied. These filters are usually safe to use on all data.

- Block Scalars
- Calculator
- Cell Data to Point Data
- Curvature
- Elevation
- Generate Surface Normals
- Gradient
- Level Scalars
- Median
- Mesh Quality
- Octree Depth Limit
- Octree Depth Scalars
- Point Data to Cell Data
- Process Id Scalars
- Python Calculator
- Random Vectors
- Resample with dataset
- Surface Flow
- Surface Vectors
- Texture Map to…
- Transform
- Warp (scalar)
- Warp (vector)

This final set of filters are those that either add no data to the output (all data of consequence is shallow copied) or the data they add is generally independent of the size of the input. These are almost always safe to add under any circumstances (although they may take a lot of time).

- Annotate Time
- Append Attributes
- Extract Block
- Extract Level
- Glyph
- Group Datasets
- Histogram
- Integrate Variables
- Normal Glyphs

- Outline
- Outline Corners
- Plot Global Variables Over Time
- Plot Over Line
- Plot Selection Over Time
- Probe Location
- Temporal Shift Scale
- Temporal Snap-to-Time-Steps
- Temporal Statistics

There are a few special case filters that do not fit well into any of the previous classes. Some of the filters, currently Temporal Interpolator and Particle Tracer, perform calculations based on how data changes over time. Thus, these filters may need to load data for two or more instances of time, which can double or more the amount of data needed in memory. The Temporal Cache filter will also hold data for multiple instances of time. Also keep in mind that some of the temporal filters such as the temporal statistics and the filters that plot over time may need to iteratively load all data from disk. Thus, it may take an impractically long amount of time even though it does not require any extra memory.

The Programmable Filter is also a special case that is impossible to classify. Since this filter does whatever it is programmed to do, it can fall into any one of these categories.

### Culling Data

When dealing with large data, it is clearly best to cull out data whenever possible, and the earlier the better. Most large data starts as 3D geometry and the desired geometry is often a surface. As surfaces usually have a much smaller memory footprint than the volumes that they are derived from, it is best to convert to a surface soon. Once you do that, you can apply other filters in relative safety.

A very common visualization operation is to extract isosurfaces from a volume using the Contour filter. The Contour filter usually outputs geometry much smaller than its input. Thus, the Contour filter should be applied early if it is to be used at all. Be careful when setting up the parameters to the Contour filter because it still is possible for it to generate a lot of data. This obviously can happen if you specify many isosurface values. High frequencies such as noise around an isosurface value can also cause a large, irregular surface to form.

Another way to peer inside of a volume is to perform a Slice on it. The Slice filter will intersect a volume with a plane and allow you to see the data in the volume where the plane intersects. If you know the relative location of an interesting feature in your large dataset, slicing is a good way to view it.

If you have little *a-priori* knowledge of your data and would like to explore the data without paying the memory and processing time for the full dataset, you can use the Extract Subset filter to subsample the data. The subsampled data can be dramatically smaller than the original data and should still be well load balanced. Of course, be aware that you may miss small features if the subsampling steps over them and that once you find a feature you should go back and visualize it with the full dataset.

There are also several features that can pull out a subset of a volume: Clip , Threshold , Extract Selection , and Extract Subset can all extract cells based on some criterion. Be aware, however, that the extracted cells are almost never well balanced; expect some processes to have no cells removed. Also, all of these filters with the exception of Extract Subset will convert structured data types to unstructured grids. Therefore, they should not be used unless the extracted cells are of at least an order of magnitude less than the source data.

When possible, replace the use of a filter that extracts 3D data with one that will extract 2D surfaces. For example, if you are interested in a plane through the data, use the Slice filter rather than the Clip filter. If you are interested in knowing the location of a region of cells containing a particular range of values, consider using the Contour filter to generate surfaces at the ends of the range rather than extract all of the cells with the Threshold  filter. Be aware that substituting filters can have an effect on downstream filters. For example, running the Histogram filter after Threshold  will have an entirely different effect than running it after the roughly equivalent Contour  filter.

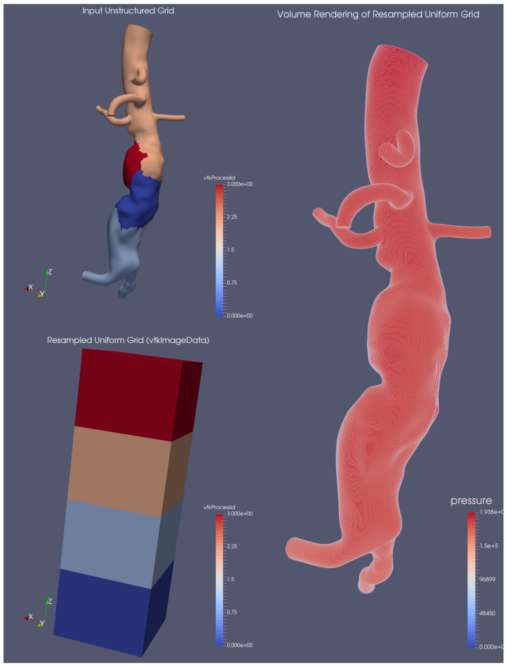
### Downsamplng

Downsampling is also frequently helpful for very large data. For example, the Extract Subset filter reduce the size of Structured DataSets by simpling taking strided samples along the i, j and k axes.

For unstructured grids the Quadric Clustering filter downsamples unstructured data into a smaller dataset with similar appearance by averaging vertices within cells of a grid. **ParaView** injects this algorithm into the display pipeline while you interact with the camera as described in the next section.

In some cases though, it can be more effective to work with structured data than unstructured data. In particular structured data volume rendering algorithms are usually much faster. In general structured data representations are very compact, adding very little overhead to the raw data values. Unstructured data types require 12 bytes of memory storage for every vertex and at least 8 bytes of storage to define each cell beyond the normal storage for cell and point aligned values. Structured types use a total of 36 bytes to represent the entire set of vertices and cells in comparison.

Fortunately you can convert from unstructured to structured easily with the Resample To Image filter. When the sizes of the cells in the input do not very widely, and can thus be approximated well by constant sized voxels, this can be very effective. This filter internally makes use of the DIY2 block-parallel communication and computation library to communicate and transfer data among the parallel nodes.



There is a companion filter Resample With DataSet which takes in a source and an input object and resamples values from the source onto the input, which does not have to be a regular grid. This also uses DIY2. This is used to assigning new values onto a particular shaped object.

### Parallel Rendering Details

Rendering is the process of synthesizing the images that you see based on your data. The ability to effectively interact with your data depends highly on the speed of the rendering. Thanks to advances in 3D hardware acceleration, fueled by the computer gaming market, we have the ability to render 3D quickly even on moderately priced computers. But, of course, the speed of rendering is proportional to the amount of data being rendered. As data gets bigger, the rendering process naturally gets slower.

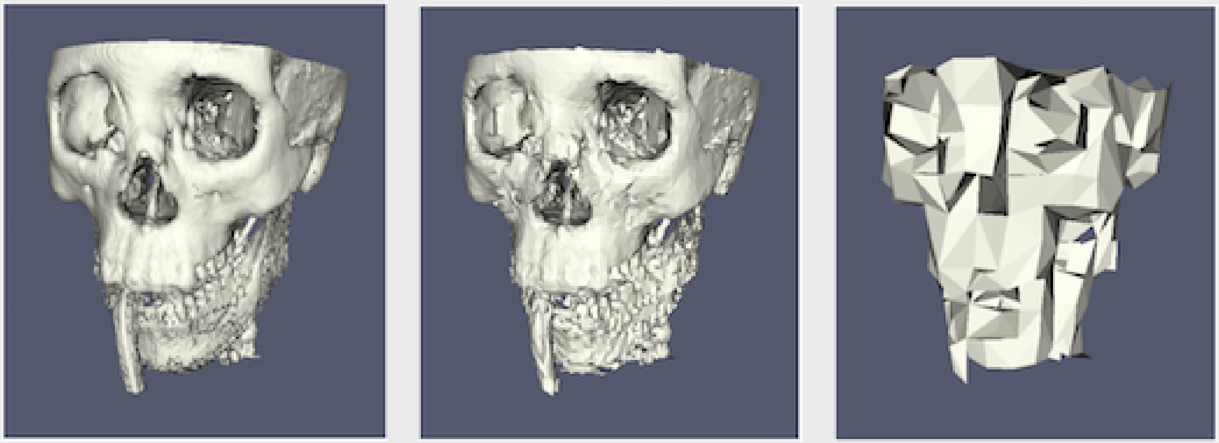
To ensure that your visualization session remains interactive, **ParaView** supports two modes of rendering that are automatically flipped as necessary. In the first mode, **still render**, the data is rendered at the highest level of detail. This rendering mode ensures that all of the data is represented accurately. In the second mode, **interactive render**, speed takes precedence over accuracy. This rendering mode endeavors to provide a quick rendering rate regardless of data size.

While you are interacting with a 3D view, for example rotating, panning, or zooming with the mouse, **ParaView** uses an interactive render. This is because during the interaction a high frame rate is necessary to make these features usable and because each frame is immediately replaced with a new rendering while the interaction is occurring so that fine details are less important during this mode. At any time when interaction of the 3D view is not taking place, **ParaView** uses a still render so that the full detail of the data is available as you study it. As you drag your mouse in a 3D view to move the data, you may see an approximate rendering while you are moving the mouse, but the full detail will be presented as soon as you release the mouse button.

The interactive render is a compromise between speed and accuracy. As such, many of the rendering parameters concern when and how lower levels of detail are used.

### Basic Rendering Settings

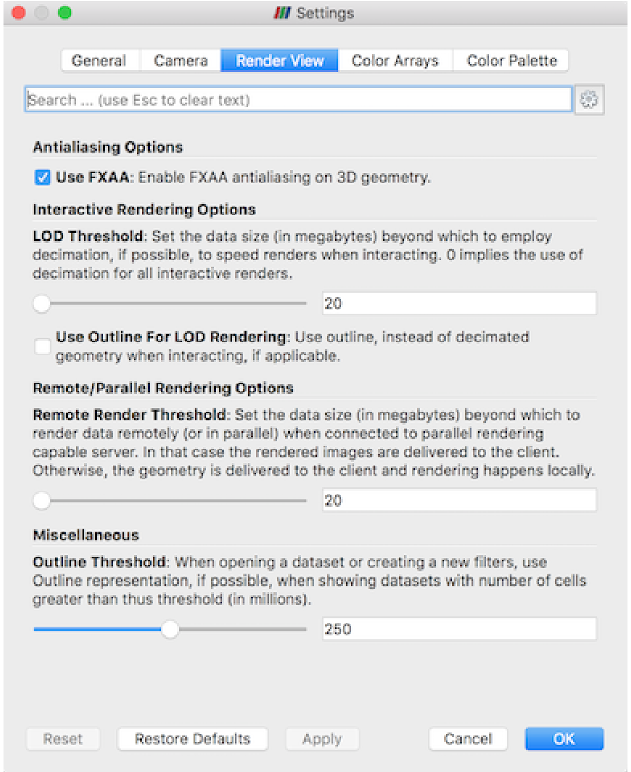
Some of the most important rendering options are the LOD parameters. During interactive rendering, the geometry may be replaced with a lower **level of detail** (**LOD**), an approximate geometry with fewer polygons.



The resolution of the geometric approximation can be controlled. In the proceeding images, the left image is the full resolution; the middle image is the default decimation for interactive rendering, and the right image is **ParaView**'s maximum decimation setting.

The 3D rendering parameters are located in the settings dialog box which is accessed in the menu from Edit → Settings (**ParaView** → Preferences on the Mac). The rendering options in the dialog are in the Render View tab.

The options pertaining to the geometric decimation for interactive rendering are located in a section labeled Interactive Rendering Options. Some of these options are considered advanced, so to access them you have to either toggle on the



**Antialiasing Options**

☑ **Use FXAA**: Enable FXAA antialiasing on 3D geometry.

**Interactive Rendering Options**

**LOD Threshold**: Set the data size (in megabytes) beyond which to employ decimation, if possible, to speed renders when interacting. 0 implies the use of decimation for all interactive renders.

20

□ **Use Outline For LOD Rendering**: Use outline, instead of decimated geometry when interacting, if applicable.

**Remote/Parallel Rendering Options**

**Remote Render Threshold**: Set the data size (in megabytes) beyond which to render data remotely (or in parallel) when connected to parallel rendering capable server. In that case the rendered images are delivered to the client. Otherwise, the geometry is delivered to the client and rendering happens locally.

20

**Miscellaneous**

**Outline Threshold**: When opening a dataset or creating a new filters, use Outline representation, if possible, when showing datasets with number of cells greater than thus threshold (in millions).

250

advanced options with the button or search for the option using the edit box at the top of the dialog. The interactive rendering options include the following.

- Set the data size at which to use a decimated geometry in interactive rendering. If the geometry size is under this threshold, **ParaView** always renders the full geometry. Increase this value if you have a decent graphics card that can handle larger data. Try decreasing this value if your interactive renders are too slow.
- Set the factor that controls how large the decimated geometry should be. This control is set to a value between 0 and 1. 0 produces a very small number of triangles but possibly with a lot of distortion. 1 produces more detailed surfaces but with larger geometry.
- Add a delay between an interactive render and a still render. **ParaView** usually performs a still render immediately after an interactive motion is finished (for example, releasing the mouse button after a rotation). This option can add a delay that can give you time to start a second interaction before the still render starts, which is helpful if the still render takes a long time to complete.
- Use an outline in place of decimated geometry. The outline is an alternative for when the geometry decimation takes too long or still produces too much geometry. However, it is more difficult to interact with just an outline.

**ParaView** contains many more rendering settings. Here is a summary of some other settings that can effect the rendering performance regardless of whether **ParaView** is run in client-server mode or not. These options are spread among several categories, and several are considered advanced.

Geometry Mapper Options

- Enable or disable the use of display lists. Display lists are internal structures built by graphics systems. They can potentially speed up rendering but can also take up memory.

Translucent Rendering Options

- Enable or disable depth peeling. Depth peeling is a technique **ParaView** uses to properly render translucent surfaces. With it, the top surface is rendered and then "peeled away" so that the next lower surface can be rendered and so on. If you find that making surfaces transparent really slows things down or renders completely incorrectly, then your graphics hardware may not be implementing the depth peeling extensions well; try shutting off depth peeling.

- Set the maximum number of peels to use with depth peeling. Using more peels allows more depth complexity but allowing less peels runs faster. You can try adjusting this parameter if translucent geometry renders too slow or translucent images do not look correct.

Miscellaneous

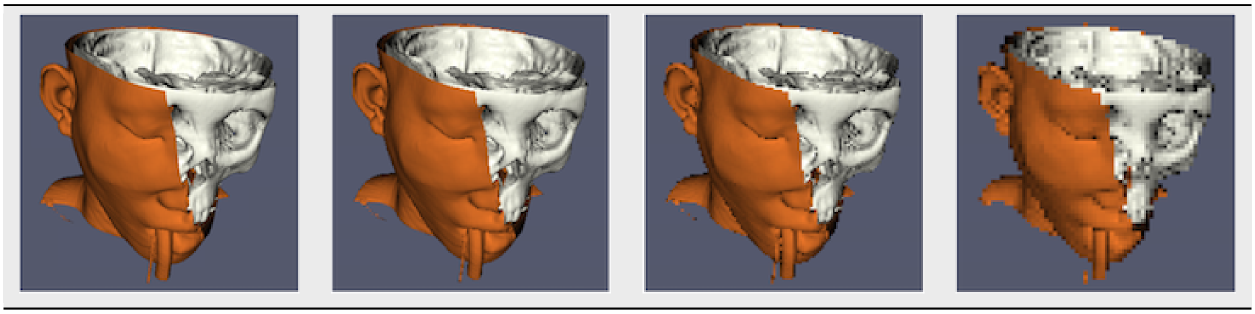
- When creating very large datasets, default to the outline representation. Surface representations usually require **ParaView** to extract geometry of the surface, which takes time and memory. For data with size above this threshold, use the outline representation, which has very little overhead, by default instead.
- Show or hide annotation providing rendering performance information. This information is handy when diagnosing performance problems.

Note that this is not a complete list of **ParaView** rendering settings. We have left out settings that do not significantly effect rendering performance. We have also left out settings that are only valid for parallel client-server rendering, which are discussed in Section 4.1.4.

### Image Level of Detail

The overhead incurred by the parallel rendering algorithms is proportional to the size of the images being generated. Also, images generated on a server must be transfered to the client, a cost that is also proportional to the image size. To help increase the frame rate during interaction, **ParaView** introduces a new LOD parameter that controls the size of the images.

During interaction while parallel rendering, **ParaView** can optionally subsample the image. That is, **ParaView** will reduce the resolution of the image in each dimension by a factor during interaction. Reduced images will be rendered, composited, and transfered. On the client, the image is inflated to the size of the available space in the GUI.

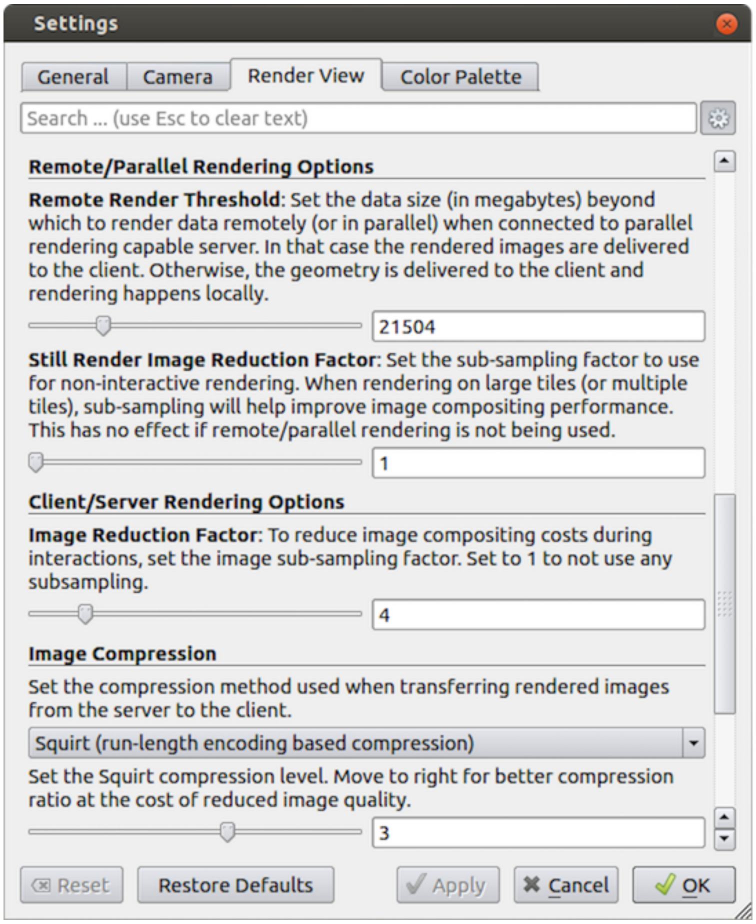


The resolution of the reduced images is controlled by the factor with which the dimensions are divided. In the preceding images, the left image has the full resolution. The following images were rendered with the resolution reduced by a factor of 2, 4, and 8, respectively.

**ParaView** also has the ability to compress images before transferring them from server to client. Compression, of course, reduces the amount of data transferred and therefore makes the most of the available bandwidth. However, the time it takes to compress and decompress the images adds to the latency.

**ParaView** contains three different image compression algorithms for client-server rendering. The oldest is a custom algorithm called **Squirt**, which stands for Sequential Unified Image Run Transfer. Squirt is a run-length encoding compression that reduces color depth to increase run lengths. The second algorithm uses the **Zlib** compression library, which implements a variation of the Lempel-Ziv algorithm. Zlib typically provides better compression than Squirt, but takes longer to perform and hence adds to the latency. The most recent addition is the **LZ4** algorithm which is tuned for fast compression and decompression.

### Parallel Render Parameters



**Remote/Parallel Rendering Options**

**Remote Render Threshold**: Set the data size (in megabytes) beyond which to render data remotely (or in parallel) when connected to parallel rendering capable server. In that case the rendered images are delivered to the client. Otherwise, the geometry is delivered to the client and rendering happens locally.

21504

**Still Render Image Reduction Factor**: Set the sub-sampling factor to use for non-interactive rendering. When rendering on large tiles (or multiple tiles), sub-sampling will help improve image compositing performance. This has no effect if remote/parallel rendering is not being used.

1

**Client/Server Rendering Options**

**Image Reduction Factor**: To reduce image compositing costs during interactions, set the image sub-sampling factor. Set to 1 to not use any subsampling.

4

**Image Compression**

Set the compression method used when transferring rendered images from the server to the client.

Squirt (run-length encoding based compression)

Set the Squirt compression level. Move to right for better compression ratio at the cost of reduced image quality.

3

Like the other 3D rendering parameters, the parallel rendering parameters are located in the settings dialog box, which is accessed in the menu from Edit → Settings (**ParaView** → Preferences on the Mac). The parallel rendering options in the dialog are in the Render View tab (intermixed with several other rendering options such as those described in Section 4.1.4). The parallel and client-server options are divided among several categories, and several are considered advanced.

Remote/Parallel Rendering Options

- Set the data size at which to render remotely in parallel or to render locally. If the geometry is over this threshold (and **ParaView** is connected to a remote server), the data is rendered in parallel remotely and images are sent back to the client. If the geometry is under this threshold, the geometry is sent back to the client and images are rendered locally on the client.
- Set the sub-sampling factor for still (non-interactive) rendering. Some large displays have more resolution than is really necessary, so this sub-sampling reduces the resolution of all images displayed.

Client/Server Rendering Options

- Set the interactive subsampling factor. The overhead of parallel rendering is proportional to the size of the images generated. Thus, you can speed up interactive rendering by specifying an image subsampling rate. When this box is checked, interactive renders will create smaller images, which are then magnified when displayed. This parameter is only used during interactive renders.

Image Compression

- Before images are shipped from server to client, they optionally can be compressed using one of the three compression algorithms: Squirt, Zlib or LZ4. To make the compression more effective, each algorithm has one or more tunable parameters that let you customize the behavior and target level of compression.
- Suggested image compression presets are provided for several common network types. When attempting to select the best image compression options, try starting with the presets that best match your connection.

### Parameters for Large Data

The default rendering parameters are suitable for most users. However, when dealing with very large data, it can help to tweak the rendering parameters. The optimal parameters depend on your data and the hardware **ParaView** is running on, but here are several pieces of advice that you should follow.

- Try turning off display lists. Turning this option off will prevent the graphics system from building special rendering structures. If you have graphics hardware, these rendering structures are important for feeding the GPUs fast enough. However, if you do not have GPUs, these rendering structures do not help much.
- If there is a long pause before the first interactive render of a particular dataset, it might be the creation of the decimated geometry. Try using an outline instead of decimated geometry for interaction. You could also try lowering the factor of the decimation to 0 to create smaller geometry.
- Avoid shipping large geometry back to the client. The remote rendering will use the power of entire server to render and ship images to the client. If remote rendering is off, geometry is shipped back to the client. When you have large data, it is always faster to ship images than to ship data (although if your network has a high latency, this could become problematic for interactive frame rates).
- Adjust the interactive image sub-sampling for client-server rendering as needed. If image compositing is slow, if the connection between client and server has low bandwidth, or if you are rendering very large images, then a higher subsample rate can greatly improve your interactive rendering performance.
- Make sure Image Compression is on. It has a tremendous effect on desktop delivery performance, and the artifacts it introduces, which are only there during interactive rendering, are minimal. Lower bandwidth connections can try using Zlib or LZ4 instead of Squirt compression. Zlib will create smaller images at the cost of longer compression/decompression times.
- If the network connection has a high latency, adjust the parameters to avoid remote rendering during interaction. In this case, you can try turning up the remote rendering threshold a bit, and this is a place where using the outline for interactive rendering is effective.
- If the still (non-interactive) render is slow, try turning on the delay between interactive and still rendering to avoid unnecessary renders.

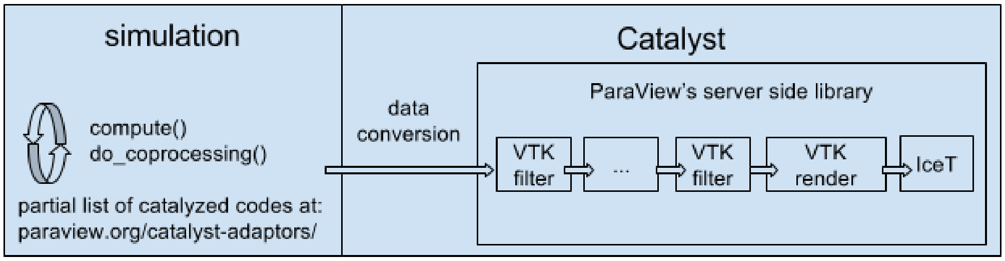
### Catalyst

For small scale data, running the **ParaView** GUI in serial mode is likely acceptable. For large scale data, interactive parallel visualization is quite useful. For very large scale data, batch mode parallel processing is more effective because of practical concerns like batch system queue wait time before a parallel job is launched and system policies that limit the size and duration of interactive jobs.

For extreme scale visualization even batch mode is beginning to be impractical though as comparatively slow disks limit the size of data that can be saved off by the simulation at runtime. In this domain **ParaView**'s Catalyst configuration is recommended as an *in situ* analysis and visualization library.

**Catalyst** is a visualization framework that packages either portions or the entirety of **ParaView**'s parallel server framework so that it can be linked into and called from arbitrary simulation codes. Slow IO systems are largely bypassed via an adaptor mechanism whereby the simulation code's data structures are translated or ideally reused directly by **ParaView** while still in RAM.

With Catalyst, full resolution simulation products stay in memory instead of being saved to and later loaded back from disk. As the simulation progresses, it periodically calls into **ParaView**. In this way it is possible to immediately generate smaller derived datasets, images, plots, etc. that would otherwise be generated during post processing after a much longer, human in the loop, delay.


simulation
compute()
do_coprocessing()
partial list of catalyzed codes at:
paraview.org/catalyst-adaptors/
data
conversion
Catalyst
ParaView's server side library
VTK
filter
...
VTK
filter
VTK
render
IceT

As explained more fully in the **ParaView** Catalyst User's Guide there are two components to making use of Catalyst. The first is to Catalyze the simulation. This is where a software developer compiles Catalyst into the simulation, adds a handful of (typically three) function calls to it, and fleshes out an adaptor template with working code that produces VTK datasets from the simulation's own internal data structures.

Once the simulation has been Catalyzed, day to day users need to define what **ParaView** should do with the data it is given. Catalyst can produce data extracts, images, statistical quantities, etc. Any of these can be made with Python or even lower level Fortran, C or C++ coding to maximize runtime efficiency, but it is more flexible and convenient to do so with recorded Catalyst scripts which are demonstrated in the next exercise.

With Catalyst scripts the simulation input deck references a Python file that defines the visualization pipeline. The script can do nearly everything that **ParaView** batch scripts do and is generally created with help from the **ParaView** GUI. To create the script one loads a representative dataset, creates a pipeline, designates the set of inputs and outputs to and output from the pipeline, and then simply saves out the Python file. Catalyst scripts are very similar to recorded Python traces.

The user interface for defining the inputs and outputs is found inside the the Catalyst Script Generator plugin. Our final exercise demonstrates with a small example.

**Exercise 4.4 (Catalyst)**

We demonstrate a Catalyst workflow using the **ParaView** binaries and the PythonFullExample code from the **ParaView** source tree. PythonFullExample is a toy simulation that uses numpy and mpi4py to update a regular grid as time evolves. The example consists of four files. Open and read these to begin.

The computational kernel is in fedatastructures.py. Inspection of the code shows that it has no particular dependency on **ParaView**, and simply makes up a structured grid in parallel with numpy and mpi4py.

From fedatastructures.py:

```
def Update(self, time):
   self.Velocity =
   numpy.zeros((self.Grid.GetNumberOfPoints(), 3))
   self.Velocity = self.Velocity + time
   self.Pressure = numpy.zeros(self.Grid.GetNumberOfCells())
```

The main loop is found in the fedriver.py. If the doCoprocessing variable is false, this falls back to a simple loop which runs the simulation to update the in-memory array over time. When the variable is true the Catalyst library is exercised via three function calls: initialize(), coprocess() and finalize().

From fedriver.py:

```
import numpy
import sys
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

import fedatastructures

grid = fedatastructures.GridClass([10, 12, 10], [2, 2, 2])
attributes = fedatastructures.AttributesClass(grid)
doCoprocessing = True

if doCoprocessing:
    import coprocessor
    coprocessor.initialize()
    coprocessor.addscript("cpscript.py")

for i in range(100):
    attributes.Update(i)
    if doCoprocessing:
        import coprocessor
        coprocessor.coprocess(i, i, grid, attributes)

if doCoprocessing:
    import coprocessor
    coprocessor.finalize()
```

fedriver.py and fedatastructures.py represent the simulation code. The Catalyst parts consist of the general purpose adaptor code, found within the coprocessor.py, and the pipeline definition, found in cpscript.py. In coprocessor.py note in particular where the coprocess function populates the dataDescription data structure to give fedatastructure's content over to **ParaView**. The coprocessor also takes in, with the addscript() call, the pipeline definition file.

From coprocessing.py:

```
if coProcessor.RequestDataDescription(dataDescription):
    import fedatastructures
    imageData = vtk.vtkImageData()
    imageData.SetExtent(\
      grid.XStartPoint, grid.XEndPoint, \
      0, grid.NumberOfYPoints-1, \
      0, grid.NumberOfZPoints-1)
    imageData.SetSpacing(grid.Spacing)

    velocity = paraview.numpy_support.numpy_to_vtk(attributes.Velocity)
    velocity.SetName("velocity")
    imageData.GetPointData().AddArray(velocity)

    pressure = numpy_support.numpy_to_vtk(attributes.Pressure)
    pressure.SetName("pressure")
    imageData.GetCellData().AddArray(pressure)
```

The starting script for this example is called cpscript.py. This simply saves out the input it is given in a format that the **ParaView** GUI can readily understand. The internals of the PipelineClass is essentially a **ParaView** batch script that takes as input not a file but the datadescription class from the adaptor.

From cpscript.py:

```
class Pipeline:
    filename_6_pvti = \
        coprocessor.CreateProducer( datadescription, "input" )

    # create a new 'Parallel ImageData Writer'
    imageDataWriter1 = \
        servermanager.writers.XMLPImageDataWriter(Input=filename_6_pvti)

    # register the writer with coprocessor
    # and provide it with information such as the filename to use,
    # how frequently to write the data, etc.
    coprocessor.RegisterWriter(imageDataWriter1, \
        filename="fullgrid_%t.pvti", freq=100)

    Slice1 = Slice( \
        Input=filename_6_pvti, guiName="Slice1", \
        Crinkleslice=0, SliceOffsetValues=[0.0], \
        Triangulatetheslice=1, SliceType="Plane" )
    Slice1.SliceType.Offset = 0.0
    Slice1.SliceType.Origin = [9.0, 11.0, 9.0]
    Slice1.SliceType.Normal = [1.0, 0.0, 0.0]

    # create a new 'Parallel PolyData Writer'
    parallelPolyDataWriter1 = \
        servermanager.writers.XMLPPolyDataWriter(Input=Slice1)

    # register the writer with coprocessor
    # and provide it with information such as the filename to use,
    # how frequently to write the data, etc.
    coprocessor.RegisterWriter(\
        parallelPolyDataWriter1, filename='slice_%t.pvtp', freq=10)

    return Pipeline()
```

Now let us run the simulation code to produce some data.

- On Mac:

```
/Applications/ParaView-x.x.x.app/Contents/MacOS/mpiexec  -np 2 /Applications/
↪ParaView-x.x.x.app/Contents/bin/pvbatch -symmetric fedriver.py
```

- On Linux:

```
/usr/local/lib/paraview-x.x.x/mpiexec -np 2 /usr/local/bin/pvbatch -symmetric␣
↪fedriver.py
```

- On Windows:

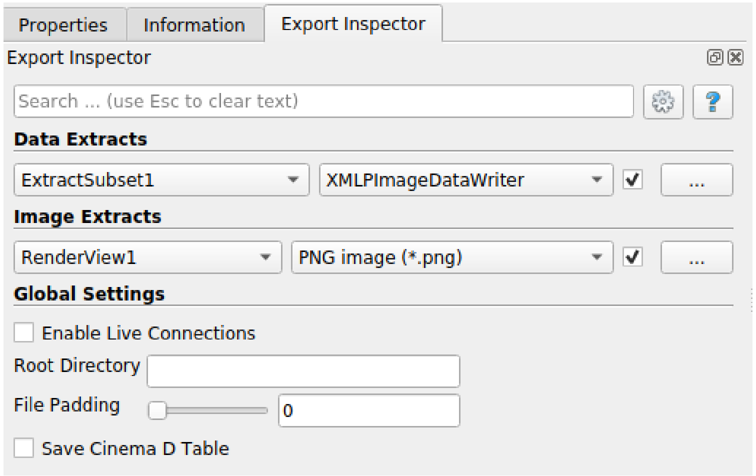
```
mpiexec -np 2 "C:/Program Files/ParaView x.x.x/bin/pvbatch" -symmetric fedriver.py
```

The example will run and produce several files, among them fullgrid_0.pvti which is a raw dump of the simulation's full output. Let us use that to customize Catalyst's output.

First open the file in the **ParaView** GUI and inspect it. You will find that this is a very simple structured grid with two variables, pressure and velocity. Now insert the File → Extract Subset filter into the pipeline to demonstrate some simple but useful processing.

Next let's use the Export Inspector to declare a set of files that you want Catalyst to produce at run time. As we will see in a moment, the **ParaView** GUI is able to generate Catalyst scripts by capturing the current state of the visualization constructed in **ParaView**. However, the state does not include file creation, which is typically done in interactive sessions via File → Save Data or File → Save Screenshot actions.

Since **ParaView** 5.6, the Export Inspector is the place for defining Catalyst's data products. Open it by clicking Export Inspector under the Catalyst menu. By default the panel will show up in a new tab near the Properties Panel. Select it, if necessary, to bring it to the front.



**Data Extracts**

ExtractSubset1 | XMLPImageDataWriter ☑ ...

**Image Extracts**

RenderView1 | PNG image (*.png) ☑ ...

**Global Settings**

□ Enable Live Connections

Root Directory

File Padding 0

□ Save Cinema D Table

The Data Extracts section at the top allows you to attach a writer to the currently active element within the visualization pipeline. Make sure ExtractSubset1 is selected in the Pipeline Browser and then choose XMLPImageDataWriter from the file format pull down to the right of this filter name in Data Extracts. To confirm that you want Catalyst to perform this particular data export at run time, click on the checkbox to the right of the file format.

If you want to change the writer's default settings, click on the configure button with the … label to the right of the checkbox. In the Save Data Options dialog that pops up, you can control things like the compression level, the frequency at which Catalyst generates files, and the filename. The default filename is composed from the producing filter's name, the time step and the file format extension. Let's change it from "ExtractSubset1_%t.pvti" to "smallgrid_%t.pvti". Here "%t" gets replaced with the iteration number when the simulation runs. Later, when we run the simulation with this script, we will get subsampled results instead of the full grid.

To add rendered images to the simulation run, configure them with the Image Extracts section of the Export Inspector. Just like with data extracts, you configure outputs for the currently active view here. Do so by making sure that "RenderView1" is active and then hit the checkbox to enable png dumps of this view.

To save your visualization session in the form of a Catalyst script, click Catalyst → Export Catalyst Script. Note that all of your choices in the Export Inspector are kept with the rest of the **ParaView** state, so File → Save State can make it easy to modify your setup after the fact.

Finally quit **ParaView** and run the simulation again. The files produced will now include png files for screen captures at each time step, and vti files corresponding to the output of the Extract Subset filter at each timestep. Simply repeat the process, adding filters and views for example, to set up any type of data generator for the simulation.

---

Note this is but one interface to using Catalyst. If simulation users are uncomfortable with **ParaView** and/or Python, it is entirely possible for simulation developers to instrument the simulation code with predefined and parameterized pipelines with or without resorting to Python. It is also possible to create minimized Catalyst editions that consist of just a small portion of the larger **ParaView** code base.

As is the case with **ParaView** itself, Catalyst is evolving rapidly. It already has more advanced offshoots than described here. One, is a live data capability in which one can connect a **ParaView** client to a running Catalyzed simulation in order to check in on its progress from time to time and do a limited amount of computational steering and debugging. Another is Cinema, which is an image based framework for deferred visualization of automatically generated simulation results organized in a database. For information on these we refer the reader to the **ParaView** mailing list and websites.

### 4.1.5 Further Reading

Thank you for participating in this tutorial. Hopefully you have learned enough to get you started visualizing large data with ParaView. Here are some sources for further reading.

The documentation page on ParaView's web site contains a list of resources available for further learning and questions: https://www.paraview.org/documentation .

The ParaView User's Guide is a good resource to have with ParaView. It provides many other instructions and more detailed descriptions on many features. The ParaView guide can be accessed from the ParaView documentation page.

The ParaView Wiki is full of information that you can use to help you set up and use ParaView: https://www.paraview.org/Wiki/ParaView.

In particular, those of you who wish to install a parallel ParaView server should consult the appropriate build and install pages: https://www.paraview.org/Wiki/Setting_up_a_ParaView_Server .

If you are interested in learning more about visualization or more specifics about the filters available in ParaView, consider picking up the following visualization textbook: [SML06].

If you plan on customizing ParaView, the previous books and web pages have lots of information. For more information about using VTK, the underlying visualization library, and Qt, the GUI library, consider the following books have more information: [KInc10], [BS08].

If you are interested about the design of parallel visualization and other features of the VTK pipeline, there are several technical papers available: [Mor13], [ALS+00], [ABM+01], [CGM+06], [ADM+07], [BGM+07].

If you are interested in the algorithms and architecture for ParaView's parallel rendering, there are also many technical articles on this as well: [MWP01], [MT03], [MAF07].

## 4.2 Classroom Tutorials

**Classroom Tutorials** contain beginning, advanced, Python and batch, and targeted tutorial lessons on how to use **ParaView**, created by Sandia National Laboratories. These **tutorials** were created by W. Alan Scott and are presented as a 3-hour class internally within Sandia National Laboratories.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY

## 4.2.1 Beginning: ParaView

### Introduction

This tutorial will describe how to start paraview, find information and help about paraview, and describe some of the more important controls used by ParaView.

The **ParaView** web site is located at https://www.paraview.org. New versions of paraview can be found here, along with different tutorials and advice. **ParaView** versions include 32 and 64 bit versions of Linux, Windows and macOS. Training (practice) data is included in all ParaView downloads, and can be accessed through **File** → **Open** → **Examples**.

**Did you know?**

paraview now has an integrated help. It is found under the menu item **Help** → **Help**.

### A simple example

### Start ParaView

- On Windows, go to **Start** → **All Programs** → **ParaView x.x.x** → and click **ParaView**.
- On Mac, in the finder, go to the ParaView directory (in the Application directory where you installed ParaView) and click on **paraview**.
- On Linux go into the **ParaView** directory (where you downloaded ParaView), then **bin** and type **paraview**.

### Startup Screen

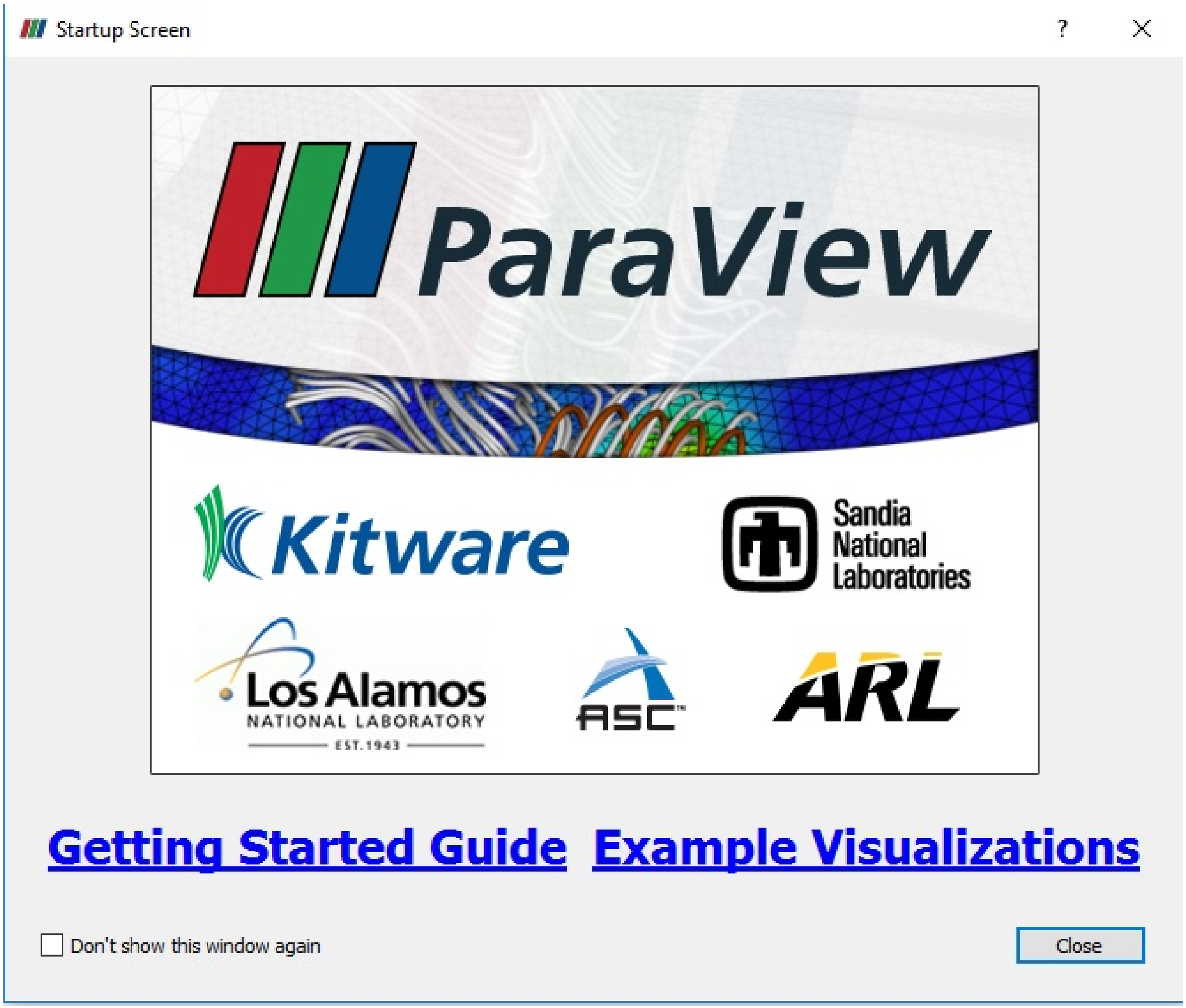
- The **Startup Screen** includes two important links. Both of these links can also be found from the **Help** menu. They are the **Getting Started Guide** and **Example Visualizations**. If desired click the **Don't show this window again**, and then click **Close**.

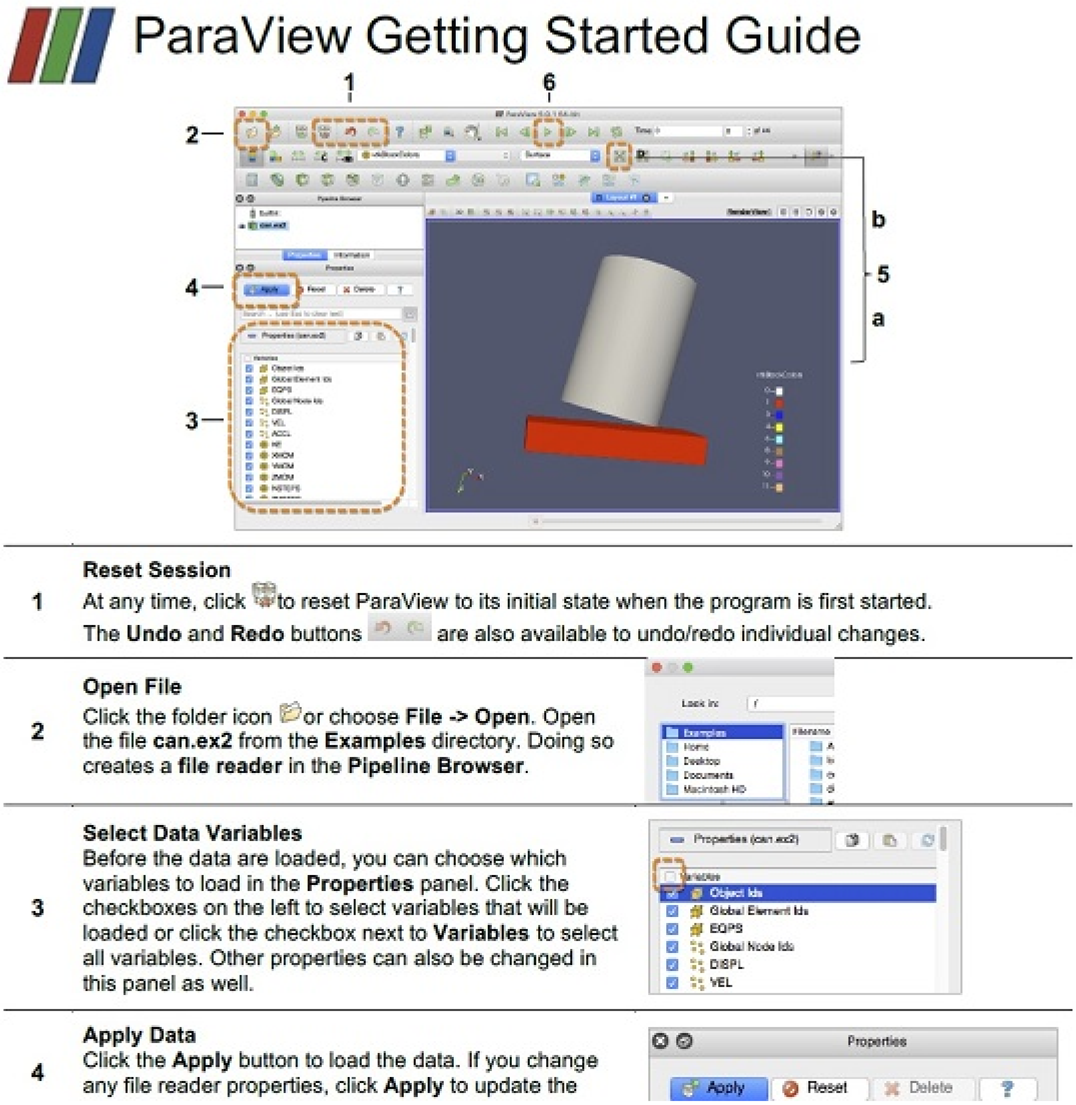
### Getting Started Guide

- The **Getting Started Guide** is a two page mini tutorial that shows fundamental **ParaView** usage.
- **Example Visualizations** provide six finished visualizations. You can then play around with a **ParaView** pipeline.

### Help Menu

- The **Help** menu looks like this. Step through each menu option, then close the **Help** menu.


Startup Screen
?
ParaView
Kitware
Sandia National Laboratories
Los Alamos NATIONAL LABORATORY EST.1943
ASC
ARL
Getting Started Guide
Example Visualizations
□ Don't show this window again
Close



## ParaView Getting Started Guide

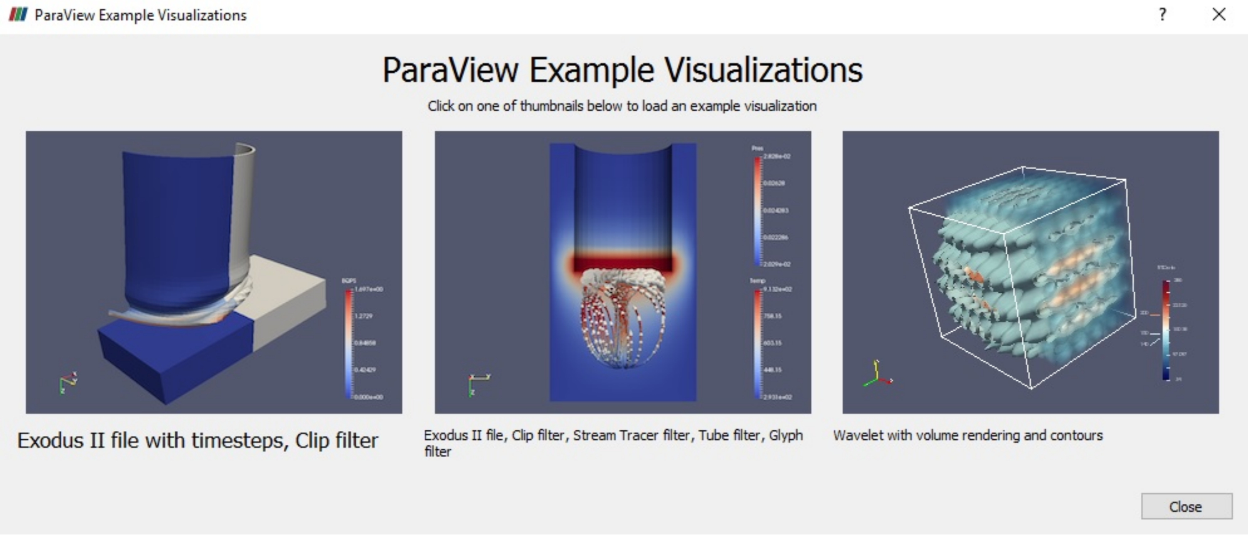
1
6
2
4
3
b
5
a
Properties
Apply
Reset
Delete
Look in:
Examples
Home
Desktop
Documents
Macintosh HD
Filename
Properties (can.ex2)
Variables
Object Ids
Global Element Ids
EQPS
Global Node Ids
DISPL
VEL
Properties

1 **Reset Session**
At any time, click to reset ParaView to its initial state when the program is first started. The **Undo** and **Redo** buttons are also available to undo/redo individual changes.

2 **Open File**
Click the folder icon or choose **File -> Open**. Open the file **can.ex2** from the **Examples** directory. Doing so creates a **file reader** in the **Pipeline Browser**.

3 **Select Data Variables**
Before the data are loaded, you can choose which variables to load in the **Properties** panel. Click the checkboxes on the left to select variables that will be loaded or click the checkbox next to **Variables** to select all variables. Other properties can also be changed in this panel as well.

4 **Apply Data**
Click the **Apply** button to load the data. If you change any file reader properties, click **Apply** to update the



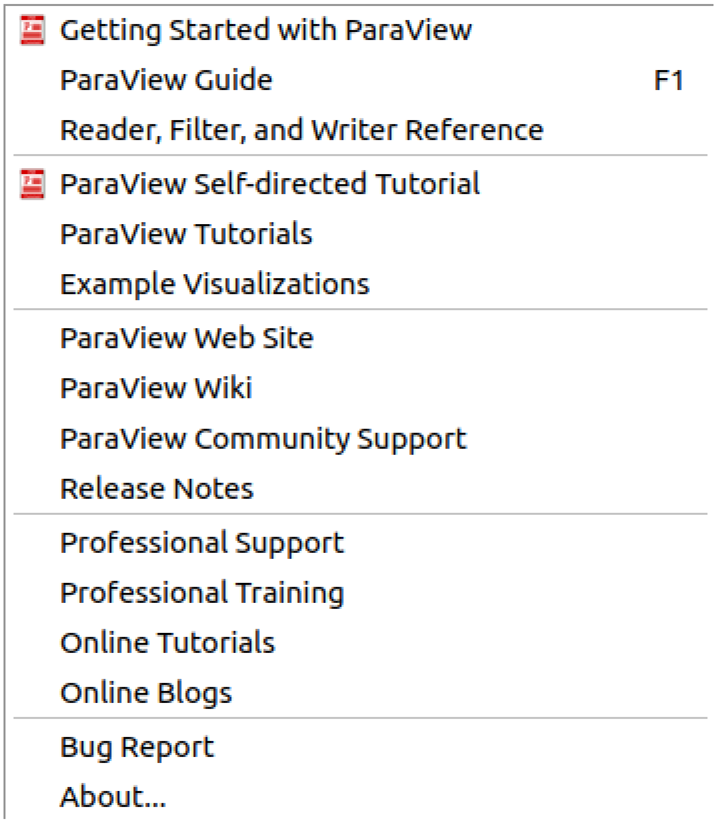
## ParaView Example Visualizations

Click on one of thumbnails below to load an example visualization

Exodus II file with timesteps, Clip filter

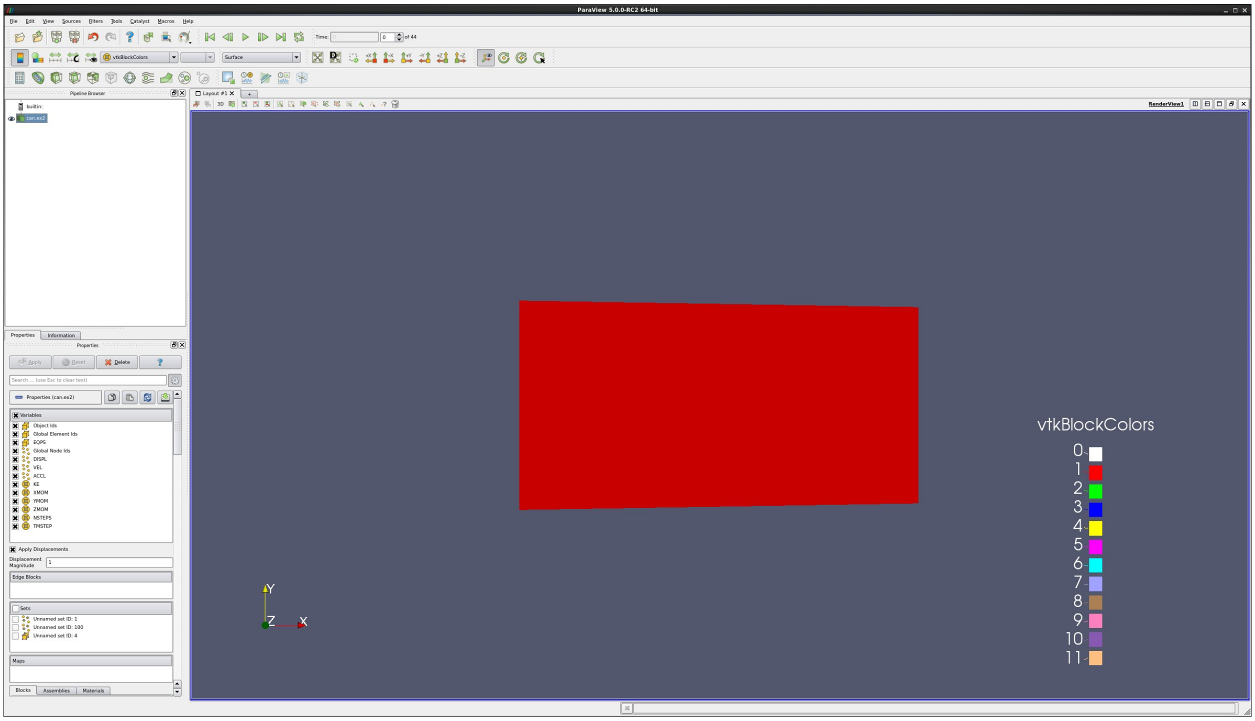
Exodus II file, Clip filter, Stream Tracer filter, Tube filter, Glyph filter

Wavelet with volume rendering and contours



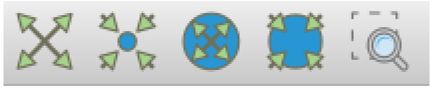
### Open can.ex2

- Open can.ex2.
  - can.ex2 is one of the datasets included with ParaView in the Examples folder.
  - In `paraview`, **File** → **Open**.
  - In the upper left corner, there is a folder called **Examples**. Go into this folder.
  - Select can.ex2.
  - **OK**.
  - Under the **Properties** tab, all **Block Arrays** will be selected.
  - **Apply**.
- Turn off the node variables for VEL. Since any variable that is selected takes up memory, and since some datasets are huge, often the user will only read in the data that is needed for a run.
  - Click **VEL**, turning the check box OFF.
  - **Apply**.
- The screen should now look like this. (The square will show up as red, since **ParaView** defaults to coloring by block, and the block we are seeing is red.) You are looking at the bottom of the plate that the can is sitting on.
- Lets move the 3d object. Grab the can using the **left** mouse button. Try the **center** button. Try again with the **right** button. Try all three again holding down the **SHIFT** key. Try again holding down the X, Y and Z keys.
- Place your mouse on a corner of the can. Now, hold the **CTRL** key down, and move the **RIGHT** mouse button up and down. You can zoom into and out of that location. Note that Macs use the **Command** key instead of the

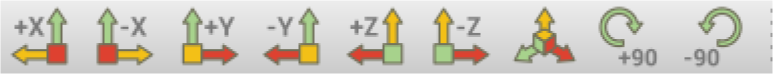


**CTRL** key.

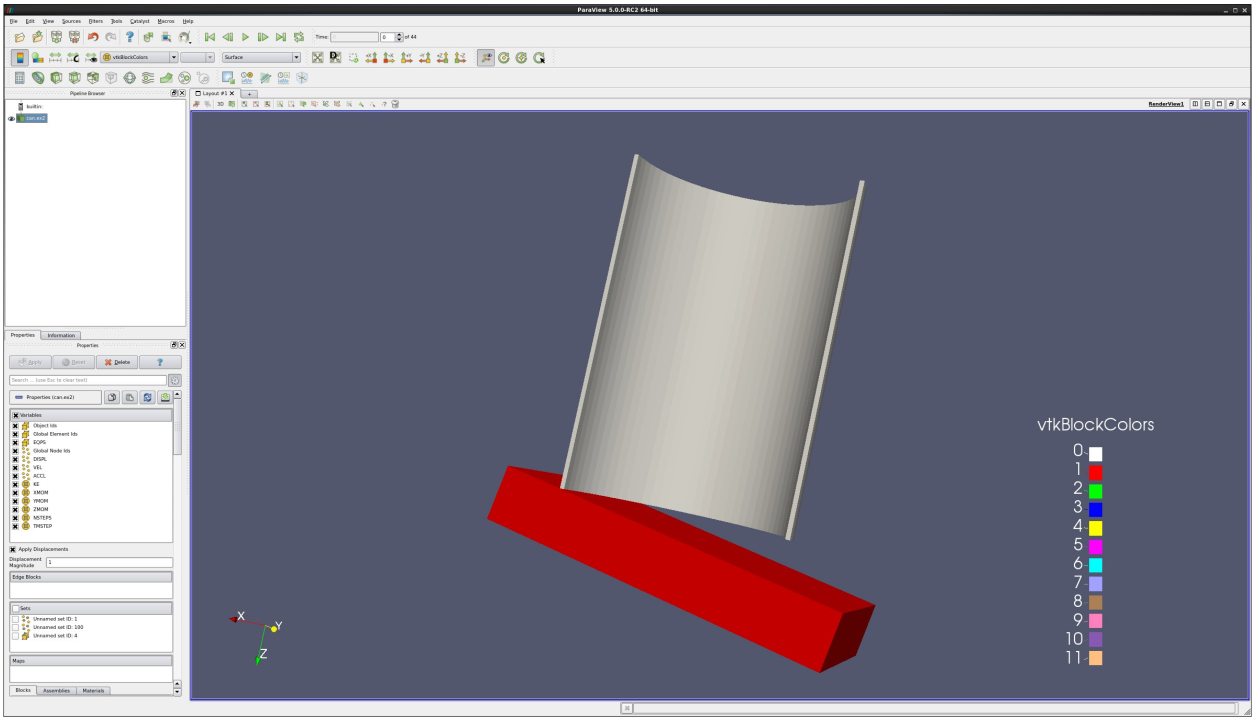
- The **Reset** icon is used to recenter your data. Select it now. To the right of this is the **Zoom to Box** icon, which looks like a magnifying glass. Select it, and rubber band select the edge of the can. **Reset**. Try using the **SHFT** key and **Zoom to Box**.

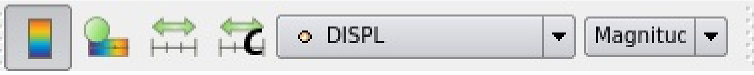


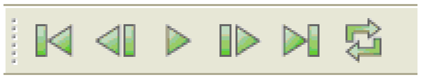
- Telling the **camera** to look up and down the **X**, **Y** and **Z** axes is done with the **+X**, **-X**, **+Y**, **-Y**, **+Z**, and **-Z** icons.



- Now the screen looks like this:
- To change the representation, change **Surface** to **Wireframe** (right below **Help**).
- Then, change it to **Surface with Edges**.
- Cycle through all of the other **representations**. In order, these are: **3D Glyphs**, **Feature Edges**, **Outline**, **Point Gaussian**, **Points**, **Slice**, **Surface**, **Surface LIC**, **Surface With Edges**, **Volume**, and **Wireframe**. Note that we usually use the **Slice filter** for **slices**.
- Finally, turn the **Representation** back to **Surface**.
- Notice that the can dataset is being painted in two colors. We are currently painting by **Block**.
- Change the variable used for color. Change this from **Solid Color**. to Acceleration (Point **ACCL**). (This is found just below the **Sources** menu.) Everything should go blue.
- Animate the can one frame forward using the **Next Frame** controls. Right above the window of the can are animation controls. Click the right arrow with a bar to its left once. The plate turns red. Next to that is the **Play** button. The leftmost control is **First Frame**. It has a matching **Last Frame**.







**Did you know?**

The can dataset has displacement information in it. We are actually running the plate into the can, and the whole object is moving.

- Notice that our color map is not set correctly. It needs to be set over the whole range of displacement, so that it grades from blue to red.

**Be careful!**

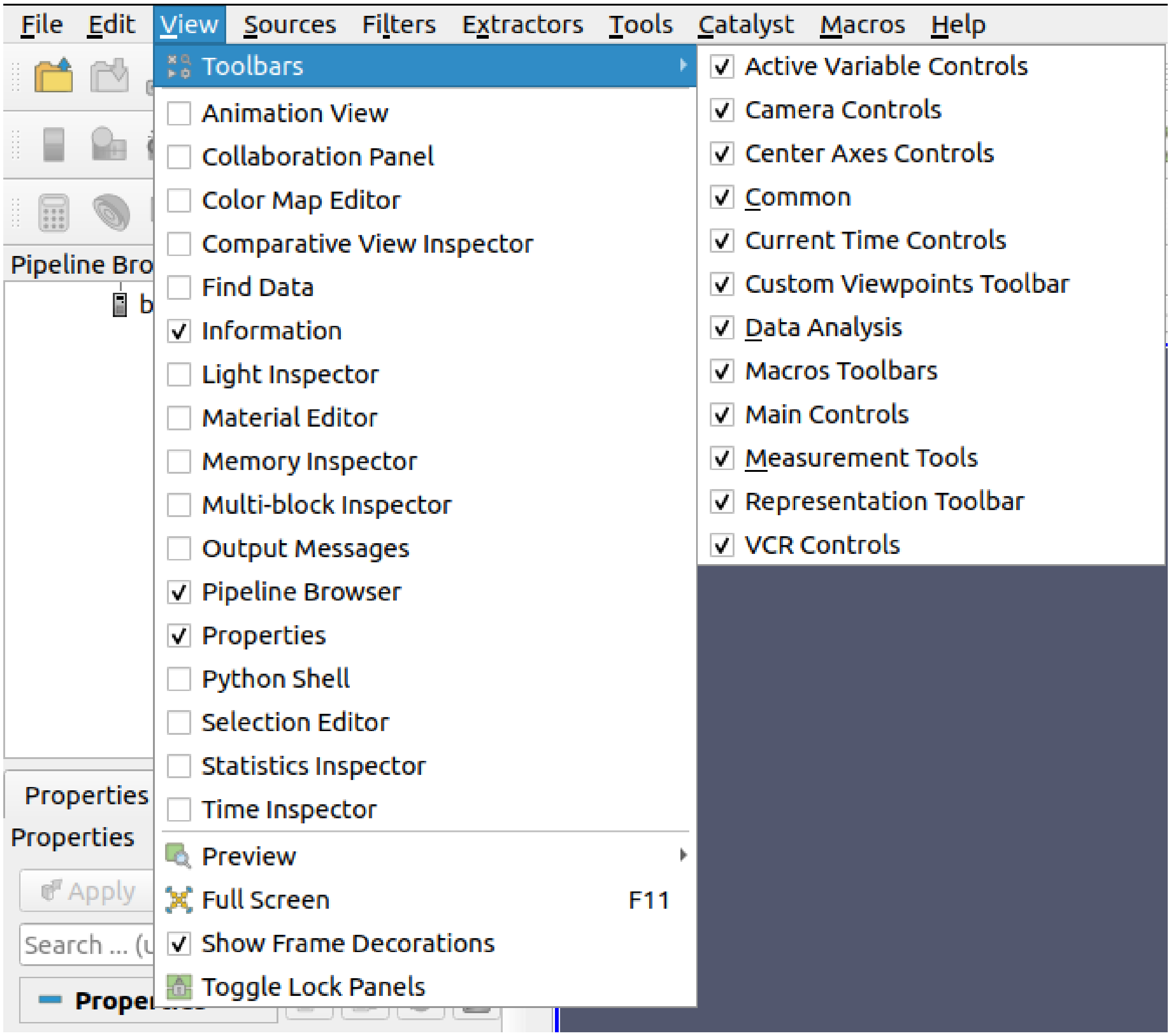
Very, very large data can take a long time to process. Don’t animate your data unless you have to with very large datasets.

- Click the **Play** icon, running to the end of the simulation.
- Click the **Rescale to Data Range** button.
- **Play**. **First Frame**.
- Click the **Rescale to Custom Data Range** button. Change the range to be 0.0 to 3.0e9.
- **Play**. **First Frame**.
- Click the **Rescale to Data Range over All Timesteps** button.

- **Play**. **First Frame**.

### Getting back GUI components

- If you accidentally close the **Properties** tab, the **Information** tab or the **Pipeline Browser** tab, open them again from the **View** menu.
- If you accidentally undock one of the tabs, just drag it back into place, wait for a gray shadow to appear, and drop it into place.



## 4.2.2 Beginning: Sources & Filters

### Introduction

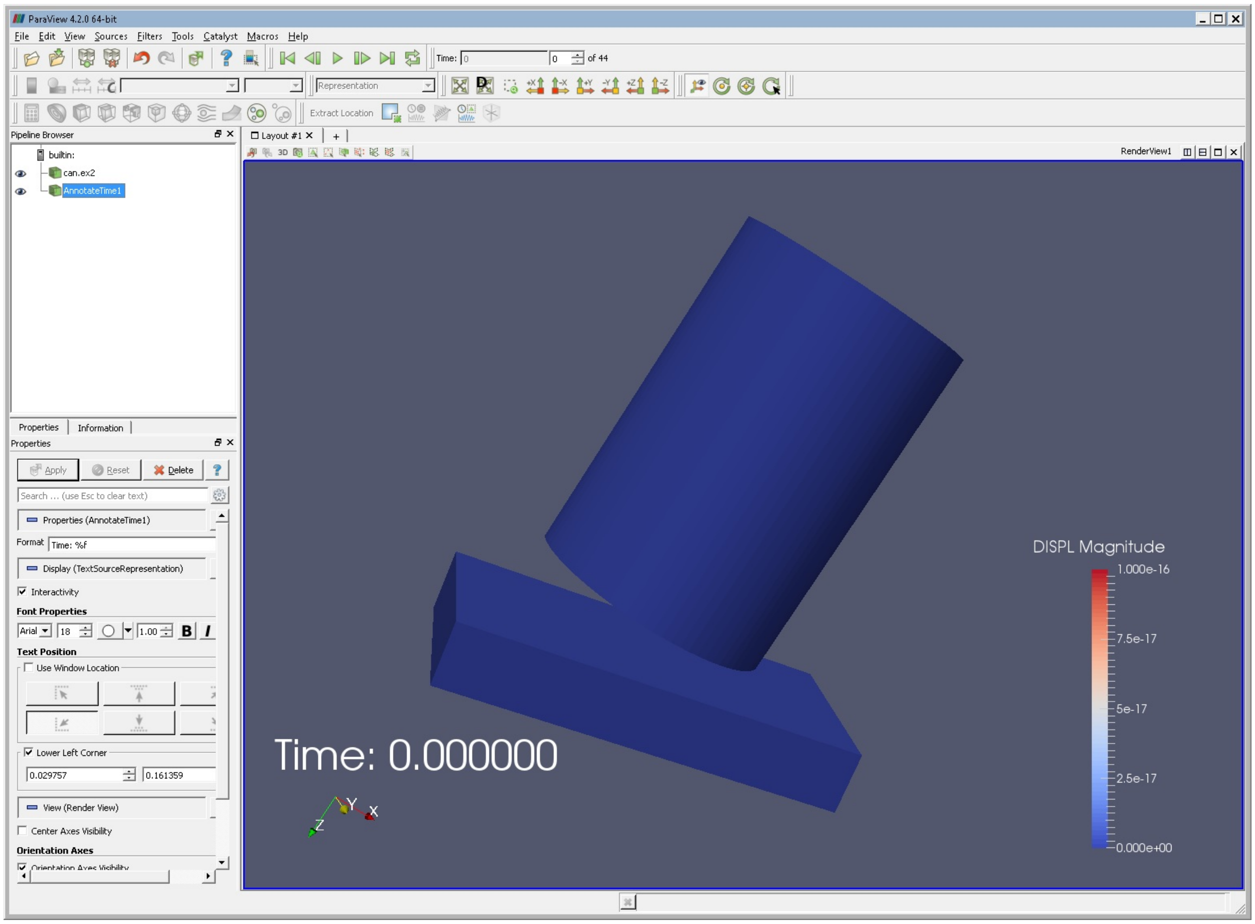
This usecase shows a user how to use sources and filters to modify the display of data.

Most examples assume that the user starts with a new model. To start over, go to the menu item **Edit** → **Reset Session**, and then re-open your data.

Data is opened by going to **File** → **Open**. Example data files can be found in the Examples directory, located in the upper left of the **Open** dialog.

### Annotate Time Source

- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- Select **Sources** → **Annotate Time**.
- **Apply**.
- Play the animation



Properties

Search ... (use Esc to clear text)

Properties (AnnotateTime1)

Format: Time: %f

Display (TextSourceRepresentation)

☑ Interactivity

Font Properties

Arial | 18 | 1.00 | B | I

Text Position

☐ Use Window Location

☑ Lower Left Corner

0.029757 | 0.161359

View (Render View)

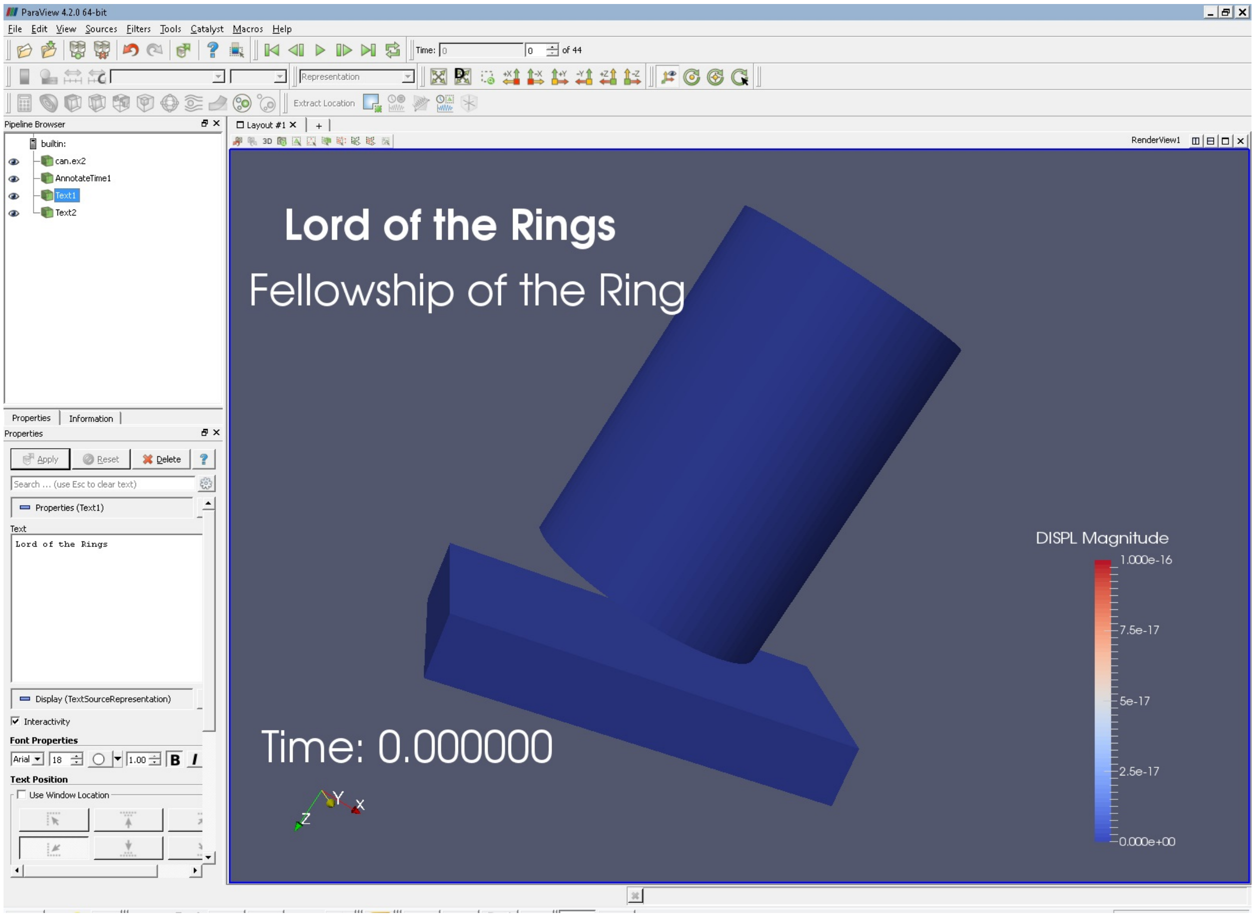
☐ Center Axes Visibility

Orientation Axes

☑ Orientation Axes Visibility

### Text Source

- This exercise continues on from the previous one. If you are starting over:
  - Open can.ex2.
  - **Apply**.
- Select **Sources** → **Text**.
- Type some text in the window.
- **Apply**.
- We now want to move the **Text** away from the **Annotate Time**.
- In the **Properties** tab under **Text Position** are six location icons. Click the **Top Center** one.
- You can manually move the text. Enable this functionality by selecting **Use Coordinates**, and then moving the text with the mouse.



Search ... (use Esc to clear text)

Properties (Text1)

Text: Lord of the Rings

Display (TextSourceRepresentation)

☑ Interactivity

Font Properties: Arial, 18, 1.00, **B**, *I*

Text Position: □ Use Window Location

### Ruler Source

- This exercise continues on from the previous one. If you are starting over:
  - Open can.ex2.
  - **Apply**.
- Rotate the can so you can see the concave surface.
- Select **Sources** → **Ruler**.
- As noted in the **Properties** tab, **1** and **2** set the starting and ending points of the ruler onto the can.
- **Apply**.

**Did you know?**

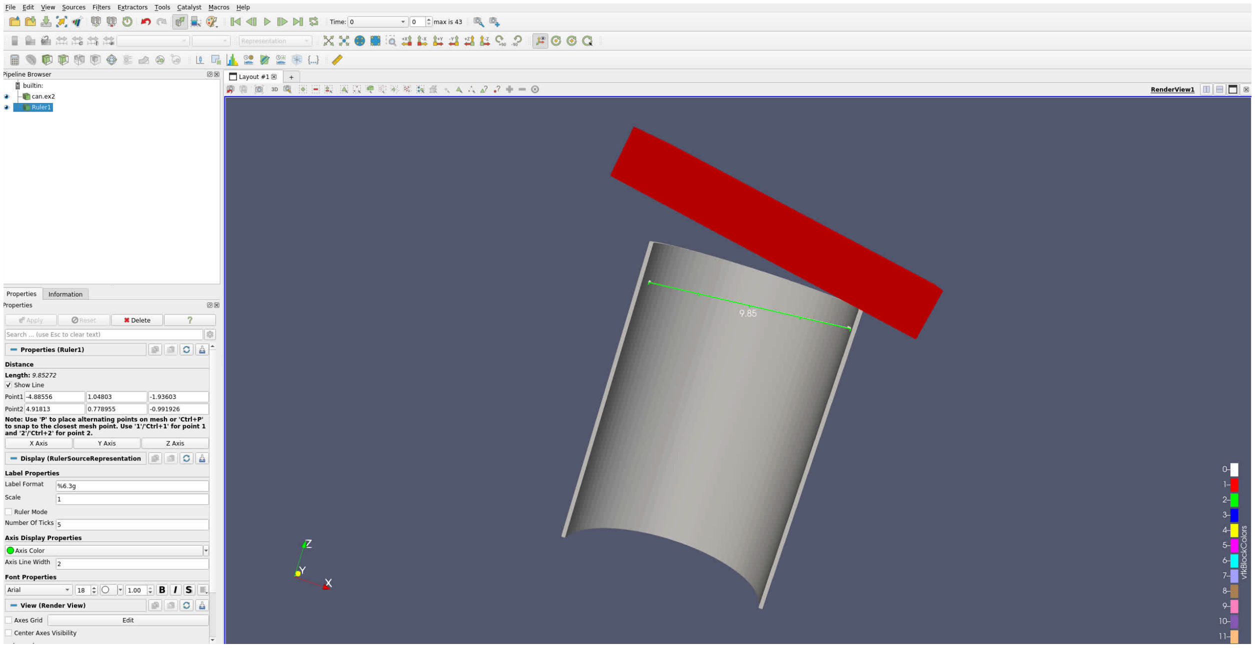
The line widget has a start and an end. The starting point is marked with a sphere, the ending point is marked with a cone.

**Did you know?**

If you turn on **Auto Apply** (see Section 4.2.3), you can interactively update the end points of the ruler.

**Did you know?**

You can use the **X**, **Y** or **Z** key and the mouse to move end points in only one axis.



Properties (Ruler1)

Distance

Length: 9.85272

☑ Show Line

| | | | |
|---|---|---|---|
| Point1 | -4.88556 | 1.04803 | -1.93603 |
| Point2 | 4.91813 | 0.778955 | -0.991926 |

Note: Use 'P' to place alternating points on mesh or 'Ctrl+P' to snap to the closest mesh point. Use '1'/'Ctrl+1' for point 1 and '2'/'Ctrl+2' for point 2.

X Axis | Y Axis | Z Axis

Display (RulerSourceRepresentation

Label Properties

Label Format: %6.3g

Scale: 1

□ Ruler Mode

Number Of Ticks: 5

Axis Display Properties

Axis Color

Axis Line Width: 2

Font Properties

Arial | 18 | 1.00 | B | I | S

View (Render View)

□ Axes Grid | Edit

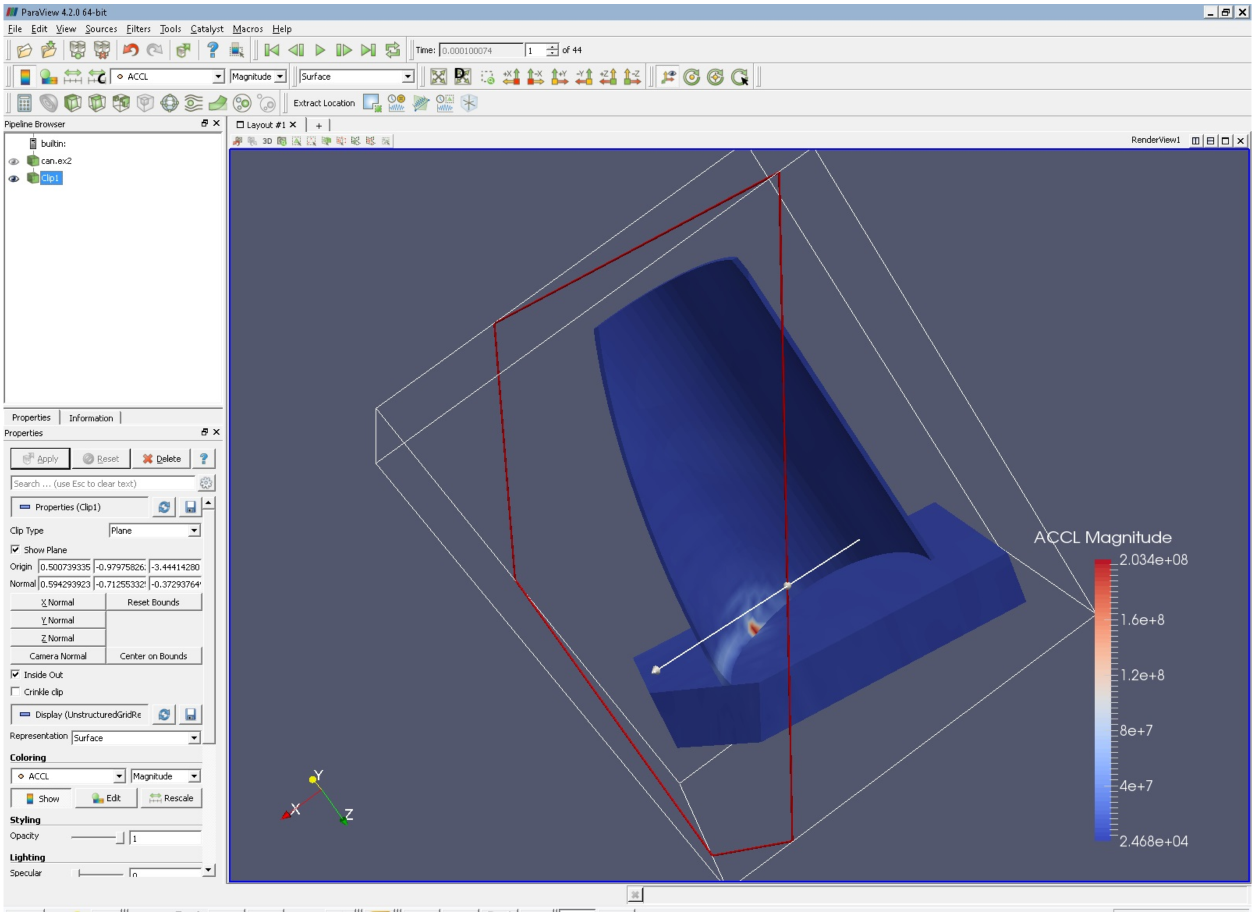
□ Center Axes Visibility

### Clip filter

- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- Select **Filters** → **Common** → **Clip** (Notice that this is also the third icon from the top on the far left of the screen.)
- **Apply**.
- Press **X Normal**.
- **Apply**.
- Try also **Y Normal**.
- **Apply**.
- Grab the arrow control at the end of the clip object with the left mouse button. Drag it around. You can also grab the red box and slide the clip plane forward and backward.
- **Apply**.
- Turn off the **Show Plane** checkbox.
- Select **Inside Out**.
- If the clip arrow control is ever hidden behind data, you can see it by clicking on the "eye" to the left of the "clip" in the **Pipeline Browser** which is located in upper left corner of the screen.

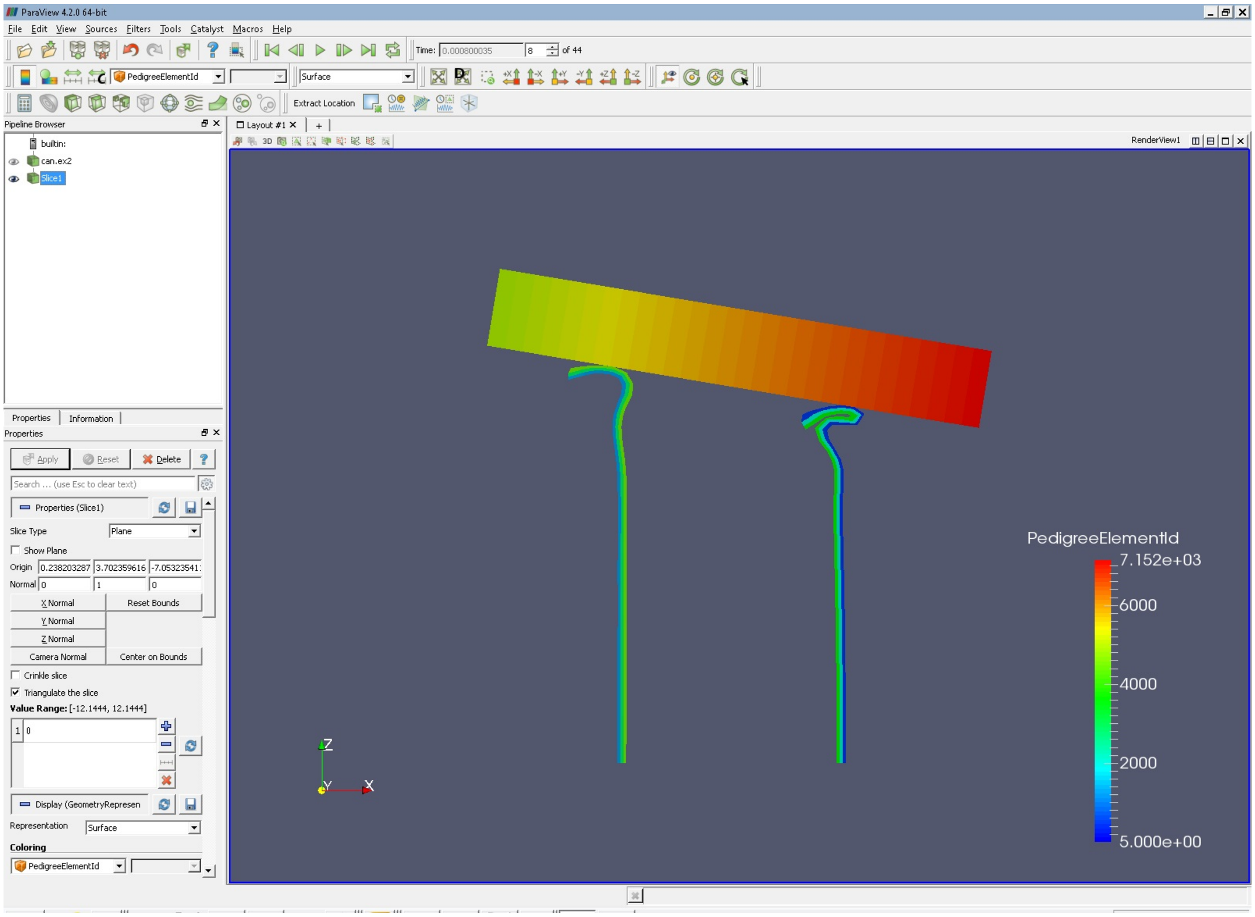
### Slice filter

- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- Select **Filters** → **Common** → **Slice**.
- **Apply**.
- Press **Y Normal**.
- **Apply**.
- Try also **Z Normal**.
- **Apply**.
- Grab the arrow control at the end of the clip object with the left mouse button. Drag it around.
- **Apply**.
- Turn off the **Show Plane** checkbox.
- We need to go to the Advanced Properties tab. Click on the little gear to the right of the search box.
- In the **Slice Offset Value** section, press **New Value**, type **1**. **Apply**. Notice that we just added a second cut plane.
- Under **Slice Offset Values**, press **Delete All**. Select **New Range**. Input From value of **-4** and To value of **4**. **OK**. **Apply**. Now we have 10 slices through our object.
- Play the animation.



Clip Type: Plane
☑ Show Plane
Origin: 0.500739335 | -0.97975826 | -3.44414280
Normal: 0.594293923 | -0.71255332 | -0.37293764
☑ Inside Out
☐ Crinkle clip
Representation: Surface
Coloring: ACCL | Magnitude
Opacity: 1
Specular: 0
Time: 0.000100074 | 1 | of 44

- If the slice arrow control is ever hidden behind data, you can see it by clicking on the “eye” to the left of the “clip” in the Pipeline Browser which is located in upper left corner of the screen.



Time: 0.000800035 8 of 44

Properties (Slice1)

Slice Type: Plane

☐ Show Plane

Origin: 0.238203287 3.702359616 -7.05323541

Normal: 0 1 0

☐ Crinkle slice

☑ Triangulate the slice

**Value Range:** [-12.1444, 12.1444]

1 0

Display (GeometryRepresen

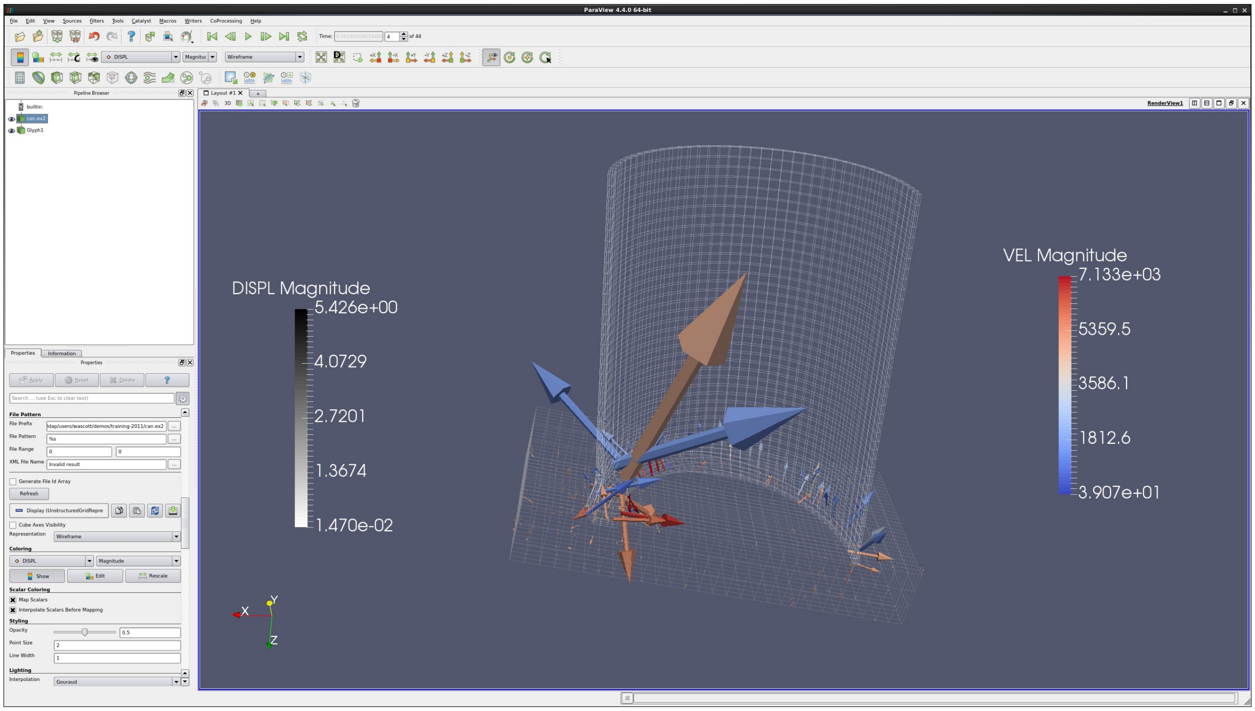
Representation: Surface

**Coloring**: PedigreeElementId

## Glyph filter

- Open can.ex2.
- **Apply**.
- Drag the can around with the right mouse button until you can see the can.
- Select **Filters** → **Common** → **Glyph**.
- Change the Vectors to **ACCL**.
- **Apply**.
- `paraview` does not set the **Scale Factor** correctly. On the **Properties** tab is an entry for **Set Scale Factor**. Change this to **1e-7**.
- **Apply**.
- Click the **Play** button at the top of the screen.
- Reset by hitting the **First Frame** button.
- We want to see the glyphs in context. Turn the visibility eyeball on the can.ex2 to “on”.
- Click on can.ex2 (it will show up as blue).

- In the **Properties** tab, Set **Representation** → **Wireframe**.
- **Play** the animation. We can now see where the accelerations are occurring as the can is crushed.
- **Reset** the animation.
- Click on the **Glyph** in the Pipeline Browser, thus giving the Glyph focus.
- In the **Properties** tab, change the **Vectors** to **VEL**.
- Then, change the **Set Scale Factor** to **3e-4**.
- **Apply**.
- Re-animate the window.
- Extra credit – Change the color of the glyphs to match the picture below.



File Pattern

File Prefix: ldap/users/wascott/demos/training-2011/can.ex2

File Pattern: %s

File Range: 0 0

XML File Name: Invalid result

□ Generate File Id Array

Display (UnstructuredGridRepre

□ Cube Axes Visibility

Representation: Wireframe

Coloring: DISPL Magnitude

Scalar Coloring

☒ Map Scalars

☒ Interpolate Scalars Before Mapping

Styling

Opacity: 0.5

Point Size: 2

Line Width: 1

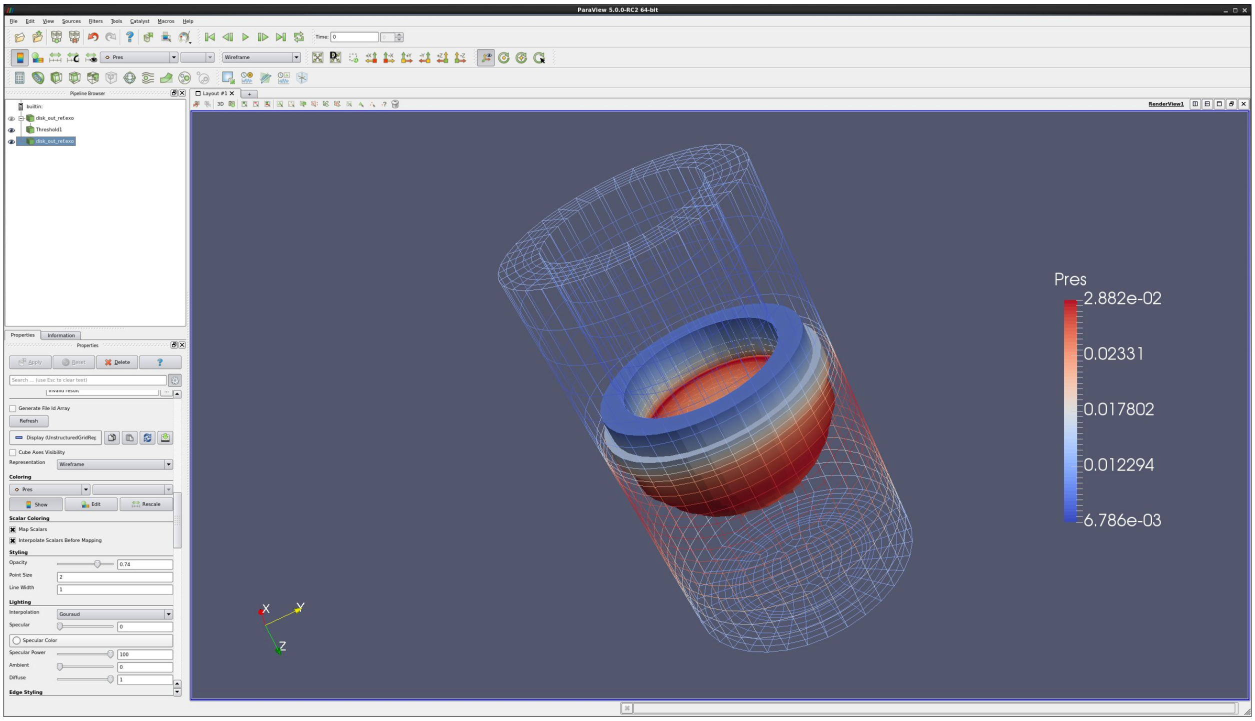
Lighting

Interpolation: Gouraud

## Threshold filter

- Open disk_out_ref.ex2.
- **Apply**.
- Spin the object around, and look inside of it.
- Press the **Threshold** button , found to the far left. (Notice that you can also find this through the filters menu.)
- Select **Scalars** “Temp” (Temperature).
  - Note: The blue recycle button will set the min and max values.
- Change the **Lower Threshold** to 400.
- **Apply**.
- Set **Coloring** to **Pres**.

- Spin the object around, and look at it.
- Now, let's place this hot section back into the cylinder.
- Let's open another version of disk_out_ref.ex2, using the **File** → **Open** menu.
- **Apply**.
- Make sure that the second disk_out_ref.ex2 is highlighted in the **Pipeline Browser**
- Set **Representation** to **Wireframe**.
- In the **Properties** tab, set **Coloring** to **Pres**
- What we have done: We have created an unstructured grid holding the cells that fit our criteria. This can make our data much bigger, and should be avoided if we are working with big data.
- Extra credit:
  - Change the outside cylinder to be volume rendered, set **Representation** to **Volume**.


ParaView 5.0.0-RC2 64-bit

Pipeline Browser: builtin:, disk_out_ref.exo, Threshold1, disk_out_ref.exo

Properties:
- ☐ Generate File Id Array
- ☐ Cube Axes Visibility
- Representation: Wireframe
- Coloring: Pres
- Scalar Coloring: ☒ Map Scalars, ☒ Interpolate Scalars Before Mapping
- Styling: Opacity 0.74, Point Size 2, Line Width 1
- Lighting: Interpolation Gouraud, Specular 0, Specular Power 100, Ambient 0, Diffuse 1

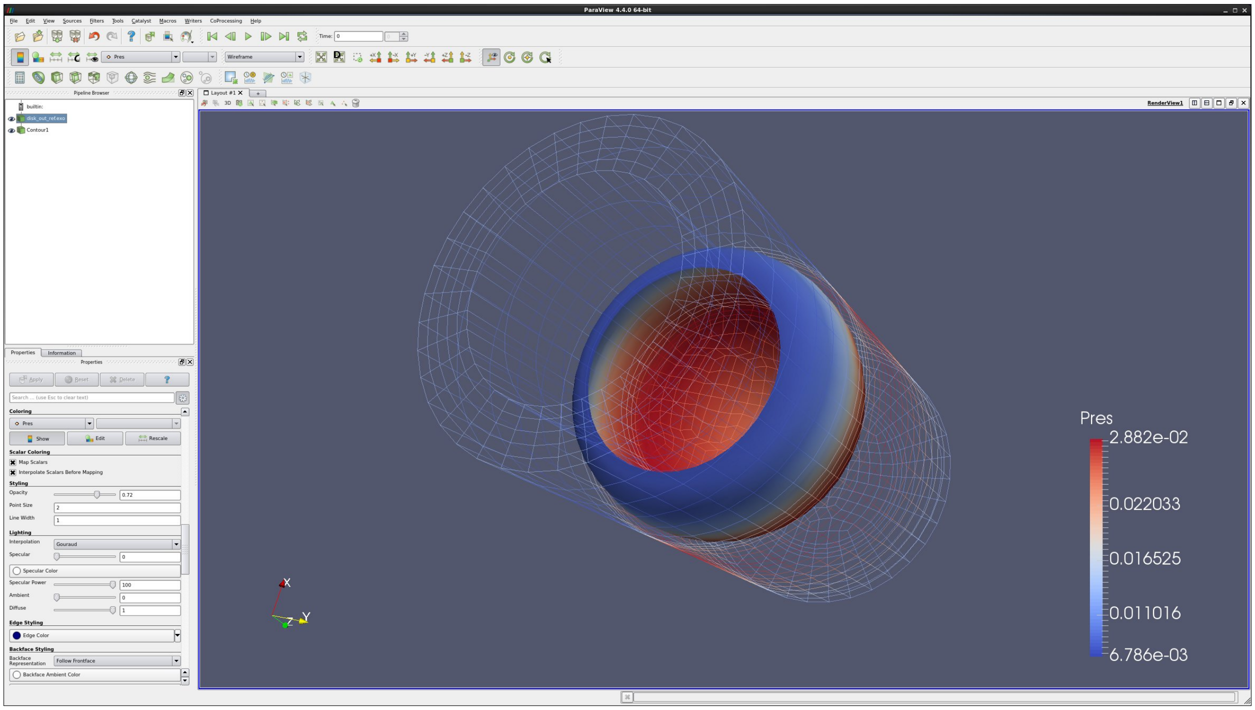
## Contour filter

- Open disk_out_ref.ex2.
- **Apply**.
- Select **Filters** → **Common** → **Contour**.
- Change **Properties**: **Contour By**: to **Temp**.
- Under **Value Range**, press **Delete All**.
- Now press **New Value** and enter **400**.
- **Apply**.

- Set **Coloring** to **Temp**.
- Why are all parts of the object the same color?
- Set **Coloring** to **Pres** (Pressure).
- This represents the location inside of the cylinder that is at temperature 400, and is colored by pressure.
- Turn the visibility back on for the disk_out_ref.ex2.
- In the **Pipeline Browser**, click disk_out_ref.ex2 (turning it white)
- Select **Representation Wireframe**.
- Set **Coloring** to **Pres**.
- Notice that this is another way to see two representations of the same object by reading the object in once and modifying it. In the Threshold Filter (above), we read in the object twice, and displayed each object differently.
- What we have done: We have created an isosurface of a specific temperature. One nice thing about isosurfaces is that they decrease the amount of data that has to fit into memory. This is handy when you are displaying big data.
- Extra credit:
  - Highlight disk_out_ref.ex2.
  - Select **Clip** filter.
  - Select **X Normal** for the clip plane. - Why has the disk_out_ref.ex2 model now turned solid? (Hint – the visibility has changed)
- More extra credit:
  - Under **contour**, Advanced Properties tab, delete the Isosurface, and use **New Range** to create 10 new surfaces. Again, the Advanced Properties tab is enabled by clicking on the gear icon to the right of the search box.
  - Next, select **Clip** filter to cut the cylinder in half.
    - What are the surfaces showing us? What are the colors showing us?
  - Highlighting clip
  - Under **Properties** tab, change the **Opacity** to **.50**.

### Clip to Scalar filter

- Open disk_out_ref.ex2.
- **Apply**.
- Select **Filters** → **Recent** → **Clip**.
- **Apply**.
- Select **Clip Type** → **Scalar**.
- Select **Scalars** → **Temp**.
- Input a **Value** of **400**.
- **Apply**.
- Select **Clip** filter again.
- Unclick **Show Plane**.



Representation: Wireframe
Time: 0

Coloring: Pres

Scalar Coloring
☒ Map Scalars
☒ Interpolate Scalars Before Mapping

Styling
Opacity: 0.72
Point Size: 2
Line Width: 1

Lighting
Interpolation: Gouraud
Specular: 0
○ Specular Color
Specular Power: 100
Ambient: 0
Diffuse: 1

Edge Styling
● Edge Color

Backface Styling
Backface Representation: Follow Frontface
○ Backface Ambient Color

- **Apply**.
- Set **Coloring** to **Temp**.
- Highlight disk_out_ref.ex2, turn the eyeball on.
- Display by **Volume**
- Set **Coloring** to **Temp**.

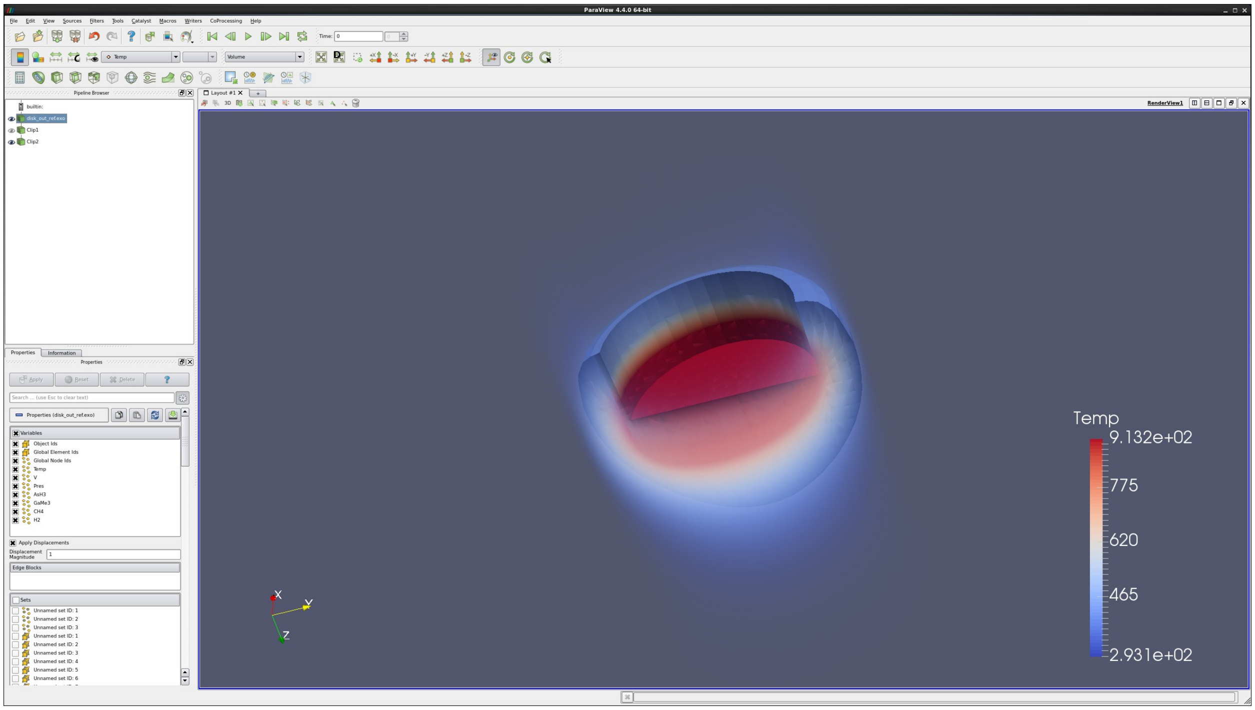
What we have done: We have clipped to a constant scalar, creating a smooth mesh. Once again, this increases the size of your data significantly.

### Cell to Point/ Point to Cell filters

These filters are used to convert a dataset from having cell data to having point data and vice versa. This is sometimes useful if a filter requires one type of data, and a user only has the other type of data. An example would be using can.ex2. You cannot get a contour of EQPS directly, since EQPS is cell data and contour only works on points. Use the filter **Cell Data to Point Data** first, then call contour.

### Stream Tracer

- Open disk_out_ref.ex2.
- **Apply**.
- Select **Filters** → **Common** → **Stream Tracer**.
- Click the **Seeds: Center on Bounds** button.
- **Apply**.
- In the **Properties** tab, set **Coloring** to **Temp**, then **V**, then **Pres**.



- If necessary, click **Color, Reset Range**.
- Extra credit:
  - On the **Properties** page, set the **Number of Points** to **40**.
  - **Apply**.
  - Select **Filters** → **Alphabetical** → **Tube**.
  - **Apply**.

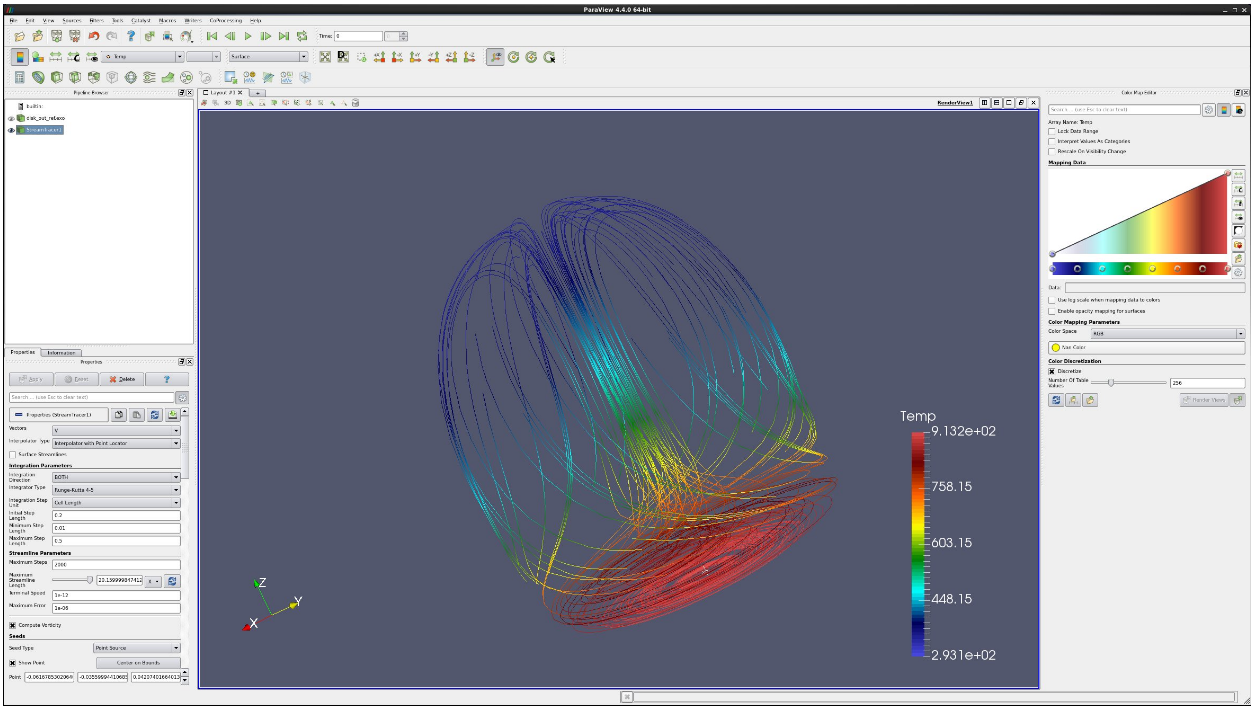
### Calculator filter

- Open disk_out_ref.ex2.
- **Apply**.
- Select **Filters** → **Common** → **Calculator**.
- Select **Attribute Mode**: **Cell Data**.
- Set **Result Array Name** to **RandomNumber**.
- In the empty line below, type **cos(GlobalElementId)*sin(GlobalElementId)**.

**Did you know?**

You can pull in the names of the variables by clicking on the **Scalars/Vectors** button.

- In the **Properties** tab, set **Coloring** to **RandomNumber**.
- We are now coloring by a pseudo random number. This shows how complex our data is.



Properties | Information

Properties (StreamTracer1)

Vectors: V
Interpolator Type: Interpolator with Point Locator
☐ Surface Streamlines

**Integration Parameters**
Integration Direction: BOTH
Integrator Type: Runge-Kutta 4-5
Integration Step Unit: Cell Length
Initial Step Length: 0.2
Minimum Step Length: 0.01
Maximum Step Length: 0.5

**Streamline Parameters**
Maximum Steps: 2000
Maximum Streamline Length: 20.159999847412
Terminal Speed: 1e-12
Maximum Error: 1e-06
☒ Compute Vorticity

**Seeds**
Seed Type: Point Source
☒ Show Point
Point: -0.0616785302064 | -0.0355999441068 | 0.04207401664013

Array Name: Temp
☐ Lock Data Range
☐ Interpret Values As Categories
☐ Rescale On Visibility Change

**Mapping Data**
☐ Use log scale when mapping data to colors
☐ Enable opacity mapping for surfaces

**Color Mapping Parameters**
Color Space: RGB
Nan Color

**Color Discretization**
☒ Discretize
Number Of Table Values: 256

- Note that you can create a vector from three scalars using the calculator. For instance, to create an X,Y,0 vector from Velocity, type **VEL_X*iHat+VEL_Y*jHat+0*kHat**.
- To get the length of a vector, use **mag(vector_name)**.
- To get the length of a vector squared, use **mag(vector_name)*mag(vector_name)**.

### Favorites

paraview allows users to place their favorite filters into the submenu named **Filters** → **Favorites**. Just use the Manage Favorites tool.

### Customize Shortcuts

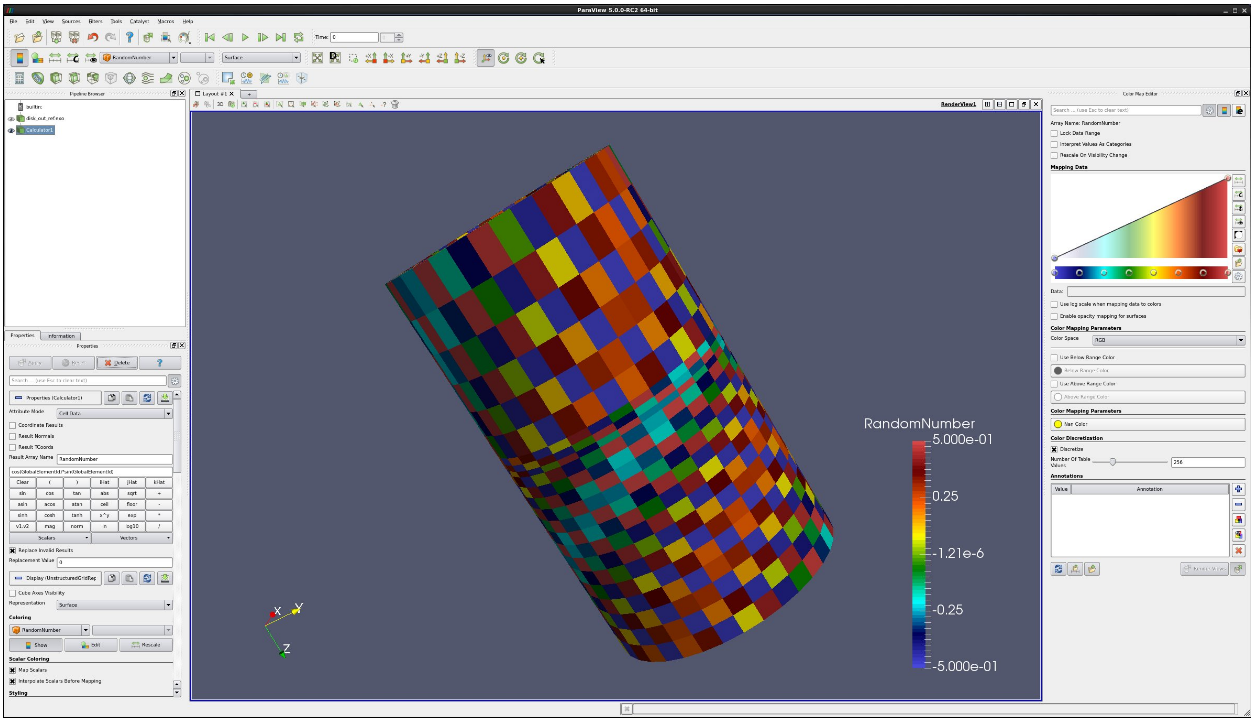
paraview allows users to add keyboard shortcuts to your favorite menu or filter. This is found under **Tools** → **Customize Shortcuts**.

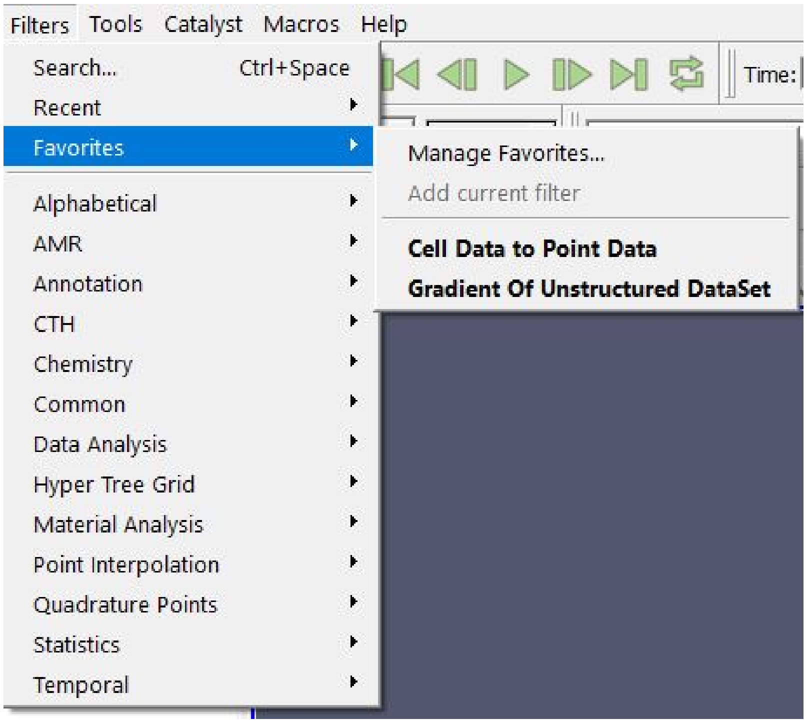
## 4.2.3 Beginning: GUI

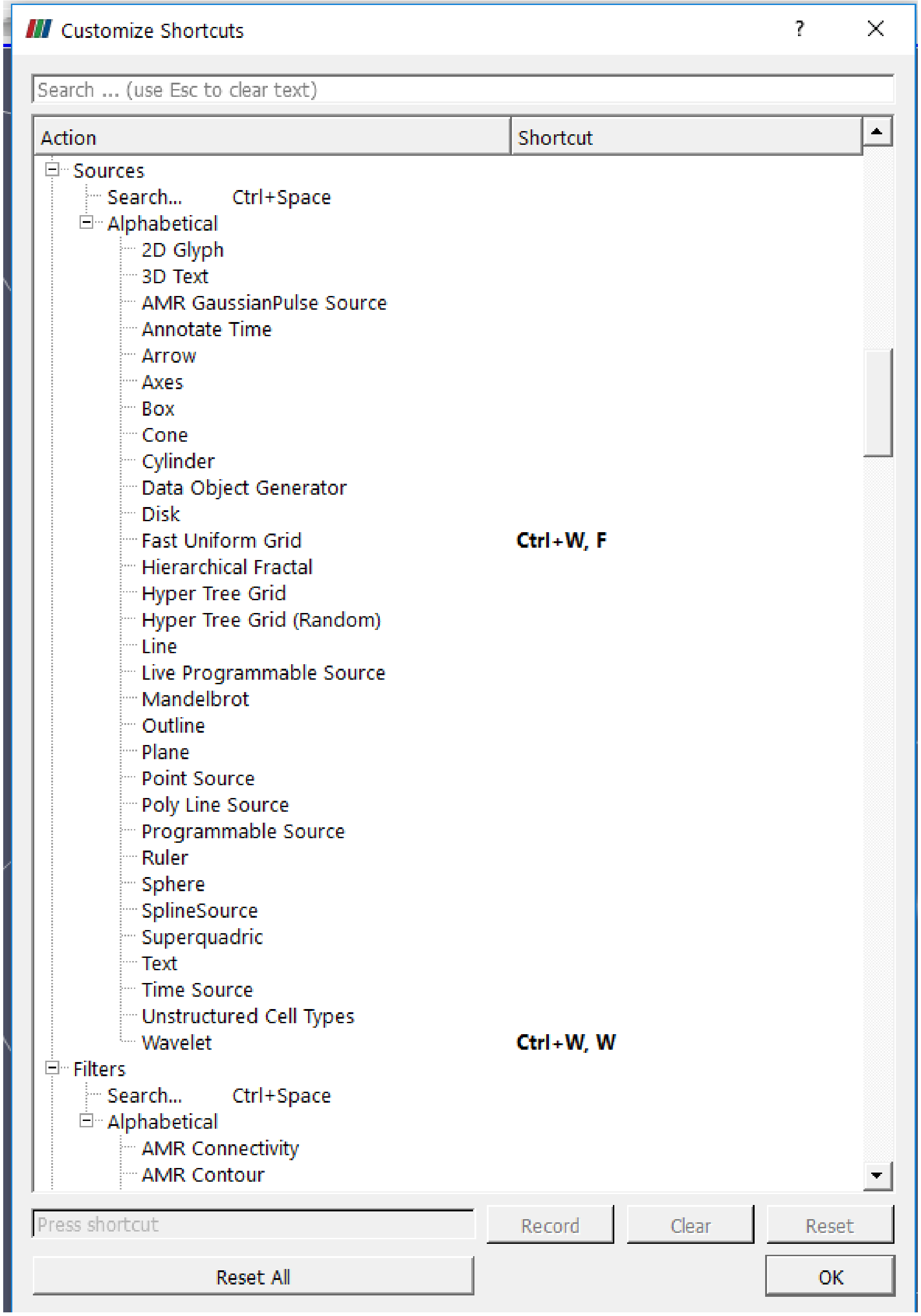
### Introduction

This usecase presents a few of the more important **ParaView** GUI features. A full list of features can be found in the ParaView Guide.

Data can be opened by going to **File** → **Open**. Example data files can be found in the Examples directory, located in the upper left of the Open dialog.


ParaView 5.0.0-RC2 64-bit
Pipeline Browser
builtin:
disk_out_ref.exo
Calculator1
Properties
Information
Apply
Reset
Delete
Layout #1
RenderView1
Color Map Editor
Array Name: RandomNumber
RandomNumber
5.000e-01
0.25
-1.21e-6
-0.25
-5.000e-01
X
Y
Z


Filters
Tools
Catalyst
Macros
Help
Search...
Ctrl+Space
Recent
Favorites
Alphabetical
AMR
Annotation
CTH
Chemistry
Common
Data Analysis
Hyper Tree Grid
Material Analysis
Point Interpolation
Quadrature Points
Statistics
Temporal
Manage Favorites...
Add current filter
Cell Data to Point Data
Gradient Of Unstructured DataSet
Time:



Customize Shortcuts

Search ... (use Esc to clear text)

| Action | Shortcut |
|---|---|
| ⊟ Sources | |
| Search... Ctrl+Space | |
| ⊟ Alphabetical | |
| 2D Glyph | |
| 3D Text | |
| AMR GaussianPulse Source | |
| Annotate Time | |
| Arrow | |
| Axes | |
| Box | |
| Cone | |
| Cylinder | |
| Data Object Generator | |
| Disk | |
| Fast Uniform Grid | **Ctrl+W, F** |
| Hierarchical Fractal | |
| Hyper Tree Grid | |
| Hyper Tree Grid (Random) | |
| Line | |
| Live Programmable Source | |
| Mandelbrot | |
| Outline | |
| Plane | |
| Point Source | |
| Poly Line Source | |
| Programmable Source | |
| Ruler | |
| Sphere | |
| SplineSource | |
| Superquadric | |
| Text | |
| Time Source | |
| Unstructured Cell Types | |
| Wavelet | **Ctrl+W, W** |
| ⊟ Filters | |
| Search... Ctrl+Space | |
| ⊟ Alphabetical | |
| AMR Connectivity | |
| AMR Contour | |

Press shortcut

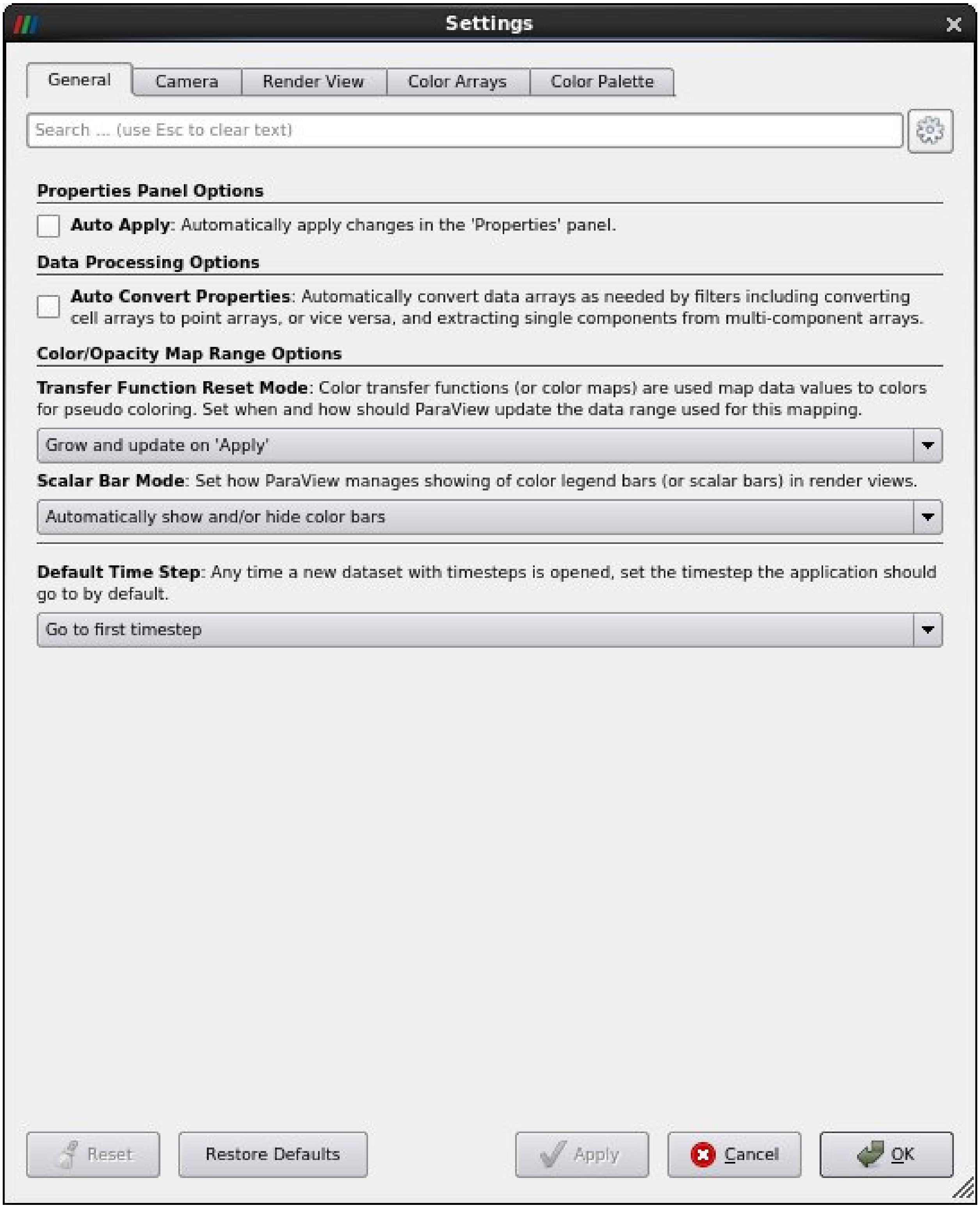
### Customize Settings

`paraview` allows users to customize settings. The most important ones are found in **Edit** → **Settings** → **General**. These are:

- **Auto Apply** - Also found as an icon below the Macros menu.
- **Auto Convert Properties** - Automatically call **Cell to Point**, **Point to Cell**, or extract components from vectors as needed.
- **Transfer Function Reset Mode** - when to update the minimum and maximum for the **Color Legend**.
- **Grow and update on Apply** - default. This means only update when told to.
- **Grow and update every timestep**. This means to update the Minimum and/or Maximum only if the current timestep exceeds these numbers. Basically, Add and grow.
- **Clamp and update every timestep**. Set the minimum and maximum every timestep, from the data this timestep. This is not recommended behavior, since it makes comparing frame to frame confusing.
- **Scalar Bar Mode** - just leave this one alone.
- **Default timestep** -
- **Go to first timestep** - default.
- **Go to last timestep**

### Information tab

- Open can.ex2.
- Press **Apply**.
- Open the **Information** tab.
  - The first section is the **File Properties** of the dataset.
  - The second section is **Data Grouping**, which tells you what blocks and sets you have in this dataset.
  - The third section is **Data Statistics** which tells you:
    - The number of cells
    - The number of points
    - The amount of memory used
    - The bounds which give the X, Y and Z bounds of the bounding box.
  - The fourth section is **Data Arrays** which gives information on each variable, including the min and max. Note that this is for the current timestep only.
  - The last section is **Time** which gives a list of all timesteps, with their associated time.



**Properties Panel Options**

□ **Auto Apply**: Automatically apply changes in the 'Properties' panel.

**Data Processing Options**

□ **Auto Convert Properties**: Automatically convert data arrays as needed by filters including converting cell arrays to point arrays, or vice versa, and extracting single components from multi-component arrays.

**Color/Opacity Map Range Options**

**Transfer Function Reset Mode**: Color transfer functions (or color maps) are used map data values to colors for pseudo coloring. Set when and how should ParaView update the data range used for this mapping.

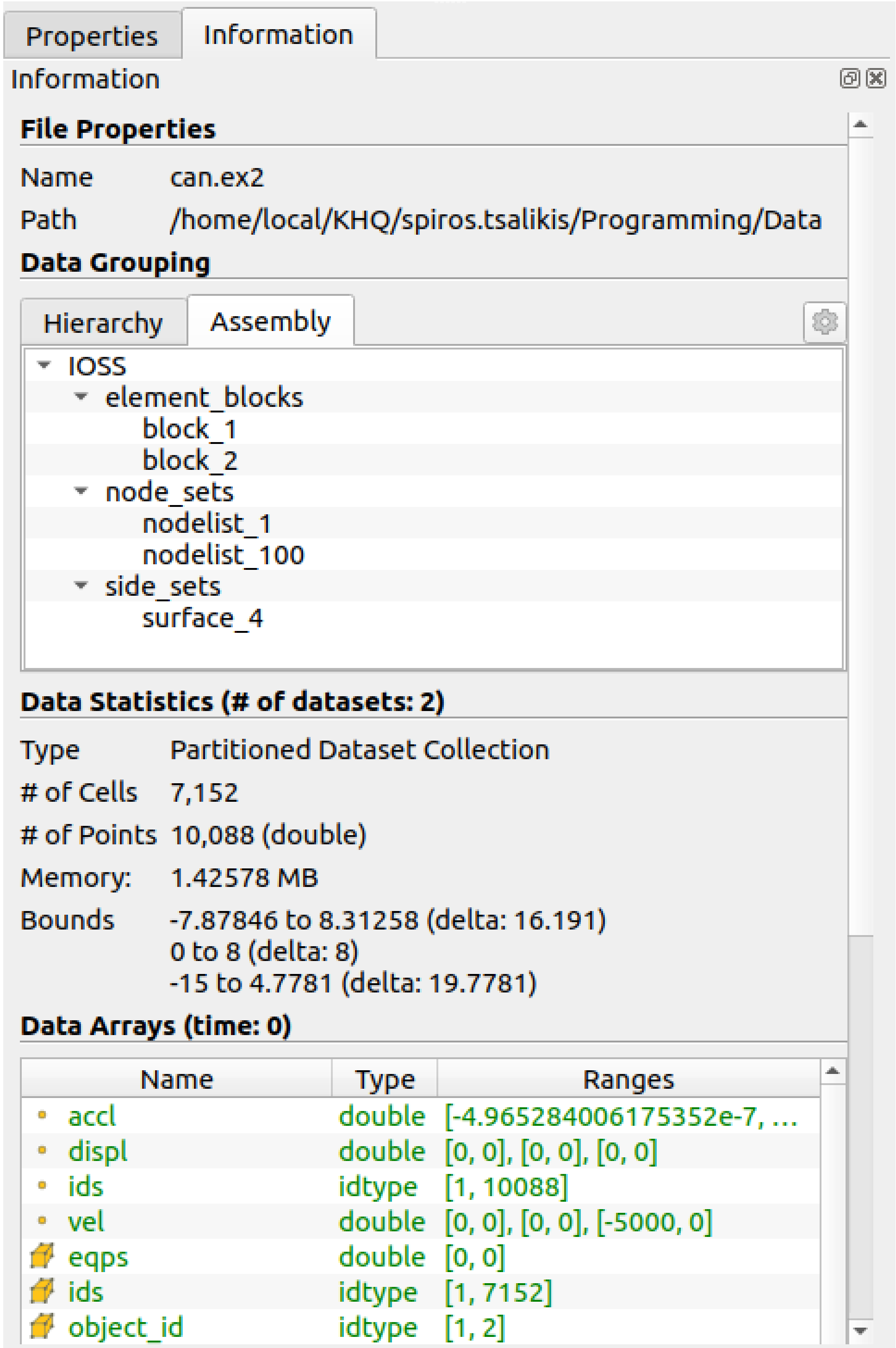
Grow and update on 'Apply'

**Scalar Bar Mode**: Set how ParaView manages showing of color legend bars (or scalar bars) in render views.

Automatically show and/or hide color bars

**Default Time Step**: Any time a new dataset with timesteps is opened, set the timestep the application should go to by default.

Go to first timestep



Information

**File Properties**

| | |
|---|---|
| Name | can.ex2 |
| Path | /home/local/KHQ/spiros.tsalikis/Programming/Data |

**Data Grouping**

```
IOSS
  element_blocks
    block_1
    block_2
  node_sets
    nodelist_1
    nodelist_100
  side_sets
    surface_4
```

**Data Statistics (# of datasets: 2)**

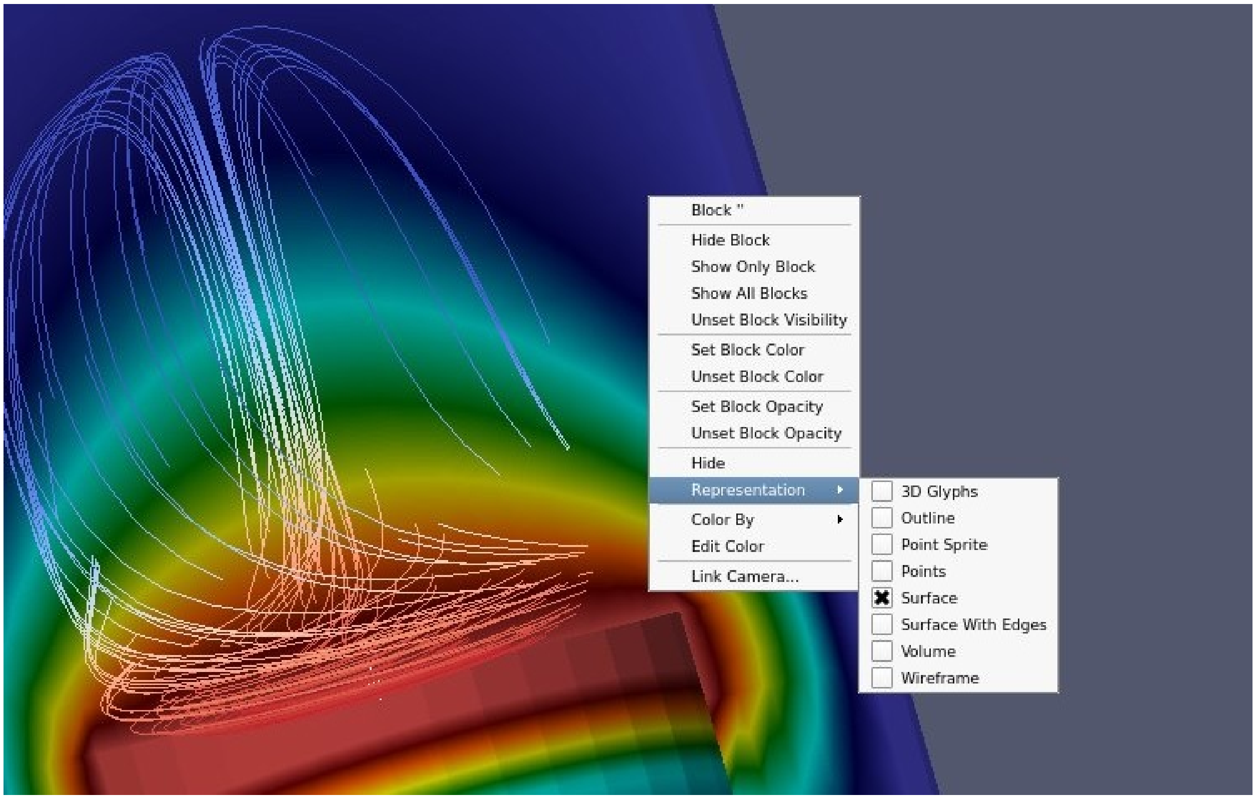
| | |
|---|---|
| Type | Partitioned Dataset Collection |
| # of Cells | 7,152 |
| # of Points | 10,088 (double) |
| Memory: | 1.42578 MB |
| Bounds | -7.87846 to 8.31258 (delta: 16.191)<br>0 to 8 (delta: 8)<br>-15 to 4.7781 (delta: 19.7781) |

**Data Arrays (time: 0)**

| Name | Type | Ranges |
|---|---|---|
| accl | double | [-4.965284006175352e-7, ... |
| displ | double | [0, 0], [0, 0], [0, 0] |
| ids | idtype | [1, 10088] |
| vel | double | [0, 0], [0, 0], [-5000, 0] |
| eqps | double | [0, 0] |
| ids | idtype | [1, 7152] |
| object_id | idtype | [1, 2] |

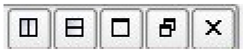
### Right click menu based commands

- Open can.ex2.
- **Apply**.
- Right click on the object.
  - Block Name
  - Block specific visibility commands
  - Block specific coloring and opacity
  - Representation and coloring commands
  - Link Camera



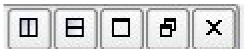
### Split windows

- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- **Split screen vertical** – i.e., one above the other. This is the little box with the horizontal line.



- Select 3D View.

- Turn the eyeball on for can, in the Pipeline Browser.
- **Split screen vertical** – i.e., one above the other. This is the little box with the horizontal line.

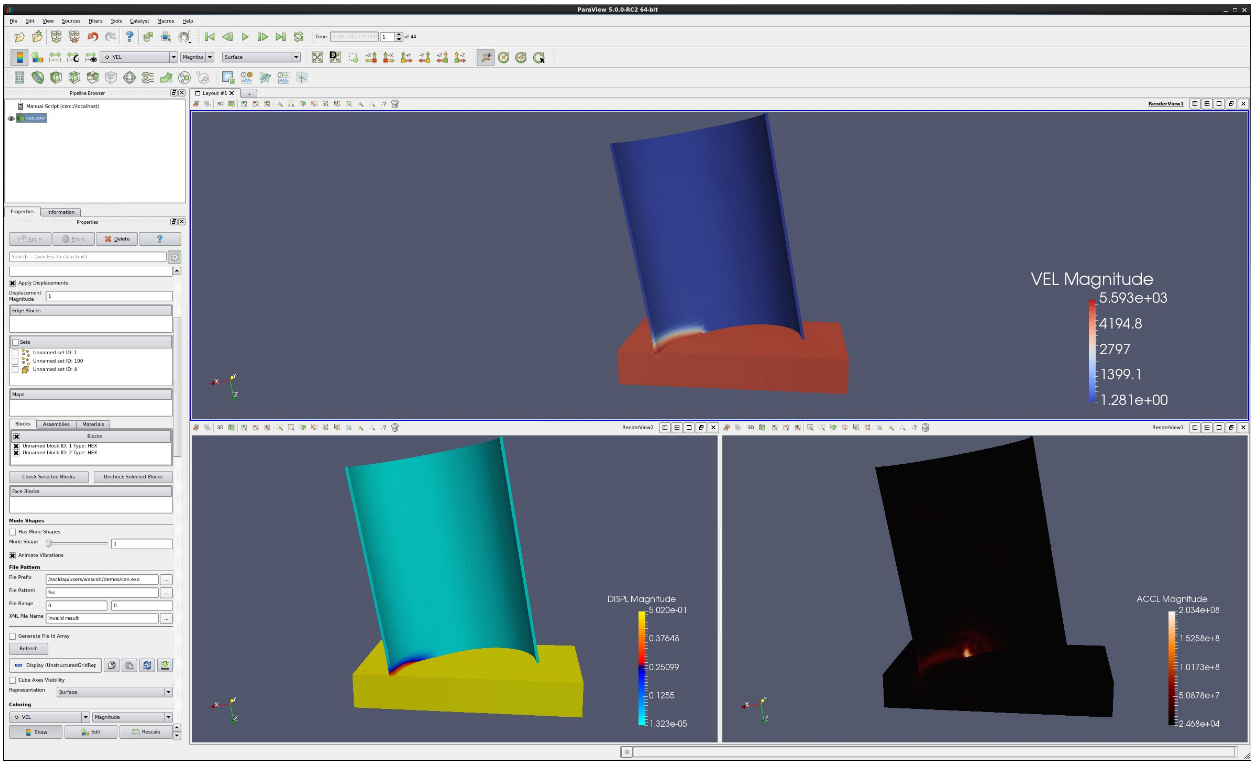


- Select 3D View.
- Turn the eyeball on for can, in the Pipeline Browser.

**Did you know?**

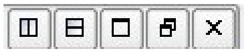
You can always **Undo** using  icon or **Redo** using  icon.

- This last command was a mistake. Lets undo it. **Undo, Undo**.
- We undid one too many commands. Lets redo it. **Redo**.
- Select the bottom viewport.
- **Split screen horizontal** – i.e., one next to the other. This is the little box with the vertical line.
- Select **3D View**.
- Turn the eyeball on for can, in the **Pipeline Browser**.
- Now lets link the cameras.
- Right click on the upper window.
- Click on **Link Camera**.
- Click on the lower right window.
- Do the same between the two bottom windows.
- Click in the lower left window, set **Coloring** to **DISPL**.
- Turn on Color Legend Visibility.
- Click in the lower right window, set **Coloring** to **ACCL**.
- Turn on Color Legend Visibility.
- Click in the upper window, set **Coloring** to **VEL**.
- Turn on Color Legend Visibility.
- Go to the last frame.
- Click on **Rescale to Data Range**.
- Go to the first frame.
- **Play**.



## Move/control windows

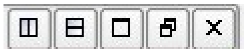
- Select the top window.
- Click on the **Maximize** button of the upper window.



- Click the **Restore** button of the upper window.
- Next, Using the left mouse button, grab the title bar of the lower left window and drag/drop it into the upper window. These two windows have now switched places.
- Finally, grab the divider between the two lower windows and drag it left and right. You can also move the divider between the upper and lower windows.

## Unlink windows

- To unlink the windows, we use the Link Manager. **Tools** → **Manage Links**.
- Select the second link.
- Click **Remove**.
- **Close**.
- Now, grab and rotate can in the three windows.
- Finally, delete the bottom two windows, using the **Close** button.



### Control the center

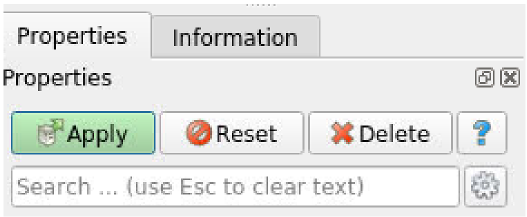
- Click on the **Show Center**  icon. Notice that this toggles the center cross.
- Click on the **Pick Center**  icon, then select a location on the can.
- Rotate the can, and notice where it is rotating.
- Click on the **Reset Center**  icon, returning the rotation location and center cross to the center of the object.

### Auto apply

ParaView now has the ability to auto apply commands. This button looks like this  and is to the left of the blue question mark.

### Properties tab

The **Properties** tab has three buttons on top: 1) **Apply** button, 2) the **Reset** button and 3) the **Delete** button. The **Reset** button will undo any Properties tab changes that a user accidentally has made. Below these three buttons is a search feature. Search will find Properties tab items, irrespective of them being standard or advanced. An example would be **Opacity**.



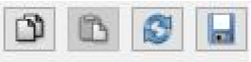
### Advanced Properties tab

The properties tab initially is in standard layout. To get to the advance layout, click the gear icon. Advanced reader or filter properties are found here. The advanced layout icon is .

### Copy/Paste/reset/Save parameters

On each section of the Properties tab there are four icons, as follows:

- Copy the state of that section of the **Properties** tab to the clipboard.
- Paste the state from the clipboard to this section of the properties tab. This allows you to copy and paste state between filters.
- Reset to factory defaults.
- Save this state as the user default.



### Move the camera

ParaView allows the user to change and store the position of the camera. Such controls as **Roll**, **Elevation** and **Azimuth** are available. The **Adjust Camera**  icon is on the left side of the row of icons at the top left of the 3d window.

The **Adjust Camera** dialog box looks like this:

Useful controls that I often use (in order) are as follows:

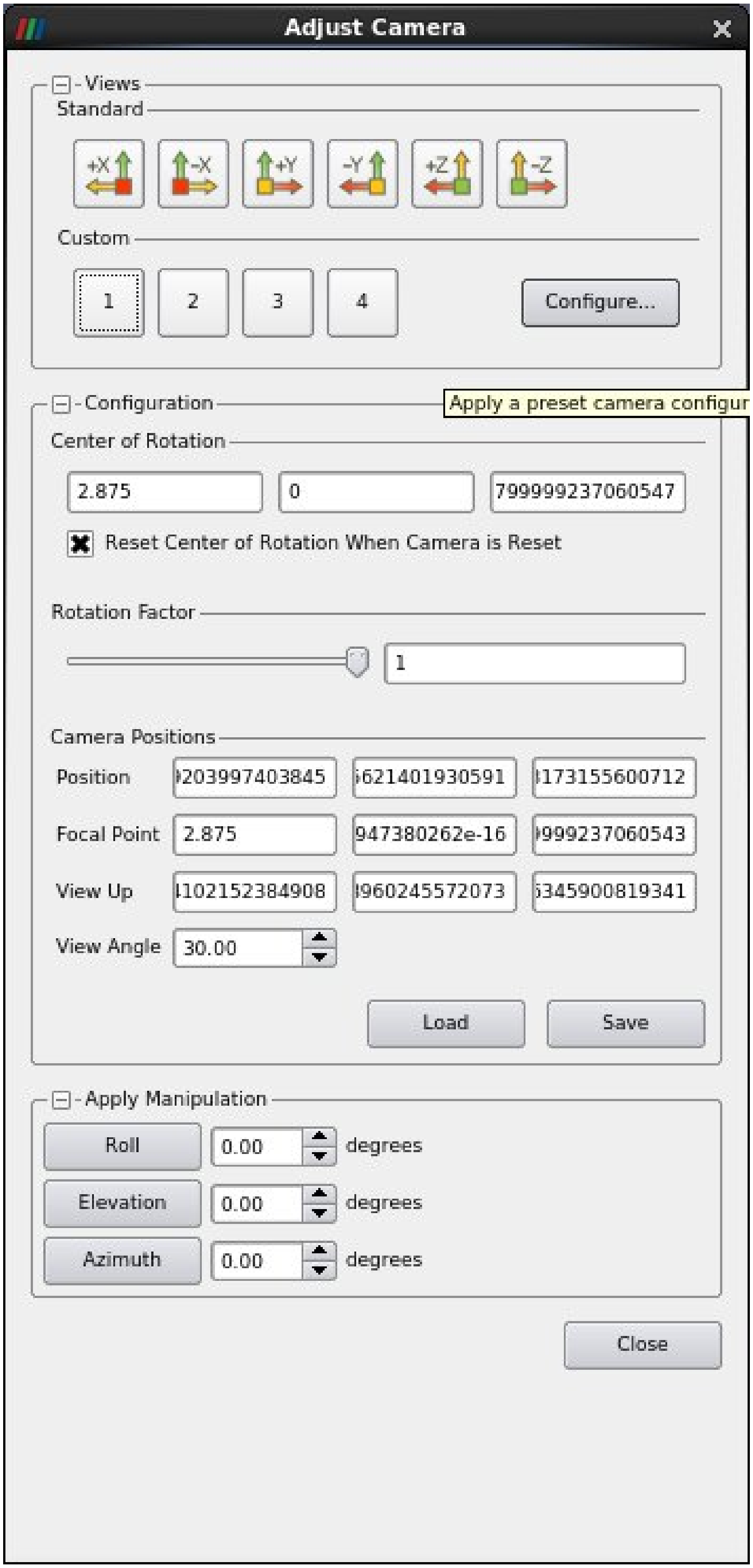
- **Custom Configure** - Save up to 4 camera positions.
- **Azimuth** - rotate around the vertical axis. Be sure to hit the button after entering a number.
- **Elevation** - rotate around the horizontal axis in the plane of the screen.
- **Roll** - rotate around the axis coming out of the screen.
- **View angle** - basically a zoom in.
- **Camera position** - where the camera is.
- **Focal point** - where the camera is looking.
- You can always recenter the object using the **Reset** icon. First, however, be sure to change **View angle** back to 30.

### Matplotlib characters

- If needed, open can.ex2..
- Select the **DISPL** variable.
- Move forward one time step using the **Next Frame** icon.
- We want to add an alpha character after DISPL.
  - Open the color editor.
  - Open the Edit Color Legend icon. It is the little color legend with the ‘e’ over it.
  - Modify the Title **DISPL** to say **DISPL $\alpha$**
- Here is how to change a 2d plot of **EQPS** to **EQPS (uV/m)**
  - Plot Over Line. Apply. Turn off all variables other than EQPS.
  - Change the **Legend Name** from **EQPS** to **EQPS ($\frac{muV}{m}$)**
- Matplotlib Mathtext formats are described here: https://matplotlib.org/2.0.2/users/mathtext.html

### Axes Grid

- Open dataset disk_out_ref.ex2.
- In the **Properties** tab, scroll down and select **Axis Grid**.
- Note that you can edit the **Axis Grid** attributes.



Center of Rotation: 2.875 | 0 | 799999237060547

☒ Reset Center of Rotation When Camera is Reset

Rotation Factor: 1

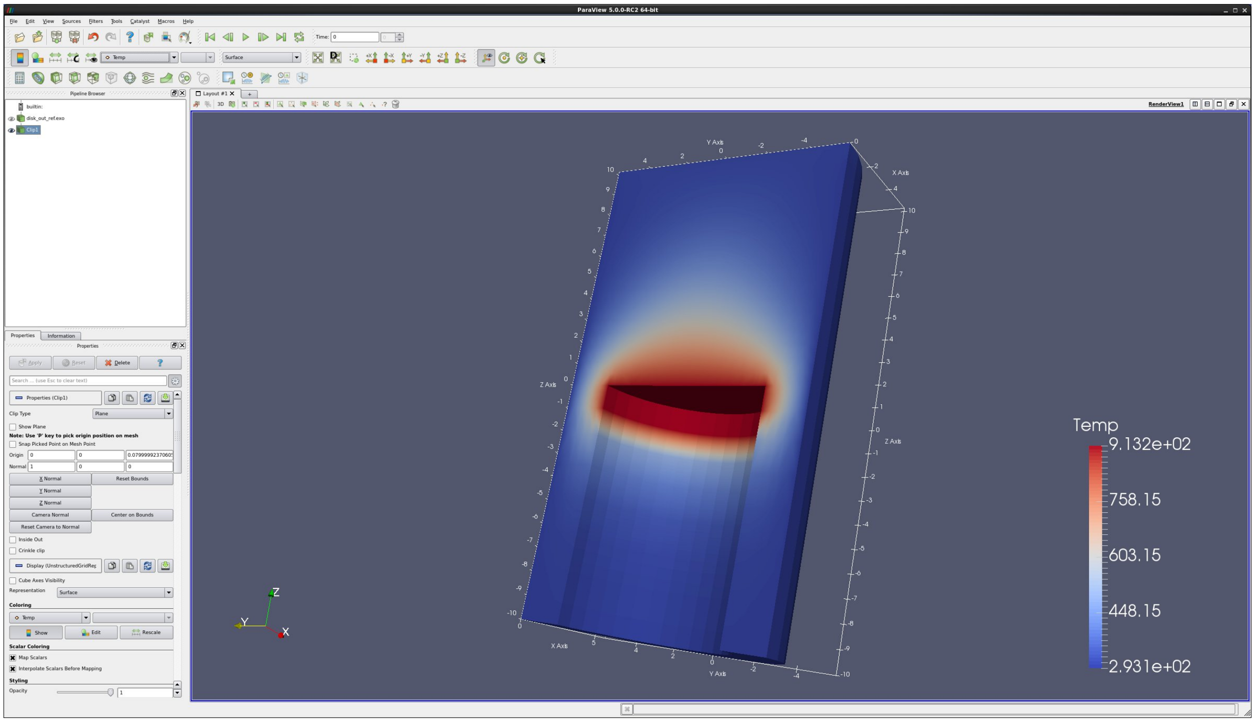
Camera Positions

| | | | |
|---|---|---|---|
| Position | 203997403845 | 621401930591 | 173155600712 |
| Focal Point | 2.875 | 947380262e-16 | 999237060543 |
| View Up | 102152384908 | 960245572073 | 345900819341 |
| View Angle | 30.00 | | |

Roll 0.00 degrees

Elevation 0.00 degrees

Azimuth 0.00 degrees



Clip Type: Plane

□ Show Plane

Note: Use 'P' key to pick origin position on mesh

□ Snap Picked Point on Mesh Point

Origin 0 0 0.0799999237060

Normal 1 0 0

□ Inside Out

□ Crinkle clip

□ Cube Axes Visibility

Representation: Surface

Coloring: Temp

Scalar Coloring

☒ Map Scalars

☒ Interpolate Scalars Before Mapping

Styling

Opacity 1

### Lighting - Specular

It is possible to change the specular highlights in ParaView. This is on the Properties tab, about half way down. It is called **Lighting: Specular**. Note that reflections can look like the center of the color map, thus specular highlights are turned off by default.

### Slice View and Layouts

- Open disk_out_ref.ex2.
- **Apply**.
- Set **Coloring** to **Temp**.
- Select **Clip**.
- Turn off the **Show Plane**.
- **Apply**.

ParaView supports numerous simultaneous layouts, or windows, into your data.

- Select the X to the right of Layout #1 (upper left side of the 3d window).

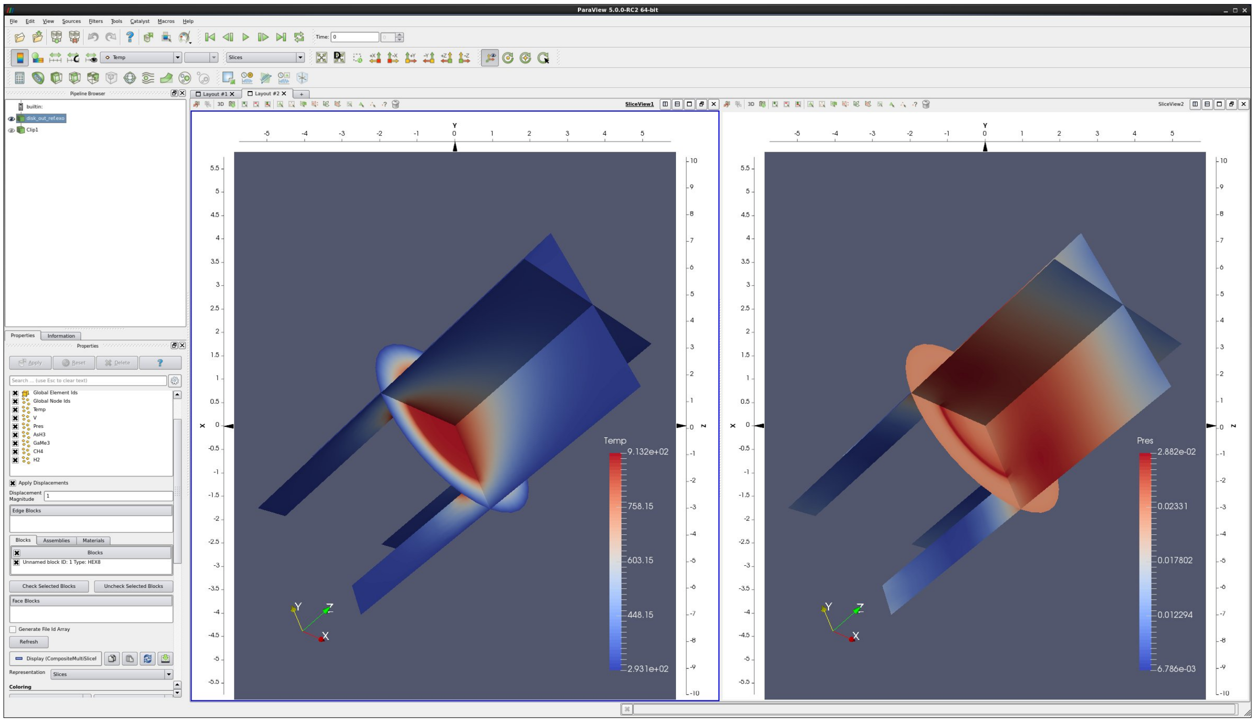
ParaView also supports different views than 3d views. Here is how to show your data as a slice view.

- Select Slice View
- Turn visibility on for disk_out_ref.ex2.
- Set **Coloring** to **Temp**.
- Left click in the window, and drag disk_out_ref around.
- Move the left, upper and right clip planes by dragging the black wedge.

You can intermix different view types.

- Split horizontal.
- Turn visibility on for disk_out_ref.
- Paint by Temp.

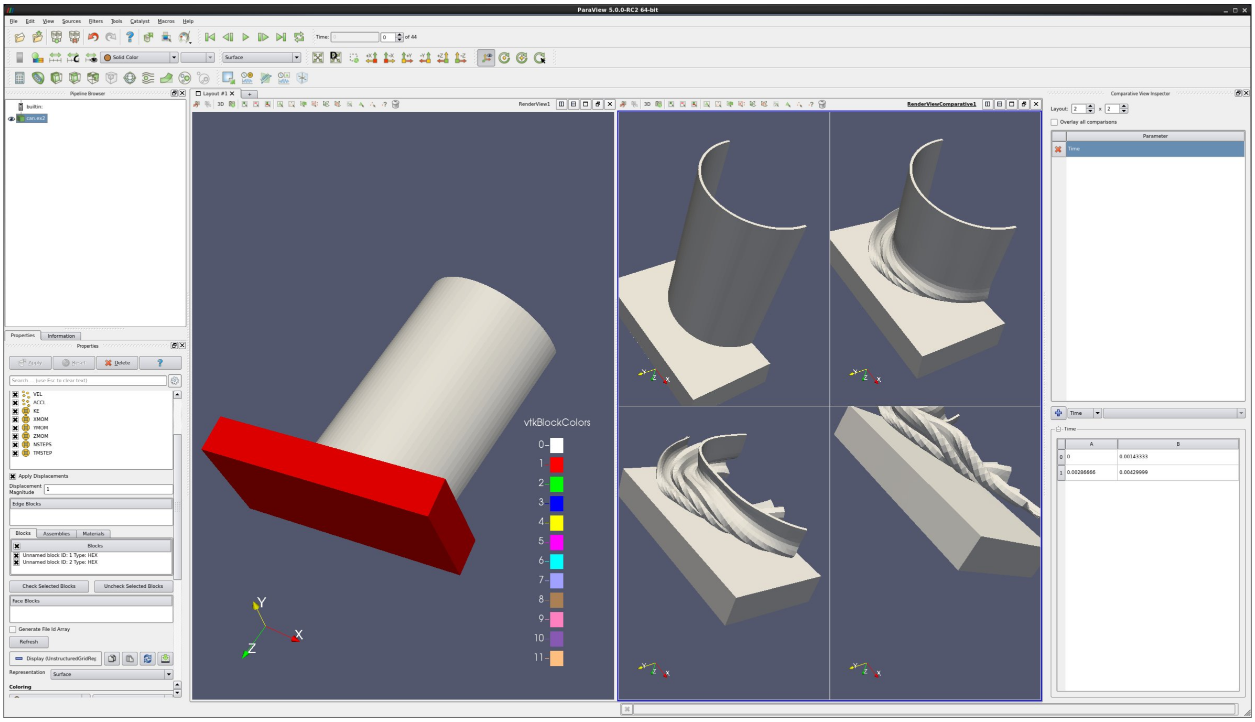
You can also connect the cameras for the different views. This can be done through the **Tools** → **Add Camera Links** menu.



## Render View (Comparison)

ParaView can compare different time steps at the same time. This is called Comparative View.

- Open can.ex2.
- **Apply**.
- Split view horizontal.
- Select **Render View (Comparative)**.
- Turn on visibility on the can.
- **View** → **Comparative View inspector**.
- Click on the blue **+**. This creates a 2X2 set of views of can.ex2 at four different timesteps.



### Customize Shortcuts

You can create shortcuts to Menu items (such as Filters) in ParaView.

- **Tools** → **Customize Shortcuts**.
- Find Wavelet. If it has a shortcut already, click Clear.
- Click in the **Press Shortcut** button.
- Now, select a shortcut, such as **CTRL W**.
- Close
- Now, click **CTRL W**, and you have a Wavelet.
- **Apply**.

## 4.2.4 Beginning: Color Maps & Palettes

### Introduction

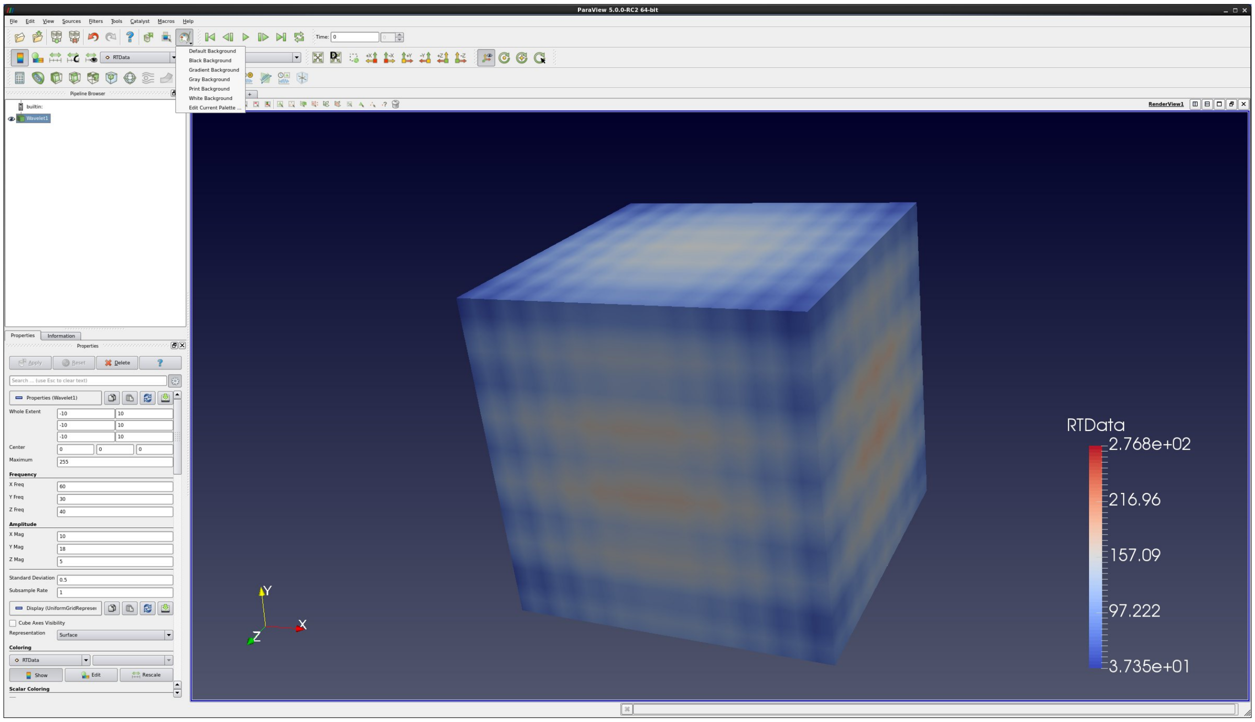
This usecase presents a few of the more important new ParaView GUI features. A full list of features can be found in the ParaView Guide.

Data can be opened by going to **File** → **Open**. Example data files can be found in the Examples directory, located in the upper left of the Open dialog.

### Color palette

ParaView allows users to easily control the color palette. This is done with the **Load a color palette** icon. . Changing color palettes does much more than just changing the background - it also changes the font colors for other annotations. Options are default gray, black, white and gradient. You can also create custom color palettes.

Here is an example of a gradient background, with the color palette menu displayed.



Time: 0

Properties (Wavelet1)

Whole Extent -10 10
-10 10
-10 10

Center 0 0 0

Maximum 255

Frequency

X Freq 60

Y Freq 30

Z Freq 40

Amplitude

X Mag 10

Y Mag 18

Z Mag 5

Standard Deviation 0.5

Subsample Rate 1

Display (UniformGridRepreser

☐ Cube Axes Visibility

Representation Surface

Coloring

RTData

A famous example of surrounding colors changing a perceived brightness of an object is Edward Adelson's Checker Shadow illusion.

### Color Map Editor

- Open can.ex2
- Select the **DISPL** variable.
- Move forward one time step using the **Next Frame** icon.
- Here is the coloring toolbar. Icons are as follows:
  - Toggle color legend visibility.
  - Edit color map.
  - Rescale to data range (set Min and Max from this timestep).
  - Rescale to custom data range (manually input Min and Max values.
  - Rescale to visible data range (set Min and Max from visible objects this timestep)
- Click on the **Edit Color Map** icon.
- The **Color Scale Editor** is used to change the colormap.

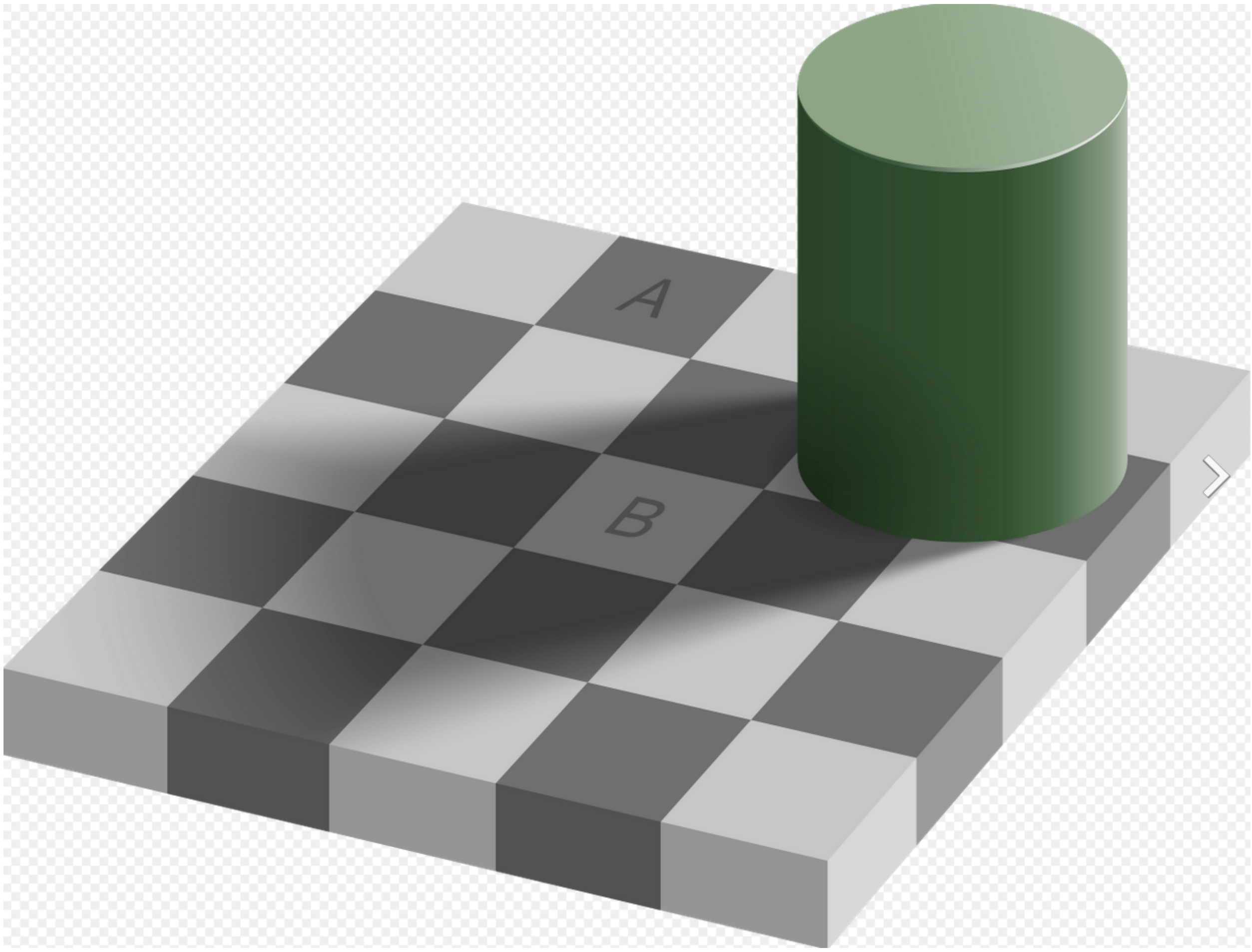


Fig. 4.11: The checker shadow illusion: the square A is exactly the same shade of grey as square B. Copyright Edward H. Adelson and Pbrks : CC BY-SA 4.0.

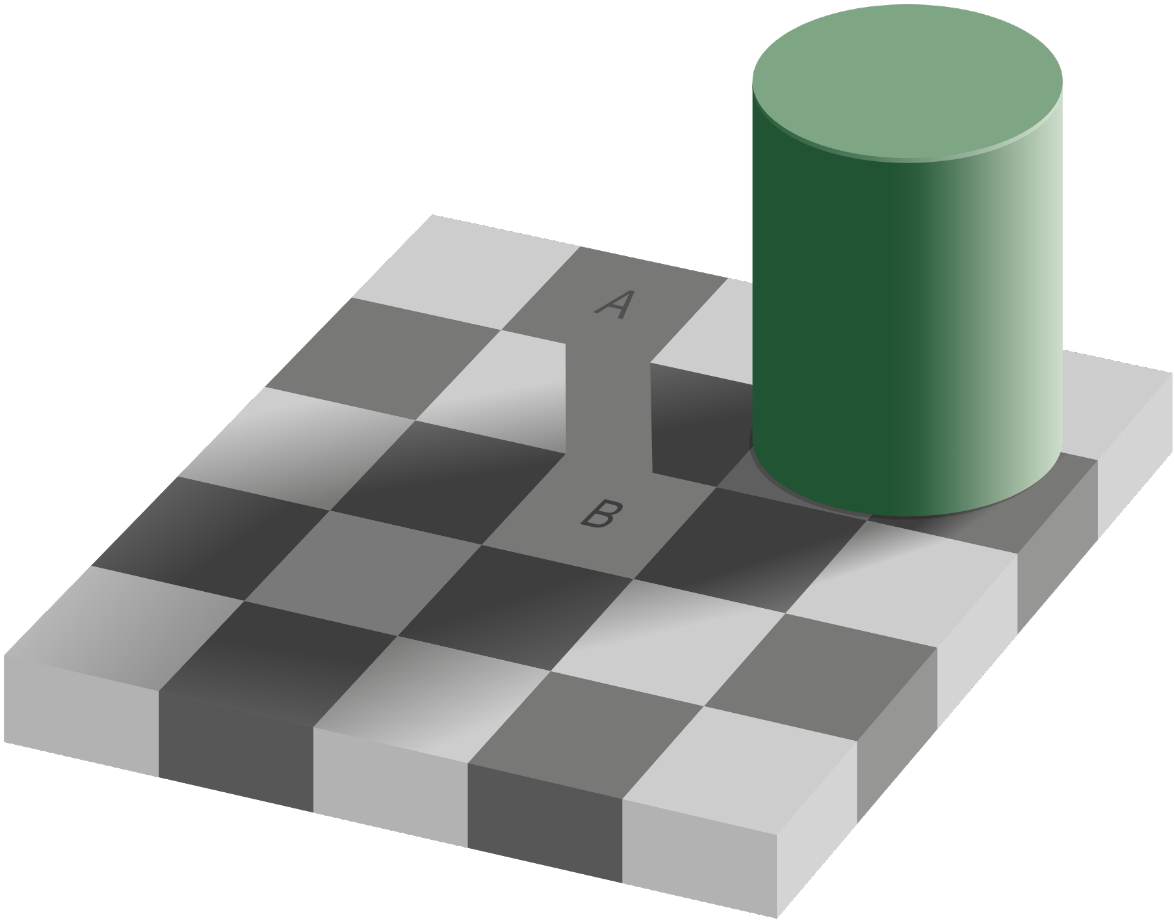
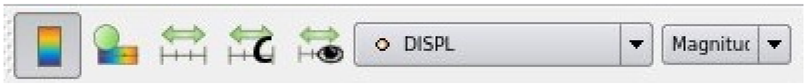
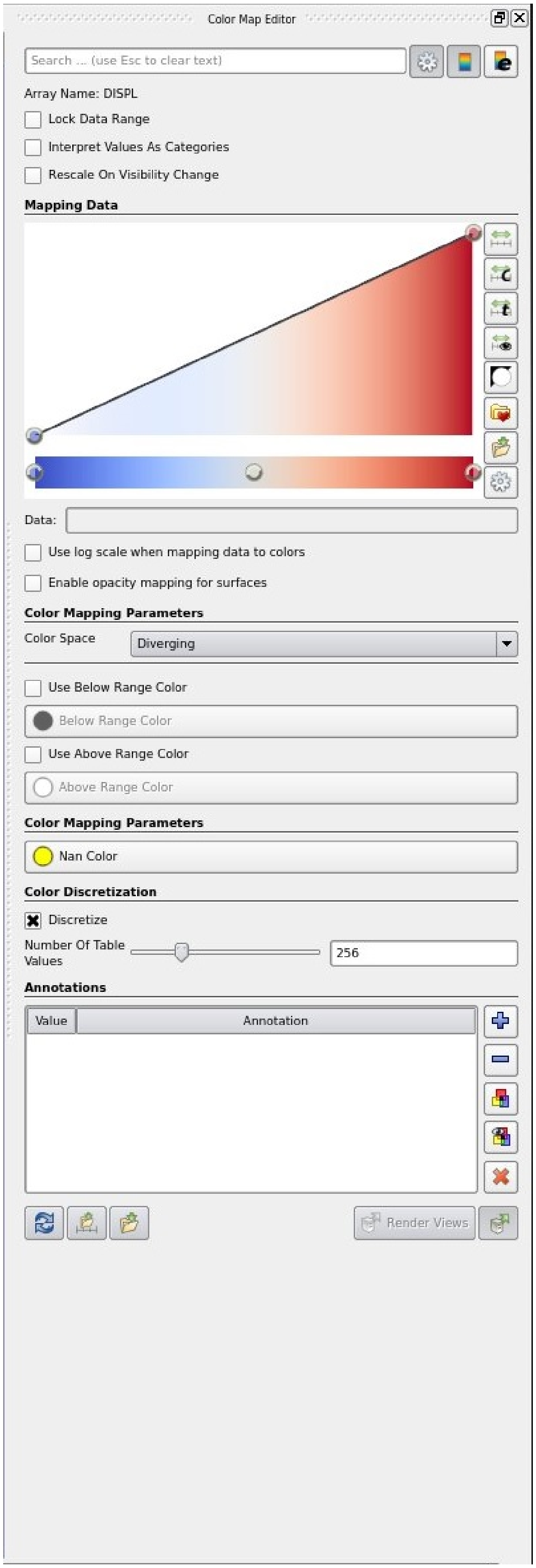


Fig. 4.12: The checker shadow illusion: drawing a connecting bar between the two squares breaks the illusion and shows that they are the same shade. Copyright Edward H. Adelson and Adrian Pingstone : Copyrighted free use.





Color Map Editor

Array Name: DISPL

☐ Lock Data Range

☐ Interpret Values As Categories

☐ Rescale On Visibility Change

Mapping Data

Data:

☐ Use log scale when mapping data to colors

☐ Enable opacity mapping for surfaces

Color Mapping Parameters

Color Space: Diverging

☐ Use Below Range Color

Below Range Color

☐ Use Above Range Color

Above Range Color

Color Mapping Parameters

Nan Color

Color Discretization

☒ Discretize

Number Of Table Values: 256

Annotations

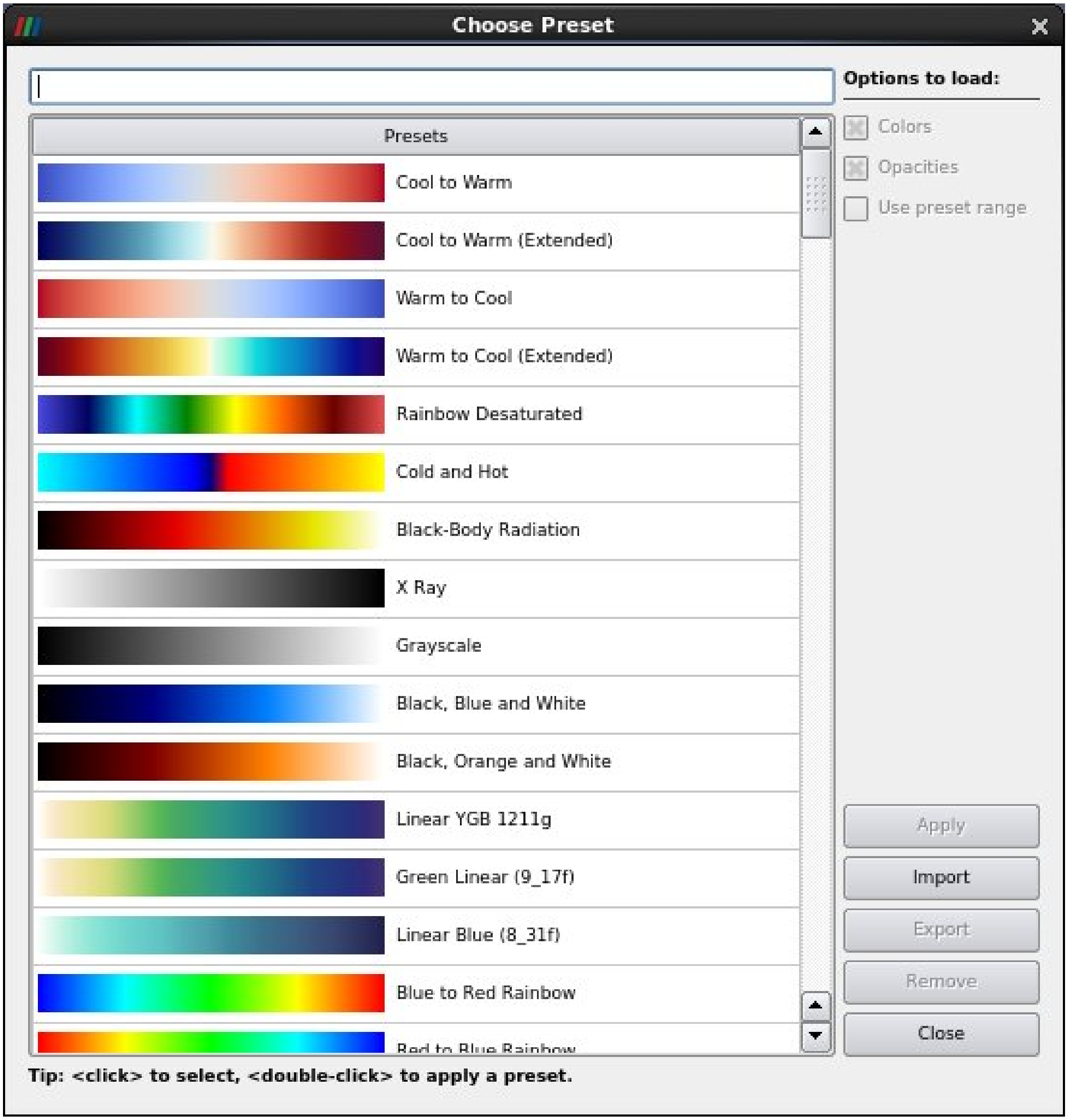
| Value | Annotation |
|---|---|

  - Select the **Use log scaling when mapping data to colors** check box. Notice what has happened to the colors on the can.
  - Unselect **Use log scaling when mapping data to colors** check box.
  - Select the **Enable opacity mapping for surfaces** check box. Notice what has happened to the colors on the can.
  - Unselect **Enable opacity mapping mapping for surfaces** check box.
  - Select the **Rescale to Data Range** icon. You already know what this does.
  - Select the **Rescale to Custom Range** icon.
  - Set custom minimum and maximum values for the Color Legend.
  - Select the **Rescale to Data Range over all timesteps** icon. This will read in all of your data, and set the min and max based upon all timesteps.
  - Select the **Rescale to Visible Range** icon. Rescales based upon what is currently visible. To see this work, click -Z, and then roll the can down slightly to hide the can itself. Click the **Rescale to Visible Range** icon.
  - Invert the transfer function.
  - Invert the color table.
  - Select the **Choose Preset** icon.
  - Choose **Cool to Warm**
  - **OK**. Notice what has happened to the colormap. This is an easier to understand color map than the rainbow one that all of us are used to.
  - Next, choose the **Black-Body Radiation** preset. Although harder to understand (requiring an understanding of the layout of the rainbow colors), this one makes prettier pictures. The best color map to use is the default one.
- Advanced. You can edit each color and opacity of the color map.
  - Nan Color - this is the color ParaView will use to paint nans in your data.
  - Click the advanced icon at the top. You can mark cells or points that are outside of the normal range of the color map.
  - Click the color legend with the **e**. This is the **Color Legend Editor**.

### Custom Color Maps

ParaView allows you to import custom color maps. This is also done on the **Color Map Editor**, **Presets** dialog. An excellent source for scientific visualization color maps is the SciVizColor website, located here: https://sciviscolor.org. Lets add a custom color map.

- Go to webpage https://sciviscolor.org.
- **Colormap**.
- **Wave & Highlighting Colormaps**.
- **Highlight Inserts**
- Select one of the middle color maps.
- Save this color map somewhere you can find it.
- Open **disk_out_ref.ex2**.



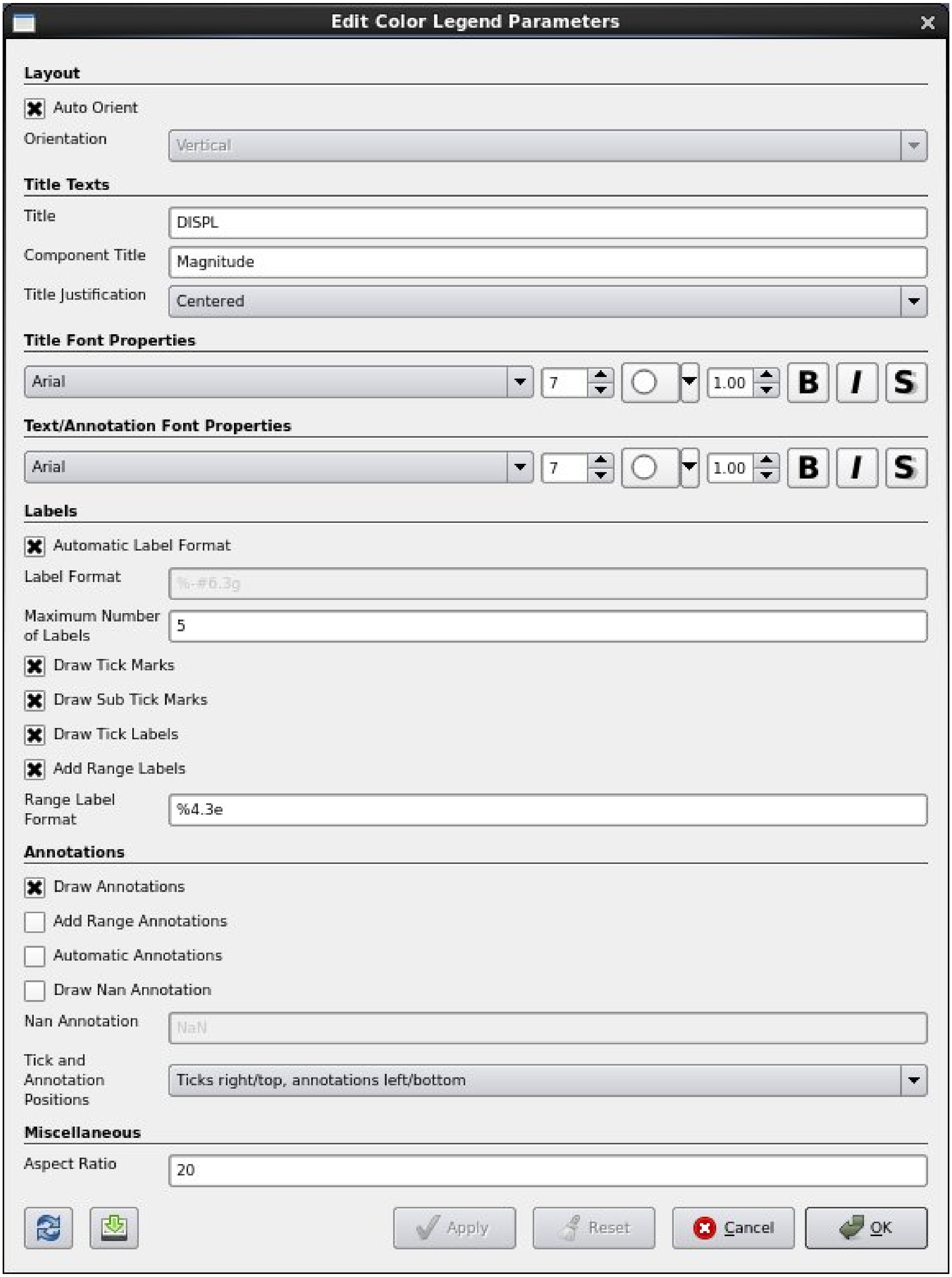
**Options to load:**

☒ Colors

☒ Opacities

□ Use preset range

**Tip: <click> to select, <double-click> to apply a preset.**



Edit Color Legend Parameters

**Layout**

☒ Auto Orient

Orientation: Vertical

**Title Texts**

Title: DISPL

Component Title: Magnitude

Title Justification: Centered

**Title Font Properties**

Arial | 7 | ○ | 1.00 | B | I | S

**Text/Annotation Font Properties**

Arial | 7 | ○ | 1.00 | B | I | S

**Labels**

☒ Automatic Label Format

Label Format: %-#6.3g

Maximum Number of Labels: 5

☒ Draw Tick Marks

☒ Draw Sub Tick Marks

☒ Draw Tick Labels

☒ Add Range Labels

Range Label Format: %4.3e

**Annotations**

☒ Draw Annotations

□ Add Range Annotations

□ Automatic Annotations

□ Draw Nan Annotation

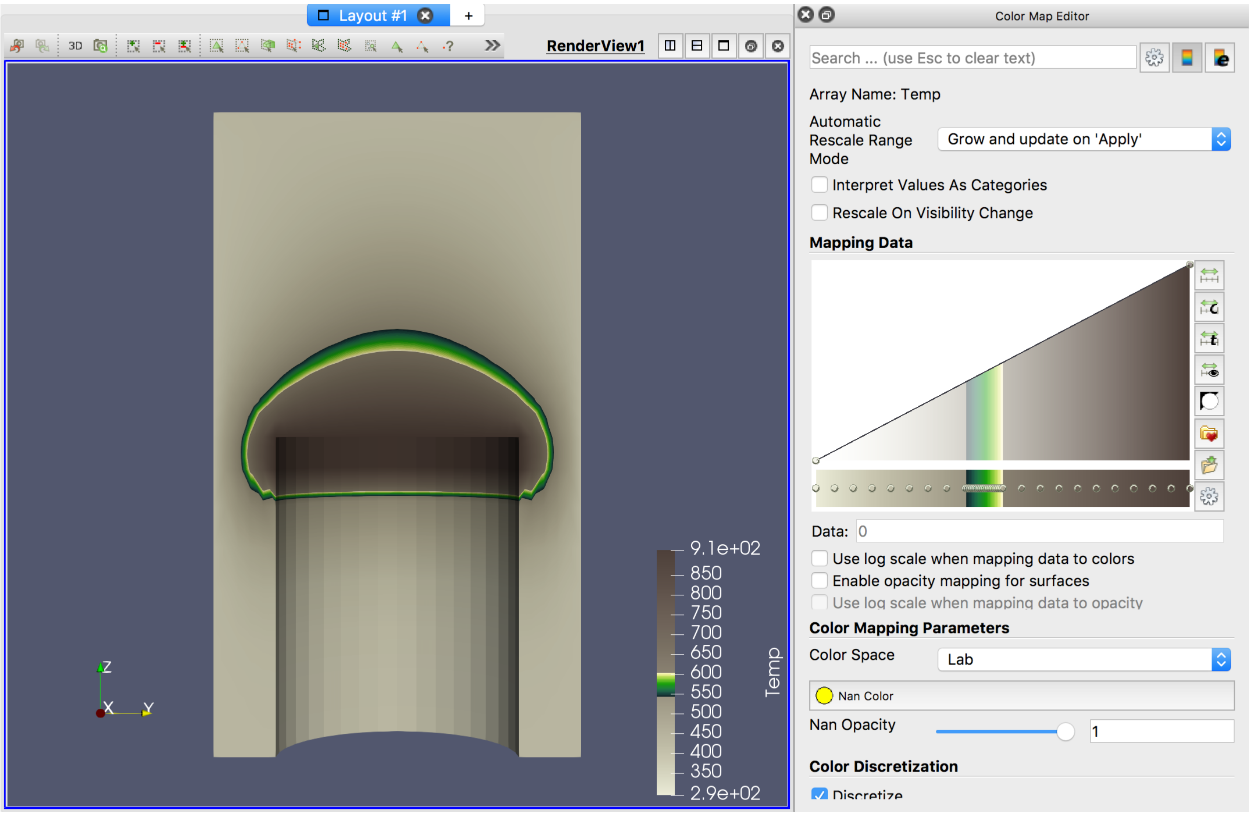
Nan Annotation: NaN

Tick and Annotation Positions: Ticks right/top, annotations left/bottom

**Miscellaneous**

Aspect Ratio: 20

- **Apply**
- **-X**
- Select **Clip** filter.
- **Apply**.
- Turn off the **Show Plane** check box.
- Select variable **Temp**
- Open the **Color Map Editor**.
- Open the **presets** dialog.
- **Import**.
- Select the color map .xml file that you downloaded.
- **Advanced**.
- Now, select the new color map.



Search ... (use Esc to clear text)

Array Name: Temp

Automatic Rescale Range Mode: Grow and update on 'Apply'

☐ Interpret Values As Categories

☐ Rescale On Visibility Change

Data: 0

☐ Use log scale when mapping data to colors

☐ Enable opacity mapping for surfaces

☐ Use log scale when mapping data to opacity

Color Space: Lab

Nan Color

Nan Opacity: 1

☑ Discretize

### Opacity mapping

- Leave the color editor open.
- Open **can.ex2**.
- **Apply**.
- **+Y**
- Go to last timestep.
- Set **Coloring** to **Accl**.
- Go to first timestep.
- Choose a black body palette.
- Enable opacity mapping for surfaces.
- Play
- **Filters** → **Alphabetical** → **Pass Arrays**.
- **Apply**.
- Change this filter to be wireframe, solid color. Opacity 0.1.
- Back on can.ex2, turn visibility on.
- Click on and change the opacity transfer function.
- Turn off the enable opacity mapping for surfaces.
- Volume render.
- Play

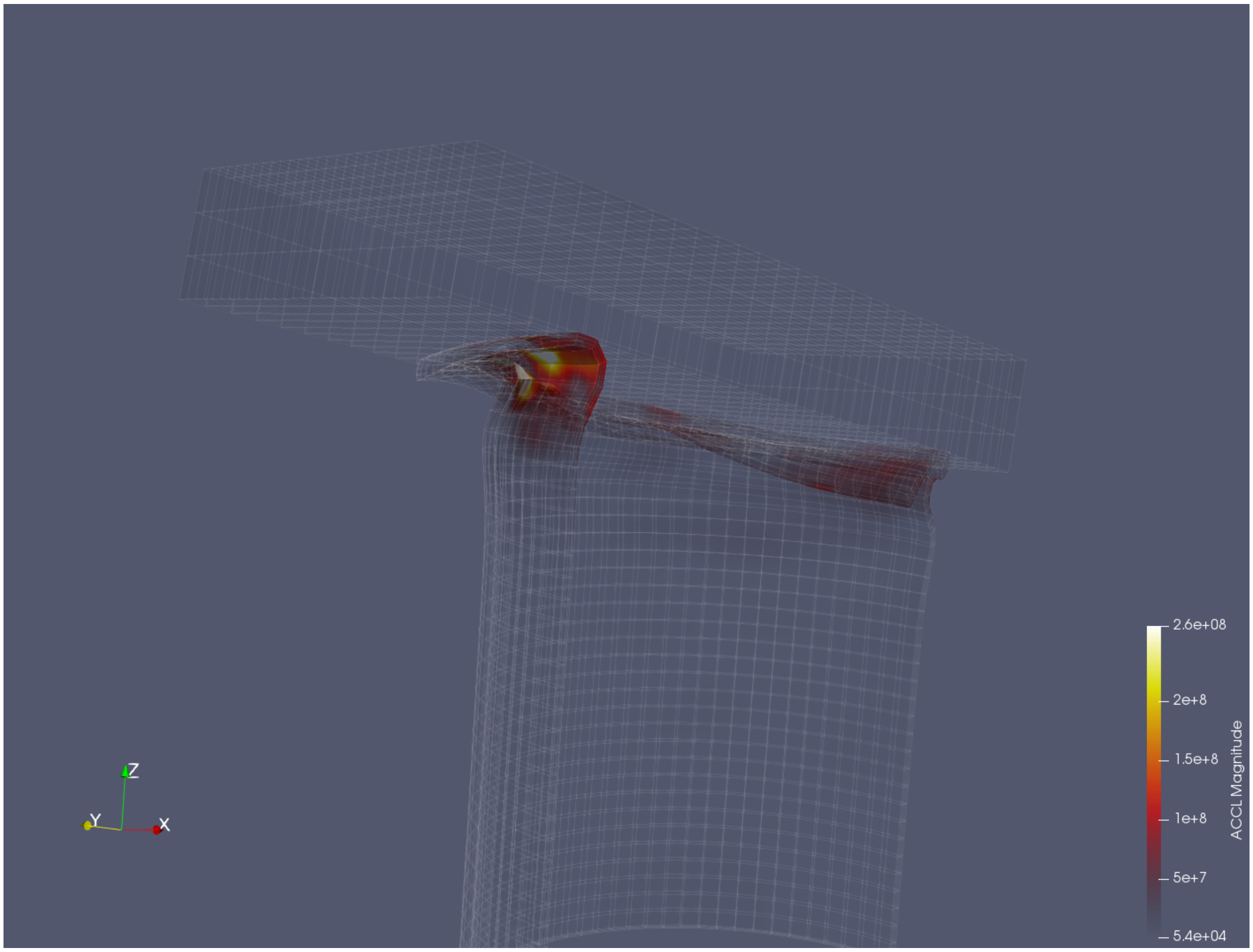
## 4.2.5 Beginning: Plotting

### Introduction

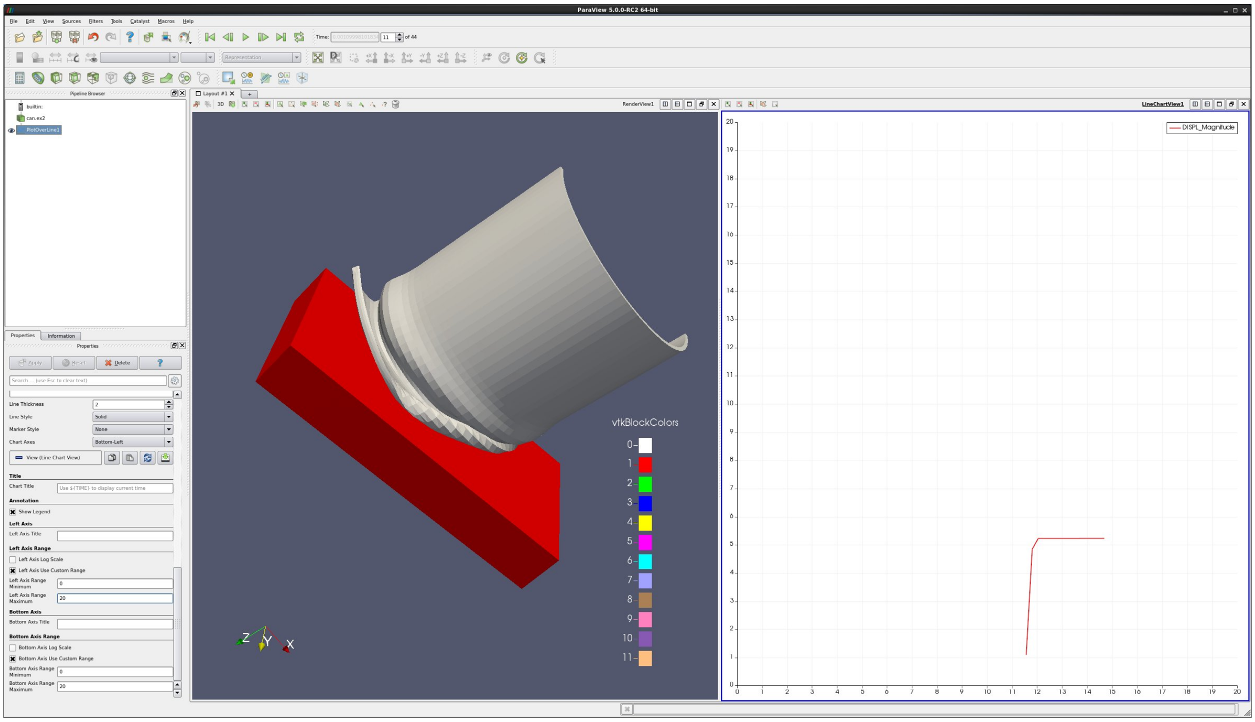
This use case shows a user how to plot cell and point data. Plotting can be along a line that cuts through your data, or a location with respect to time.

### Plot along a line

- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- Select **Filter** → **Data Analysis** → **Plot Over Line**.
- Drag the top of the line to intersect the top of the can. Note that hitting the p key will also place the line on the surface of your object. You can also use the **1** and **2** keys to set the beginning and end of the line.
- **Apply**.
- In the **Properties** tab, unselect all variables except **DISPL (Magnitude)**.
- Lower on the **Properties** tab, select **Left Axis Use Custom Range**, and enter **0.0** and **20.0**.
- Select **Bottom Axis Use Custom Range**, and enter **0.0** and **20.0**.


Z
Y
X
2.6e+08
2e+8
1.5e+8
1e+8
5e+7
5.4e+04
ACCL Magnitude

- **Play** the animation forward, and notice what happens to the plot.
- You can also add plot labels and axis labels on the properties tab.



Search ... (use Esc to clear text)

Line Thickness: 2

Line Style: Solid

Marker Style: None

Chart Axes: Bottom-Left

View (Line Chart View)

**Title**

Chart Title: Use ${TIME} to display current time

**Annotation**

☒ Show Legend

**Left Axis**

Left Axis Title:

**Left Axis Range**

□ Left Axis Log Scale

☒ Left Axis Use Custom Range

Left Axis Range Minimum: 0

Left Axis Range Maximum: 20

**Bottom Axis**

Bottom Axis Title:

**Bottom Axis Range**

□ Bottom Axis Log Scale

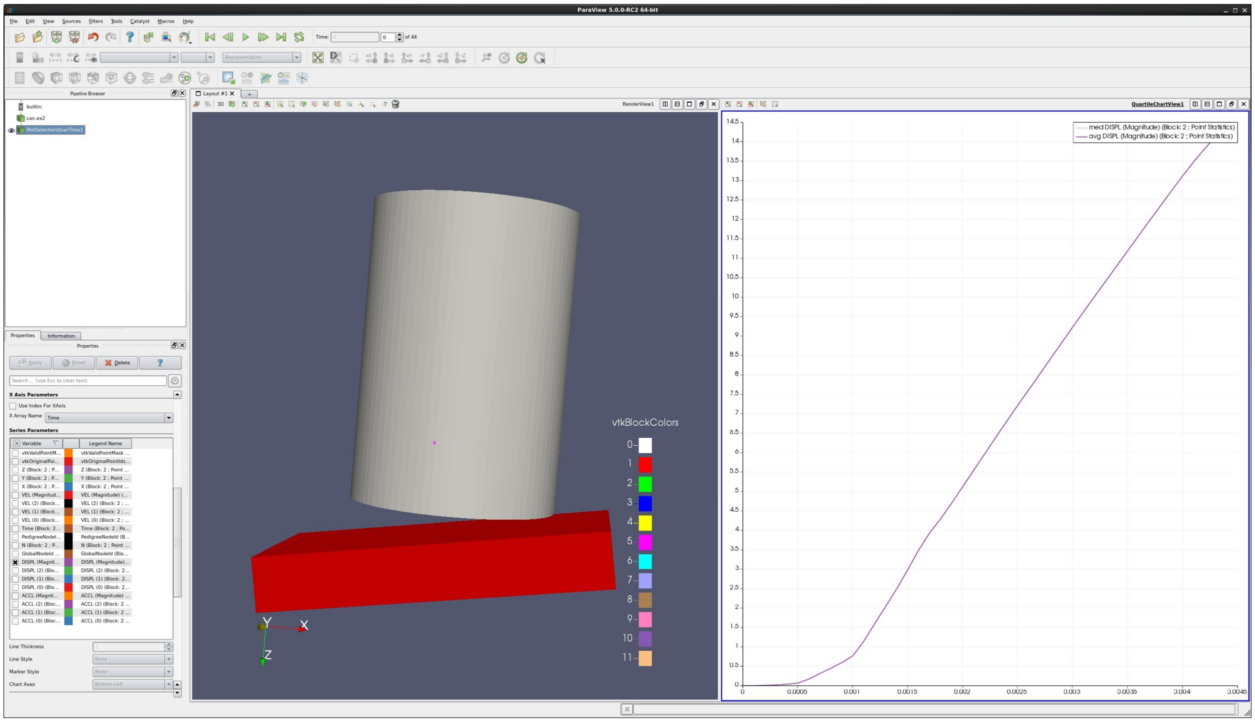
☒ Bottom Axis Use Custom Range

Bottom Axis Range Minimum: 0

Bottom Axis Range Maximum: 20

### Plot point over time

- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- We want to plot a point over time.
- First, we need to select a point.
- Click on **Select Points On** icon.
- Then, click on a point on the can.
- **Filters** → **Data Analysis** → **Plot Point Over Time**.
- **Apply**.
- In the **Properties** tab, unselect everything except **DISPL (Magnitude)**.
- You now have a plot of the displacement. at your point.
- Extra Credit:
  - Using **Select Cells On** icon, and **Plot Cell Over Time**, plot **EQPS** over time.

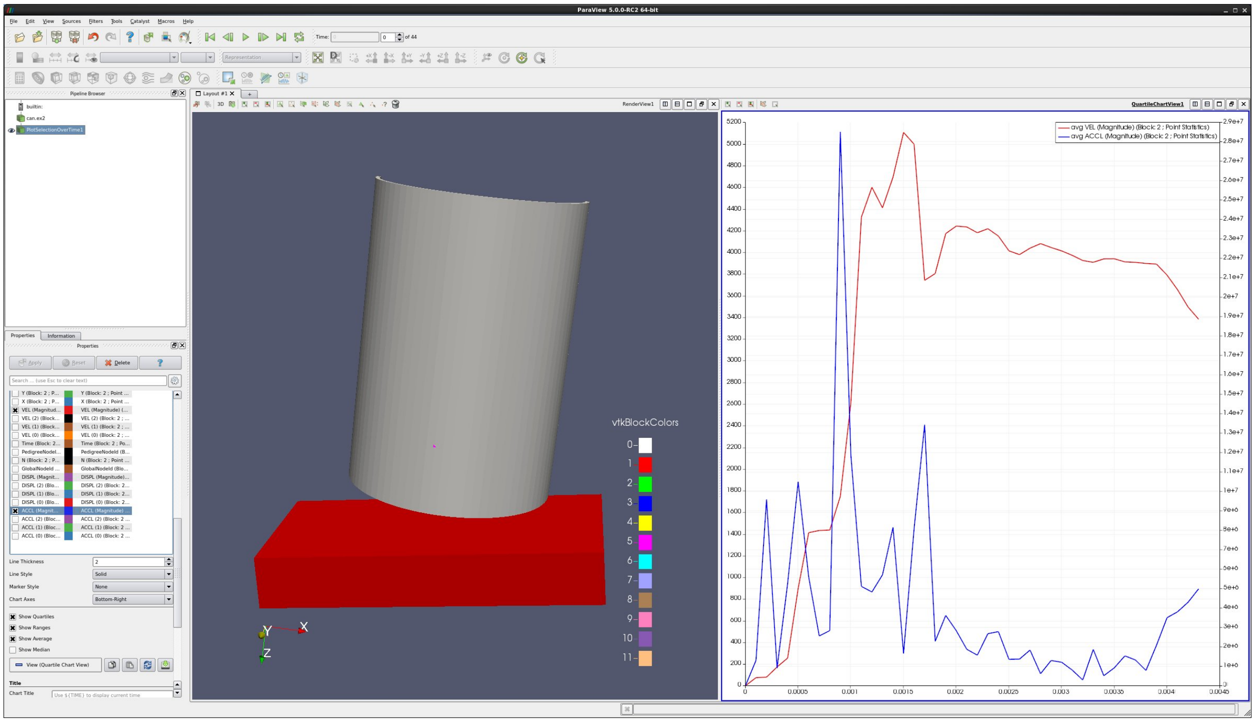


## Plot two variables at same time

- Follow the steps for **Plot Point over time**, as described above.
- In the **Properties** tab, turn off **DISPL (Magnitude)**, and turn on variables **VEL (Magnitude)** and **ACCL (Magnitude)**.
- Select the line that says **ACCL (Magnitude)**, causing this row to turn gray.
- At the bottom of the **Properties** tab, change **Chart Axes** to **Bottom-Right**.
- Note below that the ACCL (Magnitude) color has been changed to blue.

## Plot multiple points (statistics plot version)

- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- We want to plot numerous points over time. First, we need to select the points.
- Click on **Select Points On** icon. Then, rubber band select a dozen points.
- **Filters** → **Data Analysis** → **Plot Selection Over Time**.
- **Apply**.
- On the Properties tab, turn off all variables, and then turn on variables **VEL (Magnitude)** and **ACCL (Magnitude)**.
- Select the line that says **ACCL (Magnitude)**, causing this row to turn gray.



Properties | Information

Apply | Reset | Delete | ?

Search ... (use Esc to clear text)

- ☐ Y (Block: 2 ; P... 🟩 Y (Block: 2 ; Point ...
- ☐ X (Block: 2 ; P... 🟦 X (Block: 2 ; Point ...
- ☒ VEL (Magnitud... 🟥 VEL (Magnitude) (...
- ☐ VEL (2) (Block... ⬛ VEL (2) (Block: 2 ; ...
- ☐ VEL (1) (Block... 🟫 VEL (1) (Block: 2 ; ...
- ☐ VEL (0) (Block... 🟧 VEL (0) (Block: 2 ; ...
- ☐ Time (Block: 2... 🟫 Time (Block: 2 ; Po...
- ☐ PedigreeNodel... ⬛ PedigreeNodeId (B...
- ☐ N (Block: 2 ; P... ⬛ N (Block: 2 ; Point ...
- ☐ GlobalNodeId ... 🟫 GlobalNodeId (Blo...
- ☐ DISPL (Magnit... 🟪 DISPL (Magnitude)...
- ☐ DISPL (2) (Blo... 🟩 DISPL (2) (Block: 2...
- ☐ DISPL (1) (Blo... 🟦 DISPL (1) (Block: 2...
- ☐ DISPL (0) (Blo... 🟥 DISPL (0) (Block: 2...
- ☒ ACCL (Magnit... 🟦 ACCL (Magnitude) ...
- ☐ ACCL (2) (Bloc... 🟪 ACCL (2) (Block: 2 ...
- ☐ ACCL (1) (Bloc... 🟩 ACCL (1) (Block: 2 ...
- ☐ ACCL (0) (Bloc... 🟦 ACCL (0) (Block: 2 ...

Line Thickness: 2
Line Style: Solid
Marker Style: None
Chart Axes: Bottom-Right

☒ Show Quartiles
☒ Show Ranges
☒ Show Average
☐ Show Median

View (Quartile Chart View)

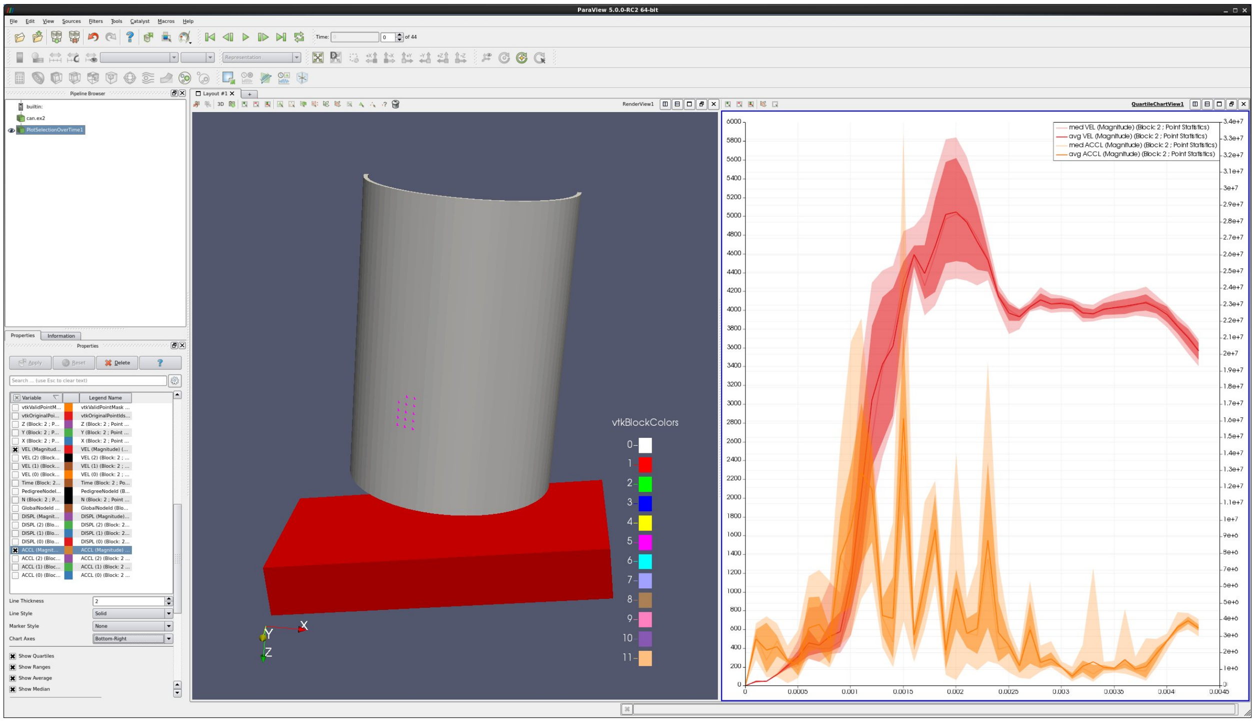
Title
Chart Title: Use ${TIME} to display current time

avg VEL (Magnitude) (Block: 2 ; Point Statistics)
avg ACCL (Magnitude) (Block: 2 ; Point Statistics)

- In the **Properties** tab, change **Chart Axes** to **Bottom-Right**.

## Plot multiple points (spaghetti plot version)

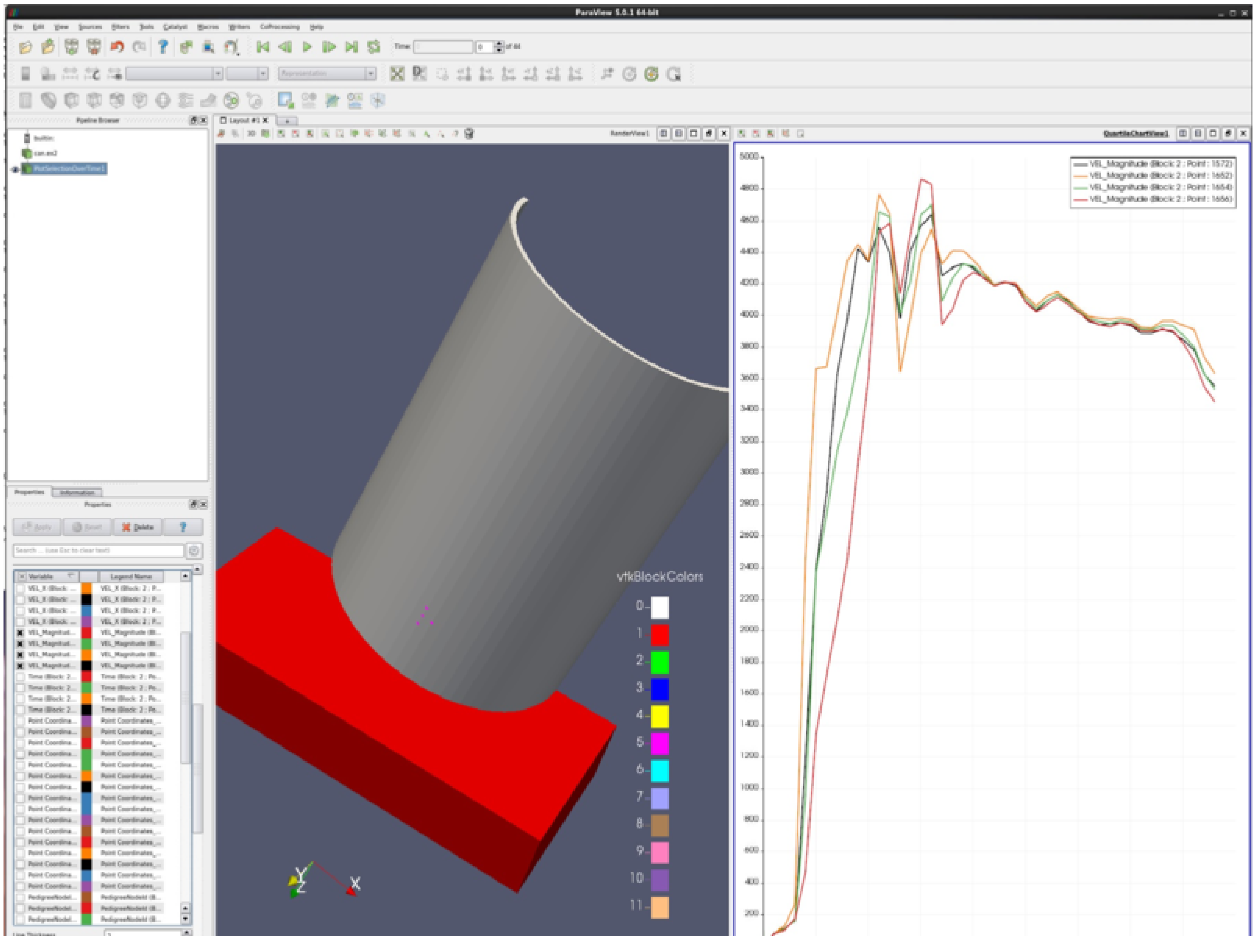
- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- We want to plot numerous points over time. First, we need to select the points.
- Click on **Select Points On** icon. Then, rubber band select about four points.
- **Filters** → **Data Analysis** → **Plot Selection Over Time**.
- **Apply**.
- On the **Properties**. tab, unselect the **Only Report Selection Statistics** checkbox.
- **Apply**.
- On the Properties tab, display section, click on **Root**, turning on all of the points or cells.
- On the Properties tab, turn off everything, then turn on all variables **VEL (Magnitude)**.



| ☒ Variable | | Legend Name |
|---|---|---|
| ☐ vtkValidPointM... | 🟧 | vtkValidPointMask ... |
| ☐ vtkOriginalPoi... | 🟥 | vtkOriginalPointIds... |
| ☐ Z (Block: 2 ; P... | 🟪 | Z (Block: 2 ; Point ... |
| ☐ Y (Block: 2 ; P... | 🟩 | Y (Block: 2 ; Point ... |
| ☐ X (Block: 2 ; P... | 🟦 | X (Block: 2 ; Point ... |
| ☒ VEL (Magnitud... | 🟥 | VEL (Magnitude) (... |
| ☐ VEL (2) (Block... | ⬛ | VEL (2) (Block: 2 ; ... |
| ☐ VEL (1) (Block... | 🟫 | VEL (1) (Block: 2 ; ... |
| ☐ VEL (0) (Block... | 🟧 | VEL (0) (Block: 2 ; ... |
| ☐ Time (Block: 2... | 🟫 | Time (Block: 2 ; Po... |
| ☐ PedigreeNodeI... | ⬛ | PedigreeNodeId (B... |
| ☐ N (Block: 2 ; P... | ⬛ | N (Block: 2 ; Point ... |
| ☐ GlobalNodeId ... | 🟫 | GlobalNodeId (Blo... |
| ☐ DISPL (Magnit... | 🟪 | DISPL (Magnitude)... |
| ☐ DISPL (2) (Blo... | 🟩 | DISPL (2) (Block: 2... |
| ☐ DISPL (1) (Blo... | 🟦 | DISPL (1) (Block: 2... |
| ☐ DISPL (0) (Blo... | 🟥 | DISPL (0) (Block: 2... |
| ☒ ACCL (Magnit... | 🟧 | ACCL (Magnitude) ... |
| ☐ ACCL (2) (Bloc... | 🟪 | ACCL (2) (Block: 2 ... |
| ☐ ACCL (1) (Bloc... | 🟩 | ACCL (1) (Block: 2 ... |
| ☐ ACCL (0) (Bloc... | 🟦 | ACCL (0) (Block: 2 ... |

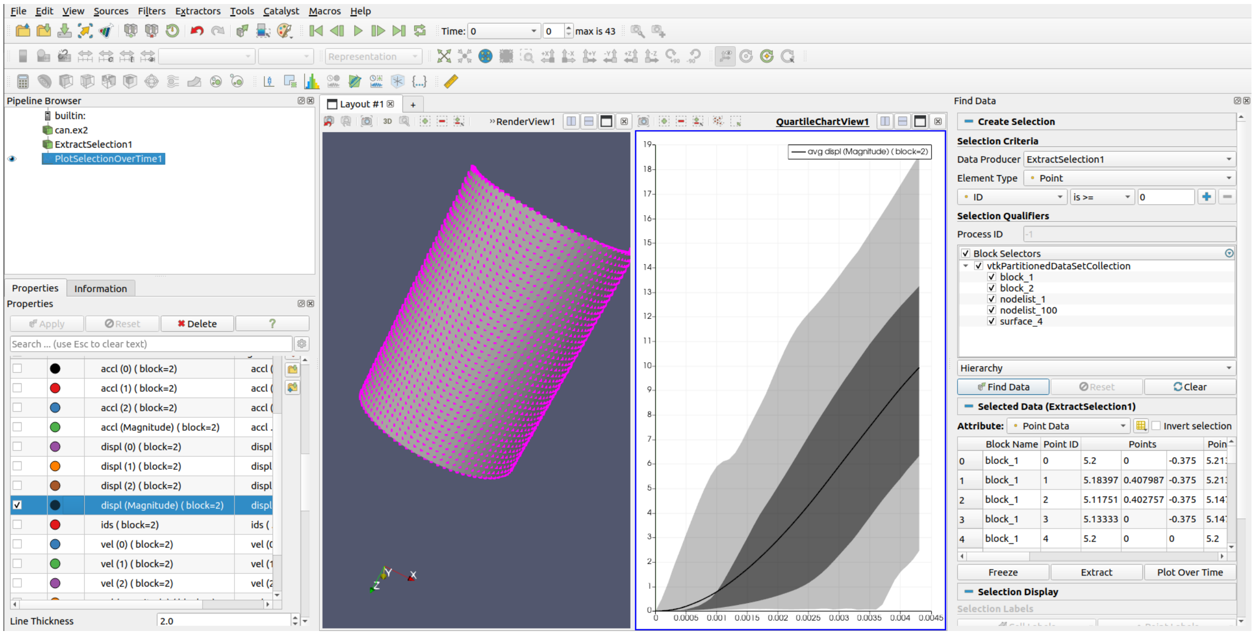
Line Thickness 2
Line Style Solid
Marker Style None
Chart Axes Bottom-Right

☒ Show Quartiles
☒ Show Ranges
☒ Show Average
☒ Show Median



### Plot min, max and average for all points of a block over time

- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- We want to plot all of the can points over time. First, we need to select the points.
- Click on **Select Block** icon.
- Select the can.
- **Filters** → **Data Analysis** → **Extract Selection**.
- **Apply**.
- We now have the can.
- Select **Edit -> Find Data**.
- Set **Data Producer** to **ExtractSelection1**.
- Set **Element Type** to Point.
- Use **ID** and **is >=** and **0**.
- **Find Data**.
- **Plot Selection Over Time**.
- **Apply**.
- In **Series Parameters**, turn off everything except **DISPL (Magnitude)**



Pipeline Browser

- builtin:
- can.ex2
- ExtractSelection1
- PlotSelectionOverTime1

Properties | Information

Properties

Apply | Reset | Delete | ?

Search ... (use Esc to clear text)

| | | | |
|---|---|---|---|
| ☐ | ● | accl (0) ( block=2) | accl ( |
| ☐ | ● | accl (1) ( block=2) | accl ( |
| ☐ | ● | accl (2) ( block=2) | accl ( |
| ☐ | ● | accl (Magnitude) ( block=2) | accl . |
| ☐ | ● | displ (0) ( block=2) | displ |
| ☐ | ● | displ (1) ( block=2) | displ |
| ☐ | ● | displ (2) ( block=2) | displ |
| ☑ | ● | displ (Magnitude) ( block=2) | displ |
| ☐ | ● | ids ( block=2) | ids ( |
| ☐ | ● | vel (0) ( block=2) | vel (( |
| ☐ | ● | vel (1) ( block=2) | vel (1 |
| ☐ | ● | vel (2) ( block=2) | vel (2 |

Line Thickness 2.0

Find Data

Create Selection

Selection Criteria

Data Producer: ExtractSelection1

Element Type: Point

ID | is >= | 0

Selection Qualifiers

Process ID: -1

☑ Block Selectors
- ☑ vtkPartitionedDataSetCollection
  - ☑ block_1
  - ☑ block_2
  - ☑ nodelist_1
  - ☑ nodelist_100
  - ☑ surface_4

Hierarchy

Find Data | Reset | Clear

Selected Data (ExtractSelection1)

Attribute: Point Data ☐ Invert selection

| | Block Name | Point ID | Points | | | Poin |
|---|---|---|---|---|---|---|
| 0 | block_1 | 0 | 5.2 | 0 | -0.375 | 5.21 |
| 1 | block_1 | 1 | 5.18397 | 0.407987 | -0.375 | 5.21 |
| 2 | block_1 | 2 | 5.11751 | 0.402757 | -0.375 | 5.14 |
| 3 | block_1 | 3 | 5.13333 | 0 | -0.375 | 5.14 |
| 4 | block_1 | 4 | 5.2 | 0 | 0 | 5.2 |

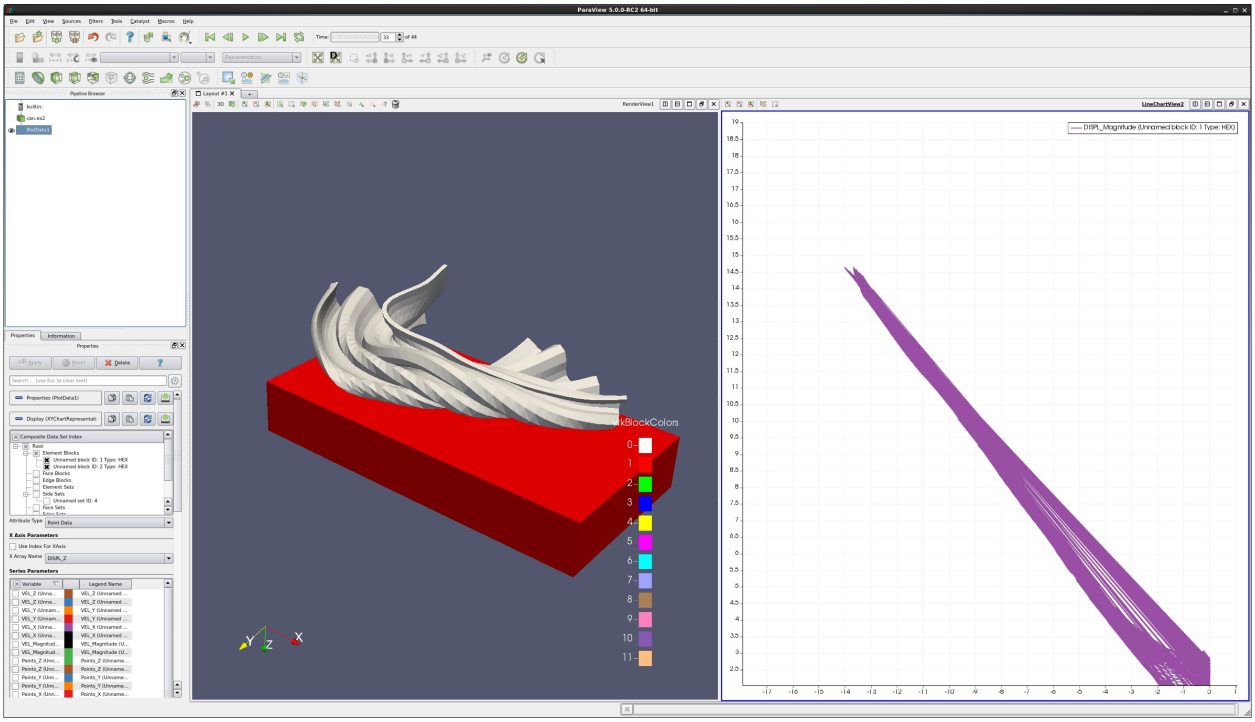
Freeze | Extract | Plot Over Time

Selection Display

Selection Labels

### Plot data

- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- Drag the can around with the left mouse button until you can see the can.
- We want to plot two variables against each other, for all points (or cells) of the can.
- **Filters** → **Data Analysis** → **Plot Data**.
- **Apply**.
- On the **Properties** tab, turn off **Use Index for X Array** and for X Array Name select **DISPL_Z**
- In **Series Parameters**, turn off everything except **DISPL (Magnitude)**
- **Play**



Properties (PlotData1)

Display (XYChartRepresentati

☒ Composite Data Set Index
- ☒ Root
  - ☒ Element Blocks
    - ☒ Unnamed block ID: 1 Type: HEX
    - ☒ Unnamed block ID: 2 Type: HEX
  - □ Face Blocks
  - □ Edge Blocks
  - □ Element Sets
  - □ Side Sets
    - □ Unnamed set ID: 4
  - □ Face Sets

Attribute Type: Point Data

**X Axis Parameters**

□ Use Index For XAxis

X Array Name: DISPL_Z

**Series Parameters**

| ☒ Variable | | Legend Name |
|---|---|---|
| □ VEL_Z (Unna... | 🟫 | VEL_Z (Unnamed ... |
| □ VEL_Z (Unna... | 🟦 | VEL_Z (Unnamed ... |
| □ VEL_Y (Unnam... | 🟧 | VEL_Y (Unnamed ... |
| □ VEL_Y (Unnam... | 🟥 | VEL_Y (Unnamed ... |
| □ VEL_X (Unna... | 🟪 | VEL_X (Unnamed ... |
| □ VEL_X (Unna... | ⬛ | VEL_X (Unnamed ... |
| □ VEL_Magnitud... | ⬛ | VEL_Magnitude (U... |
| □ VEL_Magnitud... | 🟩 | VEL_Magnitude (U... |
| □ Points_Z (Unn... | 🟩 | Points_Z (Unname... |
| □ Points_Z (Unn... | 🟫 | Points_Z (Unname... |
| □ Points_Y (Unn... | 🟦 | Points_Y (Unname... |
| □ Points_Y (Unn... | 🟧 | Points_Y (Unname... |
| □ Points_X (Unn... | 🟥 | Points_X (Unname... |

## 4.2.6 Beginning: Pictures & Movies

### Introduction

This usecase shows a user how to add pictures and movies to PowerPoint slides.

### Save Screenshot

- Start `paraview`.
- Open can.ex2.
- **Apply**.
- Rotate and position the object.
- **File** → **Save Screenshot**.
  - Save only selected view will save either the whole window, or the selected view.
  - Change the size of the output files. Note that `paraview` can save files that are significantly larger than the current view. This is done in off screen memory.
  - Leave image quality as is.
  - Override Color Palette allows a user to change the background color - for instance, to white.
  - Stereo mode allows a user to save stereo images.
- **OK**
  - Save your images as .png or .jpg. .png is a cleaner format, but .jpg's may be smaller.
- Name and save the file. **OK**

In PowerPoint:

- Add a new slide.
- Next, click **Insert** → **Picture**→ **From File…**, and pick up the file that we saved in the previous paragraph.
- You can move the file anywhere you want by holding the left mouse button and dragging.
- You can resize the picture by grabbing one of the corners and dragging the picture smaller.

### Save Animation (make a movie)

- Start `paraview`.
- Open can.ex2.
- **Apply**.
- Rotate and position the object.
- **File** → **Save Animation**.
  - Leave Frane Rate and No. of Frames alone.
  - Change the size of the output files. Note that `paraview` can save files that are significantly larger than the current view. This is done in off screen memory.
  - Set timestep range.
  - Stereo mode allows a user to save stereo images.
- **OK**
  - Save your movie. A great option is to save your images as .png or .jpg, and then post process them into .avi or .mov files. This way, you have your movies and the individual frames. .png is a cleaner format, but .jpg's may be smaller.
- Name and save the file.

- **OK**

In PowerPoint:

- Add a new slide.
- Next, click **Insert** → **Movies and Sounds** → **Movie from File**.
- Browse to the location of your movie.
- Select your file.
- **OK**.
- Decide if you want the slide show to be Automatic, or when clicked.
- Finally, remember that this file is NOT inserted into the PowerPoint slide.
- You will want this file to be on the computer – in the same location – as it was when you selected it.

A few notes on file formats for movies. You can make .avi files directly on Windows, Linux or Mac. If you are trying to make movies on Linux or Mac, or are making movies for a powerwall, use format .png or .jpg. Then, if you want .avi files for use with PowerPoint, convert these movies to .avi format as needed. One option is to use enve, which is available when downloading EnSight. Windows reads .avi files well, both using and not using PowerPoint. Linux is able to display either format if the correct video player is installed. One option is VLC. VLC is installed from .rpm files, which can be found on the net.

## 4.2.7 Advanced: MultiBlock

### Introduction

This tutorial covers multiblock processing.

Most examples assume that the user starts with a new model. To start over, go to the menu item Edit → Reset Session, and then re-open your data.

Data is opened by going to File → Open. Example data files can be found in the Examples directory, located in the upper left of the Open dialog.

### Block commands onscreen menu

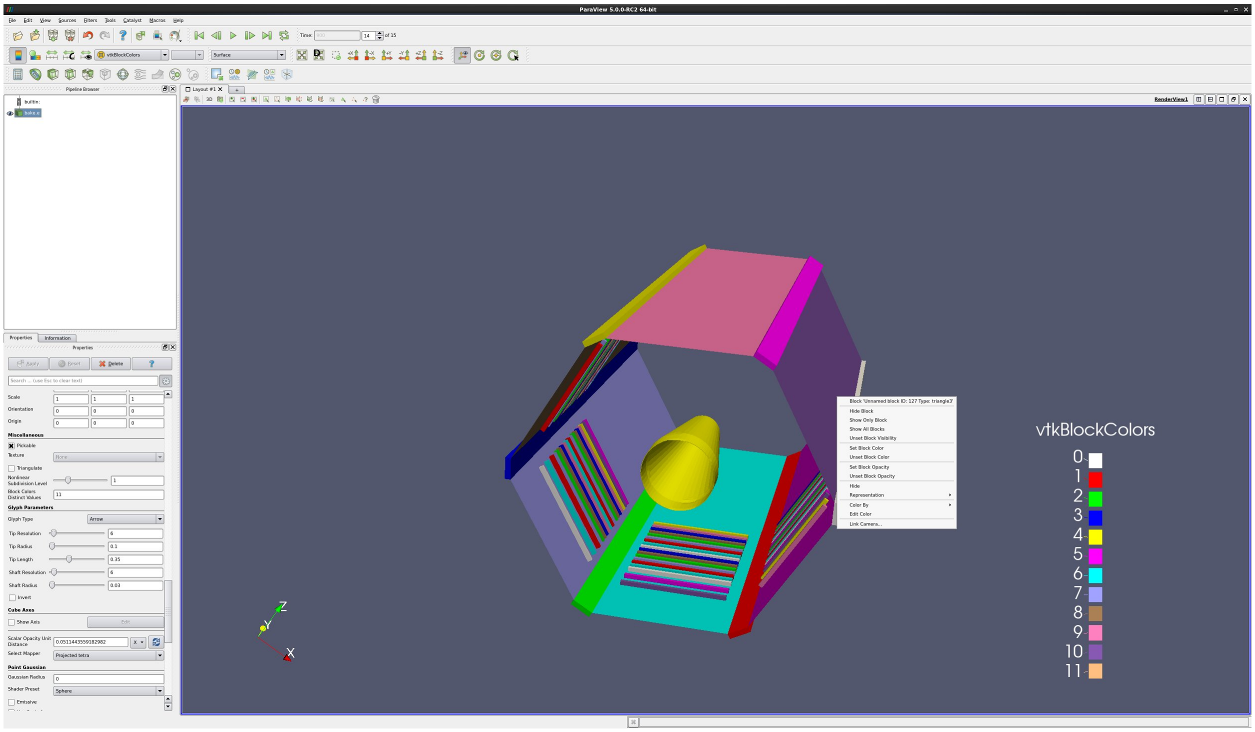
bake.e is a dataset that simulates the heating of a cone in a test facility. It has numerous blocks. **T** (for temperature) is the primary variable of interest.

- Open the dataset bake.e.
- **Apply**.
- Your dataset is now colored by block.
- Change **vtkBlockColors** to **T**.
- Your dataset is now colored by temperature.
- Play forward in time, and look into the simulation.
- Be sure to rescale to data range.
- Change color back to **vtkBlockColors**.

Notice that this model has 6 sides, and there are 12 annotation colors. Thus, they repeat. Lets make them random.

- In the **Properties** tab, click **Advanced**.

- Slide down towards the bottom, and find **Block Colors Distinct Values**, and change it to **11**.
- Right click on a side (a block). Notice that the bock name and ID are shown.
- Right click on a side. Change the block's color and opacity.
- Right click on a side. Hide, then un-hide the block.



Scale 1 1 1
Orientation 0 0 0
Origin 0 0 0

Miscellaneous

☒ Pickable
Texture None
☐ Triangulate
Nonlinear Subdivision Level 1
Block Colors Distinct Values 11

Glyph Parameters

Glyph Type Arrow
Tip Resolution 6
Tip Radius 0.1
Tip Length 0.35
Shaft Resolution 6
Shaft Radius 0.03
☐ Invert

Cube Axes

☐ Show Axis Edit
Scalar Opacity Unit Distance 0.0511443559182982 X
Select Mapper Projected tetra

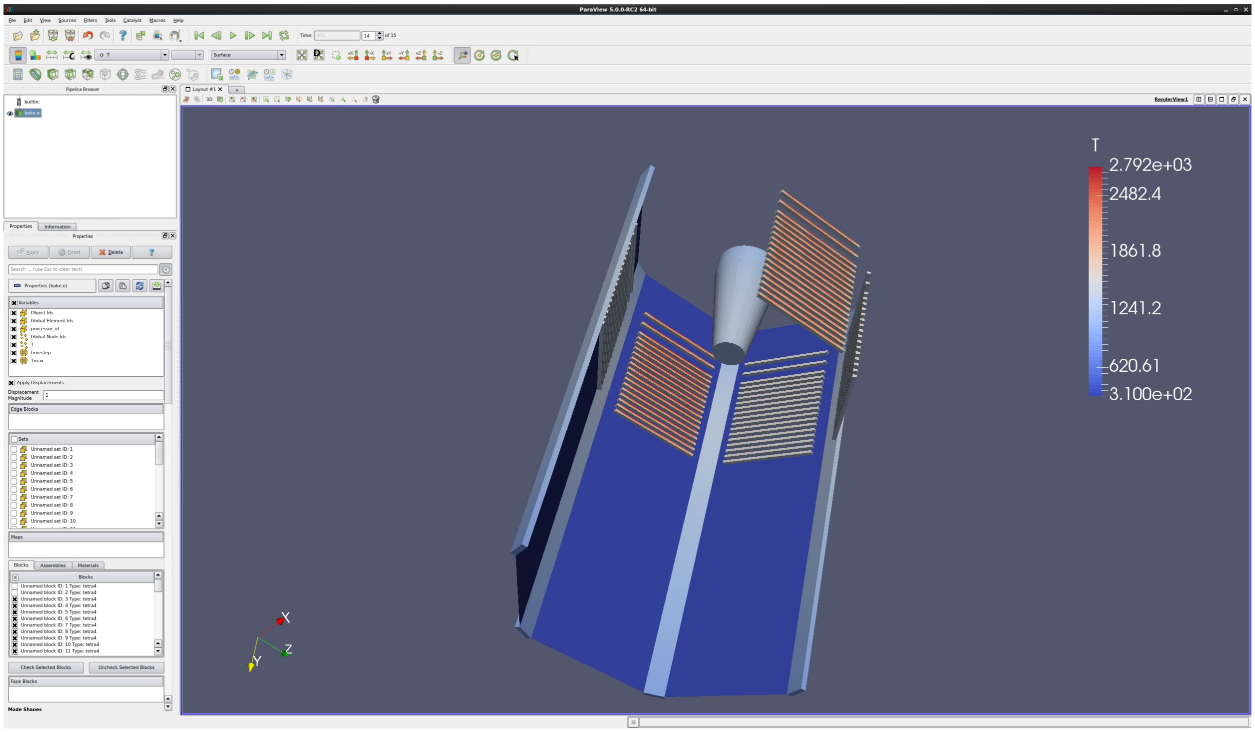
Point Gaussian

Gaussian Radius 0
Shader Preset Sphere
☐ Emissive

## Don't load some blocks

You can easily change which blocks you load. This is done through selections. Here is an example.

- Open the dataset bake.e.
- **Apply**.
- Your dataset is now colored by block.
- Choose the **Select Block** icon (or hit the **b** key), then rubber band select the blocks on the outside edge of bake. You have selected numerous blocks.
- In the **Properties** tab, slide down to the section named Blocks.
- Click **Uncheck Selected Blocks**.
- **Apply**.

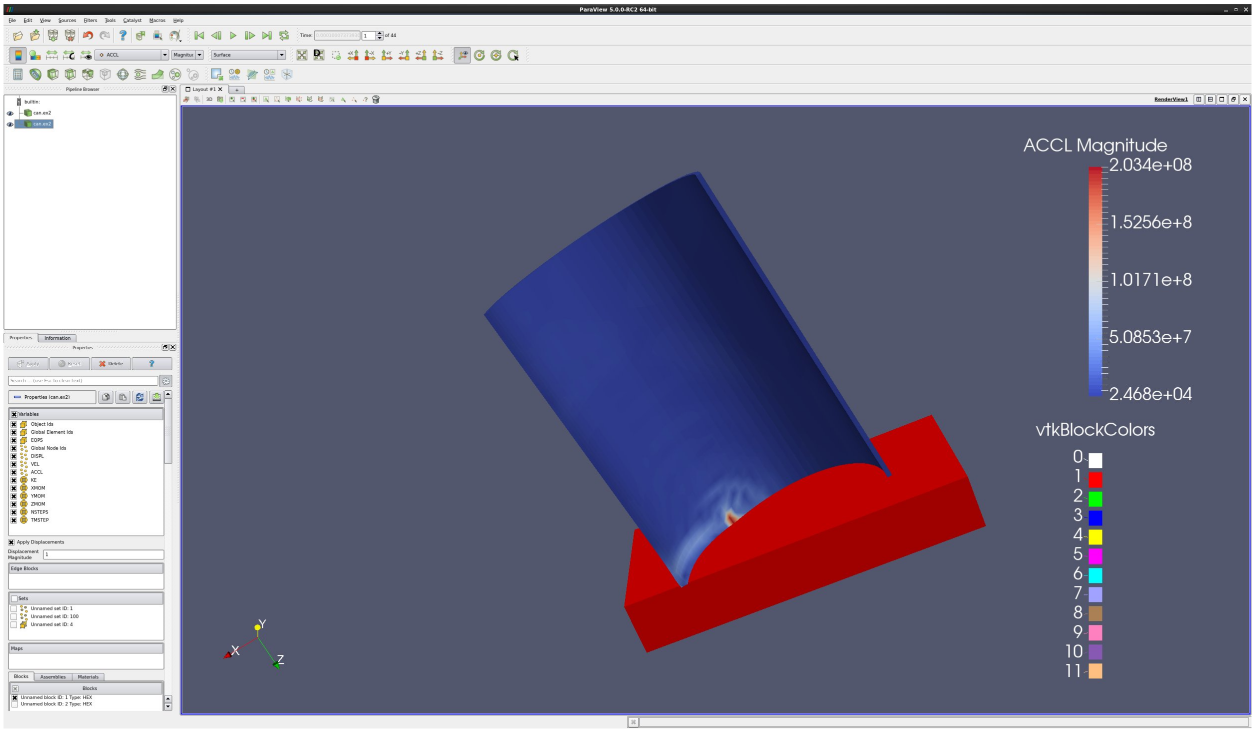


### Load the same dataset twice

- Open the dataset can.ex2.
- Only turn on block ID **2**.
- **Apply**.
- Open the dataset can.ex2.
- Only turn on block ID **1**.
- **Apply**.
- Change the timestep to 1.
- Set **Coloring** to **ACCL**.
- Play

### Extract Block and Extract Selection

Two filters that can also be used to partition your data are the **Extract Block** and **Extract Selection** filters. Instead of reading your data in twice, read it in once and Extract Block twice.



### MultiBlock inspector

The **View** → **MultiBlock inspector** can also be used to toggle visibility, and change colors and opacities of selected blocks.

## 4.2.8 Advanced: Data Analysis

### Introduction

This tutorial covers data analysis.

### Find Data Panel

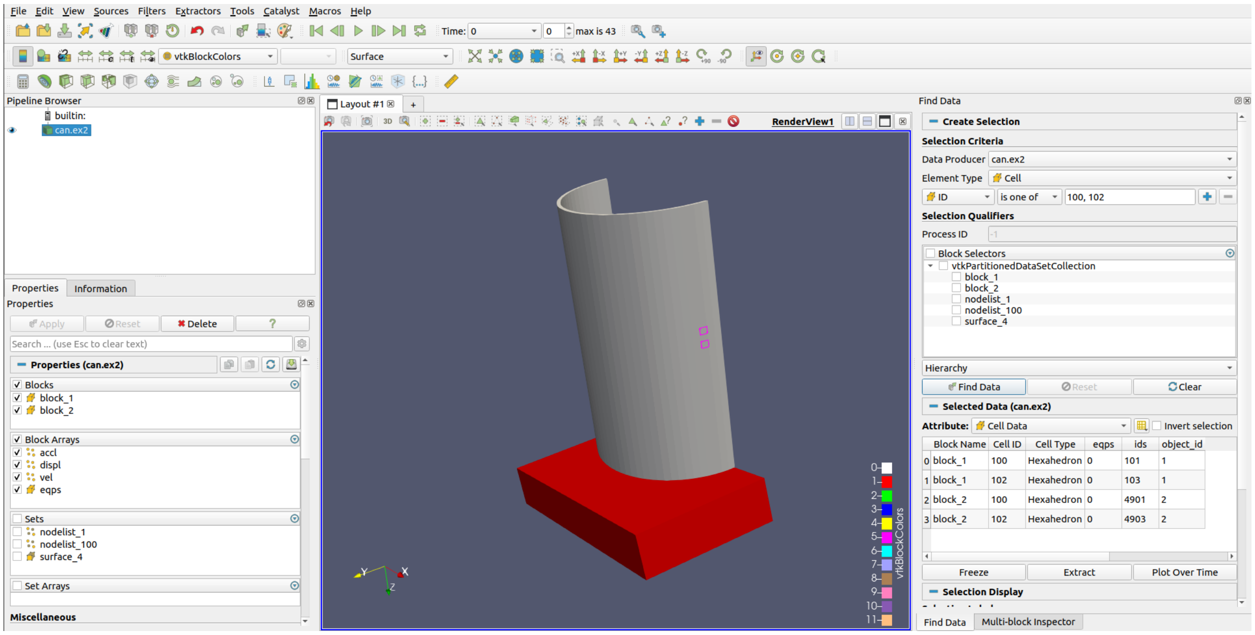
The **Find Data** panel is used to:

- Select points or cells.
- Show the data for these points or cells.
- Control label display on your data in the 3d view.
- Shortcut to run data analysis filters.

We will explore this panel using an example.

- Open a dataset and the Find Data panel.
  - Open can.ex2.
  - **Apply**.
  - **Edit** → **Find Data**. You can also open the **Find Data** panel using the icon

- We will select an **ID**.
  - Set **Data Producer** to can.ex2.
  - Set **Element Type** to **Cells**.
  - Change pulldown to **ID**. Leave as **is**. Enter 100.
  - Only select **block_1**
  - **Find Data**
- We will select two **IDs**.
  - Find **Cells** from **can.ex2**
  - Change pulldown to **ID**. Change to **is one of**. Enter 100, 102.
  - **Find Data**



Find Data

Create Selection

Selection Criteria

Data Producer: can.ex2

Element Type: Cell

ID | is one of | 100, 102

Selection Qualifiers

Process ID: -1

□ Block Selectors
- □ vtkPartitionedDataSetCollection
  - □ block_1
  - □ block_2
  - □ nodelist_1
  - □ nodelist_100
  - □ surface_4

Hierarchy

Find Data | Reset | Clear

Selected Data (can.ex2)

Attribute: Cell Data □ Invert selection

| | Block Name | Cell ID | Cell Type | eqps | ids | object_id |
|---|---|---|---|---|---|---|
| 0 | block_1 | 100 | Hexahedron | 0 | 101 | 1 |
| 1 | block_1 | 102 | Hexahedron | 0 | 103 | 1 |
| 2 | block_2 | 100 | Hexahedron | 0 | 4901 | 2 |
| 3 | block_2 | 102 | Hexahedron | 0 | 4903 | 2 |

Freeze | Extract | Plot Over Time

Selection Display

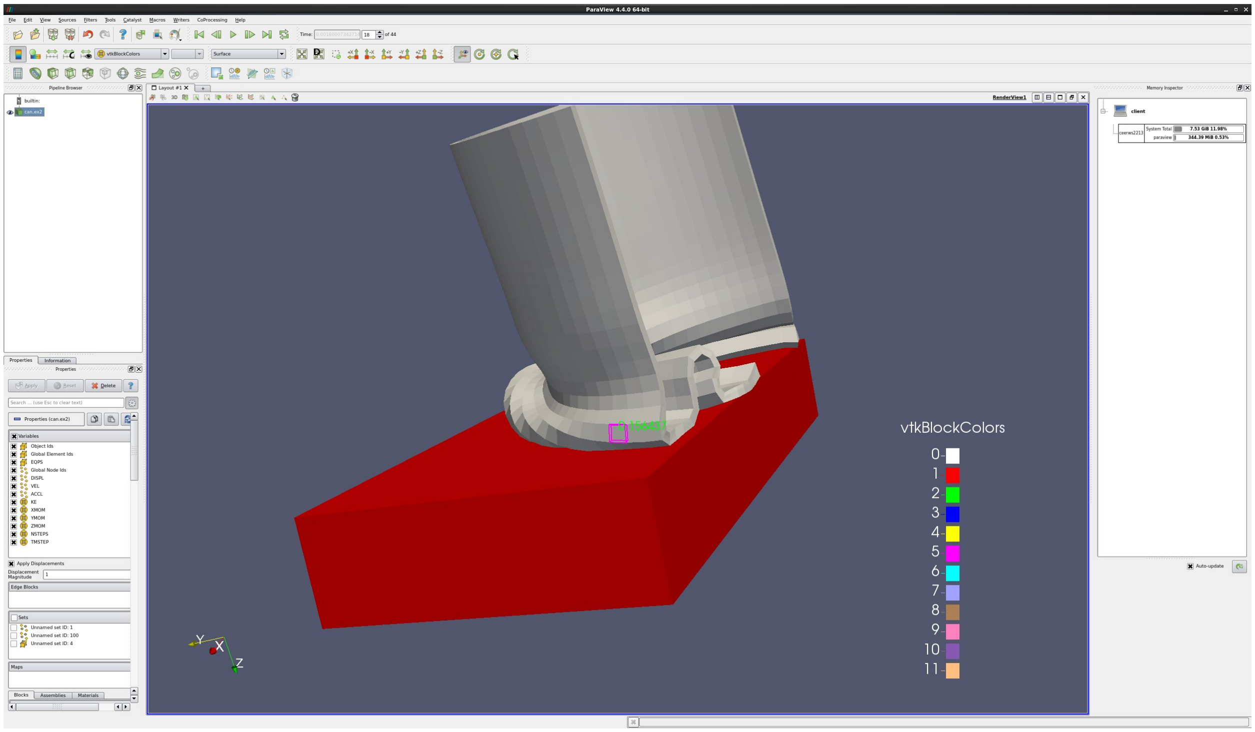
- We will select the maximum EQPS.
  - Move forward one timestep. EQPS for timestep 0 is all the same - 0.
  - Set **Data Producer** to can.ex2.
  - Set **Element Type** to Cells**.
  - Change pulldown to **EQPS**. Change to **is max**.
  - **Find Data**
  - **Play forward**

Notice that the data for the selected cell (or point) is displayed in the spreadsheet.

If desired, the maximum (or minimum) cell can be found, and this selection can be frozen. The same cell will then be selected for all timesteps. Use the **Freeze Selection** button to freeze selections.

We can also display cell or point data on the 3d view. This is done with **Cell Labels** and **Point Labels**.

- Open the **Cell Labels** pulldown, and select EQPS.



## Cell, Point and Block Selections in Render View

`paraview` has a very powerful group of selection tools. Selections are found as a group of icons in the upper left corner of the 3d view.

- More than one selection can be active at a time. Use the **CTRL** key. You can also use the **Add Selection**, **Subtract Selection** and **Toggle Selection** icons.
- Variable data can be printed on the screen for any cell that is selected. See the section on **Find Data**.
- Selections can be used as input to numerous **Data Analysis** filters.
  - **Select Cells On**. Select one or more cells on the surface of your object. Either a single click or rubber band select works. Shortcut - **s**.
  - **Select Points On**. Select one or more points on the surface of your object. Either a single click or rubber band select works. Within a tolerance, the nearest point will be selected. Shortcut - **d**.
  - **Select Cells Through**. Select cells through your dataset. This is known as a Frustum select. Shortcut - **f**.
  - **Select Points Through**. Select points through your dataset. This is also known as a Frustum select. Shortcut - **g**.
  - **Select Cells With Polygon**. Select cells on the surface of your dataset by drawing a polygon. No shortcut.
  - **Select Points With Polygon**. Select points on the surface of your dataset by drawing a polygon. No shortcut.
  - **Select Block**. Select one or more blocks. Either click or rubber band select works. Shortcut - **b**.
  - **Interactive Select Cell Data On. If your dataset is colored by a**

    cell data array of type `idtype`, interactively select all cells in that dataset with the same value as the cell under the cursor. No shortcut.

- **Interactive Select Point Data On. If your dataset is colored by a**
  point data array of type `idtype`, interactively select all points in that dataset with the same value as the point under the cursor. No shortcut.
- **Interactive Select Cells On**. Interactively select cells on the surface of your dataset. If you click on a selection, it will become permanent. You can have numerous permanent selections. No shortcut.
- **Interactive Select Points On**. Interactively select points on the surface of your dataset. If you click on a selection, it will become permanent. You can have numerous permanent selections. No shortcut.
- **Hover Cells On**. This will display all of the data available on this cell. Interactive.
- **Hover Points On**. This will display all of the data available on this point . Interactive.
- **Grow Selection**. This will grow selection to include any cell touching a previously selected cell.
- **Shrink Selection**. This will remove the selection of any cell touching an unselected cell.
- **Clear Selection**.

### Spreadsheet and histogram view

`paraview` can visualize data in a spreadsheet view. The spreadsheet view can be configured to show all data, or only selected data. You can also select a row in the spreadsheet view and it will be selected in the 3d view.

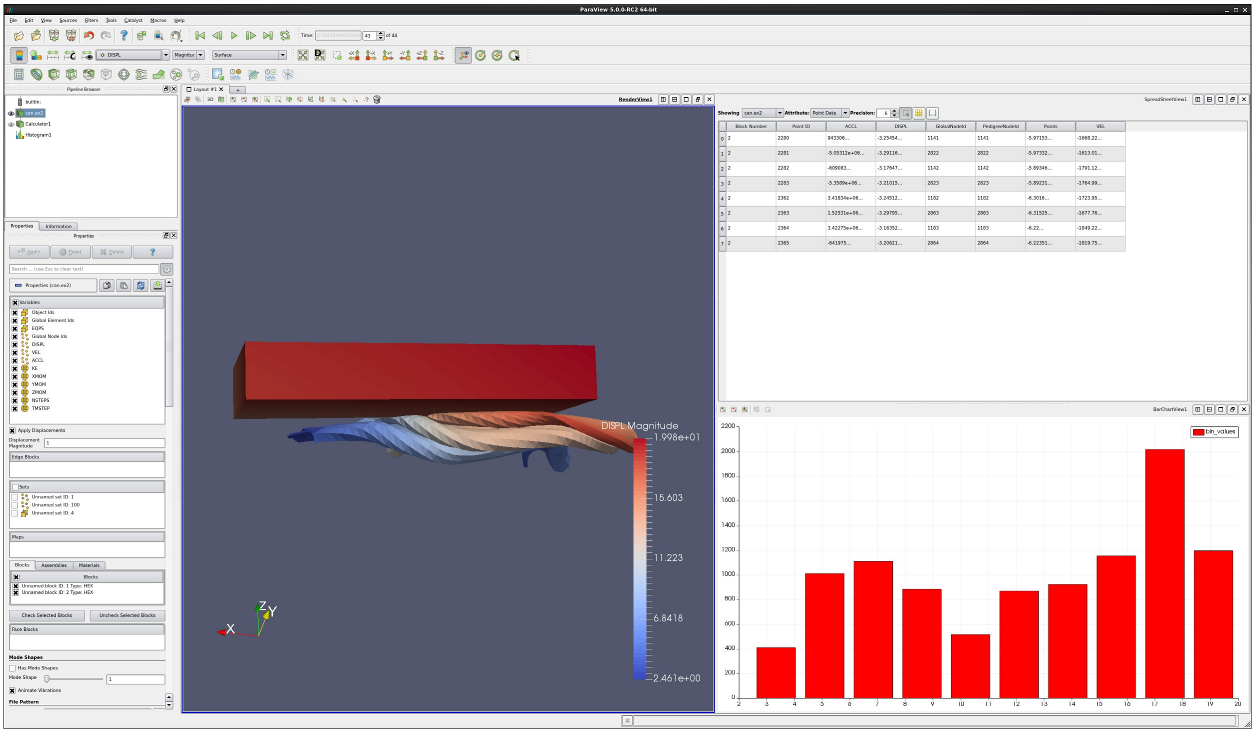
You can also show your data as a histogram. This will show you how many cells have different attributes.

- Open can.ex2.
- **Apply**.
- Highlight the can window, and **Select Cells On** on the can.
- **Split Horizontal**.
- **Split Vertical**.
- In the upper right window, select **Spreadsheet View**.
- Select the **Show Only Selected Elements** check box (next to Precision).
- **Note that the Spreadsheet can now be sorted - including in parallel.**
- Select a different cell on the can.
- Highlight the lower right window.

We want magnitude of the DISPL vector.

- Select **Filters** → **Common** → **Calculator**.
- Set Expression to **mag(DISPL)**. Result Array Name → **MagDispl**.
- **Apply**.
- **Filters** → **Data Analysis** → **Histogram**.
- **Apply**.
- Close the 3d RenderView
- Change **Select Input Array** to **MagDispl**.
- **Apply**.
- Animate forward in time one step at a time.
- **Play**.

- If desired, freeze the X and Y axis in the **Properties** tab using **Axis Use Custom Range**.



Search ... (use Esc to clear text)

Properties (can.ex2)

☒ Variables
☒ Object Ids
☒ Global Element Ids
☒ EQPS
☒ Global Node Ids
☒ DISPL
☒ VEL
☒ ACCL
☒ KE
☒ XMOM
☒ YMOM
☒ ZMOM
☒ NSTEPS
☒ TMSTEP

☒ Apply Displacements

Displacement Magnitude: 1

Edge Blocks

☐ Sets
☐ Unnamed set ID: 1
☐ Unnamed set ID: 100
☐ Unnamed set ID: 4

Maps

Blocks | Assemblies | Materials

☒ Blocks
☒ Unnamed block ID: 1 Type: HEX
☒ Unnamed block ID: 2 Type: HEX

Check Selected Blocks | Uncheck Selected Blocks

Face Blocks

Mode Shapes
☐ Has Mode Shapes
Mode Shape: 1
☒ Animate Vibrations

File Pattern

Showing: can.ex2 Attribute: Point Data Precision: 6

| | Block Number | Point ID | ACCL | DISPL | GlobalNodeId | PedigreeNodeId | Points | VEL |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 2280 | 943306... | -3.25454... | 1141 | 1141 | -5.97153... | -1668.22... |
| 1 | 2 | 2281 | -5.05312e+06... | -3.29116... | 2822 | 2822 | -5.97332... | -1613.01... |
| 2 | 2 | 2282 | -609083... | -3.17647... | 1142 | 1142 | -5.89346... | -1791.12... |
| 3 | 2 | 2283 | -5.3589e+06... | -3.21015... | 2823 | 2823 | -5.89231... | -1764.99... |
| 4 | 2 | 2362 | 3.41834e+06... | -3.24512... | 1182 | 1182 | -6.3016... | -1723.95... |
| 5 | 2 | 2363 | 1.52531e+06... | -3.29795... | 2863 | 2863 | -6.31525... | -1677.76... |
| 6 | 2 | 2364 | 3.42275e+06... | -3.16352... | 1183 | 1183 | -6.22... | -1849.22... |
| 7 | 2 | 2365 | -641975... | -3.20621... | 2864 | 2864 | -6.22351... | -1819.75... |

## Advanced Spreadsheets

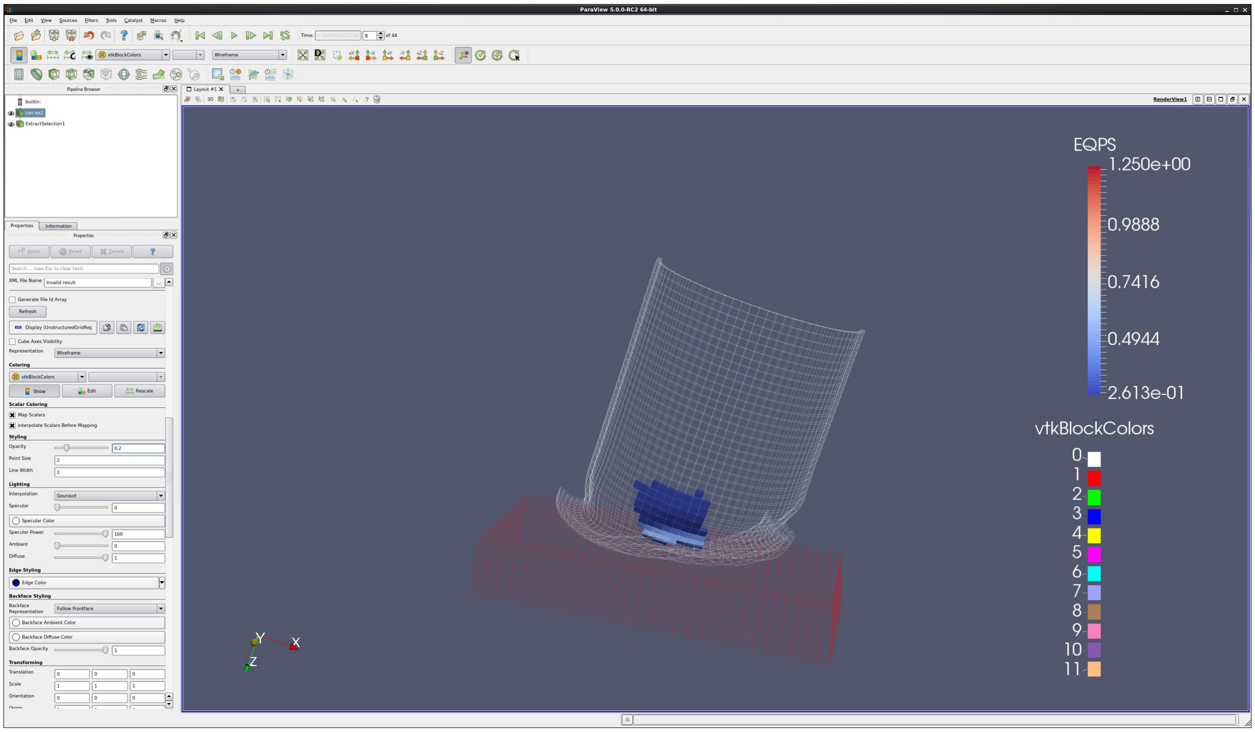
- You can either display all of your dataset's points or cells in your spreadsheet, or only those that are selected in the 3d view. With the **spreadsheet** window selected, in the **Properties** tab, click **Show only selected elements**.
- When cells or points are selected in the 3d RenderView, the appropriate row(s) in the spreadsheet view will be highlighted.
- Highlighting works both ways - if you select a row or rows in the spreadsheet view, the cell or point on your dataset will be selected.
- You can select multiple rows in the spreadsheet by holding down the **CTRL** key, and can select ranges of rows by holding down the **SHIFT** key.
- Note: The Spreadsheet can be sorted.

## Extract Selection

- Open can.ex2.
- Using the **Select Cells Through** icon, select a few dozen cells.
- Open the **Find Data** dialog
- Click **Freeze Selection**. That way, we don't have the can moving out of the frustum over time.
- Click **Extract Selection** in the **Find Data** panel.
- **Apply**.

**Did you know?**

The Extract Selection is also found under **Filters** → **Data Analysis**.



Search ... (use Esc to clear text)

XML File Name: Invalid result

□ Generate File Id Array

Refresh

Display (UnstructuredGridRep

□ Cube Axes Visibility

Representation: Wireframe

**Coloring**

vtkBlockColors

Show | Edit | Rescale

**Scalar Coloring**

☒ Map Scalars

☒ Interpolate Scalars Before Mapping

**Styling**

Opacity: 0.2

Point Size: 2

Line Width: 1

**Lighting**

Interpolation: Gouraud

Specular: 0

○ Specular Color

Specular Power: 100

Ambient: 0

Diffuse: 1

**Edge Styling**

● Edge Color

**Backface Styling**

Backface Representation: Follow Frontface

○ Backface Ambient Color

○ Backface Diffuse Color

Backface Opacity: 1

**Transforming**

Translation: 0 0 0

Scale: 1 1 1

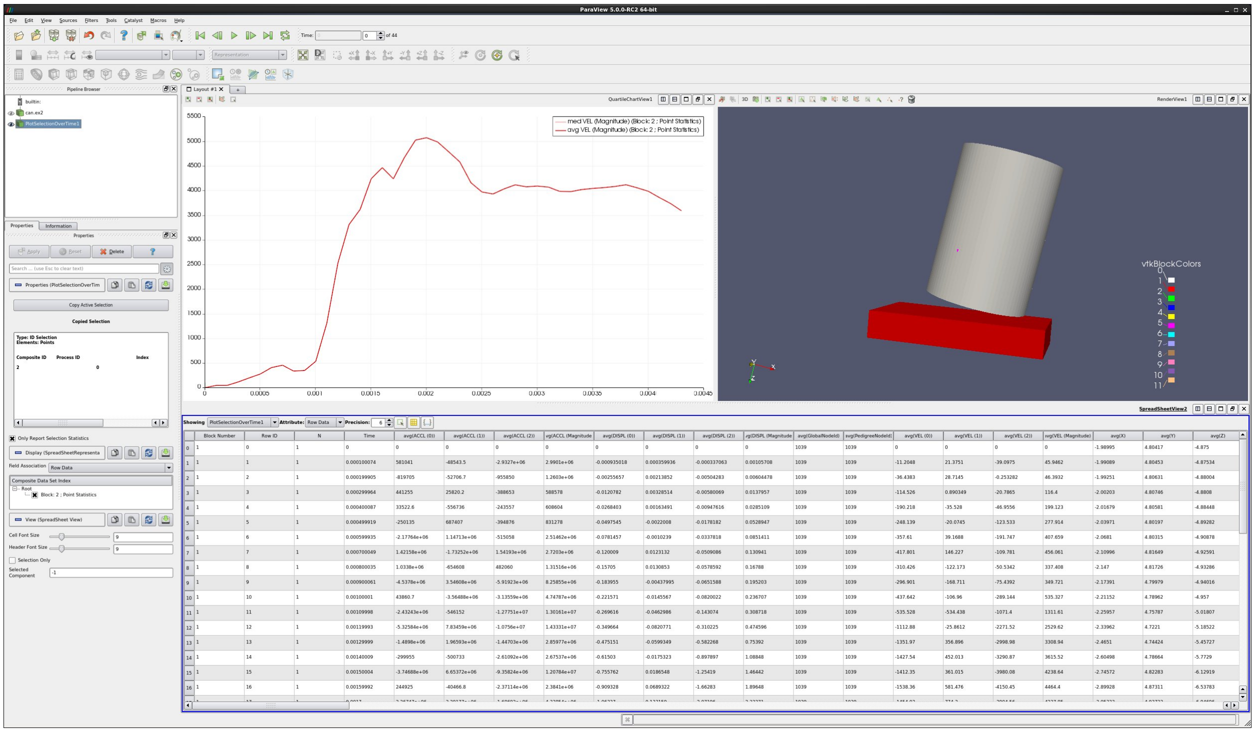
Orientation: 0 0 0

## Spreadsheet data by time

- Open can.ex2.
- On the toolbar, click the **Select Points On** icon. This will allow you to manually select a point on your dataset.
- Select a point on your dataset.
- **Filters** → **Data Analysis** → **Plot Selection Over Time**.
- **Apply**.
- Split Vertical, Spreadsheet View, and with the Spreadsheet View highlighted, turn on the eyeball in front of the **Plot Selection Over Time** filter.

**Did you know?**

You can write spreadsheet data out to a .csv file using **File** → **Export**.



Properties | Information

Apply | Reset | Delete | ?

Search ... (use Esc to clear text)

Properties (PlotSelectionOverTim

Copy Active Selection

**Copied Selection**

**Type: ID Selection**
**Elements: Points**

| Composite ID | Process ID | Index |
|---|---|---|
| 2 | 0 | |

☒ Only Report Selection Statistics

Display (SpreadSheetRepresenta

Field Association: Row Data

Composite Data Set Index
Root
☒ Block: 2 ; Point Statistics

View (SpreadSheet View)

Cell Font Size: 9

Header Font Size: 9

□ Selection Only

Selected Component: -1

**Showing** PlotSelectionOverTime1 **Attribute:** Row Data **Precision:** 6

| | Block Number | Row ID | N | Time | avg(ACCL (0)) | avg(ACCL (1)) | avg(ACCL (2)) | vg(ACCL (Magnitude | avg(DISPL (0)) | avg(DISPL (1)) | avg(DISPL (2)) | vg(DISPL (Magnitude | avg(GlobalNodeId) | avg(PedigreeNodeId) | avg(VEL (0)) | avg(VEL (1)) | avg(VEL (2)) | vg(VEL (Magnitude) | avg(X) | avg(Y) | avg(Z) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1039 | 1039 | 0 | 0 | 0 | 0 | -1.98995 | 4.80417 | -4.875 |
| 1 | 1 | 1 | 1 | 0.000100074 | 581041 | -48543.5 | -2.9327e+06 | 2.9901e+06 | -0.000935018 | 0.000359936 | -0.000337063 | 0.00105708 | 1039 | 1039 | -11.2048 | 21.3751 | -39.0975 | 45.9462 | -1.99089 | 4.80453 | -4.87534 |
| 2 | 1 | 2 | 1 | 0.000199905 | -819705 | -52706.7 | -955850 | 1.2603e+06 | -0.00255657 | 0.00213852 | -0.00504283 | 0.00604478 | 1039 | 1039 | -36.4383 | 28.7145 | -0.253282 | 46.3932 | -1.99251 | 4.80631 | -4.88004 |
| 3 | 1 | 3 | 1 | 0.000299964 | 441255 | 25820.2 | -388653 | 588578 | -0.0120782 | 0.00328514 | -0.00580069 | 0.0137957 | 1039 | 1039 | -114.526 | 0.890349 | -20.7865 | 116.4 | -2.00203 | 4.80746 | -4.8808 |
| 4 | 1 | 4 | 1 | 0.000400087 | 33522.6 | -556736 | -243557 | 608604 | -0.0268403 | 0.00163491 | -0.00947616 | 0.0285109 | 1039 | 1039 | -190.218 | -35.528 | -46.9556 | 199.123 | -2.01679 | 4.80581 | -4.88448 |
| 5 | 1 | 5 | 1 | 0.000499919 | -250135 | 687407 | -394876 | 831278 | -0.0497545 | -0.0022008 | -0.0178182 | 0.0528947 | 1039 | 1039 | -248.139 | -20.0745 | -123.533 | 277.914 | -2.03971 | 4.80197 | -4.89282 |
| 6 | 1 | 6 | 1 | 0.000599935 | -2.17764e+06 | 1.14713e+06 | -515058 | 2.51462e+06 | -0.0781457 | -0.0010239 | -0.0337818 | 0.0851411 | 1039 | 1039 | -357.61 | 39.1688 | -191.747 | 407.659 | -2.0681 | 4.80315 | -4.90878 |
| 7 | 1 | 7 | 1 | 0.000700049 | 1.42158e+06 | -1.73252e+06 | 1.54193e+06 | 2.7203e+06 | -0.120009 | 0.0123132 | -0.0509086 | 0.130941 | 1039 | 1039 | -417.801 | 146.227 | -109.781 | 456.061 | -2.10996 | 4.81649 | -4.92591 |
| 8 | 1 | 8 | 1 | 0.000800035 | 1.0338e+06 | -654608 | 482060 | 1.31516e+06 | -0.15705 | 0.0130853 | -0.0578592 | 0.16788 | 1039 | 1039 | -310.426 | -122.173 | -50.5342 | 337.408 | -2.147 | 4.81726 | -4.93286 |
| 9 | 1 | 9 | 1 | 0.000900061 | -4.5378e+06 | 3.54608e+06 | -5.91923e+06 | 8.25855e+06 | -0.183955 | -0.00437995 | -0.0651588 | 0.195203 | 1039 | 1039 | -296.901 | -168.711 | -75.4392 | 349.721 | -2.17391 | 4.79979 | -4.94016 |
| 10 | 1 | 10 | 1 | 0.00100001 | 43860.7 | -3.56488e+06 | -3.13559e+06 | 4.74787e+06 | -0.221571 | -0.0145567 | -0.0820022 | 0.236707 | 1039 | 1039 | -437.642 | -106.96 | -289.144 | 535.327 | -2.21152 | 4.78962 | -4.957 |
| 11 | 1 | 11 | 1 | 0.00109998 | -2.43243e+06 | -546152 | -1.27751e+07 | 1.30161e+07 | -0.269616 | -0.0462986 | -0.143074 | 0.308718 | 1039 | 1039 | -535.528 | -534.438 | -1071.4 | 1311.61 | -2.25957 | 4.75787 | -5.01807 |
| 12 | 1 | 12 | 1 | 0.00119993 | -5.32584e+06 | 7.83459e+06 | -1.0756e+07 | 1.43331e+07 | -0.349664 | -0.0820771 | -0.310225 | 0.474596 | 1039 | 1039 | -1112.88 | -25.8612 | -2271.52 | 2529.62 | -2.33962 | 4.7221 | -5.18522 |
| 13 | 1 | 13 | 1 | 0.00129999 | -1.4898e+06 | 1.96593e+06 | -1.44703e+06 | 2.85977e+06 | -0.475151 | -0.0599349 | -0.582268 | 0.75392 | 1039 | 1039 | -1351.97 | 356.896 | -2998.98 | 3308.94 | -2.4651 | 4.74424 | -5.45727 |
| 14 | 1 | 14 | 1 | 0.00140009 | -299955 | -500733 | -2.61092e+06 | 2.67537e+06 | -0.61503 | -0.0175323 | -0.897897 | 1.08848 | 1039 | 1039 | -1427.54 | 452.013 | -3290.87 | 3615.52 | -2.60498 | 4.78664 | -5.7729 |
| 15 | 1 | 15 | 1 | 0.00150004 | -3.74688e+06 | 6.65372e+06 | -9.35824e+06 | 1.20784e+07 | -0.755762 | 0.0186548 | -1.25419 | 1.46442 | 1039 | 1039 | -1412.35 | 361.015 | -3980.08 | 4238.64 | -2.74572 | 4.82283 | -6.12919 |
| 16 | 1 | 16 | 1 | 0.00159992 | 244925 | -40466.8 | -2.37114e+06 | 2.3841e+06 | -0.909328 | 0.0689322 | -1.66283 | 1.89648 | 1039 | 1039 | -1538.36 | 581.476 | -4150.45 | 4464.4 | -2.89928 | 4.87311 | -6.53783 |

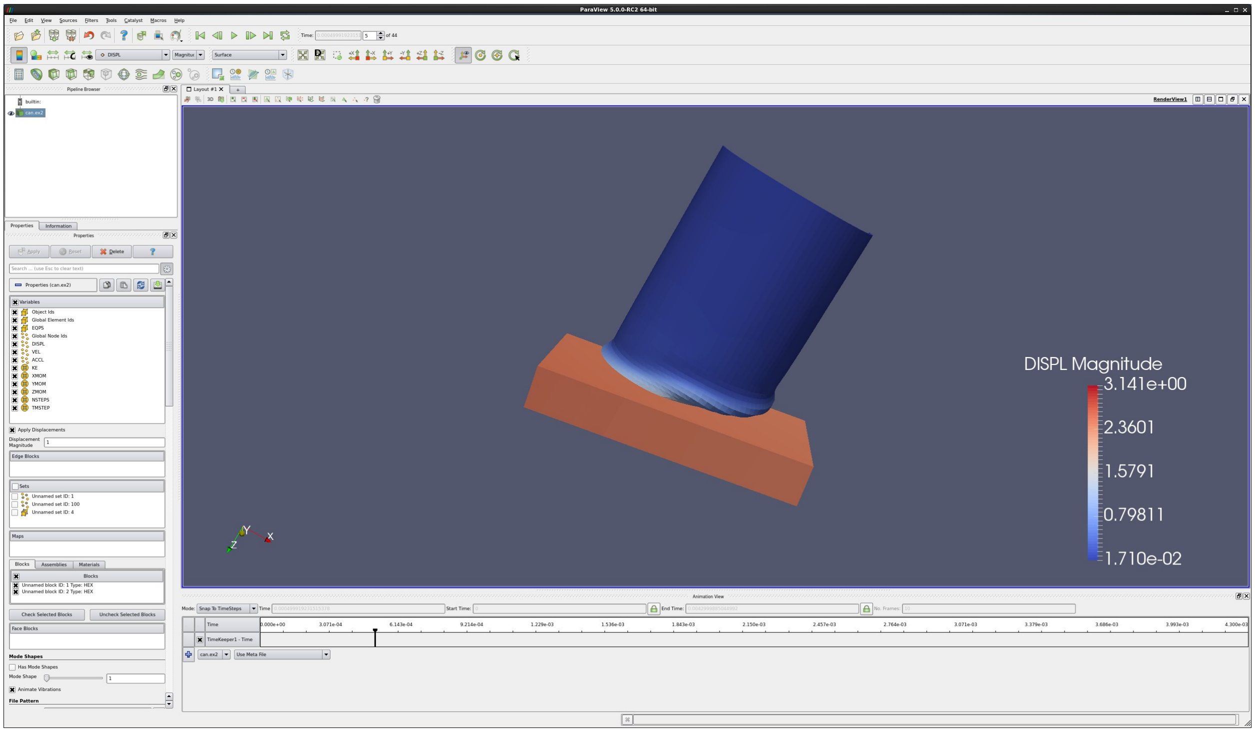
## 4.2.9 Advanced: Animations

### Introduction

This tutorial deals with advanced topics in animations.

### The Animation View

- Open can.ex2.
- **Apply**..
- Drag the can around with the left mouse button.
- Move forward to timestep number 5.
- Set **Coloring** to **DISPL**.
  - Note: If you changed the order of the steps above, you may need to click **Rescale to Data Range**.
- **View** → **Animation View**.
- Grab the slider bar and move it back and forth.



### Real Time

- **Note this functionality is deprecated**
- In the **Animation View** window, change the mode to **Real Time**, and the Duration to 100.
- **Play**. Notice that the can motion is now very slow. We are saying that we want the whole animation to last 100 seconds.
- Note: This can also be used to speed up datasets with a large number of time steps. Set the Duration to 10, and `paraview` will animate over all time in 10 seconds. Obviously, this depends on the whether `paraview` can keep up with this frame rate!

### Temporal Interpolator

- In the Animation View window, change the mode to Sequence, and the Nunber of Frames to 200.
- **Filters** → **Temporal** → **Temporal Interpolator**.
- **Play**. Notice that the can motion is now smooth. `paraview` is interpolating between frames, and making 200 time steps.
  - Note that this only works with data that has a constant mesh through the whole time sequence. AMR (Adaptive Mesh Refinement) data does not work with the Temporal Interpolator.

### Move the camera

There are four ways to move the camera. they are:

- Orbit
- Follow Data
- Interpolate Camera Locations
- Follow Path

### Orbit Camera

We are going to make the camera rotate around our data.

- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- **+Y**.
- **View** → **Animation View**.
- Add a camera. Default motion is Orbit.
- Click the blue **+**.
- A **Create Orbit** dialog will appear.
- Take the defaults.
- **OK**.
- **Play**.

### Camera Follow Data

We are going to make the camera follow the data.

- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- **+Y**.
- **View** → **Animation View**.
- Add a camera. Change **Orbit** to **Follow Data**. Click the blue **+**.
- **Play**.

**Did you know?**

The follow data option will follow the data from whatever filter is highlighted. This means that you can choose one cell, run the **Extract Selection** filter, and follow this cell. Note that you must keep visibility on for this cell. If needed, you can turn the cell's Opacity in the **Properties** tab to 1%, making it disappear. By turning visibility on for your whole dataset, you can follow the cell but display all of your data.

### Interpolate Camera Location

We are going to move the camera along a straight line. We want to move the camera to follow the can.

- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- **+Y**.
- **View** → **Animation View**.
- Add a camera. Change **Orbit** to **Interpolate Camera Position**.
- Click the blue **+**.
- Left click on the white section of the camera row that just appeared.
- An **Animation Keyframes** dialog will open.
- Left click the top **Position**.
- Use Current.
- **OK**
- **Play** to the last timestep.
- Move the can dataset back into the center of the screen. **Tip** - Don't hit **Reset**.
- Left click the bottom **Position**.
- Use Current.
- **OK**.
- **OK**.
- **Play**.

You can also create an intermediate destination for the can by going to the Animation Keyframes dialog, and selecting New. Then, follow the directions above. Experiment by adding more keyframes and different camera angles.

A way to create tracks in 3d space for use with Interpolate Camera Location is to create the points external to Interpolate Camera Location, and then copy them into place. One way to create these external points is to use a Sources/ Plane. Orient the plane in the 2d plane you want the camera to track. Now, create a Spline source. Create additional control points, and selecting each control point, use the **p** key to place them on the plane. Copy these points into the Interpolate Camera Location controls.

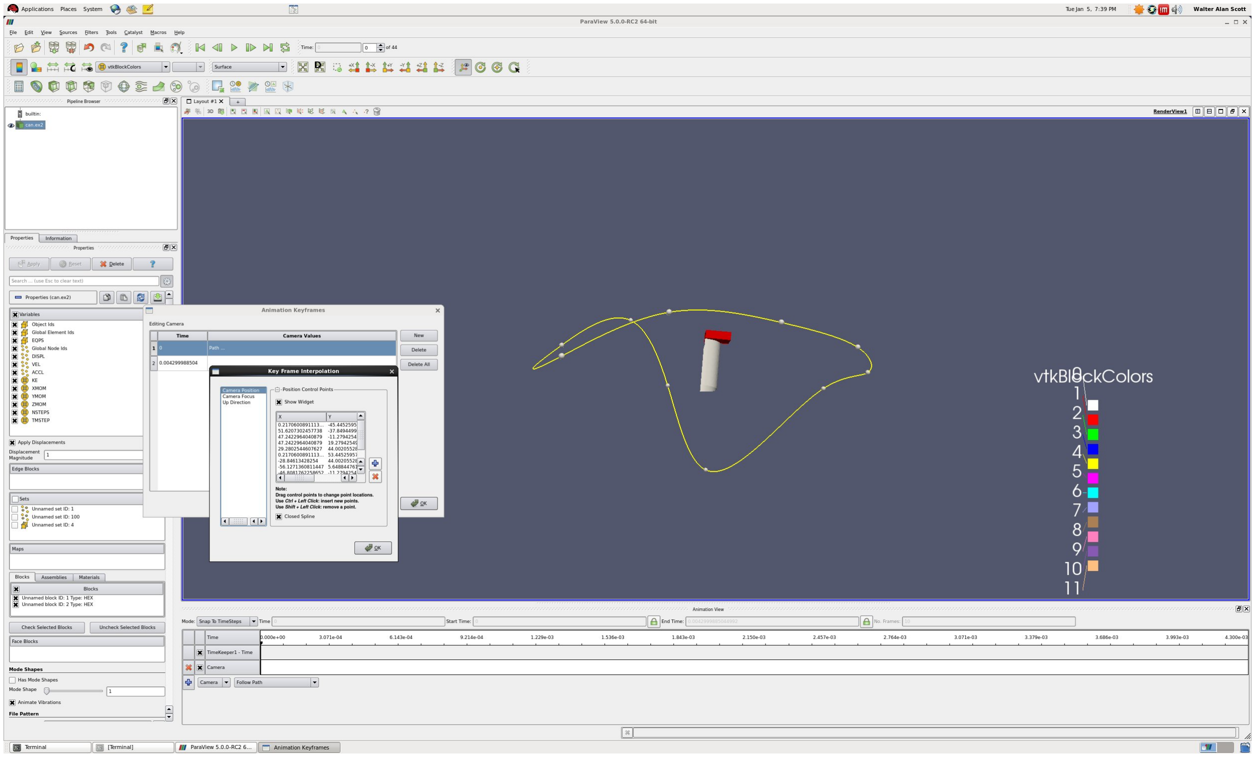
### Follow Path

We are going to move the camera along a spline. Advanced operation.

- **Edit** → **Reset Session**.
- Open can.ex2.
- **Apply**.
- **+Y**.
- **View** → **Animation View**.
- Add a camera. Change **Orbit** to **Follow Path**.

- Click the blue **+**.
- Left click on the white section of the camera row that just appeared.
- An **Animation Keyframes** dialog will open.
- Left click on **Path**.
- Click on **Camera Position**.
- Now, in the 3d view, zoom out.
- Rotate the can. You will see a yellow circle with white spheres.
- Drag the spheres around.
- **OK**.
- **OK**.
- **Play**.

One way to create tracks in 3d space for **Follow Path** is to use the same procedure listed above for **Interpolate Camera Location**.



Animation Keyframes

Editing Camera

| | Time | Camera Values |
|---|---|---|
| 1 | 0 | Path ... |
| 2 | 0.004299988504 | |

Key Frame Interpolation

Camera Position
Camera Focus
Up Direction

Position Control Points

☒ Show Widget

| X | Y |
|---|---|
| 0.2170600891113... | -45.4452595 |
| 51.6207302457738 | -37.8494499 |
| 47.2422964040879 | -11.2794254 |
| 47.2422964040879 | 19.2794254 |
| 29.2802544607627 | 44.0020552 |
| 0.2170600891113... | 53.4452595 |
| -28.84613428254 | 44.0020552 |
| -56.1271360811447 | 5.64884476 |
| -46.8081762258652 | -11.2794254 |

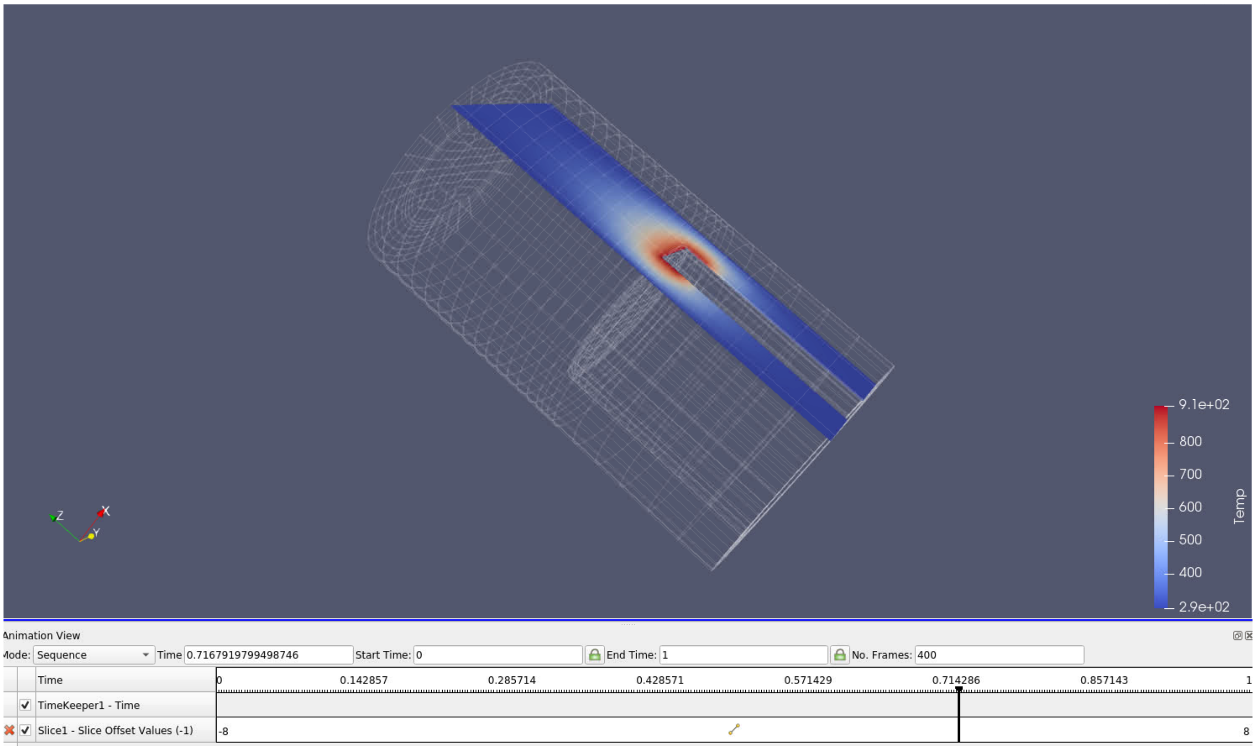
Note:
Drag control points to change point locations.
Use *Ctrl + Left Click*: insert new points.
Use *Shift + Left Click*: remove a point.

☒ Closed Spline

### Animating a Filter

In the previous example we showed how to manipulate the camera using the Animation View tools. In this example we will show how to animate a filter. Our goal is to move a slice through our dataset over time.

- Lets start from scratch. One way is to go **Edit** → **Reset Session**.
- **Ok**.
- If you closed it, bring up the **Animation View**.
- Open disk_out_ref.ex2.
- **Apply**.
- Select **Slice** filter.
- **Apply**.
- Turn off the **Show Plane**.
- In the **Animation View**, change Mode to Sequence, and number of frames to 400.
- We want to create a **Slice** track.
- Slice Offset Values.
- Click the blue **+**.
- Double click on the track. This will bring up a dialog, and will set the start and end.
- Change the starting value to **-8** and the ending value to **8**.
- **Play**.



Animation View

Mode: Sequence Time 0.7167919799498746 Start Time: 0 End Time: 1 No. Frames: 400

| | | | |
|---|---|---|---|
| | Time | 0 0.142857 0.285714 0.428571 0.571429 0.714286 0.857143 1 | |
| ☑ | TimeKeeper1 - Time | | |
| ☑ | Slice1 - Slice Offset Values (-1) | -8 | 8 |

### Data Animation

In this example we will show how to animate your dataset. Our goal is to show one dataset, then fade into another dataset. This can be handy when one physics simulation runs for an early time period, and another physics simulation runs for the later time period.

- Lets start from scratch. One way is to go **Edit** → **Reset Session**.
- **OK**.
- If you closed it, bring up the **Animation View**.
- Open can.ex2.
- **Apply**.
- Open can.ex2 again.
- **Apply**.
- Select the upper can.ex2.
- Set **Coloring** to **DISPL**.
- Go the last time step.
- Rescale to Data Range.
- Go to first time step.
- Select the lower can.ex2.
- Change the representation to **Wireframe**.

We now want to fade from the first version of the can to the second version of the can. This is done as follows:

- On the **Animation View**, on the can.ex2 pulldown menu, select the upper can.ex2.
- Right of there, use the pulldown menu to select Opacity.
- Click the blue **+**.
- Do the same for the lower can.ex2.
- Click on the upper can.ex2 white horizontal bar.
- Double click on the upper value, change it to 1.
- Double click the lower value, change it to 0.
- **OK**.
- **Play**.

You can substitute Visibility for Opacity when you add tracks to the Animation View. Then, on one dataset, you can run visibility of 1 for half of your time, and run visibility of 1 for the other dataset for the second half of your time. Thus, you will show the first simulation for the first half of your animation, and the second simulation for the second half.

### Warp Vector Filter

- If your dataset has displacement data, but it is not using a variable name that ParaView recognizes, you can still animate your data. Choose the **Filters** → **Alphabetical** → **Warp Vector filter**.

## 4.2.10 Advanced: State Management

### Introduction

This tutorial covers different types of state management.

### Save State / Load State

**ParaView** state files are saved by selecting the menu option **File** → **Save State**. These files can be saved as .pvsm files (xml) or .py files (Python). The pipeline, orientation of the dataset, and all view windows are saved. Select **File** → **Load State** to open a saved **ParaView** state file.

### Save Data

This feature will save the data output from the selected source/filter. The **Save Data** feature is selected from the main menu: **File** → **Save Data**. The data type is whatever you chose when you save the data. If a data type is missing, it probably means you are trying to output data that is not supported by this dataset format. For example, you need to have surface data, as triangles, to output .stl files.

**Save Data** can be used to:

- Create .csv files of your data which can be read into spreadsheets and other programs.
- Create .stl files. which are often used for additive manufacturing, or animation programs.
  - Run filters **Extract Surface** then **Triangulate** before saving Data.
- Create files Houdini can ingest. These are .geo files.
  - Again, run filters **Extract Surface** then **Triangulate** before saving Data.

### Export Scene

This feature will save whatever is in the viewport to a file. The Export Scene feature is selected from the main menu: File → Export Scene. Again, the data type is whatever you chose when you export scene.

**Export Scene** can be used to:

- Create .webgl files of what is displayed in the viewport. A .html file will be written out, which you can load in a web browser. This format allows users to create interactive data products that anyone can load in a web browser.
- Create .pdf files of what is displayed in the viewport.

### Traces

`paraview` can now automatically save traces, or macros. This means that you can start recording a trace, do something, stop recording a trace, and save this trace into a file. You can then use this trace file as input to the `pvbatch` program, or as a macro within `paraview`.

Example:

- **Tools** → **Python Shell** → **Trace** → **Start Trace**.
- Open disk_out_ref.ex2.
- **Apply**.
- Select **Clip**.
- Use **Z normal**.
- Unselect **Show plane**.
- **Apply**.
- Select **Slice**.
- Unselect **Show plane**.
- **Apply**.
- Set **Coloring** to **Temp**.
- In the Python Shell window, **Stop Trace**.
- **Save trace**.
- **Save as macro**.
- Call it **Super-disk**.
- Close the editor.

Now, lets run the trace.

- On the Macros toolbar, you will see **Super-disk**. Click it.

## 4.2.11 Advanced: Tips & Tricks

### Introduction

This section holds tips and tricks that don't fit anywhere else, or are small enough that they don't deserve their own tutorial.

### Creating a Custom Filter

- Open disk_out_ref.ex2.
- **Apply**.
- In the **Properties** tab, Set **Coloring** to **Temp**.
- Select **Clip** filter.
- Set **Clip type** to **Scalar**. (The Scalars and Values don't matter right now)
- **Apply**.

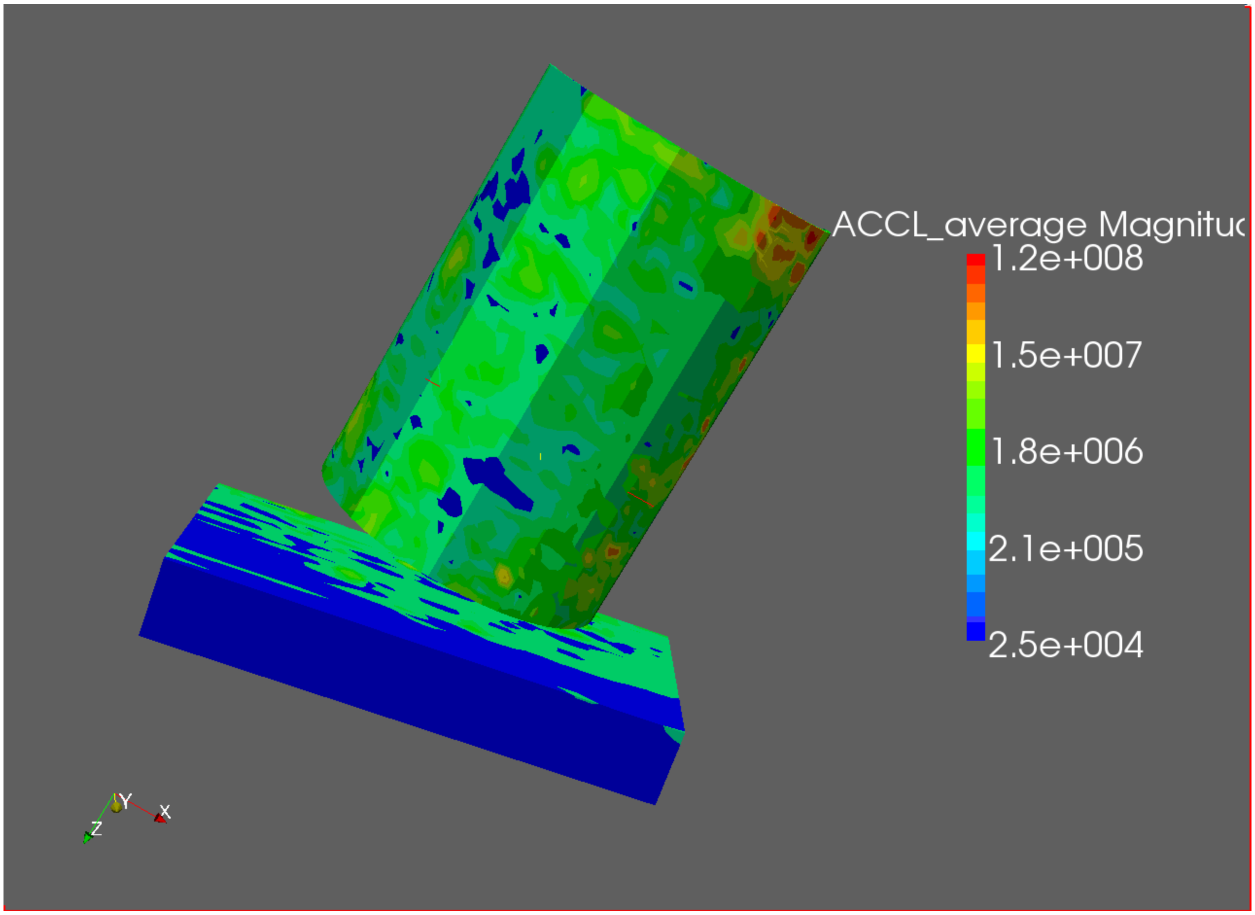
- Select **Tools** → **Create Custom Filter**.
- Name this filter ClipByScalar (IsoVolume).
- Take the default inputs.
- Take the default outputs.
- Select the clip (in the left window), and pull down the pull down menu for Property.
- Select Scalars.
- Hit the blue **+** sign.
- Do the same for **Value** and **Inside Out**.
- Finish.
- Now, delete the **Clip** filter in the pipeline browser.
- Select **Filters** → **Alphabetical** → **ClipByScalar** (sometimes incorrectly know as an IsoVolume filter).
- Turn Scalars to Temp
- Enter a Value of 400.
- **Apply**.

### Temporal Statistics Filter

- Open can.ex2.
- **Apply**.
- **Filters** → **Temporal** → **Temporal Statistics**.
- **Apply**.
- Set **Coloring** to **ACCL_average**.
- Set **Coloring** to **ACCL_maximum**.
- Set **Coloring** to **DISPL_average**.
- You can now visually see average acceleration, maximum acceleration and average displacement of each cell.
- To see the ranges of these variables over the whole mesh, look in the Information tab.

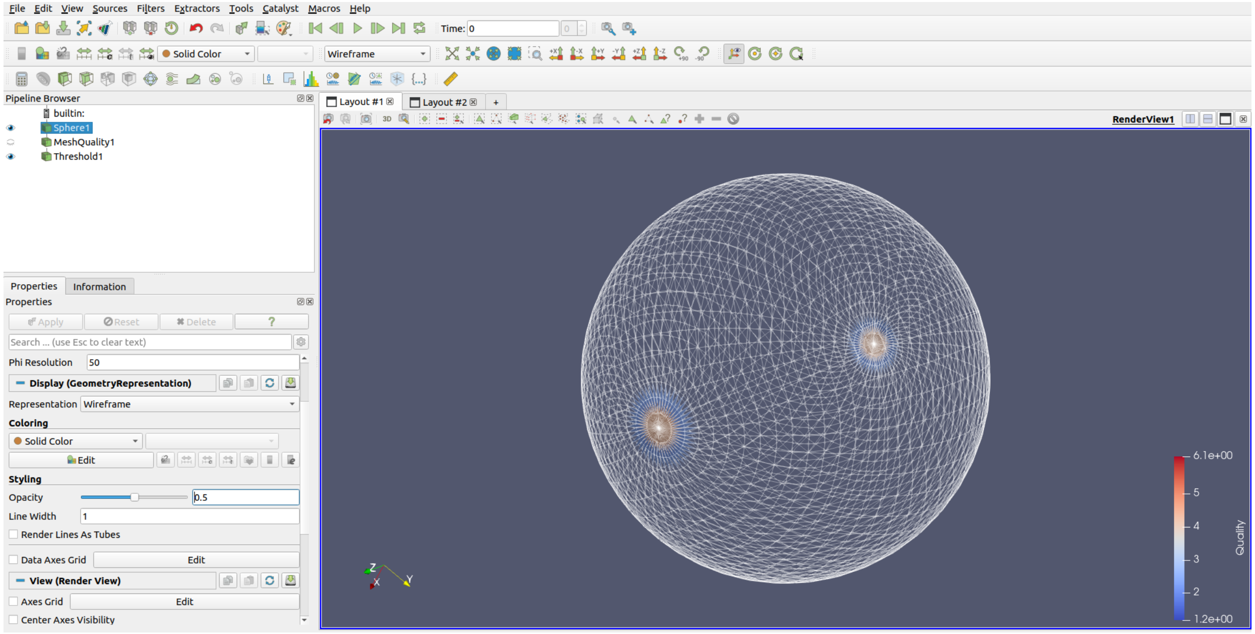
### Creating vectors from 2 or 3 scalars

- Open can.ex2.
- **Apply**.
- Select **Filters** → **Common** → **Calculator**.
- Build the vector equation using the iHat, jHat and kHat buttons on the calculator. For instance, this example will create a vector representing the acceleration in the X and Y plane.
- Set **Result Array** Name to **ACCL-XY**.
- Set **Expression** to **ACCL_X*iHat+ACCL_Y*jHat**.
- Set **Coloring** to **ACCL-XY**.


ACCL_average Magnituc
1.2e+008
1.5e+007
1.8e+006
2.1e+005
2.5e+004
Y
X
Z

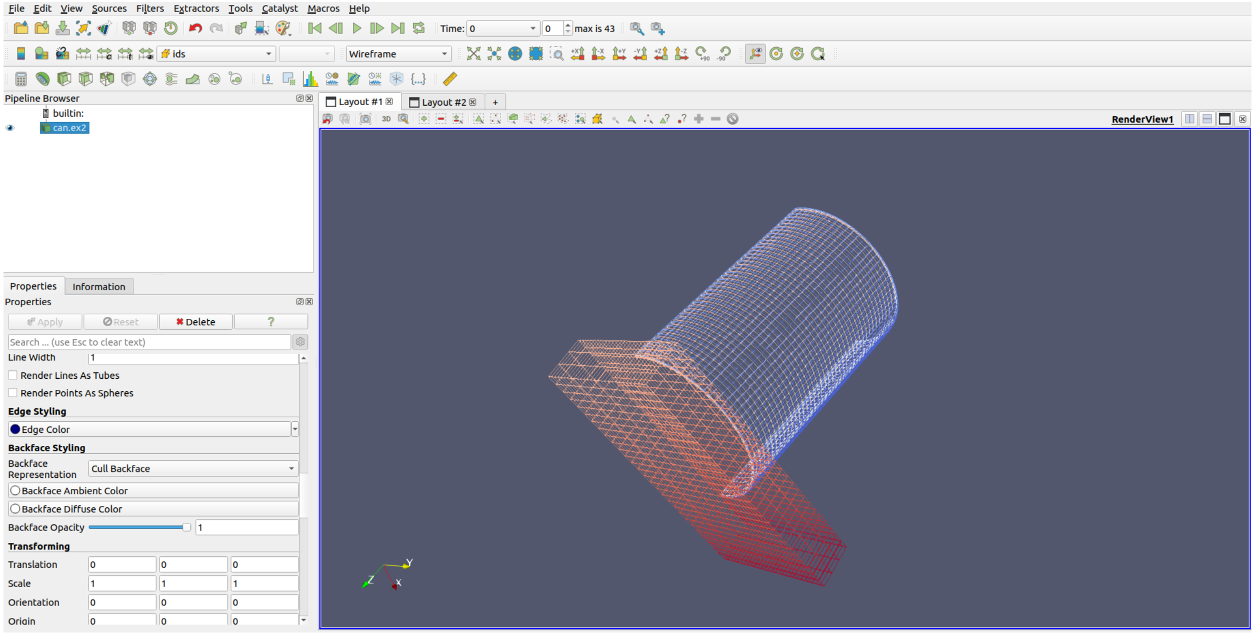
### Mesh quality

- Select **Sources** → **Alphabetical** → **Sphere**.
- In the **Properties** tab, set **Theta Resolution** and **Phi Resolution** to 50.
- **Apply**.
- Select **Filters** → **Alphabetical** → **Mesh Quality**. and use defaults.
- **Apply**.
- Next, we want to only look at those cells that are below some threshold of quality.
- **Filters** → **Common** → **Threshold**.
- Choose Scalars of "Quality", and **Lower Threshold** of 2.3 and **Upper Threshold** of 10.
- Turn visibility of **Sphere1** on.
- Set **Representation** → **Wireframe**.
- Set **Opacity** to 0.5.


Time: 0
Solid Color
Wireframe
Pipeline Browser
builtin:
Sphere1
MeshQuality1
Threshold1
Phi Resolution 50
Display (GeometryRepresentation)
Representation Wireframe
Coloring
Solid Color
Styling
Opacity 0.5
Line Width 1
☐ Render Lines As Tubes
☐ Data Axes Grid
View (Render View)
☐ Axes Grid
☐ Center Axes Visibility
File Edit View Sources Filters Extractors Tools Catalyst Macros Help
Properties
Information
Apply
Reset
Delete
?
Search ... (use Esc to clear text)
Edit
Layout #1
Layout #2
RenderView1
6.1e+00
5
4
3
2
1.2e+00
Quality
Z
X
Y

### Backface styling

It is possible to change the backface style of a wire frame object. - Open can.ex2. - **Apply**. - Set **Coloring** to **ids**. - Set **Representation** → **Wireframe**. - In the Properties tab, Click Advanced. - Slide down a few pages until you find **Backface Styling**. - Set **Backface Representation** → **Cull Backface**.



Line Width 1

□ Render Lines As Tubes

□ Render Points As Spheres

● Edge Color

Backface Representation: Cull Backface

○ Backface Ambient Color

○ Backface Diffuse Color

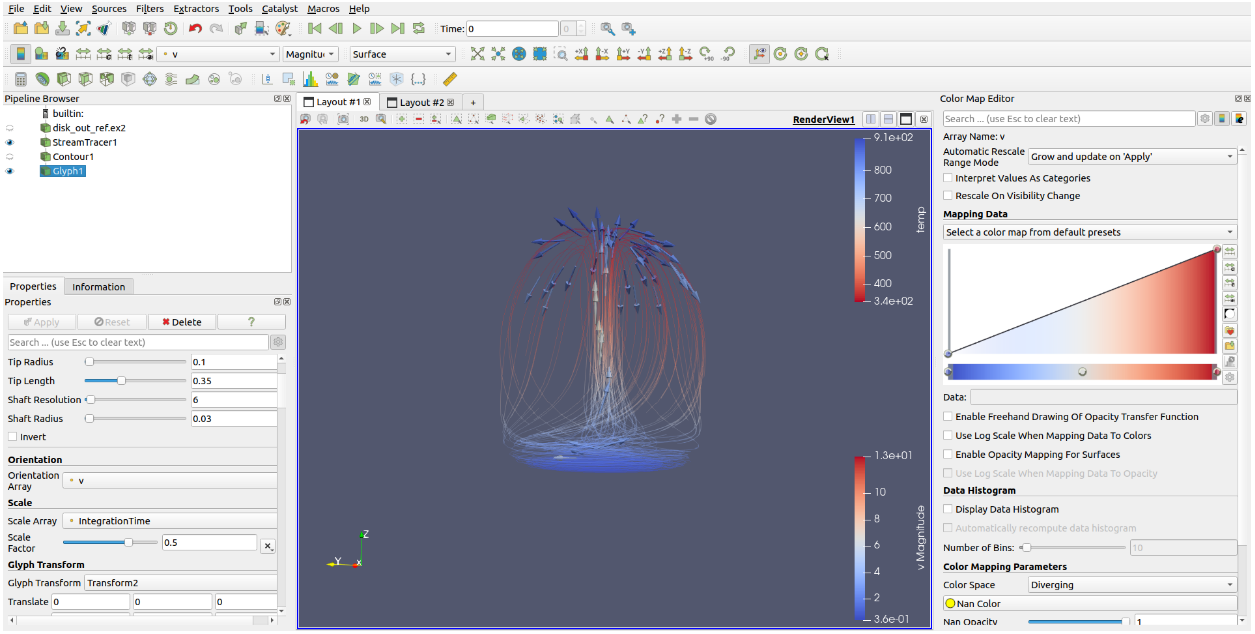
Backface Opacity 1

| | | | |
|---|---|---|---|
| Translation | 0 | 0 | 0 |
| Scale | 1 | 1 | 1 |
| Orientation | 0 | 0 | 0 |
| Origin | 0 | 0 | 0 |

## Animating a static vector field

- If you have a vector field in your data, you can animate a static dataset.
- Our goal is to create a set of streamlines from a vector field, place points on this set of streamlines, and animate the point down the streamlines. We will also add glyphs to the streamline points.
- Open disk_out_ref.ex2.
- **Apply**.
- Click **-X**.
- Select **Filters** → **Common** → **Stream tracer**. (We are already streamtracing on V).
- Set **Seed Type** to Point Cloud.
- Unckeck **Show sphere**.
- Set **Opacity** to 0.3.
- **Apply**.
- Open **View** → **Color Map Editor**.
- Click **Invert the transfer functions**.
- Select **Filters** → **Common** → **Contour**.
- Contour on **IntegrationTime**.
- **Apply**.
- Select **Filters** → **Common** → **Glyph**.
- Set **Orientation Array** to **v**.
- Set **Glyph Mode** to **All Points**.
- Set **Scale factor** to 0.5.
- Set **Coloring** to **v**.

- **Apply**.
- In the Pipeline browser, hide **Contour1**, and show **SteamTracer1** and **Contour1**.
- **View** → **Animation View**.
- Set **Mode** to **Sequence**.
- Set **No. Frames** to **100**.
- Change the pulldown box next to the blue **+** to be **Contour1**.
- Click the blue **+**.
- **Play**.



v
Magnitude
Surface

Properties

Search ... (use Esc to clear text)

Tip Radius 0.1

Tip Length 0.35

Shaft Resolution 6

Shaft Radius 0.03

□ Invert

**Orientation**

Orientation Array v

**Scale**

Scale Array IntegrationTime

Scale Factor 0.5

**Glyph Transform**

Glyph Transform Transform2

Translate 0 0 0

Search ... (use Esc to clear text)

Array Name: v

Automatic Rescale Range Mode Grow and update on 'Apply'

□ Interpret Values As Categories

□ Rescale On Visibility Change

**Mapping Data**

Select a color map from default presets

Data:

□ Enable Freehand Drawing Of Opacity Transfer Function

□ Use Log Scale When Mapping Data To Colors

□ Enable Opacity Mapping For Surfaces

□ Use Log Scale When Mapping Data To Opacity

**Data Histogram**

□ Display Data Histogram

□ Automatically recompute data histogram

Number of Bins: 10

**Color Mapping Parameters**

Color Space Diverging

Nan Color

Nan Opacity 1

## 4.2.12 Python & Batch: ParaView & Python

### Introduction

**ParaView** offers a rich and powerful Python interface. This allows users to automate processing of their data, and gives access to powerful tools in the Visualization Tool Kit (VTK). This tutorial will describe **ParaView** and Python. It shows a user how to drive **ParaView** using Python commands, and how to automate the creation and use of these commands.

### Overview

**ParaView** has a client / server architecture. The client includes the **ParaView** GUI and display. The server reads the user's data, processes the data, and passes these images to the client. We can use Python to control ParaView either in the GUI, at the client level, or directly on the server.

### A simple Python toy example within ParaView

- Start `paraview`.
- Start the Python Interpreter **Tools** → **Python Shell**
- Notes:
  - You can copy commands from elsewhere and paste them into the Python Shell.
  - Python is case sensitive. Be sure to use correct capitalization as shown below.
  - Python is indent sensitive. Be sure to not indent, as shown below.
- Lets create and display a sphere, by typing the following commands into the Python Shell

  ```
  sphere=Sphere()
  Show()
  Render()
  ```

  We have now created a sphere in the pipeline, turned on it's visibility, and re-rendered.

- Lets add a shrink filter. We hide the sphere, add the shrink filter, and re-render.

  ```
  Hide()
  Render()
  shrink = Shrink()
  Show()
  Render()
  ```

  ParaView will allow us to use either the GUI controls or Python. For instance:

  - Select the Sphere in the pipeline browser, by typing the following in the Python Shell:

    ```
    clip = Clip()
    Show()
    Render()
    ```

  - Or, we could continue in the Python as follows:

    ```
    clip = Clip()
    Hide(shrink)
    Show(clip)
    Render()
    ```

  Hide the Plane widget:

  ```
  Hide3DWidgets(proxy=clip)
  ```

**Did you know?**

- To see all commands available in ParaView:

  ```
  dir()
  ```

- To see all of the options for the Clip creator:

  ```
  dir(Clip)
  ```

- To see all of the options for the instance of the clip we created above:

```
dir(clip)
```

- A better tool to see the available commands for an item in the pipeline is ListProperties, such as:

```
clip.ListProperties()
```

  - Note that this doesn't work on instantiated controls, such as the camera. Use dir() for controls such as camera.

- And, to see the different properties of the ClipType variable, use

```
clip.ClipType.ListProperties()
```

- To see lots of detail on an instance of a command, create the instance and ask for help on that instance..

```
help(clip)
```

---

Lets look at, and change, something

- Print the Theta Resolution

```
print(sphere.ThetaResolution)
```

- Change it to 64

```
sphere.ThetaResolution=64
Show()
Render()
```

- Lets change the selected filter in the Pipeline Browser:

```
SetActiveSource(sphere)
```

- Lets delete the clip

```
Delete(clip)
```

- Lets add a filter to the sphere, without selecting it first

```
wireframe = ExtractEdges(Input=sphere)
Show()
Render()
```

### A simple Python example reading a datafile and writing a screenshot

- Within the ParaView GUI, **Edit** → **Reset Session**
- Start the Python Interpreter **Tools** → **Python Shell**

### Read in data, use a filter and save a screenshot

- Lets read in can.ex2, clip can.ex2, paint can.ex2 and save a screenshot. We use this template.

  ```
  canex2 = OpenDataFile('D:/directoryName/can.ex2')
  ```

  Here is the current path. Be sure to update for version number

  ```
  canex2 = OpenDataFile('C:/Program Files (x86)/ParaView x.x.x/data/can.ex2')
  clip = Clip()
  Hide(canex2)
  Show(clip)
  ResetCamera()
  Render()
  SaveScreenshot('D:/directoryName/picture.jpg')
  ```

See more about file readers in Section 1.2 of the User's Guide.

### Control the camera

- We want to move the camera. First, get the camera and reset the camera to a known good position.

  ```
  camera=GetActiveCamera()
  camera.SetFocalPoint(0,0,0)
  camera.SetPosition(0,0,-10)
  camera.SetViewUp(0,1,0)
  ```

- How to move the camera closer or further away

  ```
  camera.Dolly(10)
  Render()
  camera.Dolly(.1)
  Render()
  ```

- How to rotate the camera around the view direction 45 degrees, centered on the dataset. After the reset above, rotate around the X axis.

  ```
  camera.Roll(45)
  Render()
  ```

- How to rotate the camera around the vector up, centered on the Y axis. After the reset above, rotate around the Y axis.

  ```
  camera.Yaw(45)
  Render()
  ```

- How to rotate the camera vertically around the camera point

  ```
  camera.Pitch(45)
  Render()
  ```

- How to rotate the camera around the vector up, centered on the dataset. After the reset above, rotate around the Y axis.

```
camera.Azimuth(45)
Render()
```

- How to rotate the camera around the X axis, centered on the dataset. After the reset above, rotate around the Y axis.

```
camera.Elevation(45)
Render()
```

- How to reset the camera

```
ResetCamera()
```

- Available commands are found using:

```
dir(camera) (after you have created the animationScene1
variable)
dir(GetActiveCamera())
```

### Control time

- We want to move forward one timestep, so min and max are set correctly for a variable

```
animationScene1 = GetAnimationScene()
animationScene1.GoToNext()
```

- Playing through all time is done with the following command.

```
animationScene1.Play()
```

- We need to acquire the available timesteps:

```
tk = GetTimeKeeper()
timesteps = tk.TimestepValues
```

- First timestep is found using either of these methods:

```
animationScene1.GoToFirst()
animationScene1.AnimationTime = timesteps[0]
```

- Last timestep is found using either of these methods:

```
animationScene1.GoToLast()
animationScene1.AnimationTime = timesteps[-1]
```

- Moving to a specific timestep (such as timestep 10) is done as follows:

```
# index starts with 0
animationScene1.AnimationTime = timesteps[9]
```

- To find out how many timesteps we have, you use the len command. Continued on from above.

```
numTimesteps = len(timesteps)
```

- Available commands are found using:

```
# after you have created the animationScene1 variable
dir(animationScene1)
dir(GetAnimationScene())
```

### Paint by a variable

- We want to color by the variable. Be sure to Show the Clip, and not the Can. Steps are, move forward one timestep, get the renderview, get the display, get the variables, ColorBy.

```
animationScene1 = GetAnimationScene()
animationScene1.GoToNext()
renderView1 = GetActiveViewOrCreate('RenderView')
canex2Display = Show(clip, renderView1)
```

- Get point var names

```
canex2 = GetActiveSource()
print(canex2.PointVariables.GetAvailable())
```

- Get Cell var names

```
print(canex2.ElementPointVariables.GetAvailable())
```

- For point vars

```
vars = canex2.PointVariables.GetAvailable()
print(vars)
ColorBy(canex2Display, ('POINTS', vars[0]))
```

- For cell vars

```
vars = canex2.ElementVariables.GetAvailable()
print(vars)
ColorBy(canex2Display, ('CELLS', vars[0]))
# Actually not needed
Render()
```

Information on reading variable information is found in Section 1.3.3 of the User's Guide.

### Scale Around Dataset Center - A userful example

- We want to create a script that allows us to scale a dataset around it's center. This example shows how to get the active source, get the bounds, and transform the camera.

```
scale_factor = 2
indata = GetActiveSource()

bounds = indata.GetDataInformation().GetBounds()
center = ((bounds[0] + bounds[1])/2, (bounds[2] + bounds[3])/2,(bounds[4] +␣
↪bounds[5])/2)
```

(continues on next page)

(continued from previous page)

```
transform_to_center = Transform()
transform_to_center.Transform.Translate = [-center[0],
-center[1], -center[2]]
Hide()

scale = Transform()
scale.Transform.Scale = [scale_factor, scale_factor,
scale_factor]
Hide()

transform_from_center = Transform()
transform_from_center.Transform.Translate = [center[0],
center[1], center[2]]
Show()

Render()
```

### Trace Recorder

`paraview` includes a tool to automatically generate Python scripts for us. It is called the Trace Recorder. An example is as follows.

- Read in can.ex2, clip can, paint by EQPS, change the camera to +Y, write out a screenshot and write out a movie. The steps are:
  - **Tools** → **Start Trace** Select **Show Incremental Trace**.
  - Open can.ex2.
  - **OK**.
  - Turn all variables on.
  - **Apply**.
  - **+Y**.
  - Select **Clip**.
  - **Y Normal**.
  - Unselect **Show Plane**.
  - **Apply**.
  - Set **Coloring** to **EQPS**.
  - Last timestep.
  - Rescale to Data Range.
  - Set First timestep.
  - **File** → **Save Screenshot**. Save as .png.
  - **File** → **Save Animation**. Save as .avi.
  - **Tools** → **Stop Trace**
  - **File** → **Save** to a known location.

> Another way to find Python for `paraview` is through Save State. This should be a last resort, but it may include commands that the Trace Recorder missed. '"File → Save State → Python State File.

### Running Scripts

ParaView allows a user to run a script. This is done as follows:

- **Tools** → **Python Shell**
- **Run Script**

Now, browse to your script, and select OK.

### Macros

`paraview` can save and use Python scripts that have been placed in a known location. When you create a trace, you have the option to **File** → **Save As Macro**. You also have the option on the Macros menu to **Add new macro**. Macros will be added to the Macro toolbar at the top of the ParaView GUI. You can edit and delete these Macros through the **Macro menu**.

> As an example, lets add the Python script that we created above.

- **Macros** → **Add new macro**, find your macro, and click **OK**.
- Click on your **Macro** on the toolbar.

### Python Help

More information can about can be found throughout the User's Guide.

Additional information can be found in Section 4.1.3 of Self-directed Tutorial.

## 4.2.13 Python & Batch: Python Calculator, Programmable Source & Filter

### Introduction

**ParaView** has three filters that give a user access to python math functions as well as the underlying VTK library. These are the Python Calculator, the Programmable Source and Filter. This tutorial explains both.

### Python Calculator

The Python Calculator allows a user to apply calculations that are available in Python. These include such functions as get volume of cells, get area of cells, get the cross product, dot product, curl, etc. The whole formula must fit on one line. Lets go over the Python Calculator with an example.

- Lets create a new point variable, loaded with 5.
  - Open can.ex2.
  - Turn all variables on.
  - **Apply**.
  - **Filters** → **Alphabetical** → **Python Calculator**.
  - Change Expression to **5**.

  - Set **Array Association** to **Point Data**
  - Change **Array Name** to **Calculated Variable**.
  - **Apply**.
  - Paint by **Calculated Variable**.
- Lets create a variable equal to two times displacement.
  - Change Expression to **DISPL*2**.
  - **Apply**.
  - Go to last timestep.
  - Rescale to Data Range.
  - Note: A more complete way to access displacement data. We explicitly pull **DISPL** from the first input to the filter. Don't forget that the first input is [0], the second is [1], etc.
  - Change Expression to **(inputs[0].PointData['DISPL']) * 2**.
  - **Apply**.
  - Rescale to Data Range.
- Here is how to multiply a vector by a global variable. Lets multiply DISPL by the timestep (TMSTEP)
  - Change expression to **inputs[0].FieldData['NSTEPS'][time_index] * DISPL**.
  - **Apply**.
  - Rescale to Data Range.
  - To recap, If a function requires a variable input, use the string above. If the function needs the input mesh, use inputs[].

**Did you know?**

Python uses square brackets for arrays, and parenthesis to group parts of your formula. Also, to designate a string (i.e., variable name), use either single or double quotation marks.

- Lets create a variable to contains the cell volume.
  - Change expression to **volume(inputs[0])**.
  - **Apply**.
  - Rescale to Data Range.
- Numerous functions can be combined in one expression. For instance, a nonsensical expression would be to take a sin of a divergence of a curl. Here is the expression:
  - Change expression to **sin(divergence(curl('ACCL')))**. Apply. Rescale to Data Range.

Interesting are functions available through the Python Calculator:

- area(dataset)
- aspect(dataset)
- cos(array)
- cross(X,Y) where X and Y are two 3D vector arrays
- curl(array)

- divergence(array)
- dot(a1,a2)
- eigenvalue and eigenvector(array)
- gradient(array)
- max(array)
- mean(array)
- min(array)
- norm(array)
- sin(array)
- strain(array)
- volume(array)
- vorticity(array)

The complete list can be found in Section 1.5.9 of the User's Guide.

### Programmable Filter

The Programmable Filter is used to modify data in the pipeline using Python. An example of a Programmable Filter dividing a variable by 2:

- Divide **ACCL** by 2
  - Open can.ex2.
  - Turn all variables on.
  - **Apply**.
  - Select **Filters** → **Alphabetical** → **Programmable Filter**.
  - Leave Output Data
  - Set Type as **Same as Input**.
  - Enter the following into the Script window:

    ```
    input0 = inputs[0]
    dataArray = input0.PointData["ACCL"] / 2.0
    output.PointData.append(dataArray, "ACCL_half")
    ```

  - Set **Coloring** by **ACCL_half**
- Subtract two datasets from each other. Use two instances of disk_out_ref.ex2 as two datasets, subtract **GaMe3** from **AsH3**.
  - Open disk_out_ref.ex2.
  - Turn all variables on.
  - **Apply**.
  - Open disk_out_ref.ex2.
  - Turn all variables on.
  - **Apply**.

  - Highlight both datasets. Use the key with the mouse to select more than one input.
  - Select **Filters** → **Alphabetical** → **Programmable Filter**.
  - Leave Output Data
  - Set Type as **Same as Input**.
  - Enter the following into the Script window:

    ```
    v_0 = inputs[0].PointData['AsH3']
    v_1 = inputs[1].PointData['GaMe3']
    output.PointData.append(v_1 - v_0, 'difference')
    ```

  - Set **Coloring** to **difference**

- Create a tensor

  ```
  from paraview.vtk.numpy_interface import dataset_adapter as dsa
  import numpy
  def make_tensor(xx,yy,zz, xy, yz, xz):

        t = numpy.vstack([xx,yy,zz,xy, yz,
        xz]).transpose().view(dsa.VTKArray)
        t.DataSet = xx.DataSet
        t.Association = xx.Association
        return t

  xx = inputs[0].PointData["sigma_xx"]
  yy = inputs[0].PointData["sigma_yy"]
  zz = inputs[0].PointData["sigma_zz"]
  xy = inputs[0].PointData["sigma_xy"]
  yz = inputs[0].PointData["sigma_yz"]
  xz = inputs[0].PointData["sigma_xz"]
  output.PointData.append(make_tensor(xx,yy,zz,xy,yz,xz), "tensor")
  ```

- Subtract two timesteps from each other
  - Load can.ex2
  - Select **Filters** → **Alphabetical** → **Force Time**.
  - Set the timestep to the one whose displacement you want to be zero.
  - The output of this filter is the can.ex2 dataset "frozen" at the chosen timestep. It will not change as you advance through time.
  - Select both can.ex2 (using the **CTRL** key) and the **ForceTime1** sources in the Pipeline Browser. Note that the original can.ex2 source will update as the timestep changes, but the **ForceTime1** source will not change.
  - Select **Filters** → **Alphabetical** → **Python Calculator**.
  - Substract the displacement in the "frozen" dataset from the current timestep in can.ex2.
  - Set the expression to **inputs[0].PointData['DISPL'] - inputs[1].PointData['DISPL']**
  - The order of inputs into the **Python Calculator** is not well defined, so you may need to swap the indices in inputs[0] and inputs[1] to get the correct sign on the result, but this should work.

  - (If desired,) Finally, add a Plot Selection over Time filter. This filter will run over all time steps, subtracting the data from the current timestep from the data in the "frozen" timestep produced by the Force Time filter, and plot the result in a graph. The first timestep should have a value of 0.

- Read eigenvector, calculate an eigenvalue, and place it into three variables

```
# ParaView Programmable Filter script.  ParaView 5.0.1.
#
# This code will read in an eigenvector, calculate an
#    eigenvalue, and place it into three variables.
#
# Be sure to correct the input variables below.  Also, note that
#   the code uses ZX, not XZ.
#
# This code only works with any multiblock or vtk
#   datasets (including ones with only one block - i.e.,
#   Exodus datasets as input).
#
# Usage:  Run the Programmable Filter.
#   Cut and paste this file in the section named "Script".
#   Leave "Output Data Set Type" as "Same as Input".
#   Click Apply button
#
# Written by Jeff Mauldin and Alan Scott
#

import numpy as np

def process_composite_dataset(input0):
  # Pick up input arrays
  xxar = input0.CellData["EPSXX"]
  xyar = input0.CellData["EPSXY"]
  zxar = input0.CellData["EPSZX"]
  yyar = input0.CellData["EPSYY"]
  yzar = input0.CellData["EPSYZ"]
  zzar = input0.CellData["EPSZZ"]

  #print `xxar`
  #print len(xxar.Arrays)

  # Set output arrays to same type as input array.
  # Do a multiply to make sure we don't just have a
  # pointer to the original.
  outarray0 = xxar*0.5
  outarray1 = xxar*0.5
  outarray2 = xxar*0.5


  # Run a for loop over all blocks
  numsubarrays = len(xxar.Arrays)
  for ii in range(0, numsubarrays):
    # pick up input arrays for each block.
    xxarsub = xxar.Arrays[ii]
    xyarsub = xyar.Arrays[ii]
```

(continues on next page)

(continued from previous page)

```
    zxarsub = zxar.Arrays[ii]
    yyarsub = yyar.Arrays[ii]
    yzarsub = yzar.Arrays[ii]
    zzarsub = zzar.Arrays[ii]

    #print `xxarsub`

    # Transpose and calculate the principle strain.
    strain = np.transpose(
        np.array(
          [ [xxarsub, xyarsub, zxarsub],
            [xyarsub, yyarsub, yzarsub],
            [zxarsub, yzarsub, zzarsub] ] ),
              (2,0,1))

    principal_strain = np.linalg.eigvalsh(strain)

    # Move principle strain to temp output arrays for this block
    outarray0.Arrays[ii] = principal_strain[:,0]
    outarray1.Arrays[ii] = principal_strain[:,1]
    outarray2.Arrays[ii] = principal_strain[:,2]

  #ps0 = principal_strain[:,0]
  #print "ps0 len: " + str(len(ps0))

  # Finally, move the temp arrays to output arrays
  output.CellData.append(outarray0, "principal_strain_0")
  output.CellData.append(outarray1, "principal_strain_1")
  output.CellData.append(outarray2, "principal_strain_2")


def process_unstructured_dataset(input0):
  # Pick up input arrays
  xxar = input0.CellData["EPSXX"]
  xyar = input0.CellData["EPSXY"]
  zxar = input0.CellData["EPSZX"]
  yyar = input0.CellData["EPSYY"]
  yzar = input0.CellData["EPSYZ"]
  zzar = input0.CellData["EPSZZ"]

  #print `xxar`
  #print len(xxar.Arrays)

  # Transpose and calculate the principle strain.
  strain = np.transpose(
        np.array(
          [ [xxar, xyar, zxar],
            [xyar, yyar, yzar],
            [zxar, yzar, zzar] ] ),
              (2,0,1))

  principal_strain = np.linalg.eigvalsh(strain)
```

(continues on next page)

(continued from previous page)

```
  #ps0 = principal_strain[:,0]
  #print "ps0 len: " + str(len(ps0))

  # Finally, move the temp arrays to output arrays
  output.CellData.append(principal_strain[:,0],
     "principal_strain_0")
  output.CellData.append(principal_strain[:,1],
     "principal_strain_1")
  output.CellData.append(principal_strain[:,2],
     "principal_strain_2")

input0 = inputs[0]

if input0.IsA("vtkCompositeDataSet"):
  process_composite_dataset(input0)
elif input0.IsA("vtkUnstructuredGrid"):
  process_unstructured_dataset(input0)
else:
  print "Bad dataset type for this script"
```

Important note: Since the **Programmable Source** and **Programmable Filter** work at the server level, paraview.simple cannot be loaded or used.

Mode details about the **Programmable Filter** can be found in Section 2.5 of the Reference Manual.

Additionally, this is list of interesting Blog Posts related to the **Programmable Filter**:

- https://blog.kitware.com/improved-vtk-numpy-integration/
- https://blog.kitware.com/improved-vtk-numpy-integration-part-2/
- https://blog.kitware.com/improved-vtk-numpy-integration-part-3/
- https://blog.kitware.com/mpi4py-and-vtk/
- https://blog.kitware.com/improved-vtk-numpy-integration-part-4/
- https://blog.kitware.com/improved-vtk-numpy-integration-part-5/

## 4.2.14 Python & Batch: pvpython and pvbatch

### Introduction

ParaView can run without opening the ParaView GUI, and is controlled through Python. There are two Python interfaces - pvpython and pvbatch.

### pvpython

pvpython is the Python interface to ParaView. You can think of pvpython as ParaView with a Python interface. As we did with the Python Shell, you can manually type in commands to pvpython. The first thing you will want to do is import paraview simple, as follows:

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

pvpython can also read Python command files. Type **pvpython –help** for arguments. Running pvpython files looks like this:

```
/pathTopvpython/pvpython /pathToPythonCommandFile/commandFile.py

# Example
D:/alan/paraview/pvpython D:/alan/scripts/disk_out_ref-A.py
```

- You will notice that pvpython will run the script and then exit. The output of the script is a screenshot or other data product.
- Anywhere that needs editing in the scripts above will be marked by the string **editMeHere**.
- You will need to hard code in the paths to your data, and paths for output products.
- The first time you run a script with pvpython, the output will be a postage stamp sized window. You can change this by finding and uncommenting the line **renderView1.ViewSize**.
- Try making and running a script of your own. Alternatively, here is an example. Cut and paste the following into a file named greenSphere.py:

  ```
  #!/usr/bin/env pvpython

  from paraview.simple import *

  # Lets create a sphere
  sphere=Sphere()
  Show()
  Render()

  # get active view
  renderView1 = GetActiveViewOrCreate('RenderView')
  renderView1.ViewSize = [1670, 1091]

  # get display properties
  sphere1Display = GetDisplayProperties(sphere, view=renderView1)

  # change solid color
  sphere1Display.AmbientColor = [0.0, 1.0, 0.0]
  sphere1Display.DiffuseColor = [0.0, 1.0, 0.0]

  # save screenshot
  SaveScreenshot('greenSphereScreenshot.png', renderView1,
  ImageResolution=[1670, 1091])
  ```

- Run this as follows:

  ```
  /pathTopvpython/pvpython greenSphere.py
  ```

### pvbatch

pvbatch is like pvpython, with two exceptions. pvbatch only accepts commands from input scripts, and pvbatch will run in parallel if it was built using MPI. Input is exactly like pvpython.

#### Generic user specific section

#### This section describes how to use pvbatch when on Windows PCs.

- If you are training on Linux, pvbatch will exist in the bin directory.
- If you are training on OS X, open a terminal window, and cd to /Applications/ParaView x.x.x/Contents/bin. pvbatch will be located here.
- If you are training on Windows, pvbatch does not exist. But, for a single process, such as this training, pvpython will substitute.

#### Lets create a Python trace.

Since we are on Windows, we will create a python trace, and use pvpython to process it.

- Read exodus data, screenshot, movie.
  - Run paraview.
  - Select **Tools** → **Start Trace**
  - Open can.ex2.
  - Turn all variables on.
  - **Apply**.
  - **+Y**.
  - Go forward one timestep.
  - Set **Coloring** to **EQPS**.
  - Select **File** → **Save Screenshot**.
  - **File** → **Save Animation**.
  - **Tools** → **Stop trace**.
  - Save this script on your desktop
  - Edit the file and change the following:
    - Correct the path to the input data and output screenshots or animations (not necessary, since you made the trace)
    - Uncomment the line that says renderView*.ViewSize. Change this to something reasonable (maybe 1920x1080)

### Let's now batch run this Python trace.

- Delete the Screenshot and Animation you made above. We want to recreate these.
- Open a **CMD** window. (On the Start button, type cmd, then click on Command Prompt.)
- cd to the ParaView bin directory

```
cd C:/Program Files (x86)/ParaView x.x.x/bin**
```

- Use `pvpython` to process our trace. Notice that `pvpython` understands forward slashes.

```
pvpython.exe C:/Users/myUserName/Desktop/trainingExampleScriptA.py
```

- Open the Screenshot and Animation that you just made.

### Let's edit the trace to accept arguments

- Edit the python trace.
- Right above the ExodusIIReader, enter the following code:

```
datasetIn = sys.argv[1]
directoryOut = sys.argv[2]
imageFilesOut = sys.argv[3]
print("datasetIn = " + datasetIn)
print("directoryOut = " + directoryOut)
print("imageFilesOut = " + imageFilesOut)
```

- Edit the **canex2 = ExodusIIReader** line as follows:

```
canex2 = ExodusIIReader(FileName=[datasetIn])
```

- Edit the **SaveScreenshot(...)** line as follows:

```
SaveScreenshot(directoryOut+imageFilesOut+'.png', renderView1,
ImageResolution=[1425, 1324])
```

- Edit the **SaveAnimation(...)** line as follows:

```
SaveAnimation(directoryOut+imageFilesOut+'.avi', renderView1,
ImageResolution=[1424, 1324], FrameWindow=[0, 43])
```

- Now, run in a command window as follows:

```
pvpython.exe "c:/Users/myUserName/Desktop/trainingExampleScriptA.py" "C:/Users/
↪myUsername/Desktop" "coolVizA"
```

### Generic Python Script for pvbatch on Linux

Here is an example running `pvbatch`, without having to make a trace.

- Make a file greenSphere.py, as described in the `pvpython` section.
- Copy greenSphere.py to be redSphere.py.
- Edit redSphere.py
  - Change **AmbientColor** and DiffuseColor to be [1.0, 0.0, 0.0]
  - Change the output file from greenSphereScreenshot.png to redShpereScreenshotScreenshot.png
- Make a file runner.sh
- Edit as follows:

```
#!/usr/bin/bash

/pathTopvbatch/pvbatch greenSphere.py
/pathTopvbatch/pvbatch redSphere.py
```

- Run as follows:

```
source runner.sh
```

### Sandia National Labs specific section

This section is specific to the clusters and environment at Sandia National Laboratories.

### pvbatch on the clusters

ssh into one of the clusters. `pvbatch` can be run on the login nodes, and magically will acquire compute nodes and run your batch visualization in parallel. You will find test scripts at /projects/viz/training/paraview. These scripts are run as follows:

```
/projects/viz/paraview/bin/pvbatch_chama_mesa
```

```
This is version x.x.x of |pvbatch|.
Incorrect number of argument supplied. Expecting 4 but have 0

Usage: /projects/viz/paraview/bin/pvbatch_chama_mesa <Nodes> <Minutes> <HERT estimate>␣
↪batchFileFullPath
```

An example is:

```
/projects/viz/paraview/bin/pvbatch_chama_mesa 1 10 FY123456 /projects/viz/training/
↪paraview/whipple-A.py
```

### pvbatchOnNode on the clusters

`pvbatch` can be run on the same nodes as your simulation. Ask ParaView help for more information on how to use this feature.

### Example scripts

Here are four examples. We are going to create scripts using the trace recorder, then run these scripts using `pvbatch`.

- Read exodus data, screenshot, movie.
  - Run `paraview`.
  - **Tools** → **Start Trace**.
  - Read g1s1.e.16.[0-15].
  - Go to last timestep.
  - Go back one timestep.
  - Set **Coloring** to **EQPS**.
  - **File** → **Save Screenshot**.
  - **File** → **Save Animation**.
  - **Tools** → **Stop trace**.
  - Save this script on your cluster.
  - Edit the file, change the following:
    - Correct the path to the input data and output screenshots or animations.
    - Uncomment the line that says **renderView*.ViewSize**. Change this to something reasonable (maybe 1920x1080).
- Read exodus data, Clip, Slice, screenshot, movie.
  - Run `paraview`.
  - **Tools** → **Start Trace**.
  - Read g1s1.e.16.[0-15].
  - Go to last timestep.
  - Go back one timestep.
  - Set **Coloring** to **EQPS**.
  - Select **Filters** → **Common** → **Clip**.
  - Slice using **Y Normal**.
  - **File** → **Save Screenshot**.
  - **File** → **Save Animation**.
  - **Tools** → **Stop trace**.
  - Save this script on your cluster.
  - Edit the file, change the following:
    - Correct the path to the input data and output screenshots or animations

        - Uncomment the line that says **renderView*.ViewSize**. Change this to something reasonable (maybe 1920x1080).
        - Add the following line above **ColorBy(....,('EQPS'))**.

          ```
          g1s110fpse16Display.SetScalarBarVisibility(renderView1, False)
          ```

- Read exodus data, 2d plots, screenshot, movie
    - Run `paraview`.
    - **Tools** → **Start Trace**.
    - Read g1s1.e.16.[0-15].
    - Select point if possible.
    - Plot selection (or plot over line, if necessary).
    - Plot EQPS.
    - **File** → **Save Screenshot**.
    - **File** → **Save Animation**.
    - **Tools** → **Stop trace**.
    - Edit the file, change the following:
        - Correct the path to the input data and output screenshots or animations.
        - Uncomment the line that says **renderView*.ViewSize**. Change this to something reasonable (maybe 1920x1080).
- Read cth data, extractCTHPart, screenshot, movie.
    - Run `paraview`.
    - **Tools** → **Start Trace**.
    - Read cth-med/spcth.[0-3].
    - ExtractCTHParts - 1.
    - ExtractCTHParts - 2.
    - **Tools** → **Stop trace**.

## 4.2.15 Targeted: ParaView & CTH

### Introduction

This tutorial shows how to visualize CTH datasets (called spcth files). CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories. A more complete writeup on CTH is here: http://www.sandia.gov/CTH/. CTH Spyplot files are Eulerian, or structured mesh datasets. They can be flat mesh or adaptive mesh refinement (i.e., AMR).

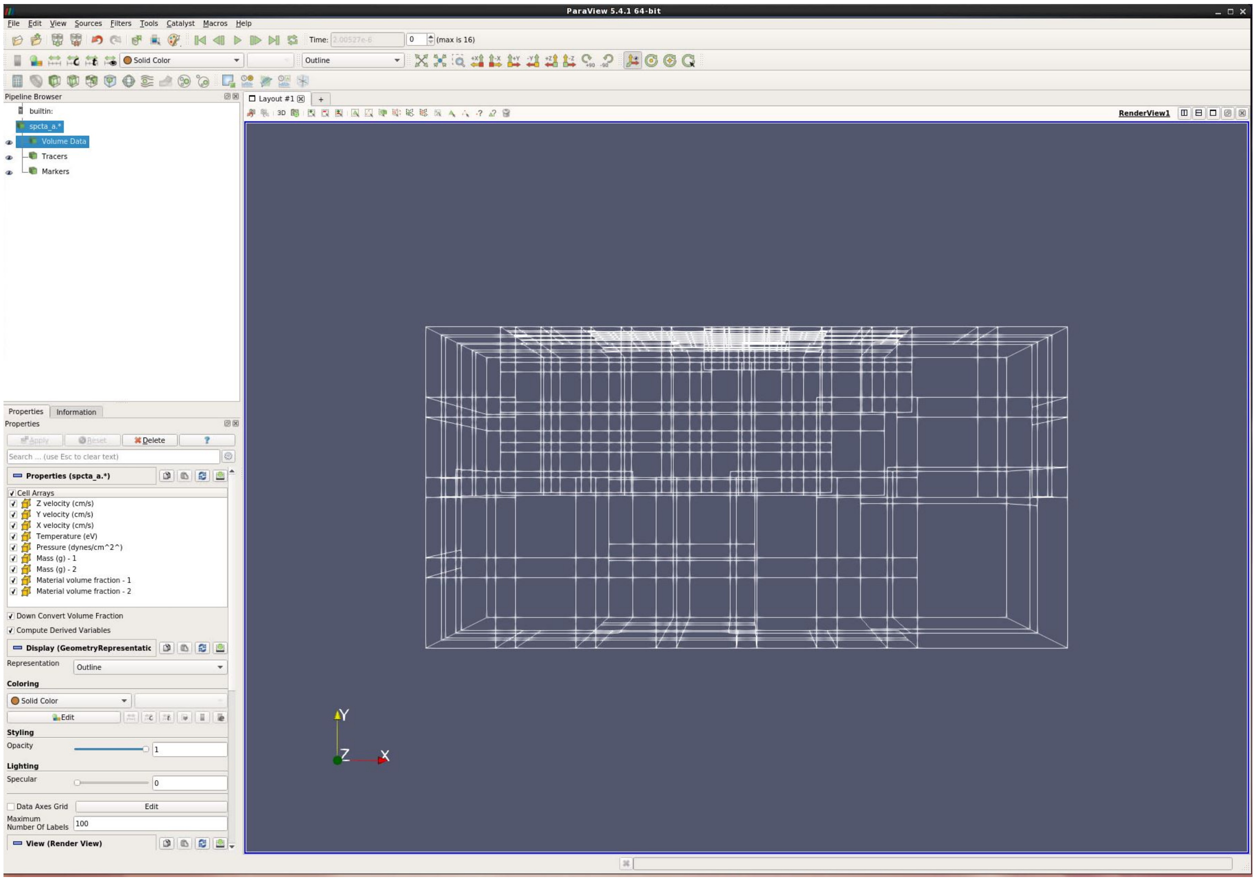
### ParaView's CTH reader

`paraview` reads spcth files in an very efficient manor. All material volume arrays go from 0 (representing 0%) to 1 (representing 100%). These are actually stored in the files as floats (4 bytes) or doubles (8 bytes). `paraview` reads these into unsigned bytes, which range from 0 to 255. Generally, you want a surface (contour or clip by scalar) at about 10%.

### Creating contours with the Extract CTH Parts filter

Our goal is to create a contour where Material Volume Fractions is at 10%.

- Start `paraview`.
- Open **spcth_a.0.[0-3]**.
- Turn all **variables** on.
- **Apply**.
- Notes:
  - There are three types of data that have been read in.
  - **Volume data** is the structured data from the CTH simuliation
  - **Tracers** are points in the mesh that move with a material, and can include variable information.
  - **Markers** are groups of points that move with a material.
  - Notice the **Down Convert Volume Fraction** has been checked. This means that ParaView will store volume fractions as an unsigned byte.
- **Y-**
- Note: Since we will have two materials touching each other, our default **Volume Fraction Value** will have some cells visible on top of each other. If this is not desired, change the value to **0.5**.
- Coloring by point will be smoother than color by cell. Thus, we will convert variables to be point for anything we color.
- Filters/CTH/**Extract CTH Parts**.
- Turn off all Volume Arrays, except **Material volume fraction - 1**.
- **Apply**.
- **Filters** → **Alphabetical** → **Cell Data to Point Data**.
- **Apply**.
- Set **Coloring** to **Pressure**.
- In the Pipeline Browser, select Volume Data (i.e., the raw cth data again).
- **Filters** → **CTH** → **Extract CTH Parts**.
- Turn off all Volume Arrays, except **Material volume fraction - 2**.
- **Apply**.
- Change Representation to "Surface With Edges".
- Change Representation back to "Surface".
- With the **left mousebutton**, grab the 3d view and drag down. This will rotate the block face down.
- With the **right mousebutton**, grab the 3d view and drag down. This will move closer to the block face.



Time: 2.00527e-6 0 (max is 16)

Solid Color

Outline

builtin:
- spcta_a.*
  - Volume Data
  - Tracers
  - Markers

**Properties (spcta_a.*)**

☑ Cell Arrays
- ☑ Z velocity (cm/s)
- ☑ Y velocity (cm/s)
- ☑ X velocity (cm/s)
- ☑ Temperature (eV)
- ☑ Pressure (dynes/cm^2^)
- ☑ Mass (g) - 1
- ☑ Mass (g) - 2
- ☑ Material volume fraction - 1
- ☑ Material volume fraction - 2

☑ Down Convert Volume Fraction

☑ Compute Derived Variables

**Display (GeometryRepresentatic**

Representation: Outline

**Coloring**

Solid Color

**Styling**

Opacity: 1

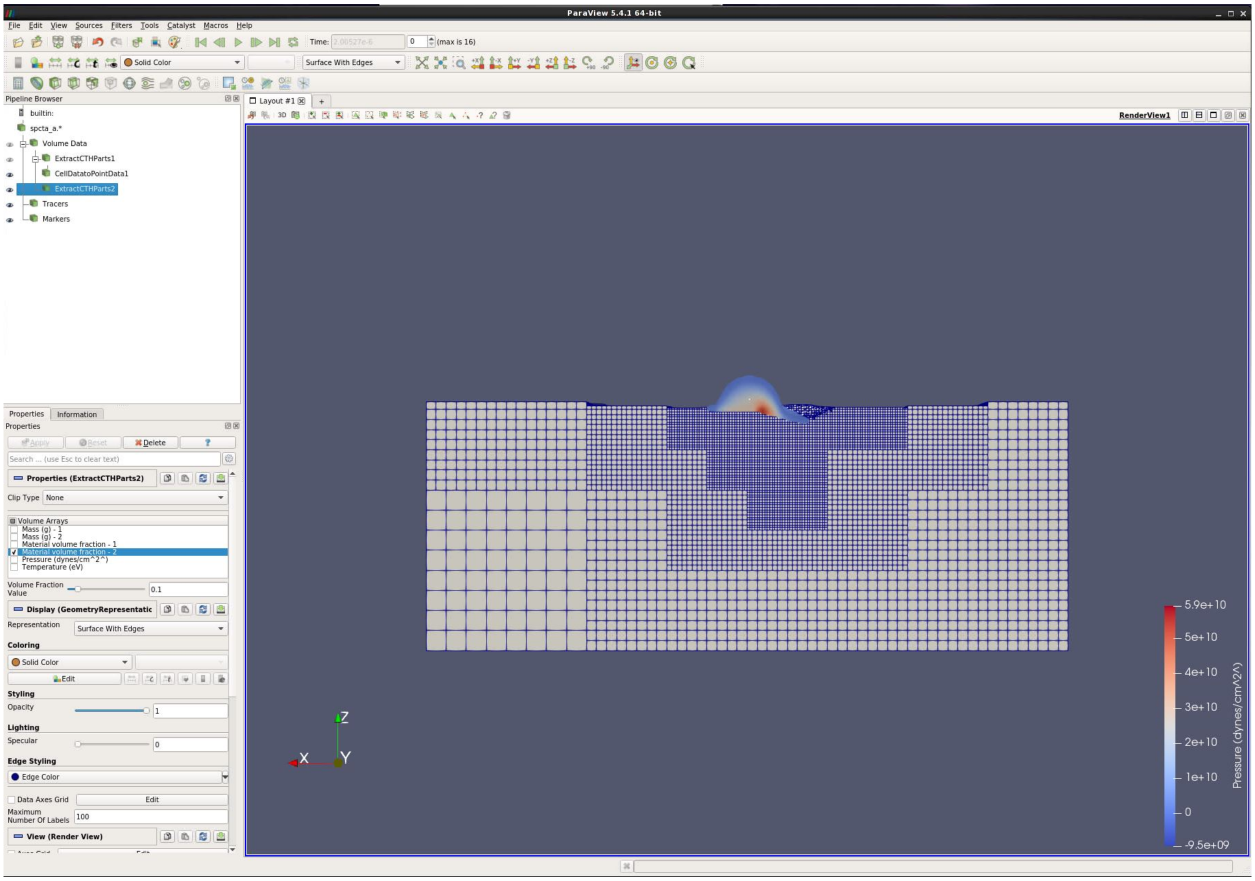
**Lighting**

Specular: 0

☐ Data Axes Grid

Maximum Number Of Labels: 100

**View (Render View)**



ParaView 5.4.1 64-bit

Time: 2.00527e-6 0 (max is 16)

Pipeline Browser

- builtin:
  - spcta_a.*
    - Volume Data
      - ExtractCTHParts1
        - CellDatatoPointData1
      - ExtractCTHParts2
    - Tracers
    - Markers

Properties (ExtractCTHParts2)

Clip Type: None

Volume Arrays

- ☐ Mass (g) - 1
- ☐ Mass (g) - 2
- ☐ Material volume fraction - 1
- ☑ Material volume fraction - 2
- ☐ Pressure (dynes/cm^2^)
- ☐ Temperature (eV)

Volume Fraction Value: 0.1

Display (GeometryRepresentatic

Representation: Surface With Edges

Coloring: Solid Color

Styling: Opacity 1

Lighting: Specular 0

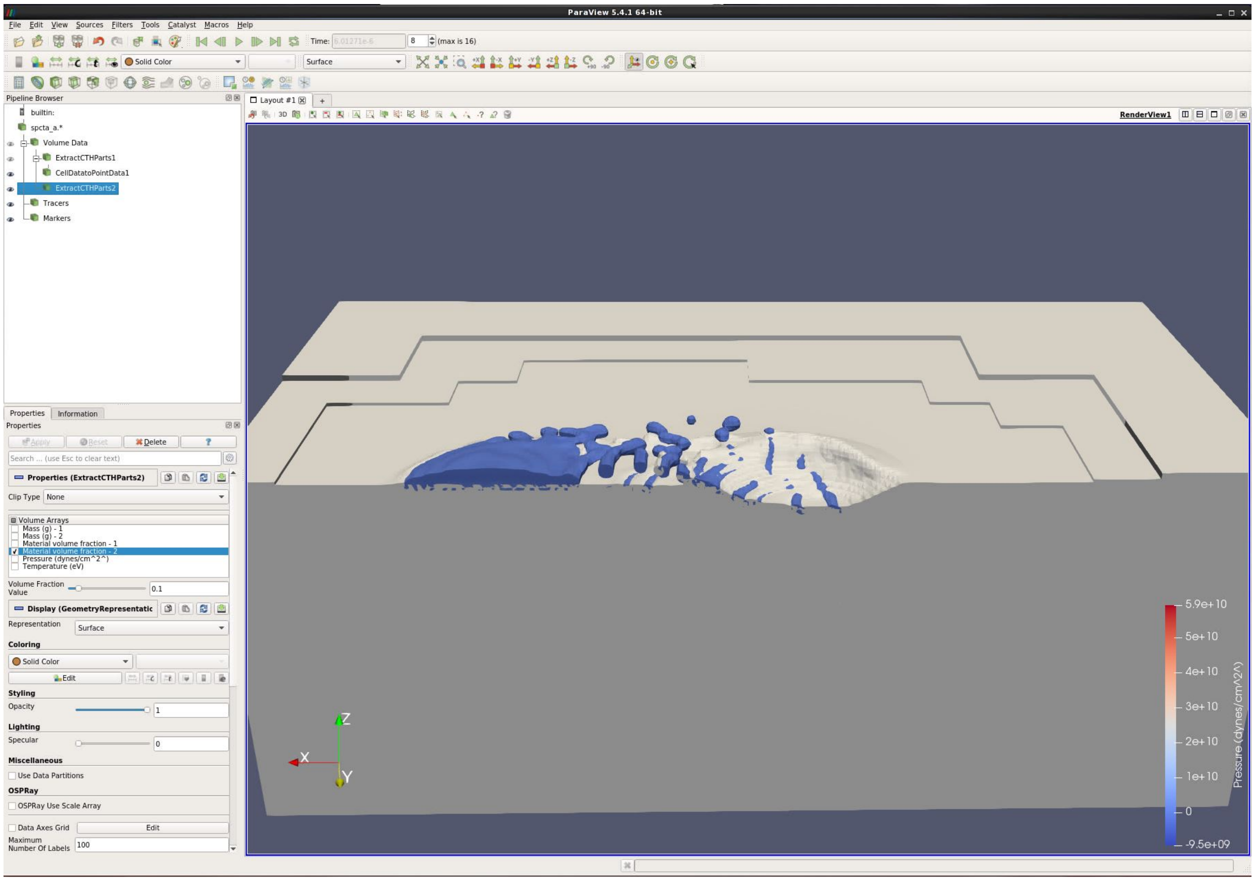
Edge Styling: Edge Color

☐ Data Axes Grid

Maximum Number Of Labels: 100

View (Render View)

- **Play** forward to about the 8th timestep.



Properties (ExtractCTHParts2)

Clip Type: None

Volume Arrays
- ☐ Mass (g) - 1
- ☐ Mass (g) - 2
- ☐ Material volume fraction - 1
- ☑ Material volume fraction - 2
- ☐ Pressure (dynes/cm^2^)
- ☐ Temperature (eV)

Volume Fraction Value: 0.1

Display (GeometryRepresentatic

Representation: Surface

Coloring: Solid Color

Styling
Opacity: 1

Lighting
Specular: 0

Miscellaneous
☐ Use Data Partitions

OSPRay
☐ OSPRay Use Scale Array

☐ Data Axes Grid

Maximum Number Of Labels: 100

### Creating a filled isovolume with the Clip by Scalar filter

Our goal is to create a solid 3d isovolume where Material Volume Fractions is at 10%. We want to do this so we can then take slices through the object.

Lets start from scratch.

- **Edit** → **Reset Session**.
- Open **spcth_a.0.[0-3]**.
- Turn all **variables** on.
- **Apply**.

We now want to move our data to point based, as opposed to cell based data.

- **Filters** → **Alphabetical** → **Cell Data to Point Data**.
- **Apply**

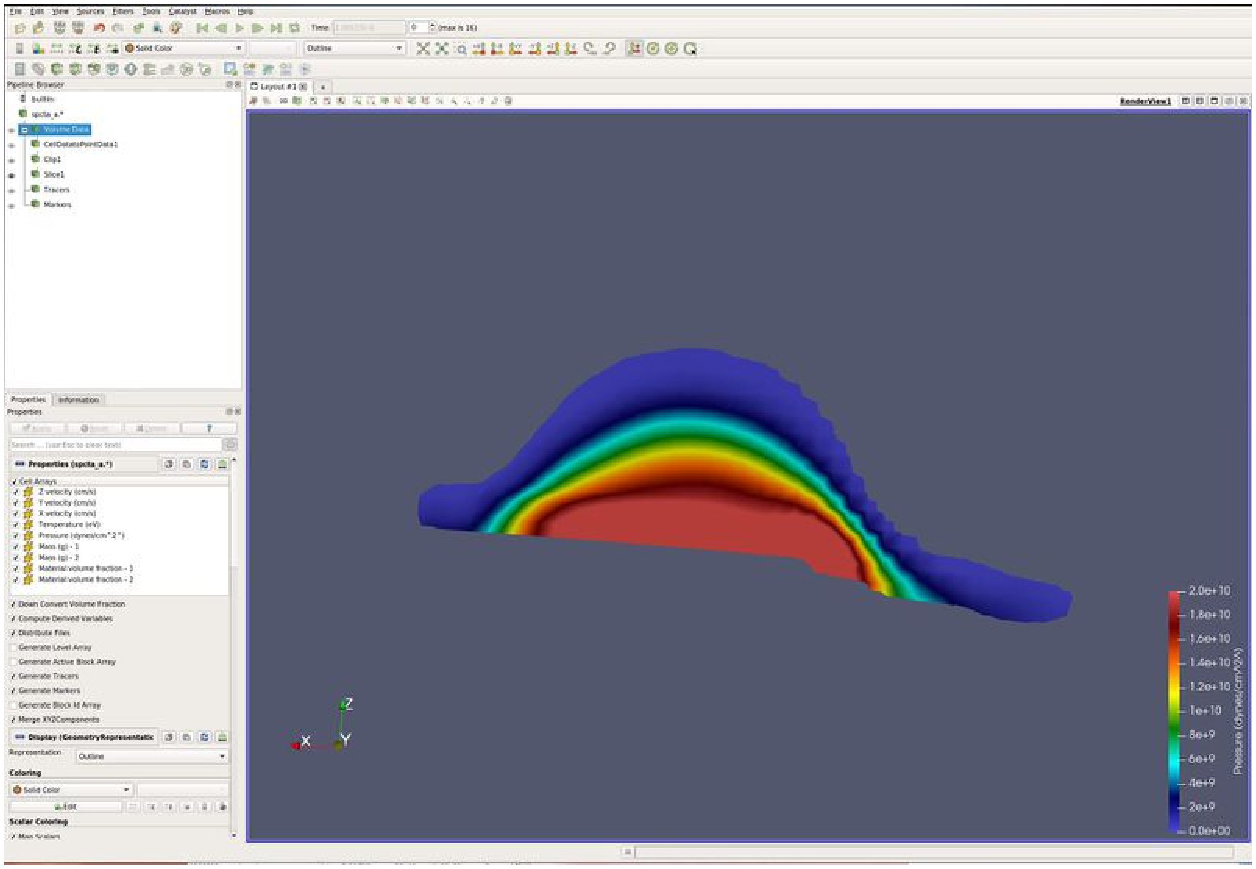
We now extract one material volume. ParaView has actually read in Material volume fraction data as an unsigned byte, running from 0 to 255. 10% full is about 25, 50% is 128. Lets create an isovolume at 10%.

- **Filters** → **Common** → **Clip**.
- Set **Clip Type** to **Scalar**.

- Set **Scalars** to **Material volume fraction - 1**.
- Set **Value** to **25**.
- **Apply**.
- Turn off **visibility** of the **Cell Data to Point Data** filter.
- **Filters** → **Common** → **Slice**.
- **Y Normal**.
- **Apply**.
- **Unselect** the Show Plane check box.
- Set **Coloring** to **Pressure**.

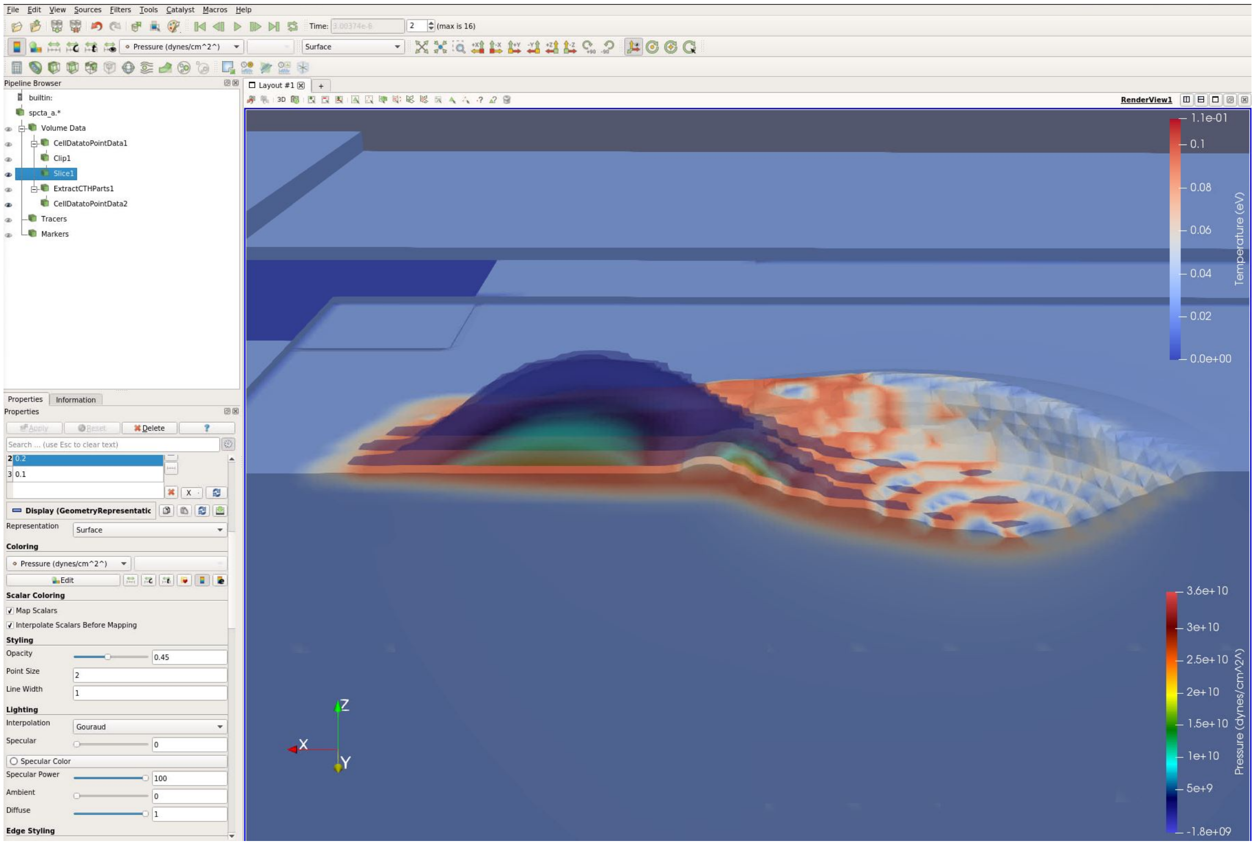
We now want to change the colormap.

- **View** → **Color Map Editor**. (There is a shortcut icon below the Edit menu.)
- Click the **Preset** icon (little folder with a heart).
- **Rainbow Desaturated**.
- **Apply**.
- **Close**.
- Click **Rescale to Custom Data Range** (small icon below Sources).
- Set range from **0 to 2e10**.



- Select the **Volume Data** in the pipeline browser.
- **Filters** → **CTH** → **Extract CTH Parts**.
- Select only **Material volume fraction - 2**.
- **Apply**.
- **Filters** → **Alphabetical** → **Cell Data to Point Data**. (This will smooth out the colors on each cell.)
- **Apply**.
- Set **Coloring** to **Temperature (eV)**

This image has three slices, opacity turned to 0.5 and the Extract CTH Parts filter is being painted by temperature.



Properties
2 0.2
3 0.1
Display (GeometryRepresentatic
Representation: Surface
Coloring
Pressure (dynes/cm^2^)
Scalar Coloring
☑ Map Scalars
☑ Interpolate Scalars Before Mapping
Styling
Opacity: 0.45
Point Size: 2
Line Width: 1
Lighting
Interpolation: Gouraud
Specular: 0
○ Specular Color
Specular Power: 100
Ambient: 0
Diffuse: 1
Edge Styling

### Analyzing fragments with the Material Interface filter

Our next goal is to analyze the fragments created by this simulation. To do this, we use the Material Interface Filter.

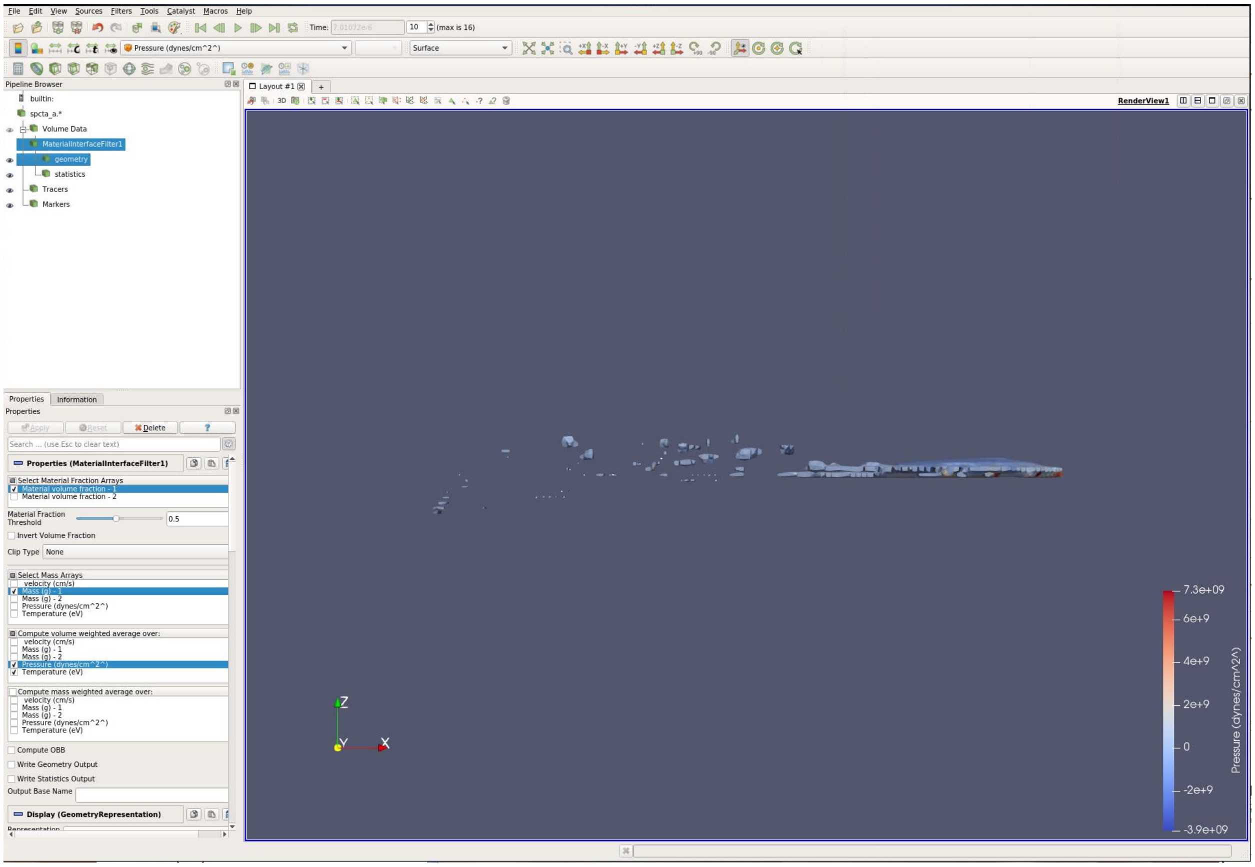
Lets start again from scratch.

- **Edit** → **Reset Session**.
- Open **spcth_a.0.[0-3]**.
- Turn all variables on.
- **Apply**.

This filter is tricky. Do exactly as follows.

- **Filters** → **CTH** → **Material Interface Filter**.

- In **Select Material Fraction Arrays** select **Material volume fraction - 1** (which is the ball).
- In **Select Mass Arrays** select **Mass (g) - 1** (which is the mass array for Material volume fraction - 1).
- In **Compute volume weighted average over** select **Pressure** and **Temperature**.
- Leave **Compute mass weighted average over** empty.
- **Apply**
- Go to the 10th timestep.
- Paint by **Pressure**.
- **+Y**



Properties (MaterialInterfaceFilter1)

Select Material Fraction Arrays
☑ Material volume fraction - 1
☐ Material volume fraction - 2

Material Fraction Threshold 0.5

☐ Invert Volume Fraction

Clip Type None

Select Mass Arrays
☐ velocity (cm/s)
☑ Mass (g) - 1
☐ Mass (g) - 2
☐ Pressure (dynes/cm^2^)
☐ Temperature (eV)

Compute volume weighted average over:
☐ velocity (cm/s)
☐ Mass (g) - 1
☐ Mass (g) - 2
☑ Pressure (dynes/cm^2^)
☑ Temperature (eV)

☐ Compute mass weighted average over:
☐ velocity (cm/s)
☐ Mass (g) - 1
☐ Mass (g) - 2
☐ Pressure (dynes/cm^2^)
☐ Temperature (eV)

☐ Compute OBB
☐ Write Geometry Output
☐ Write Statistics Output
Output Base Name

Display (GeometryRepresentation)

Representation

- Turn off visibility for the **geometry** in the Pipeline Browser. This is done by clicking the eyeball.

Anywhere there is a dot is the center of a fragment. We have statistics for each of these fragments.

- Turn on visibility for the **geometry** in the Pipeline Browser.

Now, lets look at the statistics in the spreadsheet view.

- **split vertical**. (This icon is found in the upper right corner of the 3d window)
- **Spreadsheet View**
- In the **Pipeline Browser**, click the eyeball for **statistics**

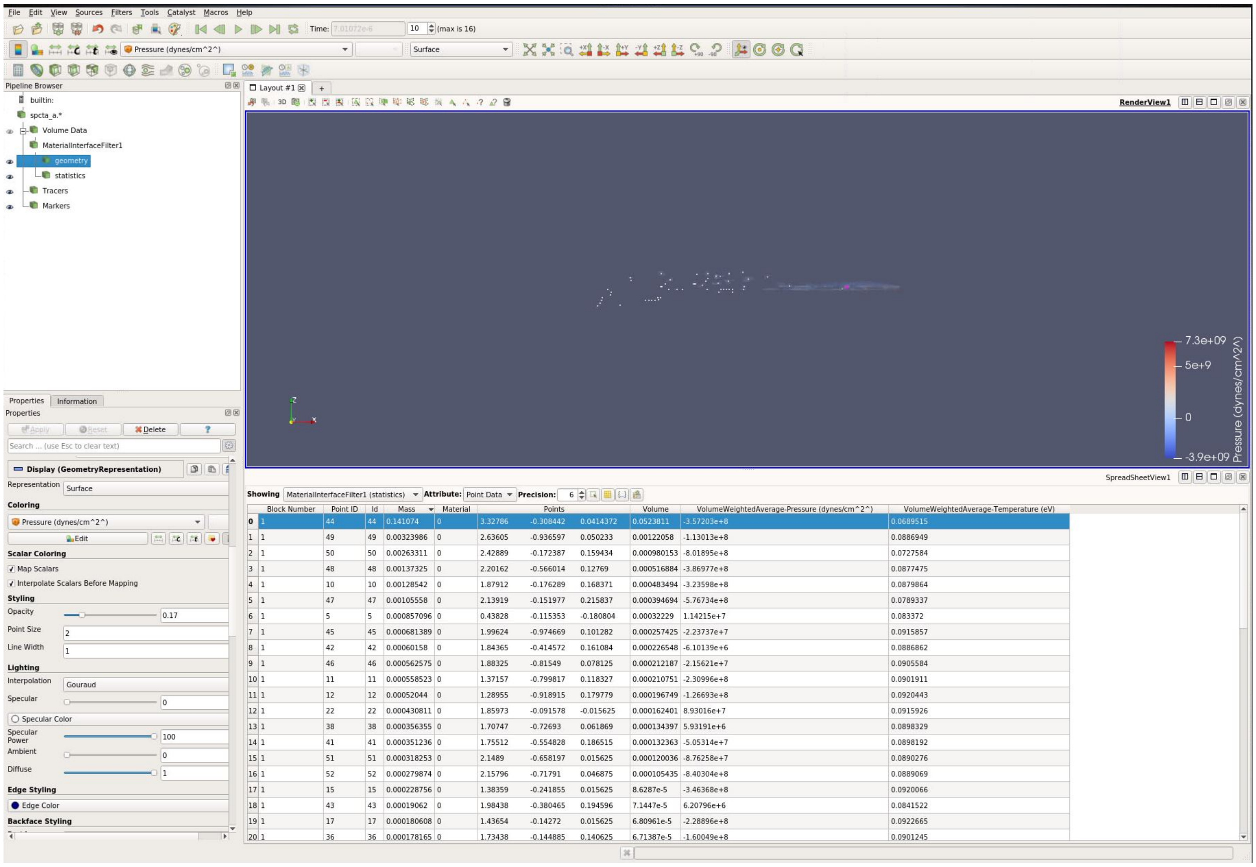
Each row represents a fragment.

- In the Spreadsheet View, click on the **Mass** column. You have now sorted the spreadsheet.

The biggest fragment is the top row.

- Click on the top row. It is now selected in the 3d view above.
- Click in the 3d view, then turn off visibility for the **geometry** in the Pipeline Browser. The geometry of the fragment is hiding the selected statistic point.

Note that you can also change the opacity of the geometry, so you can see this selected point.



Representation: Surface

Coloring: Pressure (dynes/cm^2^)

Scalar Coloring: ☑ Map Scalars, ☑ Interpolate Scalars Before Mapping

Styling: Opacity 0.17; Point Size 2; Line Width 1

Lighting: Interpolation Gouraud; Specular 0; Specular Power 100; Ambient 0; Diffuse 1

Showing MaterialInterfaceFilter1 (statistics) Attribute: Point Data Precision: 6

| | Block Number | Point ID | Id | Mass | Material | Points | | | Volume | VolumeWeightedAverage-Pressure (dynes/cm^2^) | VolumeWeightedAverage-Temperature (eV) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 44 | 44 | 0.141074 | 0 | 3.32786 | -0.308442 | 0.0414372 | 0.0523811 | -3.57203e+8 | 0.0689515 |
| 1 | 1 | 49 | 49 | 0.00323986 | 0 | 2.63605 | -0.936597 | 0.050233 | 0.00122058 | -1.13013e+8 | 0.0886949 |
| 2 | 1 | 50 | 50 | 0.00263311 | 0 | 2.42889 | -0.172387 | 0.159434 | 0.000980153 | -8.01895e+8 | 0.0727584 |
| 3 | 1 | 48 | 48 | 0.00137325 | 0 | 2.20162 | -0.566014 | 0.12769 | 0.000516884 | -3.86977e+8 | 0.0877475 |
| 4 | 1 | 10 | 10 | 0.00128542 | 0 | 1.87912 | -0.176289 | 0.168371 | 0.000483494 | -3.23598e+8 | 0.0879864 |
| 5 | 1 | 47 | 47 | 0.00105558 | 0 | 2.13919 | -0.151977 | 0.215837 | 0.000394694 | -5.76734e+8 | 0.0789337 |
| 6 | 1 | 5 | 5 | 0.000857096 | 0 | 0.43828 | -0.115353 | -0.180804 | 0.00032229 | 1.14215e+7 | 0.083372 |
| 7 | 1 | 45 | 45 | 0.000681389 | 0 | 1.99624 | -0.974669 | 0.101282 | 0.000257425 | -2.23737e+7 | 0.0915857 |
| 8 | 1 | 42 | 42 | 0.00060158 | 0 | 1.84365 | -0.414572 | 0.161084 | 0.000226548 | -6.10139e+6 | 0.0886862 |
| 9 | 1 | 46 | 46 | 0.000562575 | 0 | 1.88325 | -0.81549 | 0.078125 | 0.000212187 | -2.15621e+7 | 0.0905584 |
| 10 | 1 | 11 | 11 | 0.000558523 | 0 | 1.37157 | -0.799817 | 0.118327 | 0.000210751 | -2.30996e+8 | 0.0901911 |
| 11 | 1 | 12 | 12 | 0.00052044 | 0 | 1.28955 | -0.918915 | 0.179779 | 0.000196749 | -1.26693e+8 | 0.0920443 |
| 12 | 1 | 22 | 22 | 0.000430811 | 0 | 1.85973 | -0.091578 | -0.015625 | 0.000162401 | 8.93016e+7 | 0.0915926 |
| 13 | 1 | 38 | 38 | 0.000356355 | 0 | 1.70747 | -0.72693 | 0.061869 | 0.000134397 | 5.93191e+6 | 0.0898329 |
| 14 | 1 | 41 | 41 | 0.000351236 | 0 | 1.75512 | -0.554828 | 0.186515 | 0.000132363 | -5.05314e+7 | 0.0898192 |
| 15 | 1 | 51 | 51 | 0.000318253 | 0 | 2.1489 | -0.658197 | 0.015625 | 0.000120036 | -8.76258e+7 | 0.0890276 |
| 16 | 1 | 52 | 52 | 0.000279874 | 0 | 2.15796 | -0.71791 | 0.046875 | 0.000105435 | -8.40304e+8 | 0.0889069 |
| 17 | 1 | 15 | 15 | 0.000228756 | 0 | 1.38359 | -0.241855 | 0.015625 | 8.6287e-5 | -3.46368e+8 | 0.0920066 |
| 18 | 1 | 43 | 43 | 0.00019062 | 0 | 1.98438 | -0.380465 | 0.194596 | 7.1447e-5 | 6.20796e+6 | 0.0841522 |
| 19 | 1 | 17 | 17 | 0.000180608 | 0 | 1.43654 | -0.14272 | 0.015625 | 6.80961e-5 | -2.28896e+8 | 0.0922665 |
| 20 | 1 | 36 | 36 | 0.000178165 | 0 | 1.73438 | -0.144885 | 0.140625 | 6.71387e-5 | -1.60049e+8 | 0.0901245 |

## Showing cell data

Our next goal is to drill into and show individual cells' data. This is done with the the Hover Cells On function.

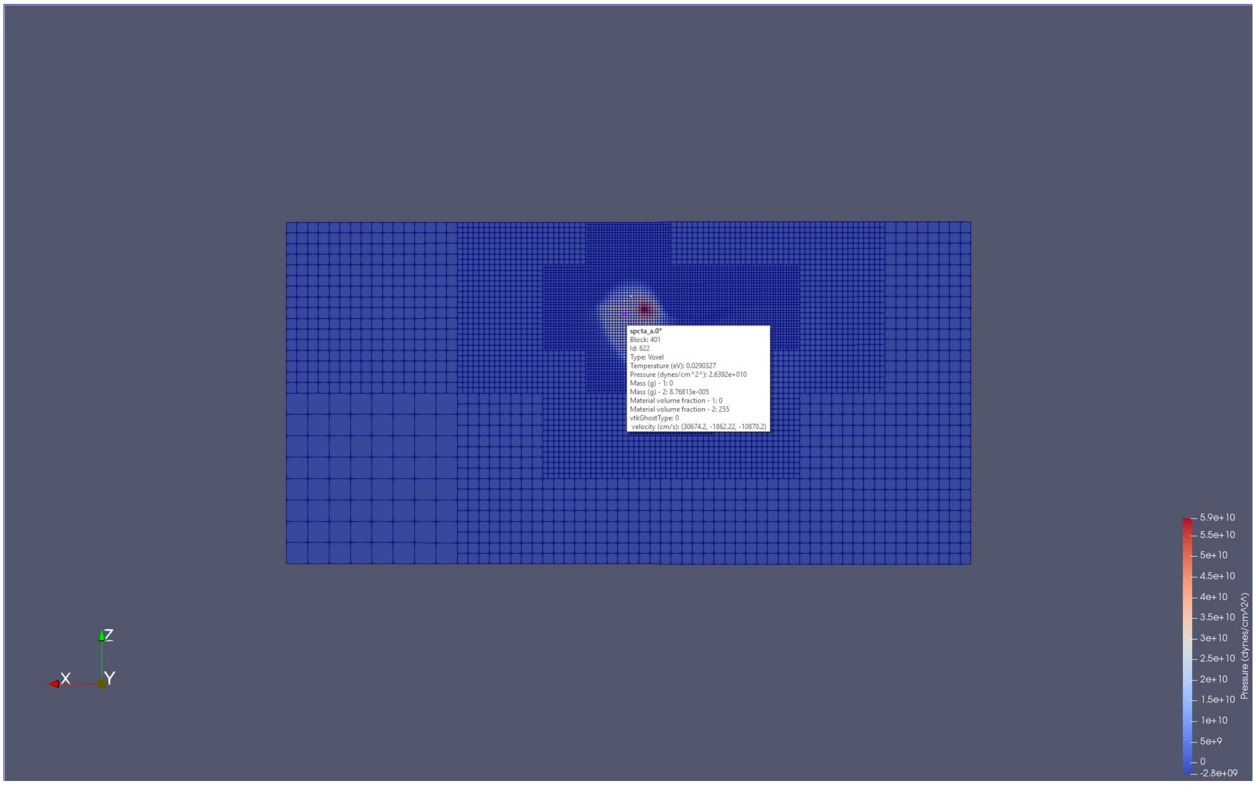
Lets start again from scratch.

- Edit/Reset Session.
- Open **spcth_a.0.[0-3]**.
- Turn all **variables** on.
- **Apply**.

We need a solid surface in which to work.

- Representation: **Surface with Edges**
- Set **Coloring** to **Pressure**
- **-Y**

- Now, click on the **Hover Cells On** icon, upper left section of the View.
- Hover, and stop, over the cells. The cell data will be displayed on screen.
  - **Hover Cells On** is the only selection that works, and provides data.
- If you want to look at cells in the middle of your dataset, use the **Clip**, **Slice** or **Clip by Scalar** filter to get to your data.



```
spcta_a.0*
Block: 401
Id: 622
Type: Voxel
Temperature (eV): 0.0290327
Pressure (dynes/cm^2^): 2.6392e+010
Mass (g) - 1: 0
Mass (g) - 2: 8.76813e-005
Material volume fraction - 1: 0
Material volume fraction - 2: 255
vtkGhostType: 0
 velocity (cm/s): (30674.2, -1862.22, -10870.2)
```

### Plot over line

You can also create a 2d plot along a line through your dataset. Here is how to do it.

Lets start again from scratch.

- **Edit** → **Reset Session**.
- Open **spcth_a.0.[0-3]**.
- Turn all **variables** on.
- **Apply**.

Again, we need a solid surface in which to work.

- Set **Representation** to **Surface with Edges**.
- Set **Coloring** to **Pressure**.
- **-Y**.

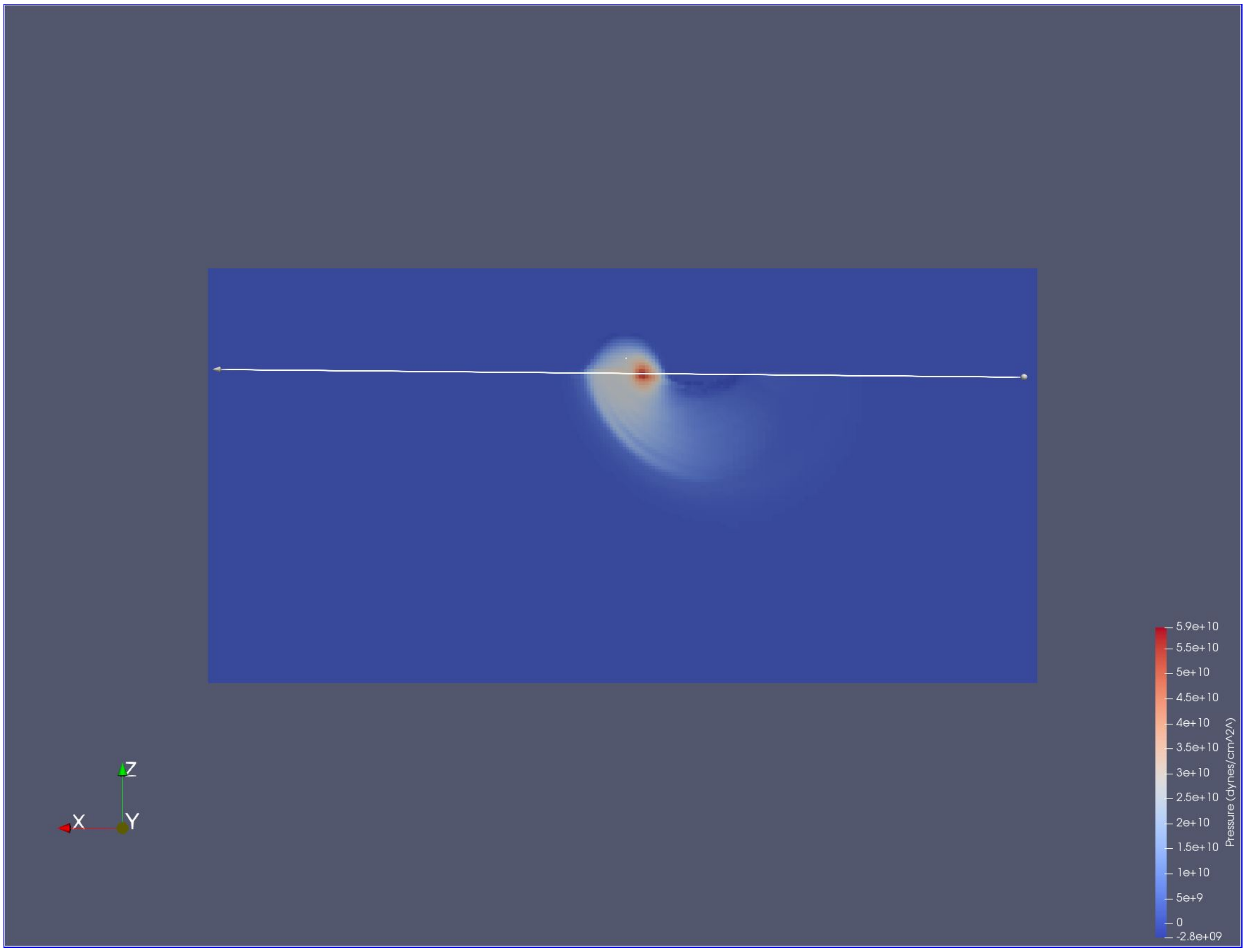
You can now see where the aluminum ball is hitting the steel block.

- **Filters** → **Data Analysis** → **Plot Over Line**.

- **Apply**.

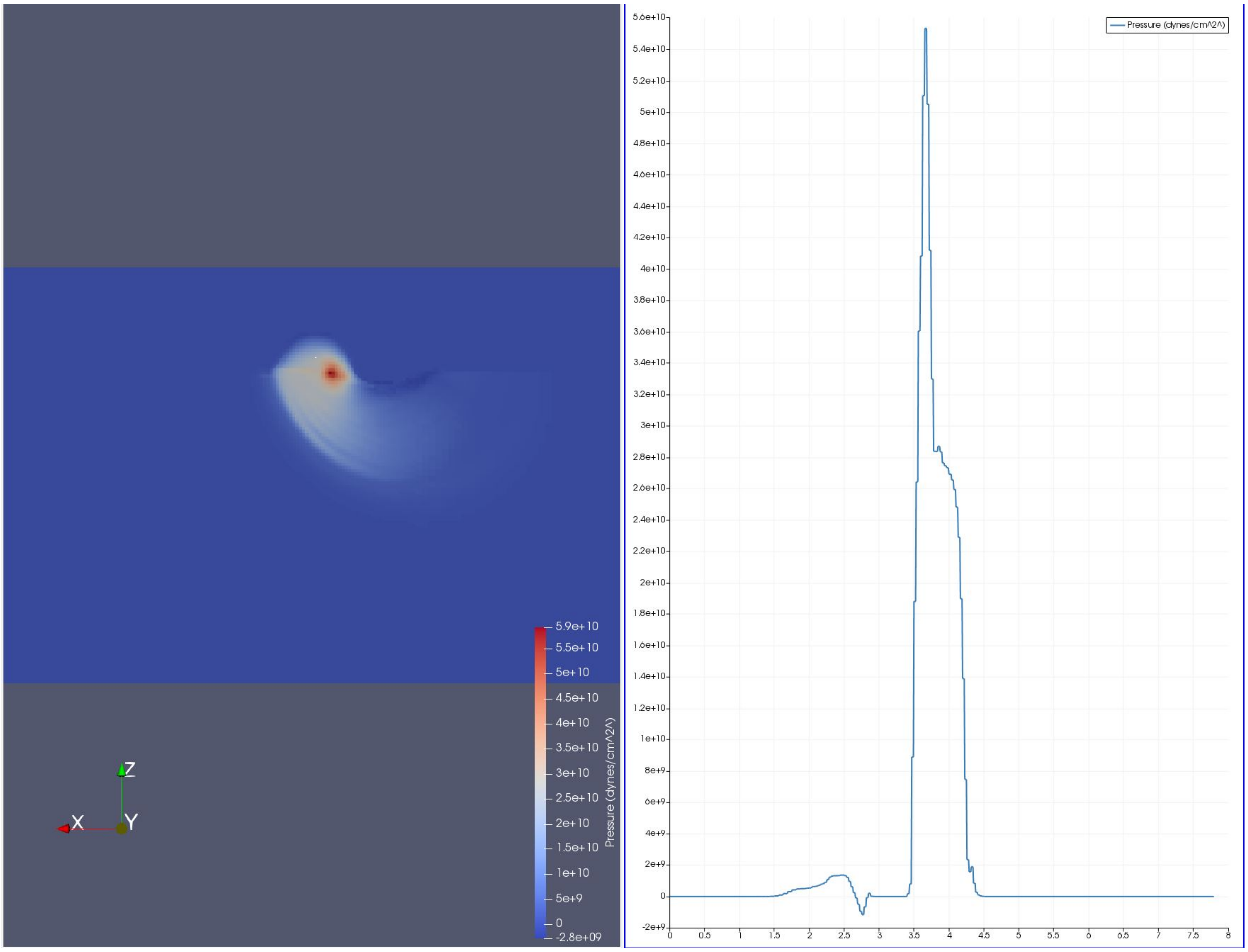
We want to overlay the line horizontally across the area of interest, just inside of the block. To do this, we will set it on the surface and then move it.

- Click X Axis. The line is now horizontal, through your data. The base of the arrow is on the right, the head is to the left.
- Place the cursor at the right edge of the block, 3/4 of the way to the top. Hit the **1** key. You have now moved the base.
- Place the cursor at the left edge of the block, 3/4 of the way to the top. Hit the **2** key. You have now moved the head.



- Make sure the **Plot Over Line** filter is still selected.
- **Apply**.
- In the **Properties** tab, un-select all variables that you don't want to plot.

More details on how to run **Plot** filters can be found in Section 4.2.5 of Tutorials.


5.9e+10
5.5e+10
5e+10
4.5e+10
4e+10
3.5e+10
3e+10
2.5e+10
2e+10
1.5e+10
1e+10
5e+9
0
-2.8e+09
Pressure (dynes/cm^2^)
Z
X
Y
Pressure (dynes/cm^2^)
5.6e+10
5.4e+10
5.2e+10
5e+10
4.8e+10
4.6e+10
4.4e+10
4.2e+10
4e+10
3.8e+10
3.6e+10
3.4e+10
3.2e+10
3e+10
2.8e+10
2.6e+10
2.4e+10
2.2e+10
2e+10
1.8e+10
1.6e+10
1.4e+10
1.2e+10
1e+10
8e+9
6e+9
4e+9
2e+9
0
-2e+9
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
6.5
7
7.5
8

### Calculating density

Density is calculated in the CTH reader for rectilinear grids, but not for AMR. Here is how to calculate it for AMR grids.

Lets start again from scratch.

- **Edit** → **Reset Session**.
- Open **spcth_a.0.[0-3]**.
- Turn all **variables** on.
- Since we need cell volume fraction going from 0 to 1, unselect **Down Convert Volume Fraction**.
- **Apply**.

Now, we calculate the volume of each cell.

- **Cell Size**.
- Only keep **Compute Volume** checked.
- **Apply**.

Now, we calculate the density.

- Select **Filters** → **Common** → **Calculator**.
- Set **Attribute Type** to **Cell Data**.
- Result Array **Density**.
- Set **Expression** to **(Mass(g) – 2) / (Volume *(Material Volume Fraction – 2))**.
  - The goal is divide the mass in a cell by the volume of that cell that is of this material.

## 4.2.16 Targeted: Computational Fluid Dynamics

### Introduction

This tutorial shows common visualization techniques for cfd datasets. We will be using the dataset disk_out_ref.ex2, found in Paraview under File/ Open/ Examples. It has a vector field in it called V. Note that we will reset session (i.e., start from scratch) every section.

### Slices

- Open disk_out_ref.ex2.
- **Apply**.
- **+X**
- Select **Filters** → **Common** → **Slice**.
- **Apply**.
- Unselect the **Show Plane**.
- Set **Coloring** to **v**.
- In the pipeline browser, select **disk_out_ref.ex2**
- Select **Filters** → **Common** → **Slice**.

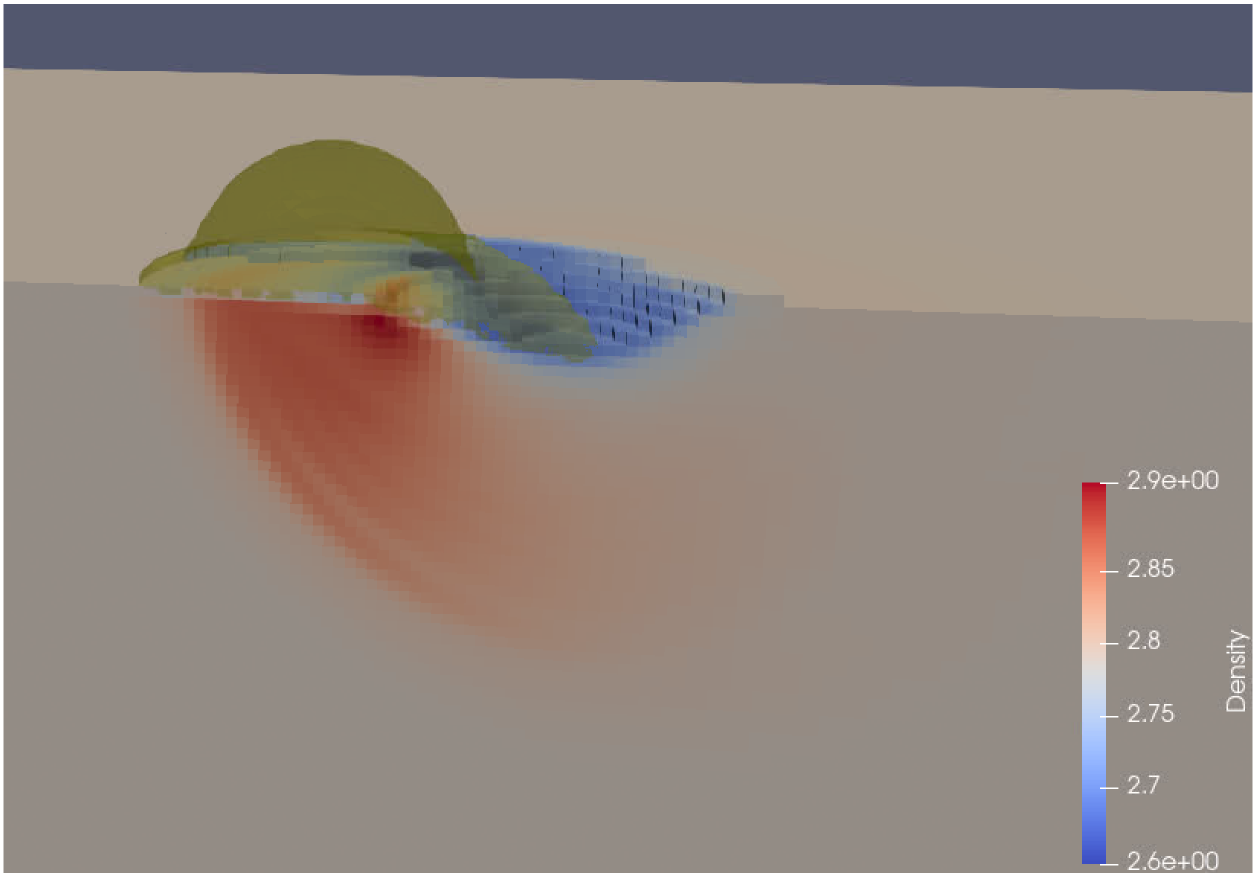
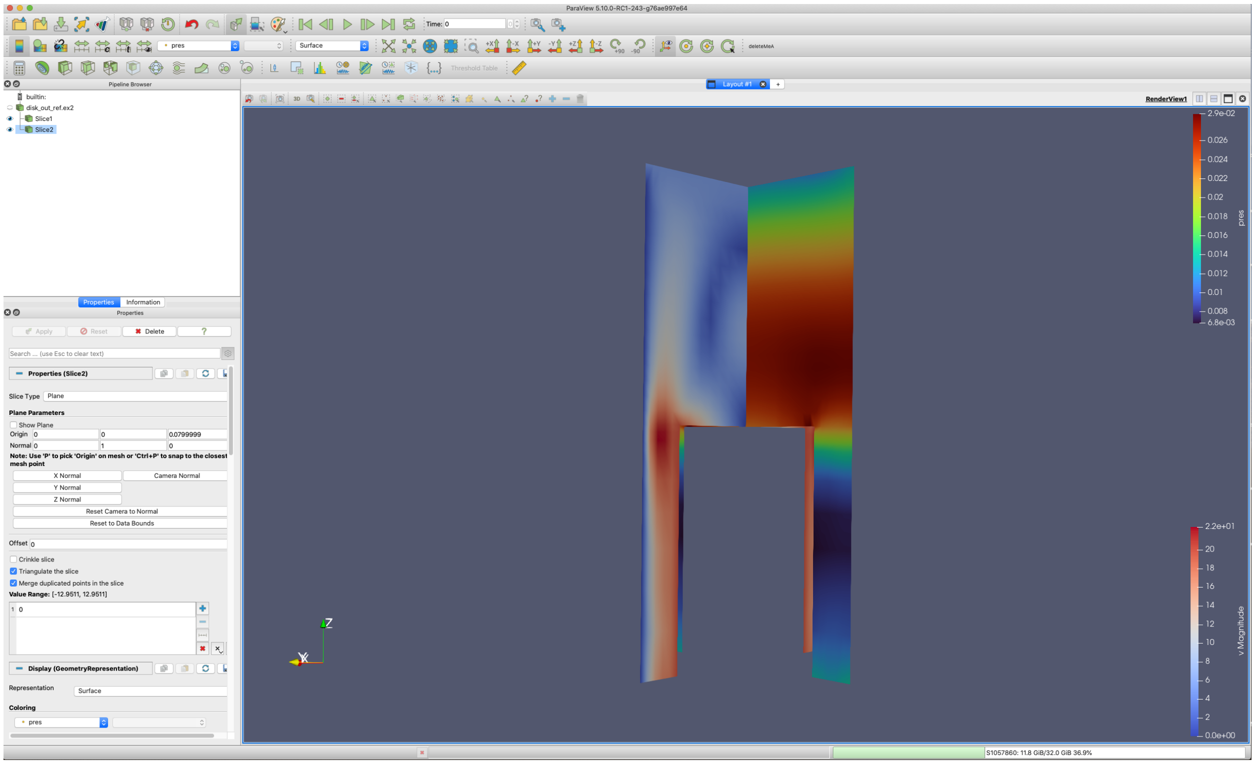


Fig. 4.13: Ball sliding across aluminum plate. Ball is yellow, plate is painted by Density, showing density changes.

- **Y Normal**.
- **Apply**.
- Unselect the **Show Plane**.
- Set **Coloring** to **pres**.
- **View** → **Color Map Editor** → **Presets** (the little envelope with a heart) → **Turbo**.
- With the mouse, rotate the slices around so you can see both surfaces.
- **Edit** → **Reset Session**. There is also a shortcut icon just above where you have been changing colors. It looks like a green counterclockwise snake eating it's tail.


Properties (Slice2)

Slice Type: Plane

Plane Parameters

□ Show Plane

Origin: 0, 0, 0.0799999

Normal: 0, 1, 0

Note: Use 'P' to pick 'Origin' on mesh or 'Ctrl+P' to snap to the closest mesh point

Offset: 0

□ Crinkle slice

☑ Triangulate the slice

☑ Merge duplicated points in the slice

Value Range: [-12.9511, 12.9511]

Display (GeometryRepresentation)

Representation: Surface

Coloring: pres

## Stream Tracers - lines and tubes

- Open disk_out_ref.ex2.
- **Apply**.
- **+X**
- Select **Filters** → **Common** → **Stream Tracer**.
- **v**.
- Set **Seed Type** to **Point Cloud**.
- Uncheck **Show Sphere**.
- Set **Coloring** to **v**.
- **Apply**.

Lines don't color as nicely as surfaces. Lets add a tube filter around each streamline.

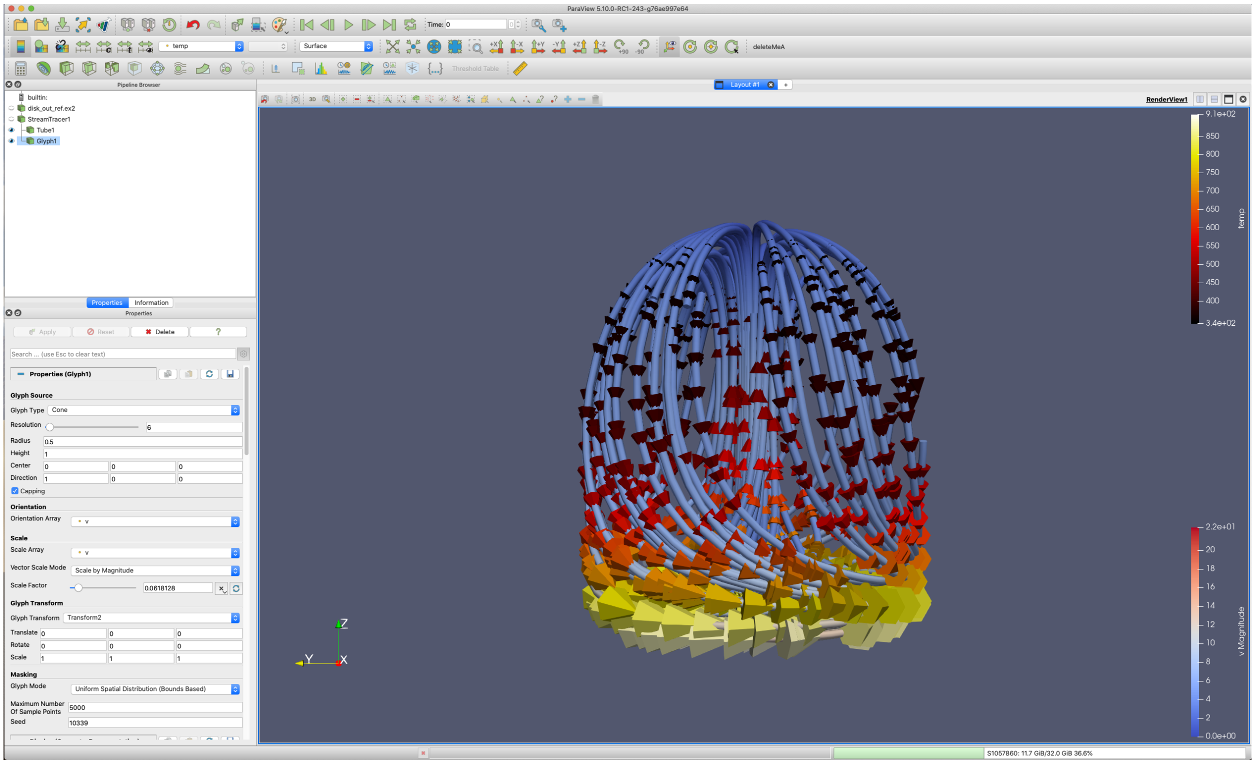
- Select **Filters** → **Search**
- Type **Tube**.
- **Apply**.

Now, we want to know which directions the particles are moving. We will use a glyph filter. Note we place the glyph filter on the streamline, not the tube.

- Select **StreamTracer1** in the Pipeline Browser.
- Select **Filters** → **Common** → **Glyph**.
- Set **Glyph Type** to **Cone**.
- Set **Orientation Array** to **v**.
- Set **Scale Array** to **v**.
- **!Very Important!**: In the **Scale Factor** select the recycle button to the right.
- **Apply**.
- Set **Coloring** to **temp**.
- **View** → **Color Map Editor** → **Presets** (the little envelope with a heart) → **Black Body Radiation**.

Lets save this really cool image as a screenshot.

- **File** → **Save Screenshot**.
- Add a file name.
- **OK**.
- **OK**.
- **Edit** → **Reset Session**.



Properties (Glyph1)

Glyph Source

Glyph Type: Cone
Resolution: 6
Radius: 0.5
Height: 1
Center: 0 0 0
Direction: 1 0 0
☑ Capping

Orientation

Orientation Array: v

Scale

Scale Array: v
Vector Scale Mode: Scale by Magnitude
Scale Factor: 0.0618128

Glyph Transform

Glyph Transform: Transform2
Translate: 0 0 0
Rotate: 0 0 0
Scale: 1 1 1

Masking

Glyph Mode: Uniform Spatial Distribution (Bounds Based)
Maximum Number Of Sample Points: 5000
Seed: 10339

### Stream Tracers with Custom Source

We want to create stream tracers from any arbitrary source. This can be a line, spline, circle, elipse or any other curving line. An extreme example would be a cylinder cut by a plane.

- Open disk_out_ref.ex2.
- **Apply**.
- **+X**
- Set **vtkBlockColors** to **Solid Color**.
- Set **Opacity** to **0.3**.
- Sources/Alphabetical/**Elipse**.
- Set **Center** to **0,0,7**.
- Set **Major Radius Vector** to **3,0,0**.
- Set **Ratio** to **0.3**.
- **Apply**.
- Select **Filters** → **Search**
- Type **Tube**.
- **Apply**.
- Select **disk_out_ref.ex2** in the **Pipeline Browser**.
- Select **Filters** → **Alphabetical** → **Stream Tracer with Custom Source**.
- Set **Seed Source** to **Elipse**.
- **Apply**.
- Set **Coloring** to **Solid Color**.

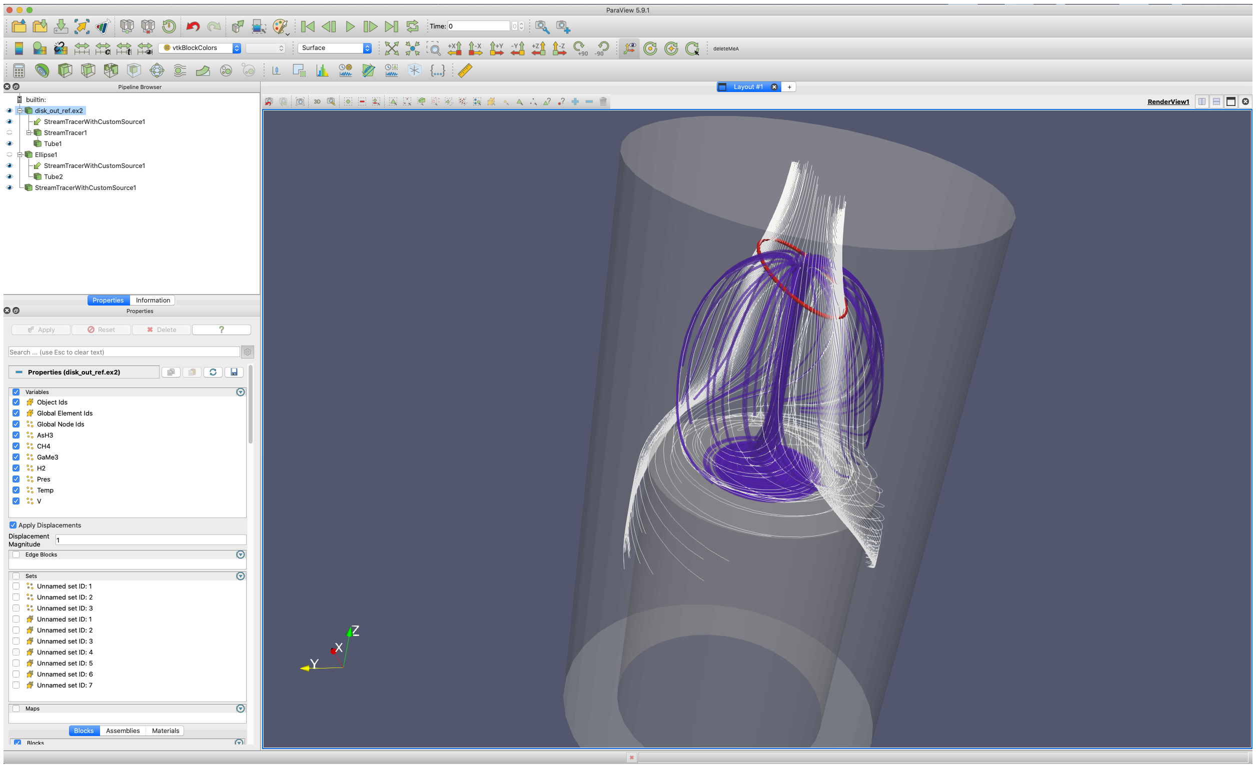
The image below is a merging of Stream Tracers with lines and tubes, and Stream Tracer with Custom Source. I have also played with colors to make it look nicer. If interested, replicating is left to the user.

### Glyphs perpendicular to a slice

- **Edit** → **Reset Session**.
- Open disk_out_ref.ex2.
- **Apply**.
- **+X**.

Lets create a half slice. This will be used as the seed plane for glyphs.

- Select **Filters** → **Common** → **Slice**.
- **Z Normal**.
- Set Origin to **0, 0, 5**.
- Uncheck **Show Plane**.
- **Apply**.
- Select **Filters** → **Common** → **Clip**.



Properties (disk_out_ref.ex2)

- ☑ Variables
  - ☑ Object Ids
  - ☑ Global Element Ids
  - ☑ Global Node Ids
  - ☑ AsH3
  - ☑ CH4
  - ☑ GaMe3
  - ☑ H2
  - ☑ Pres
  - ☑ Temp
  - ☑ V
- ☑ Apply Displacements
- Displacement Magnitude: 1
- ☐ Edge Blocks
- ☐ Sets
  - ☐ Unnamed set ID: 1
  - ☐ Unnamed set ID: 2
  - ☐ Unnamed set ID: 3
  - ☐ Unnamed set ID: 1
  - ☐ Unnamed set ID: 2
  - ☐ Unnamed set ID: 3
  - ☐ Unnamed set ID: 4
  - ☐ Unnamed set ID: 5
  - ☐ Unnamed set ID: 6
  - ☐ Unnamed set ID: 7
- ☐ Maps

Blocks | Assemblies | Materials

- ☑ Blocks

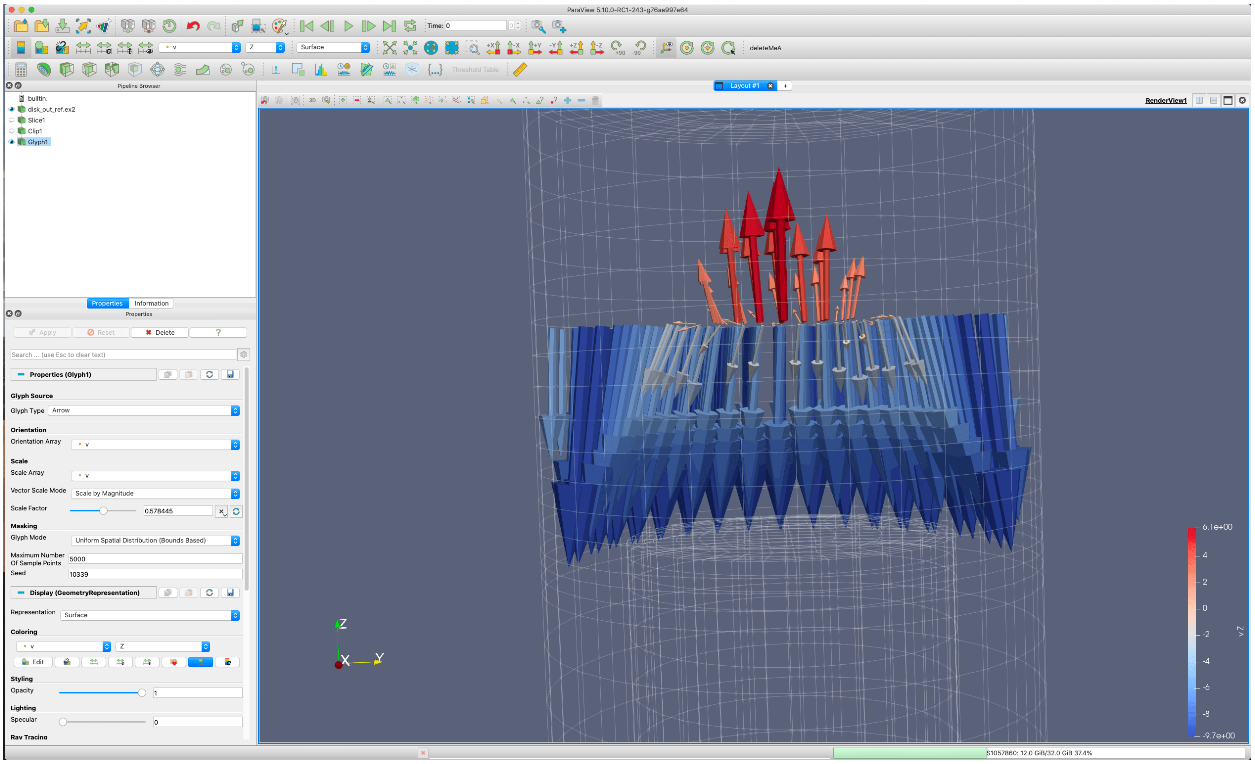
- Uncheck **Show Plane**.
- **Apply**.

Now, apply glyphs.

- Select **Filters** → **Common** → **Glyph**.
- Set **Glyph Type** to **Arrow**.
- Set **Orientation Array** to **v**.
- Set **Scale Array** to **v**.
- **!Very Important!**: In **Scale Factor** select the recycle button to the right.
- **.5X**.
- **.5X**.
- **Apply**.
- Set **Coloring** to **v**.
- Change **Magnitude** to **Z**.

Let's put these glyphs back into context by showing the original dataset.

- Select **disk_out_ref.ex2** in the Pipeline Browser.
- Set **Representation** to **Wireframe**.
- In the **pipeline browser**, click on the **eyeball** next to **disk_out_ref.ex2**.
- Set **Opacity** to **0.3**.
- Use the mouse to zoom into the glyph vectors.



ParaView 5.10.0-RC1-243-g76ae997e64

Pipeline Browser: builtin: / disk_out_ref.ex2 / Slice1 / Clip1 / Glyph1

Properties (Glyph1)

Glyph Source
Glyph Type: Arrow

Orientation
Orientation Array: v

Scale
Scale Array: v
Vector Scale Mode: Scale by Magnitude
Scale Factor: 0.578445

Masking
Glyph Mode: Uniform Spatial Distribution (Bounds Based)
Maximum Number Of Sample Points: 5000
Seed: 10339

Display (GeometryRepresentation)
Representation: Surface

Coloring: v, Z

Styling
Opacity: 1

Lighting
Specular: 0

## Flow in a fluid

To show a velocity profile we need to sample the dataset with a line, and then create glyphs off of this line. This can be done using a trick in ParaView, i.e., the Plot over Line filter. Note that a Resample to Line filter will be added in ParaView 5.11 or so.

- **Edit** → **Reset Session**.
- Open disk_out_ref.ex2.
- **Apply**.

Lets sample over a line.

- Select **Filters** → **Data Analysis** → **Plot over Line**.
- **Y Axis**.
- Change the **Z** component of **Point1** and **Point2** to **1**.
- Change the **Resolution** to **40**.
- **Apply**.
- Close the **LineChartView**.
- In the **Pipeline Browser**, turn visibility off for **disk_out_ref.ex2**.

We now have a line sampled through the fluid. Lets calculate the negative Z component of V (so it goes the opposite direction on the line from V). That way we can have two profiles, one with V, and one with Vz.

- Set **Filters** → **Common** → **Calculator**.
- Change **Result** to **Vz**.
- Set **Expression** to **0*iHat+0*jHat+-v_Z*kHat**.

- **Apply**.

Now we want to create two Glyphs - one from the **Calculator** filter, and one directly from the **Plot over Line** filter.

- **Calculator1** should still be highlighted in the Pipeline Browser.
- Select **Filters** → **Common** → **Glyph**.
- Set **Glyph Type** to **Arrow**.
- Set **Orientation Array** to **Vz**.
- Set **Scale Array** to **Vz**.
- **!Very Important!**: In **Scale Factor** select the recycle button to the right.
- **.5X**.
- **.5X**.
- **Apply**.
- Click on the **Color Editor** icon.
- Change the color to Orange.
- **Apply**.
- In the Pipeline Browser select the **Plot over Line** filter.
- Select **Filters** → **Common** → **Glyph**.
- Set **Glyph Type** to **Arrow**.
- Set **Orientation Array** to **v**.
- Set **Scale Array** to **v**.
- **!Very Important!**: In **Scale Factor** select the recycle button to the right.
- **Apply**.

Let's put these glyphs back into context by showing the original dataset.

- Select **disk_out_ref.ex2** in the Pipeline Browser.
- Click on the **eyeball** next to **disk_out_ref.ex2**.
- Set **Opacity** to **0.3**.

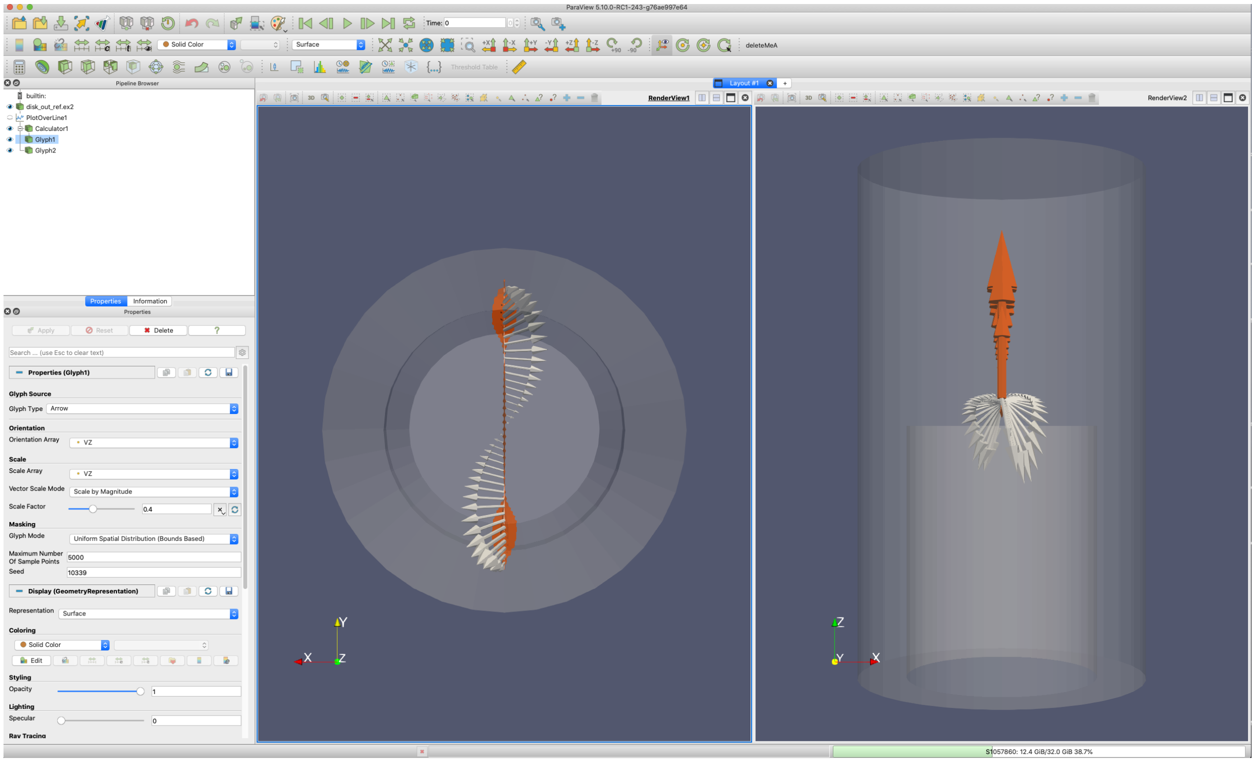
Just to create a nice image, I'm going to split the views horizontally, and show this visualization also from the side.

You can write the data sampled down the line to a .csv file, where you can post process it with tools such as Excel. Here is how to do it.

- Select **PlotOverLine** in the Pipeline Browser.
- Split screen vertical.
- **Spreadsheet view**.
- Now, click on the **Export Scene** icon, and write the **Spreadsheet** down to a **.csv file**.

Lets save this really cool image as a screenshot.

- **File** → **Save Screenshot**.
- Add a file name.
- **OK**.


Pipeline Browser

builtin:
disk_out_ref.ex2
PlotOverLine1
Calculator1
Glyph1
Glyph2

Properties (Glyph1)

Glyph Source
Glyph Type: Arrow

Orientation
Orientation Array: VZ

Scale
Scale Array: VZ
Vector Scale Mode: Scale by Magnitude
Scale Factor: 0.4

Masking
Glyph Mode: Uniform Spatial Distribution (Bounds Based)
Maximum Number Of Sample Points: 5000
Seed: 10339

Display (GeometryRepresentation)

Representation: Surface

Coloring
Solid Color

Styling
Opacity: 1

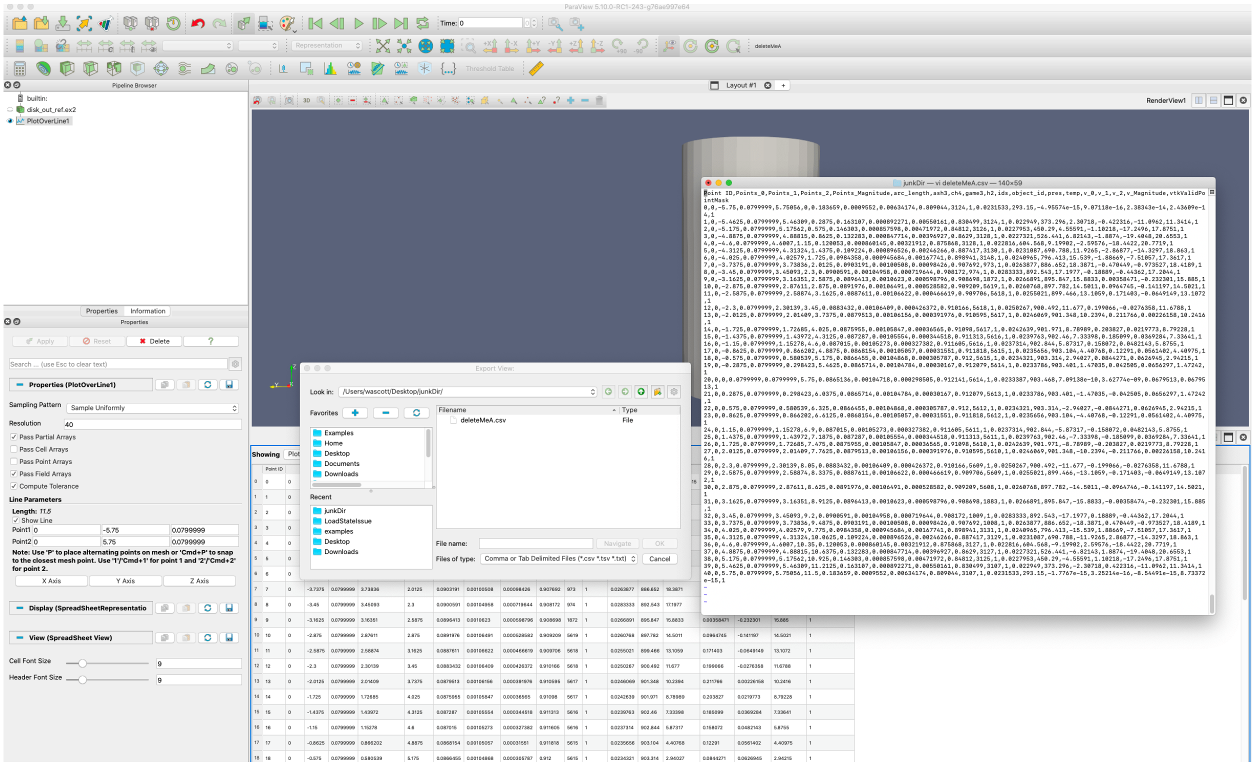
Lighting
Specular: 0

Ray Tracing

- We want to save both views. Click **Save All Views**.
- **OK**.

## 2D plots through a fluid

- **Edit** → **Reset Session**.
- Open disk_out_ref.ex2.
- **Apply**.
- **+X**.
- Select **Filters** → **Data Analysis** → **Plot over Line**.
- **Y Axis**.
- Change the **Z** component of **Point1** and **Point2** to **4**.
- **Apply**.
- In the **Properties Tab**, turn all variables off other than **v_Z**.
- Click on the **RenderView**, the left **view**.
- Select **disk_out_ref.ex2** in the Pipeline Browser.
- Set **Opacity** to **0.3**.



Look in: /Users/wascott/Desktop/junkDir/

File name:

Files of type: Comma or Tab Delimited Files (*.csv *.tsv *.txt)

Properties (PlotOverLine1)

Sampling Pattern: Sample Uniformly

Resolution: 40

☑ Pass Partial Arrays
□ Pass Cell Arrays
□ Pass Point Arrays
☑ Pass Field Arrays
☑ Compute Tolerance

Line Parameters

Length: *11.5*

☑ Show Line

| | | | |
|---|---|---|---|
| Point1 | 0 | -5.75 | 0.0799999 |
| Point2 | 0 | 5.75 | 0.0799999 |

Note: Use 'P' to place alternating points on mesh or 'Cmd+P' to snap to the closest mesh point. Use '1'/'Cmd+1' for point 1 and '2'/'Cmd+2' for point 2.

Display (SpreadSheetRepresentatio

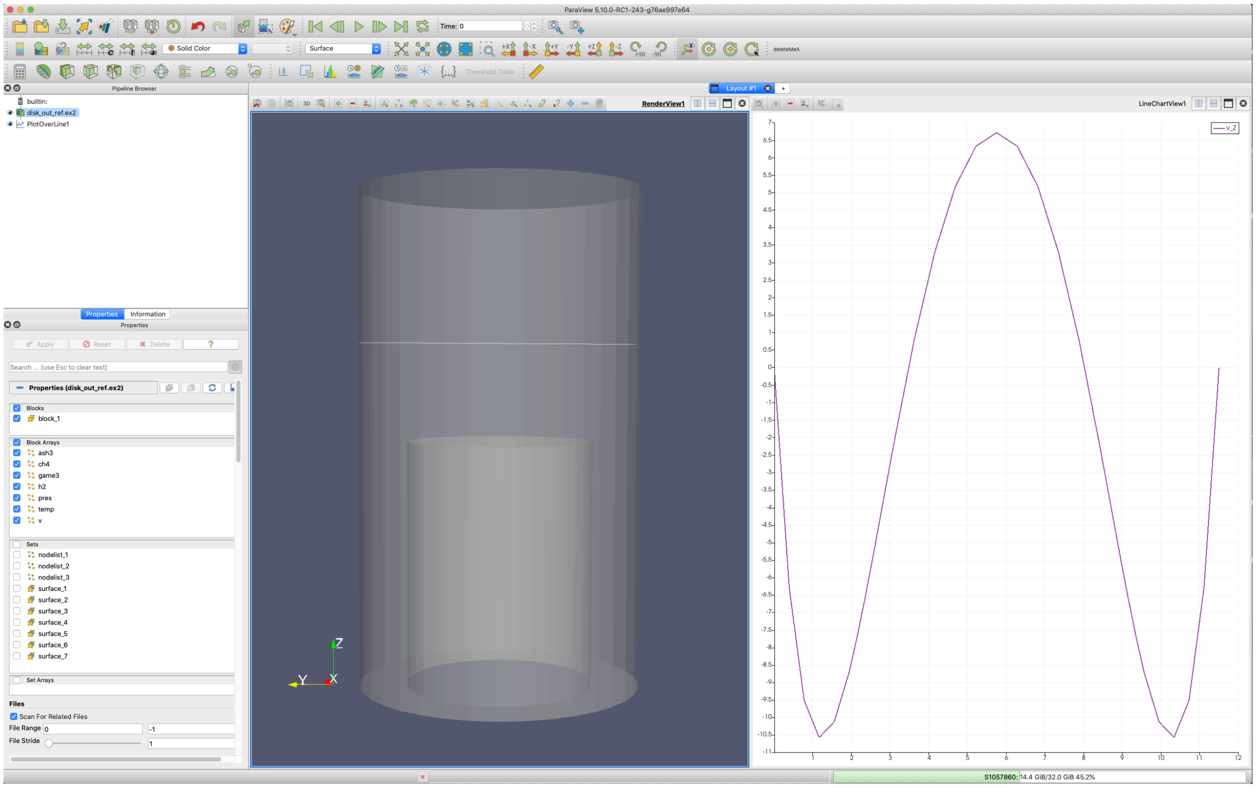
View (SpreadSheet View)

Cell Font Size: 9

Header Font Size: 9

junkDir — vi deleteMeA.csv — 140×59

```
Point ID,Points_0,Points_1,Points_2,Points_Magnitude,arc_length,ash3,ch4,game3,h2,ids,object_id,pres,temp,v_0,v_1,v_2,v_Magnitude,vtkValidPo
intMask
0,0,-5.75,0.0799999,5.75056,0,0.183659,0.0009552,0.00634174,0.809044,3124,1,0.0231533,293.15,-4.95574e-15,9.07118e-16,2.38343e-14,2.43609e-1
4,1
1,0,-5.4625,0.0799999,5.46309,0.2875,0.163107,0.000892271,0.00550161,0.830499,3124,1,0.022949,373.296,2.30718,-0.422316,-11.0962,11.3414,1
2,0,-5.175,0.0799999,5.17562,0.575,0.146303,0.000857598,0.00471972,0.84812,3126,1,0.0227953,450.29,4.55591,-1.10218,-17.2496,17.8751,1
3,0,-4.8875,0.0799999,4.88815,0.8625,0.132283,0.000847714,0.00396927,0.8629,3128,1,0.0227321,526.441,6.82143,-1.8874,-19.4048,20.6553,1
4,0,-4.6,0.0799999,4.6007,1.15,0.120053,0.000860145,0.00321912,0.875868,3128,1,0.022816,604.568,9.19902,-2.59576,-18.4422,20.7719,1
5,0,-4.3125,0.0799999,4.31324,1.4375,0.109224,0.000896526,0.00246266,0.887417,3130,1,0.0231087,690.788,11.9265,-2.86877,-14.3297,18.863,1
6,0,-4.025,0.0799999,4.02579,1.725,0.0984358,0.000945684,0.00167741,0.898941,3148,1,0.0240965,796.413,15.539,-1.88669,-7.51057,17.3617,1
7,0,-3.7375,0.0799999,3.73836,2.0125,0.0903191,0.00100508,0.00098426,0.907692,973,1,0.0263877,886.652,18.3871,-0.470449,-0.973527,18.4189,1
8,0,-3.45,0.0799999,3.45093,2.3,0.0900591,0.00104958,0.000719644,0.908172,974,1,0.0283333,892.543,17.1977,-0.18889,-0.44362,17.2044,1
9,0,-3.1625,0.0799999,3.16351,2.5875,0.0896413,0.0010623,0.000598796,0.908698,1872,1,0.0266891,895.847,15.8833,0.00358471,-0.232301,15.885,1
10,0,-2.875,0.0799999,2.87611,2.875,0.0891976,0.00106491,0.000528582,0.909209,5619,1,0.0260768,897.782,14.5011,0.0964745,-0.141197,14.5021,1
11,0,-2.5875,0.0799999,2.58874,3.1625,0.0887611,0.00106622,0.000466619,0.909706,5618,1,0.0255021,899.466,13.1059,0.171403,-0.0649149,13.1072
,1
12,0,-2.3,0.0799999,2.30139,3.45,0.0883432,0.00106409,0.000426372,0.910166,5618,1,0.0250267,900.492,11.677,0.199066,-0.0276358,11.6788,1
13,0,-2.0125,0.0799999,2.01409,3.7375,0.0879513,0.00106156,0.000391976,0.910595,5617,1,0.0246069,901.348,10.2394,0.211766,0.00226158,10.2416
,1
14,0,-1.725,0.0799999,1.72685,4.025,0.0875955,0.00105847,0.00036565,0.91098,5617,1,0.0242639,901.971,8.78989,0.203827,0.0219773,8.79228,1
15,0,-1.4375,0.0799999,1.43972,4.3125,0.087287,0.00105554,0.000344518,0.911313,5616,1,0.0239763,902.46,7.33398,0.185099,0.0369284,7.33641,1
16,0,-1.15,0.0799999,1.15278,4.6,0.087015,0.00105273,0.000327382,0.911605,5616,1,0.0237314,902.844,5.87317,0.158072,0.0482143,5.8755,1
17,0,-0.8625,0.0799999,0.866202,4.8875,0.0868154,0.00105057,0.00031551,0.911818,5615,1,0.0235656,903.104,4.40768,0.12291,0.0561402,4.40975,1
18,0,-0.575,0.0799999,0.580539,5.175,0.0866455,0.00104868,0.000305787,0.912,5615,1,0.0234321,903.314,2.94027,0.0844271,0.0626945,2.94215,1
19,0,-0.2875,0.0799999,0.298423,5.4625,0.0865714,0.00104784,0.00030167,0.912079,5614,1,0.0233786,903.401,1.47035,0.042505,0.0656297,1.47242,
1
20,0,0,0.0799999,0.0799999,5.75,0.0865136,0.00104718,0.000298505,0.912141,5614,1,0.0233387,903.468,7.09138e-10,3.62774e-09,0.0679513,0.06795
13,1
21,0,0.2875,0.0799999,0.298423,6.0375,0.0865714,0.00104784,0.00030167,0.912079,5613,1,0.0233786,903.401,-1.47035,-0.042505,0.0656297,1.47242
,1
22,0,0.575,0.0799999,0.580539,6.325,0.0866455,0.00104868,0.000305787,0.912,5612,1,0.0234321,903.314,-2.94027,-0.0844271,0.0626945,2.94215,1
23,0,0.8625,0.0799999,0.866202,6.6125,0.0868154,0.00105057,0.00031551,0.911818,5612,1,0.0235656,903.104,-4.40768,-0.12291,0.0561402,4.40975,
1
24,0,1.15,0.0799999,1.15278,6.9,0.087015,0.00105273,0.000327382,0.911605,5611,1,0.0237314,902.844,-5.87317,-0.158072,0.0482143,5.8755,1
25,0,1.4375,0.0799999,1.43972,7.1875,0.087287,0.00105554,0.000344518,0.911313,5611,1,0.0239763,902.46,-7.33398,-0.185099,0.0369284,7.33641,1
26,0,1.725,0.0799999,1.72685,7.475,0.0875955,0.00105847,0.00036565,0.91098,5610,1,0.0242639,901.971,-8.78989,-0.203827,0.0219773,8.79228,1
27,0,2.0125,0.0799999,2.01409,7.7625,0.0879513,0.00106156,0.000391976,0.910595,5610,1,0.0246069,901.348,-10.2394,-0.211766,0.00226158,10.241
6,1
28,0,2.3,0.0799999,2.30139,8.05,0.0883432,0.00106409,0.000426372,0.910166,5609,1,0.0250267,900.492,-11.677,-0.199066,-0.0276358,11.6788,1
29,0,2.5875,0.0799999,2.58874,8.3375,0.0887611,0.00106622,0.000466619,0.909706,5609,1,0.0255021,899.466,-13.1059,-0.171403,-0.0649149,13.107
2,1
30,0,2.875,0.0799999,2.87611,8.625,0.0891976,0.00106491,0.000528582,0.909209,5608,1,0.0260768,897.782,-14.5011,-0.0964746,-0.141197,14.5021,
1
31,0,3.1625,0.0799999,3.16351,8.9125,0.0896413,0.0010623,0.000598796,0.908698,1883,1,0.0266891,895.847,-15.8833,-0.00358474,-0.232301,15.885
,1
32,0,3.45,0.0799999,3.45093,9.2,0.0900591,0.00104958,0.000719644,0.908172,1009,1,0.0283333,892.543,-17.1977,0.18889,-0.44362,17.2044,1
33,0,3.7375,0.0799999,3.73836,9.4875,0.0903191,0.00100508,0.00098426,0.907692,1008,1,0.0263877,886.652,-18.3871,0.470449,-0.973527,18.4189,1
34,0,4.025,0.0799999,4.02579,9.775,0.0984358,0.000945684,0.00167741,0.898941,3131,1,0.0240965,796.413,-15.539,1.88669,-7.51057,17.3617,1
35,0,4.3125,0.0799999,4.31324,10.0625,0.109224,0.000896526,0.00246266,0.887417,3129,1,0.0231087,690.788,-11.9265,2.86877,-14.3297,18.863,1
36,0,4.6,0.0799999,4.6007,10.35,0.120053,0.000860145,0.00321912,0.875868,3127,1,0.022816,604.568,-9.19902,2.59576,-18.4422,20.7719,1
37,0,4.8875,0.0799999,4.88815,10.6375,0.132283,0.000847714,0.00396927,0.8629,3127,1,0.0227321,526.441,-6.82143,1.8874,-19.4048,20.6553,1
38,0,5.175,0.0799999,5.17562,10.925,0.146303,0.000857598,0.00471972,0.84812,3125,1,0.0227953,450.29,-4.55591,1.10218,-17.2496,17.8751,1
39,0,5.4625,0.0799999,5.46309,11.2125,0.163107,0.000892271,0.00550161,0.830499,3107,1,0.022949,373.296,-2.30718,0.422316,-11.0962,11.3414,1
40,0,5.75,0.0799999,5.75056,11.5,0.183659,0.0009552,0.00634174,0.809044,3107,1,0.0231533,293.15,-1.7767e-15,3.25214e-16,-8.54491e-15,8.73372
e-15,1
~
~
~
```



Properties (disk_out_ref.ex2)

☑ Blocks
☑ block_1

☑ Block Arrays
☑ ash3
☑ ch4
☑ game3
☑ h2
☑ pres
☑ temp
☑ v

□ Sets
□ nodelist_1
□ nodelist_2
□ nodelist_3
□ surface_1
□ surface_2
□ surface_3
□ surface_4
□ surface_5
□ surface_6
□ surface_7

□ Set Arrays

Files

☑ Scan For Related Files

File Range: 0 -1

File Stride: 1

### Contours on a slice

- **Edit** → **Reset Session**.
- Open disk_out_ref.ex2.
- **Apply**.
- **+X**.
- Select Filters → Common → Slice**.
- Uncheck **Show Plane**.
- **Apply**.

We need the magnitude for the Contour filter.

- Select Filters → Common → Calculator**.
- Result Array Name **vMag**.
- Set **Expression** to **mag(v)**.
- **Apply**.

Now, draw contours on the 2d slice.

- Select **Filters** → **Common** → **Contour**.
- Contour by vMag.
- Delete the **Value**, and create a new set using the **Add a Range of Values** icon.
- **Apply**.

A nice visualization is to turn visibility on for **Slice** and paint by **v**, and change **Contour** to be a **Solid Color**, and make that color **White**.

Here is an example with additional **streamlines**, **tubes** and **glyphs**.

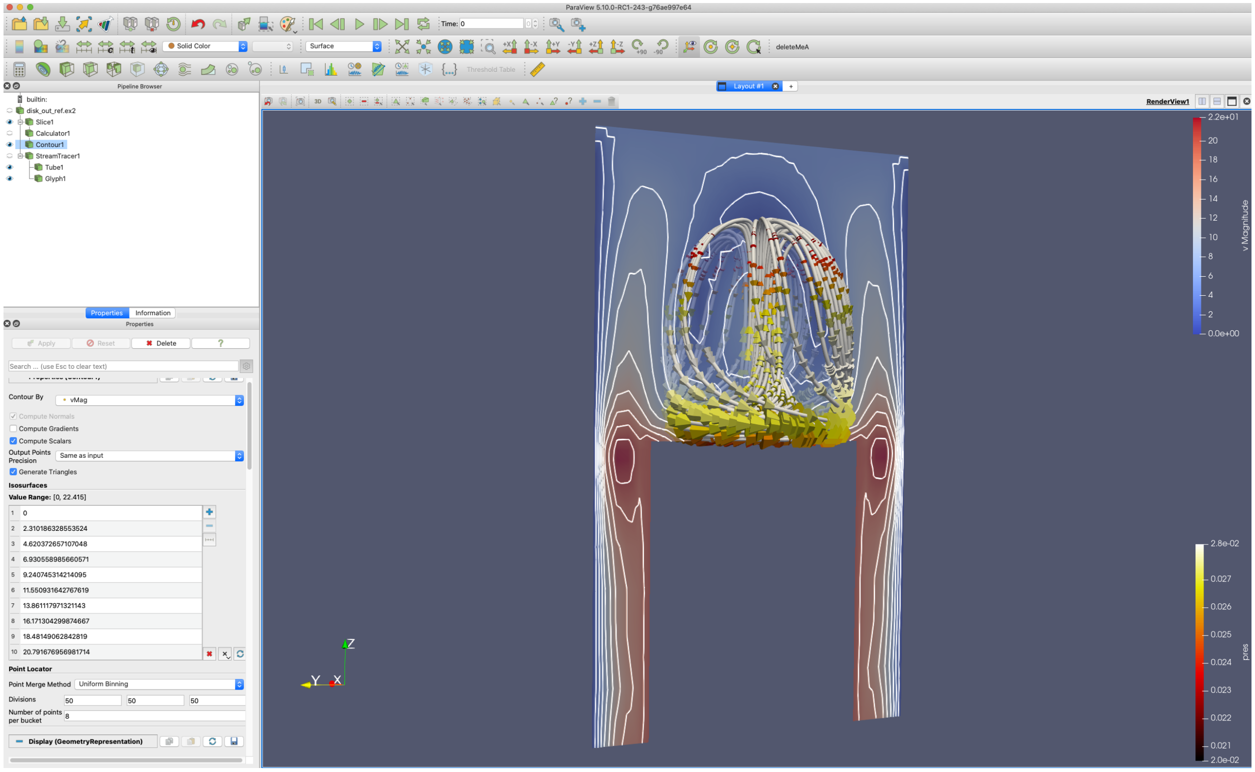
### Gradient, Divergence, Vorticity and Q Criterion

The **Gradient** filter (**Advanced** Properties tab) provides Gradient, Divergence, Vorticity and and Q Criterion. Here is am example, using disk_out_ref.ex2.

### Probing a fluid

There are numerous ways to probe the cells and points of a fluid. One is with the **Hover Points On** and **Hover Cells On** icons just above the Renderview. Another is with Interactive Select Cells or Points On. Then, in the Find Data, turn on Cell or Point Labels. Yet another is with the Probe filter. Here is how to use the probe filter.

- **Edit** → **Reset Session**.
- Open disk_out_ref.ex2.
- **Apply**.
- **+X**.
- Select **Filters** → **Common** → **Clip**.
- Uncheck **Show Plane**.
- Uncheck **Invert**.



Contour By: vMag

☑ Compute Normals
☐ Compute Gradients
☑ Compute Scalars

Output Points Precision: Same as input

☑ Generate Triangles

**Isosurfaces**

**Value Range:** [0, 22.415]

| | |
|---|---|
| 1 | 0 |
| 2 | 2.310186328553524 |
| 3 | 4.620372657107048 |
| 4 | 6.930558985660571 |
| 5 | 9.240745314214095 |
| 6 | 11.550931642767619 |
| 7 | 13.861117971321143 |
| 8 | 16.171304299874667 |
| 9 | 18.48149062842819 |
| 10 | 20.791676956981714 |

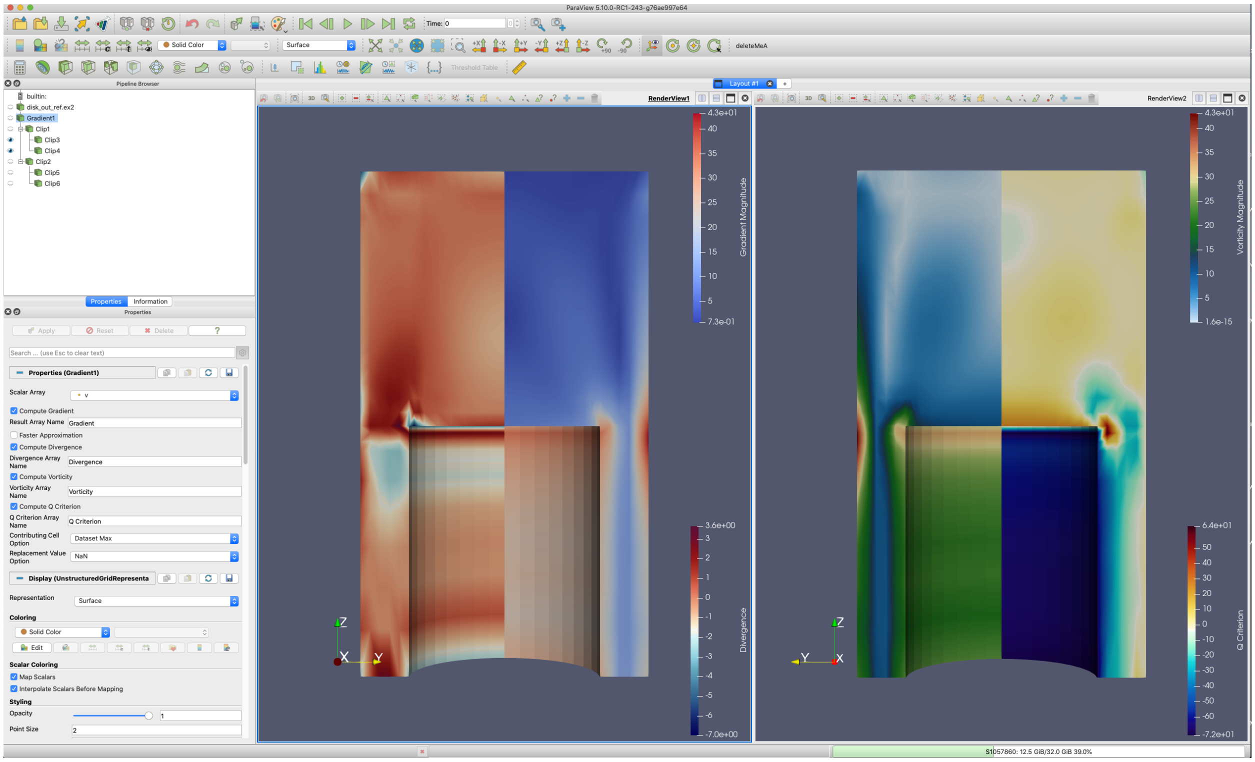
**Point Locator**

Point Merge Method: Uniform Binning

Divisions: 50 50 50

Number of points per bucket: 8

**Display (GeometryRepresentation)**



**Properties (Gradient1)**

Scalar Array: v

☑ Compute Gradient

Result Array Name: Gradient

☐ Faster Approximation
☑ Compute Divergence

Divergence Array Name: Divergence

☑ Compute Vorticity

Vorticity Array Name: Vorticity

☑ Compute Q Criterion

Q Criterion Array Name: Q Criterion

Contributing Cell Option: Dataset Max

Replacement Value Option: NaN

**Display (UnstructuredGridRepresenta**

Representation: Surface

**Coloring**

Solid Color

**Scalar Coloring**

☑ Map Scalars
☑ Interpolate Scalars Before Mapping

**Styling**

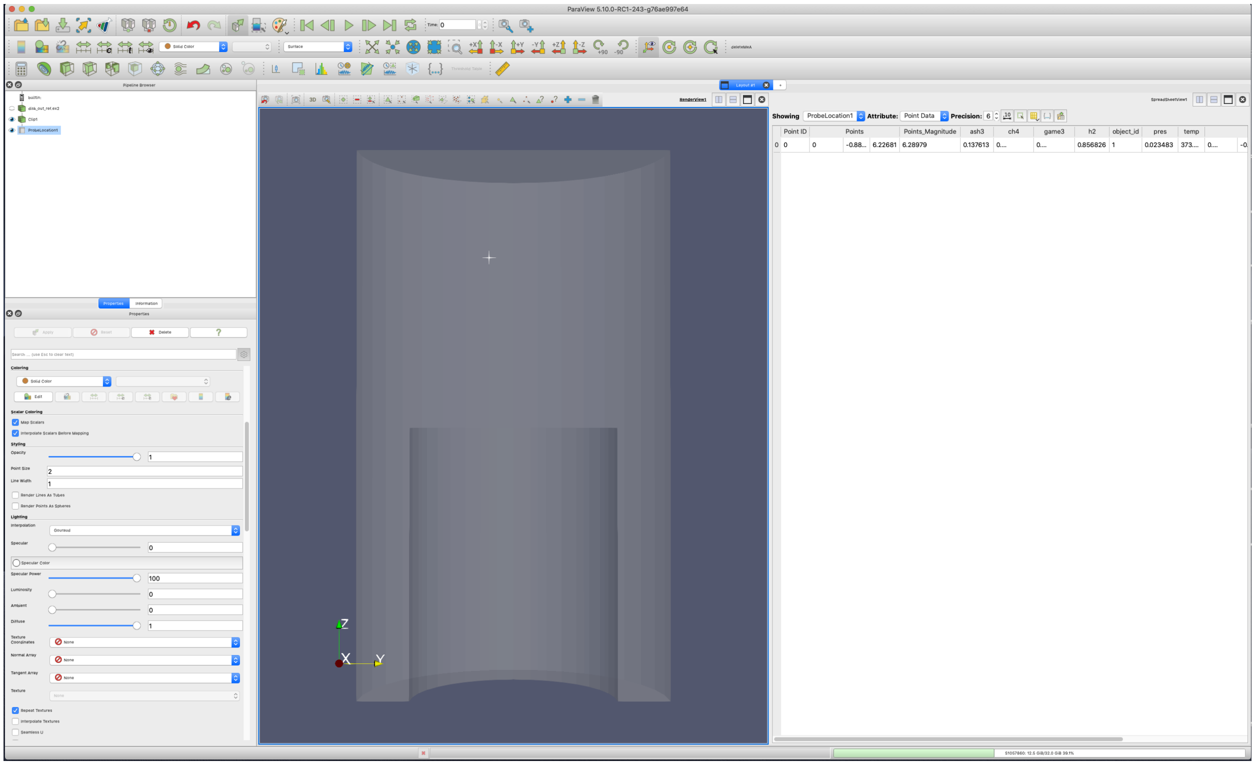
Opacity: 1

Point Size: 2

- **Apply**.

The **Probe** filter works much better with **Auto Apply** turned on. This is the icon that looks like a tree growing out of a cube.

- **Auto Apply** on
- Select **Filters** → **Data Analysis** → **Probe**.
- If needed, select the **RenderView** window, giving it focus.
- Now, move over **disk_out_ref.ex2**, updating the probed location with the **p** key. The **probed** data will show in the Spreadsheet view.



builtin:
disk_out_ref.ex2
Clip1
ProbeLocation1

**Showing** ProbeLocation1 **Attribute:** Point Data **Precision:** 6

| | Point ID | Points | | | Points_Magnitude | ash3 | ch4 | game3 | h2 | object_id | pres | temp | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | -0.88... | 6.22681 | 6.28979 | 0.137613 | 0.... | 0.... | 0.856826 | 1 | 0.023483 | 373.... | 0.... | -0. |

## Animating a static vector field

If you have a vector field in your data, you can animate a static dataset. Our goal is to create a set of streamlines from a vector field, place points on this set of streamlines, and animate the point down the streamlines. We will also add glyphs to the streamline points.

- **Edit** → **Reset Session**.
- Open disk_out_ref.ex2.
- **Apply**.
- Click the **-X** icon.
- **Stream tracer** filter. (We are already streamtracing on V).
- Set **Seed Type** to **Point Cloud**.
- Optional - change the Opacity to 0.4.
- **Apply**.

- Select **Filters** → **Common** → **Contour**.
- Contour on **IntegrationTime**.
- **Apply**.
- Select **Filters** → **Common** → **Glyph**.
- Vectors **V**.
- No Scale Array.
- Scale 1.
- Set **Glyph Mode** to **All Points**.
- **Apply**.
- Select **View** → **Animation View**.
- Set **Mode** to **Sequence**.
- Set **No. Frames** to **100**.
- Change the pulldown box next to the blue **+** to be **Contour**.
- Click the blue **+**. Note it works better if you use 0 for the start.
- Now, click the play button.
- In the pipeline browser, I also turned off visibility for the Contour filter.

Lets save this as a movie.

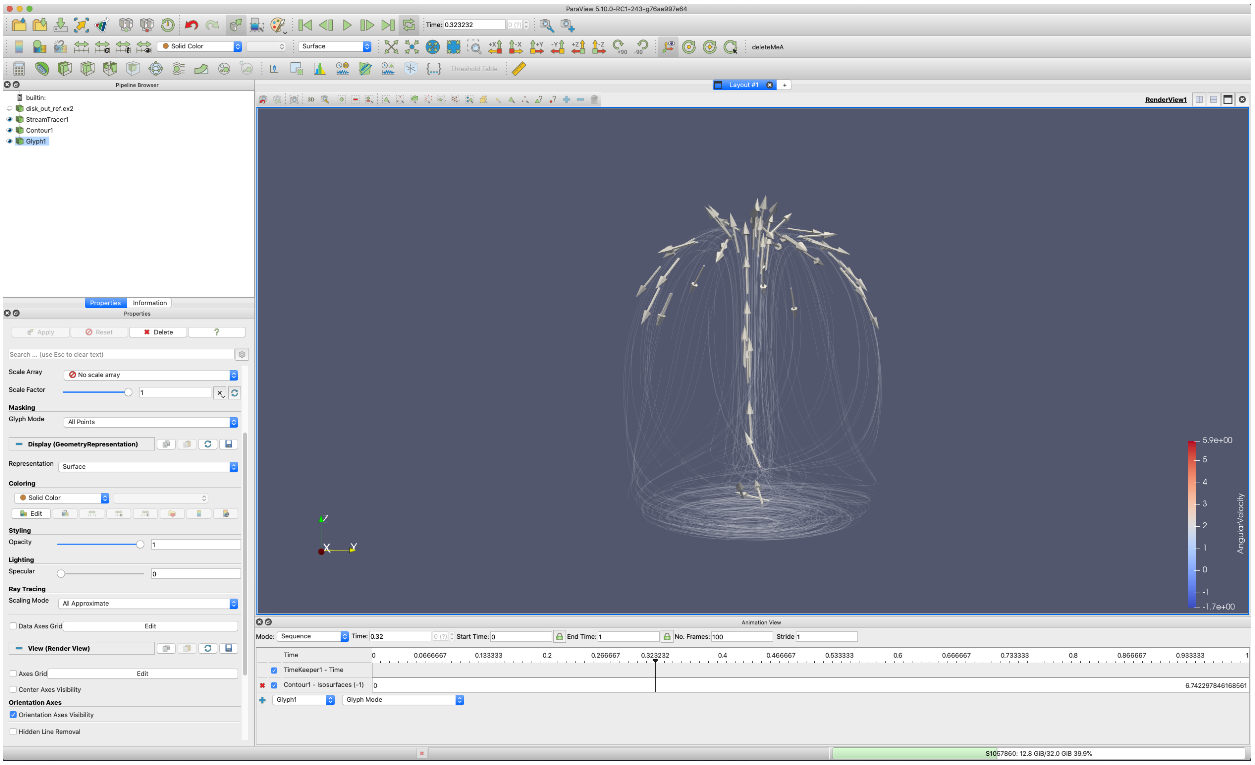
- **File** → **Save Animation**.
- Add a file name.
- Save as a **.avi**.
- **OK**.
- **OK**.

### Volume Rendering

We are going to paint the fluid by volume rendered temperature and we will try to make it look like fire. To give context, we are also going to extract the exterior surface, and clip the disk_out_ref in half. We will paint this exterior surface black.

Note: Volume rendering is very resource intensive. It is possible to display a dataset using surface that chokes using Volume Rendering. The solution is to grab more nodes of your cluster, thus picking up more memory.

- **Edit** → **Reset Session**.
- Open disk_out_ref.ex2
- **Apply**.
- Select **Filters** → **Alphabetical** → **Extract Surface**.
- **Apply**.
- Select **Filters** → **Common** → **Clip**.
- Deselect **Invert**.
- **Apply**.



Scale Array: No scale array
Scale Factor: 1
Masking
Glyph Mode: All Points
Representation: Surface
Coloring: Solid Color
Opacity: 1
Specular: 0
Scaling Mode: All Approximate
□ Data Axes Grid
□ Axes Grid
□ Center Axes Visibility
☑ Orientation Axes Visibility
□ Hidden Line Removal

Mode: Sequence  Time: 0.32  Start Time: 0  End Time: 1  No. Frames: 100  Stride 1

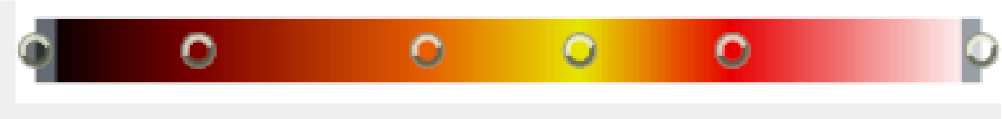
| Time | 0 | 0.0666667 | 0.133333 | 0.2 | 0.266667 | 0.323232 | 0.4 | 0.466667 | 0.533333 | 0.6 | 0.666667 | 0.733333 | 0.8 | 0.866667 | 0.933333 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ☑ TimeKeeper1 - Time | | | | | | | | | | | | | | | | |
| ☑ Contour1 - Isosurfaces (-1) | 0 | | | | | | | | | | | | | | | 6.742297846168561 |

Glyph1  Glyph Mode

- Set **Solid Color** to **black**.
- Set **Representation** to **Wireframe**.
- Open disk_out_ref.ex2 again.
- **Apply**.
- Set **Coloring** to **temp**.
- Set **Representation** to **Volume**.

Since the goal is to make it look like fire, we will finetune the **Color Map** settings.

- Select **View** → **Color Map Editor**
- Change **presets** (looks like a folder with a heart) to be **Black Body Radiation**.
- The **Color Transfer function** (the lower 1D colored line) should already have 4 points.
- Add a point in the orange area
- Add another at the top of the black.



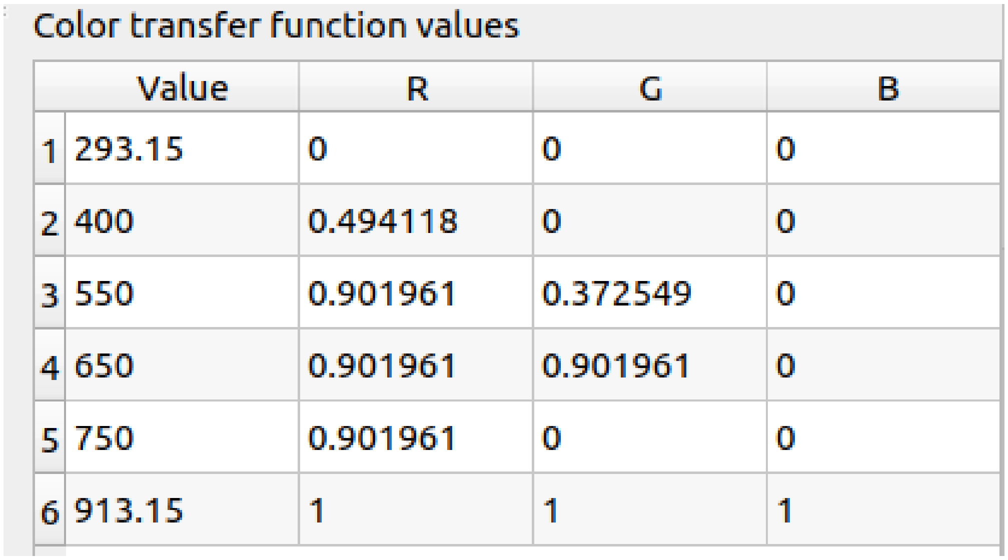
**Did you know?**

- You can create a point on the color scale by clicking in the window.
- You can select a point on the color scale by clicking on it.

- You can move between points using the **Tab** and **Shift** keys.
- You can delete points using the delete key.

---

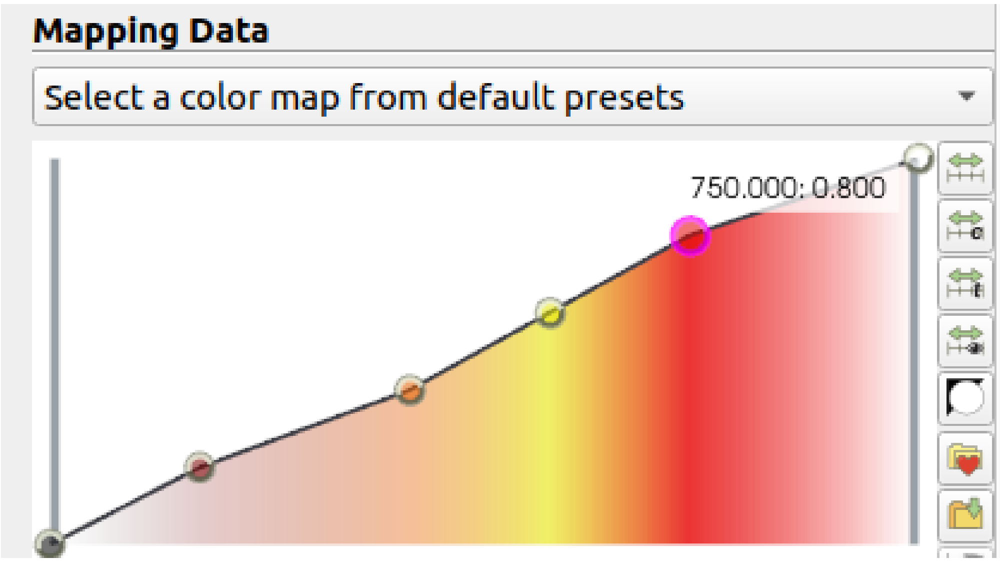
- The temperature should be set by the physical laws of black ody curve. Thus, manually specify the color transfer function values of the 6 points as follows by clicking the **advanced** button.



Color transfer function values

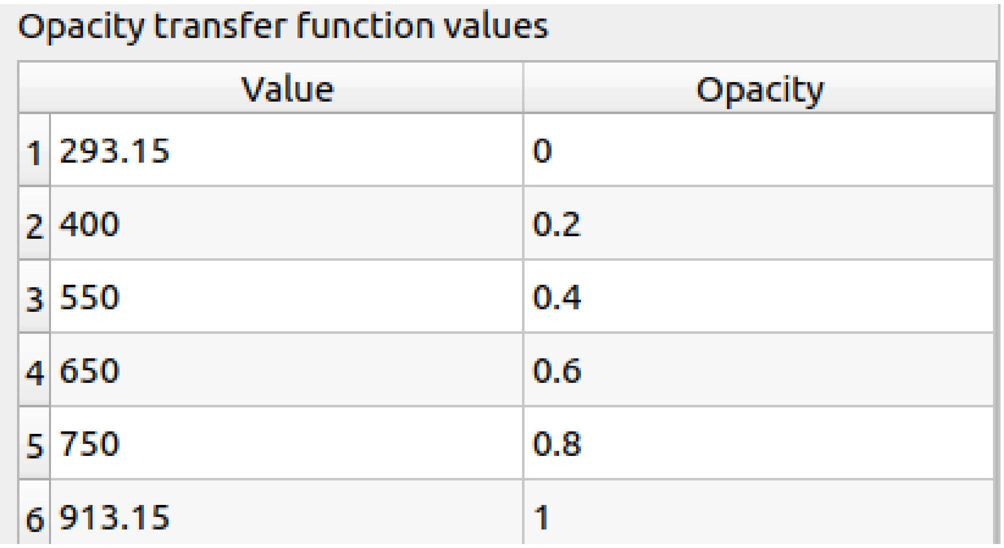
| | Value | R | G | B |
|---|---|---|---|---|
| 1 | 293.15 | 0 | 0 | 0 |
| 2 | 400 | 0.494118 | 0 | 0 |
| 3 | 550 | 0.901961 | 0.372549 | 0 |
| 4 | 650 | 0.901961 | 0.901961 | 0 |
| 5 | 750 | 0.901961 | 0 | 0 |
| 6 | 913.15 | 1 | 1 | 1 |

- The **Opacity Transfer function** (the upper 2D colored line) should already have 2 points.
- Add 4 extra points as seen bellow.



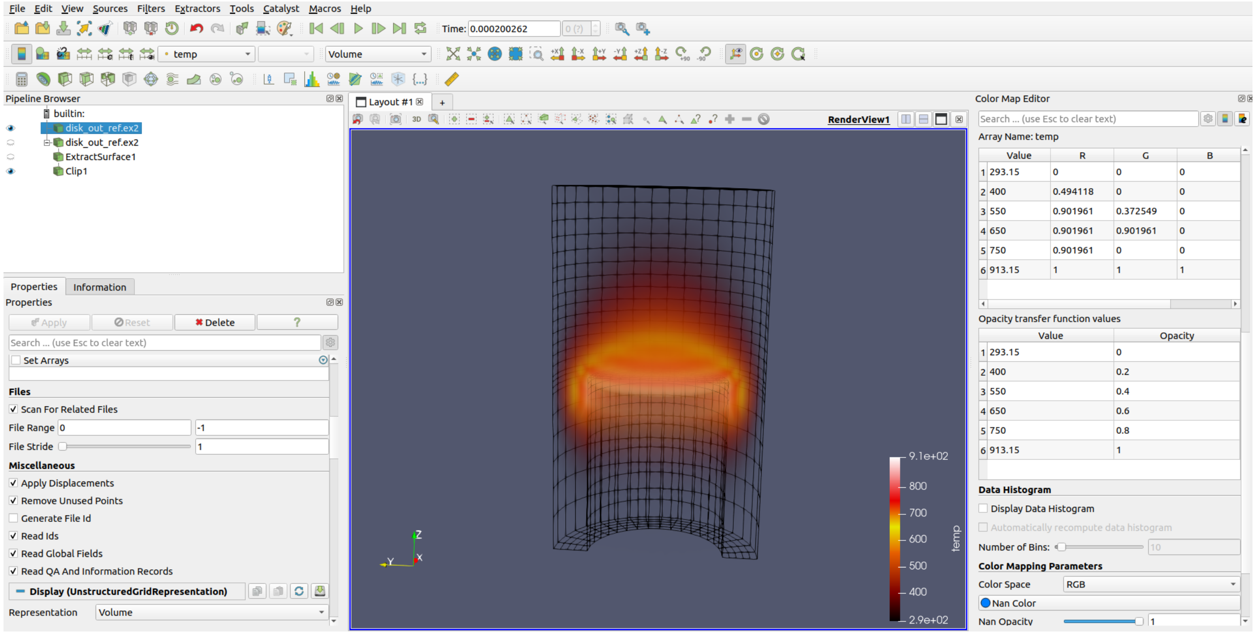
- The Opacity requires a bit of artistic license. What we are trying to do is show the different temperatures inside of the flame. Also, we may want to show differing amounts of soot - which will be point number1. Thus, manually specify the opacity transfer function values of the 6 points as follows.

The end result of the volume rendered fire should look like this:



Opacity transfer function values

|   | Value | Opacity |
|---|---|---|
| 1 | 293.15 | 0 |
| 2 | 400 | 0.2 |
| 3 | 550 | 0.4 |
| 4 | 650 | 0.6 |
| 5 | 750 | 0.8 |
| 6 | 913.15 | 1 |



Pipeline Browser

- builtin:
- disk_out_ref.ex2
- disk_out_ref.ex2
- ExtractSurface1
- Clip1

Properties | Information

Properties

Apply | Reset | Delete | ?

Search ... (use Esc to clear text)

□ Set Arrays

**Files**

☑ Scan For Related Files

File Range 0 -1

File Stride 1

**Miscellaneous**

☑ Apply Displacements

☑ Remove Unused Points

□ Generate File Id

☑ Read Ids

☑ Read Global Fields

☑ Read QA And Information Records

**Display (UnstructuredGridRepresentation)**

Representation Volume

Color Map Editor

Search ... (use Esc to clear text)

Array Name: temp

|   | Value | R | G | B |
|---|---|---|---|---|
| 1 | 293.15 | 0 | 0 | 0 |
| 2 | 400 | 0.494118 | 0 | 0 |
| 3 | 550 | 0.901961 | 0.372549 | 0 |
| 4 | 650 | 0.901961 | 0.901961 | 0 |
| 5 | 750 | 0.901961 | 0 | 0 |
| 6 | 913.15 | 1 | 1 | 1 |

Opacity transfer function values

|   | Value | Opacity |
|---|---|---|
| 1 | 293.15 | 0 |
| 2 | 400 | 0.2 |
| 3 | 550 | 0.4 |
| 4 | 650 | 0.6 |
| 5 | 750 | 0.8 |
| 6 | 913.15 | 1 |

**Data Histogram**

□ Display Data Histogram

□ Automatically recompute data histogram

Number of Bins: 10

**Color Mapping Parameters**

Color Space RGB

Nan Color

Nan Opacity 1

## 4.2.17 Targeted: Particle Simulations

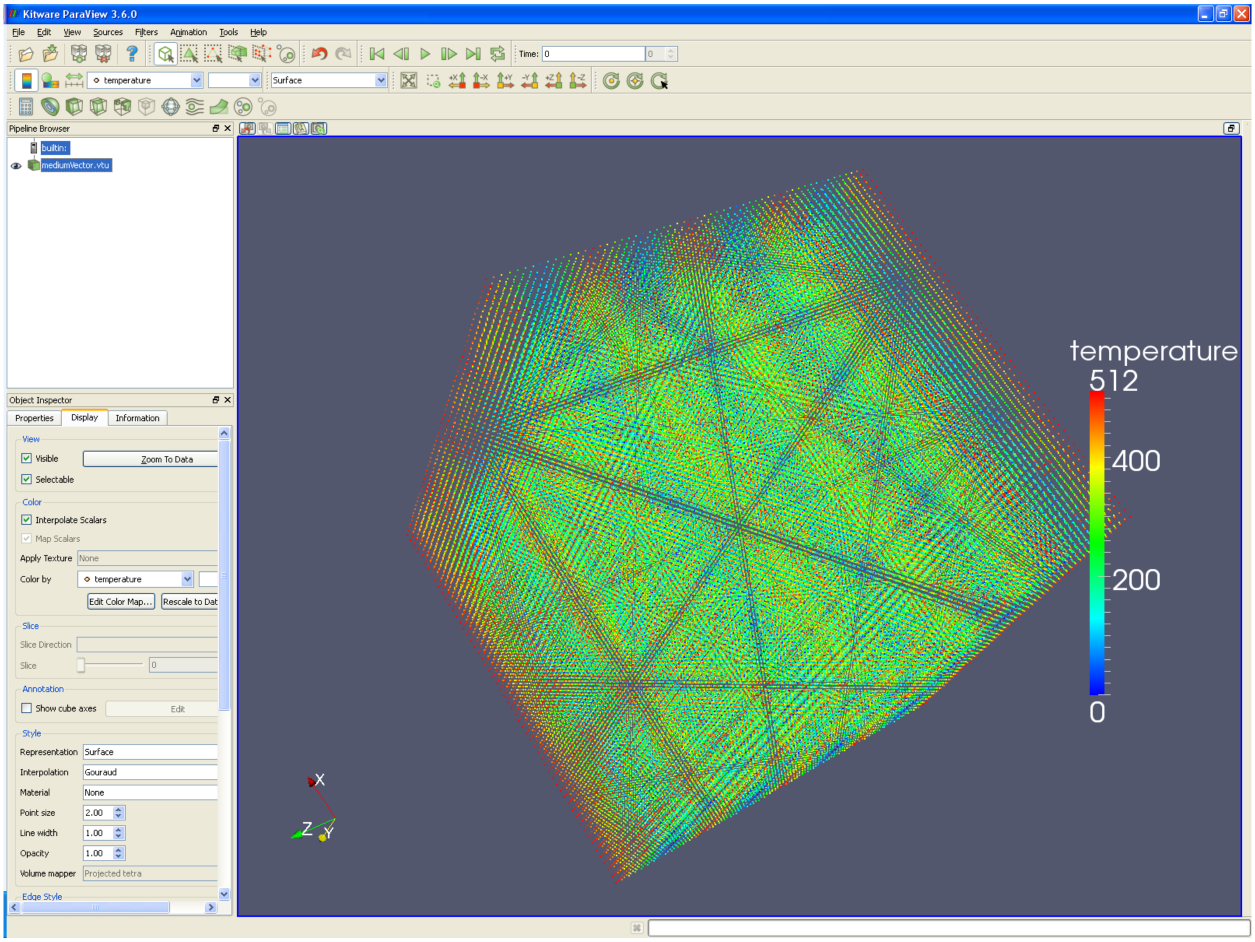
### Introduction

Particle simulations consist of point data, rather than cell or element data. An example may be electrons in a field, or water molecules in a fluid. Particle datasets can be combined with other datasets to show particle motion in a solid or liquid.

This tutorial uses a dataset named mediumVector.vtu. I will try to get it on the Kitware web site. If you want to make it yourself, it is just a vtk dataset, using the format listed at the end of this tutorial. I then read it into ParaView, and used the reflect filter in the X, then Y, then Z direction. I believe I then did the same reflections again. Finally, this file was written out to disk as a .vtu file.

### Displaying particles

- Run `paraview`.
- Open mediumVector.vtu.
- In the **Properties** tab, set the **Point size** to 2 (or 1, depending on your preference). The following picture holds about 250,000 points.

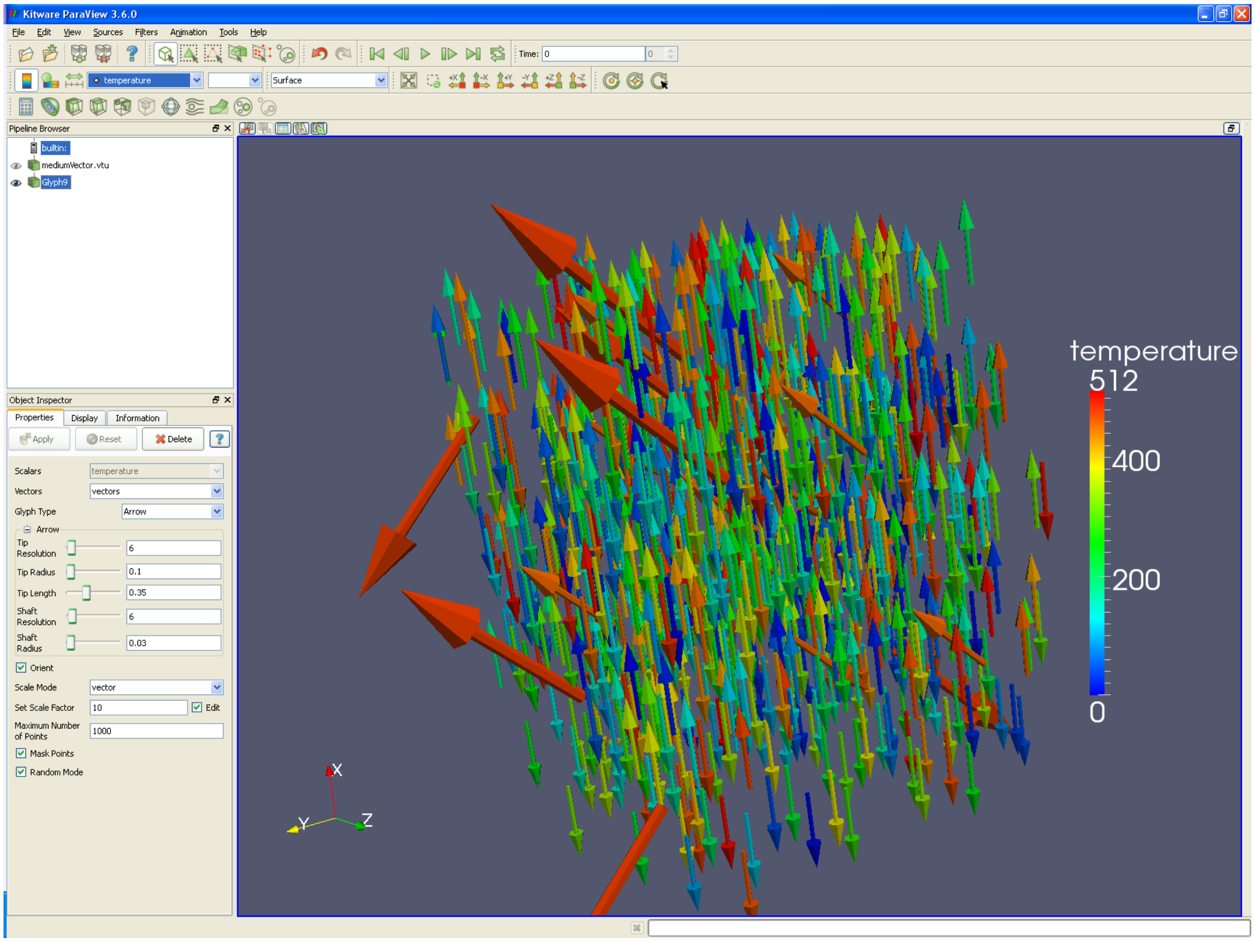


Time: 0

☑ Visible
☑ Selectable
☑ Interpolate Scalars
☑ Map Scalars
Apply Texture: None
Color by: temperature
Slice Direction:
Slice: 0
☐ Show cube axes
Representation: Surface
Interpolation: Gouraud
Material: None
Point size: 2.00
Line width: 1.00
Opacity: 1.00
Volume mapper: Projected tetra

### Glyphing particles

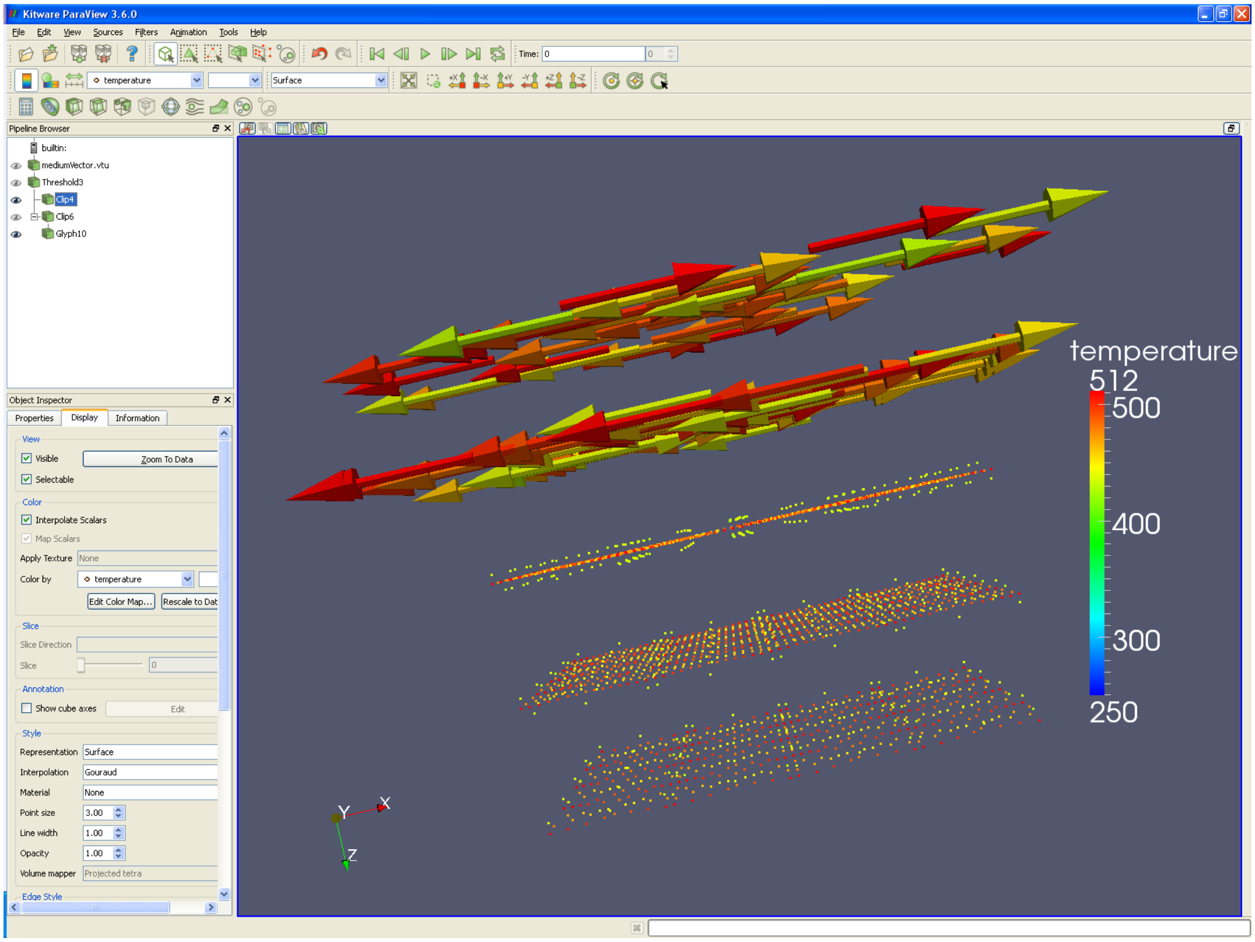
- You may want to replace these particles with spheres or arrows, using the glyph filter. The glyph filter will depopulate your dataset to a reasonable number of items, and this reasonable number is set by the **Maximum number of points** input box. **Random mode** often creates better looking output, but is not useful if you animate through numerous time steps. The following picture has arrows for particles, using vectors for direction and length, and colored by temperature.



Scalars: temperature
Vectors: vectors
Glyph Type: Arrow
Arrow
Tip Resolution: 6
Tip Radius: 0.1
Tip Length: 0.35
Shaft Resolution: 6
Shaft Radius: 0.03
☑ Orient
Scale Mode: vector
Set Scale Factor: 10 ☑ Edit
Maximum Number of Points: 1000
☑ Mask Points
☑ Random Mode

### Thresholding particles

- Another filter that is useful is the **Filters** → **Alphabetical** → **Threshold**. Using the **Threshold filter**, a user can select only those particles that are of interest. For instance, the following picture only displays particles that have a temperature of 430 to 511 degrees. Using the **Filters** → **Common** → **Clip**, the upper half of the picture has arrow glyphs, and the lower half has the particles themselves - all colored by temperature.



Time: 0

☑ Visible

☑ Selectable

☑ Interpolate Scalars

☑ Map Scalars

Apply Texture None

Color by temperature

Slice Direction

Slice 0

☐ Show cube axes

Representation Surface

Interpolation Gouraud

Material None

Point size 3.00

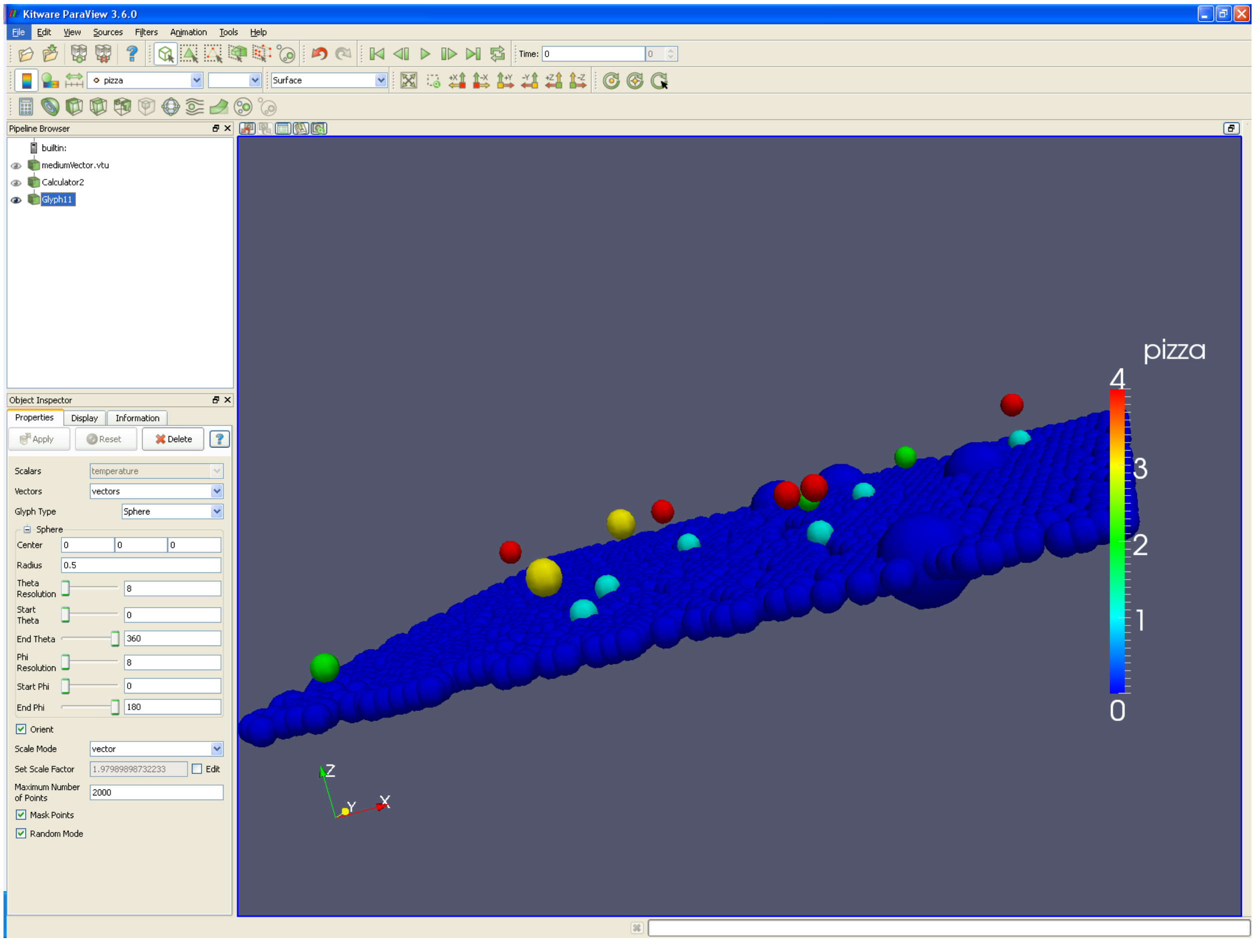
Line width 1.00

Opacity 1.00

Volume mapper Projected tetra

### Scatter Plot

- Finally, you can make a scatter plot through the **Filters** → **Common** → **Calculator**. For instance, if we want to preserve the X and Y coordinates of our points, but use number of pizzas for the Z component, we would do the following:
  - Select **Filters** → **Common** → **Calculator**.
  - Check the **Coordinate Results** check box.
  - Set **Expression** to **coordsX*iHat+coordsY*jHat+pizza*kHat**. This creates a vector, as follows: (X,Y,pizzas). (Think of the iHat as “This is an X”, the * as tying the coordinates and variables together, and the + sign as the comma between X,Y, and Z.)



Scalars: temperature
Vectors: vectors
Glyph Type: Sphere
⊟ Sphere
Center: 0 0 0
Radius: 0.5
Theta Resolution: 8
Start Theta: 0
End Theta: 360
Phi Resolution: 8
Start Phi: 0
End Phi: 180
☑ Orient
Scale Mode: vector
Set Scale Factor: 1.97989898732233 □ Edit
Maximum Number of Points: 2000
☑ Mask Points
☑ Random Mode

### Example data file

This data is written to a .vtk file. Note that I have deleted a lot of the data, which will need to be recreated. Also note that I added some random variable data - for instance, some locations have 1 or 2 pizzas. This actually creates a strange dataset - with many points per cell, but it worked for display purposes.

```
# vtk DataFile Version 2.0
Unstructured Grid Example
ASCII
```

(continues on next page)

(continued from previous page)

```
DATASET UNSTRUCTURED_GRID
POINTS 512 float
0 0 0  1 0 0  2 0 0  3 0 0  4 0 0  5 0 0  6 0 0  7 0 0
0 1 0  1 1 0  2 1 0  3 1 0  4 1 0  5 1 0  6 1 0  7 1 0
...
0 0 1  1 0 1  2 0 1  3 0 1  4 0 1  5 0 1  6 0 1  7 0 1
0 1 1  1 1 1  2 1 1  3 1 1  4 1 1  5 1 1  6 1 1  7 1 1
...
0 0 2  1 0 2  2 0 2  3 0 2  4 0 2  5 0 2  6 0 2  7 0 2
0 1 2  1 1 2  2 1 2  3 1 2  4 1 2  5 1 2  6 1 2  7 1 2
...
CELL_TYPES 52
1
1
...
POINT_DATA 512
SCALARS temperature float 1
LOOKUP_TABLE default
000.0 001.0 002.0 003.0 004.0 005.0 006.0 007.0 008.0 009.0
010.0 011.0 012.0 013.0 014.0 015.0 016.0 017.0 018.0 019.0
...
SCALARS pizza float 1
LOOKUP_TABLE default
000.0 000.0 000.0 000.0 000.0 000.0 000.0 000.0 000.0 000.0
000.0 000.0 000.0 000.0 000.0 000.0 000.0 000.0 000.0 000.0
...
VECTORS vectors float
1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0
1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0  1 0 0
...
```

## 4.2.18 Targeted: ParaView Web

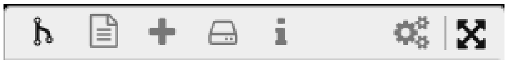
### Introduction

ParaViewWeb, is a Web framework to build applications with interactive scientific visualization inside the Web browser. Those applications can leverage a VTK and/or ParaView backend for large data processing and rendering.

There are example prototypes that have been built using the ParaViewWeb framework. Here is a webpage that has 4 of them - https://pvw.kitware.com/.

### Visualizer

Visualizer is a web based prototype of ParaView using the ParaView Web framework. Click on the Visualizer icon. This brings up a web client, connecting to a backend server located at Kitware headquarters, in Clifton Park, New York. Visualizer can connect to any ParaView server, but in this case they use Kitware headquarters.

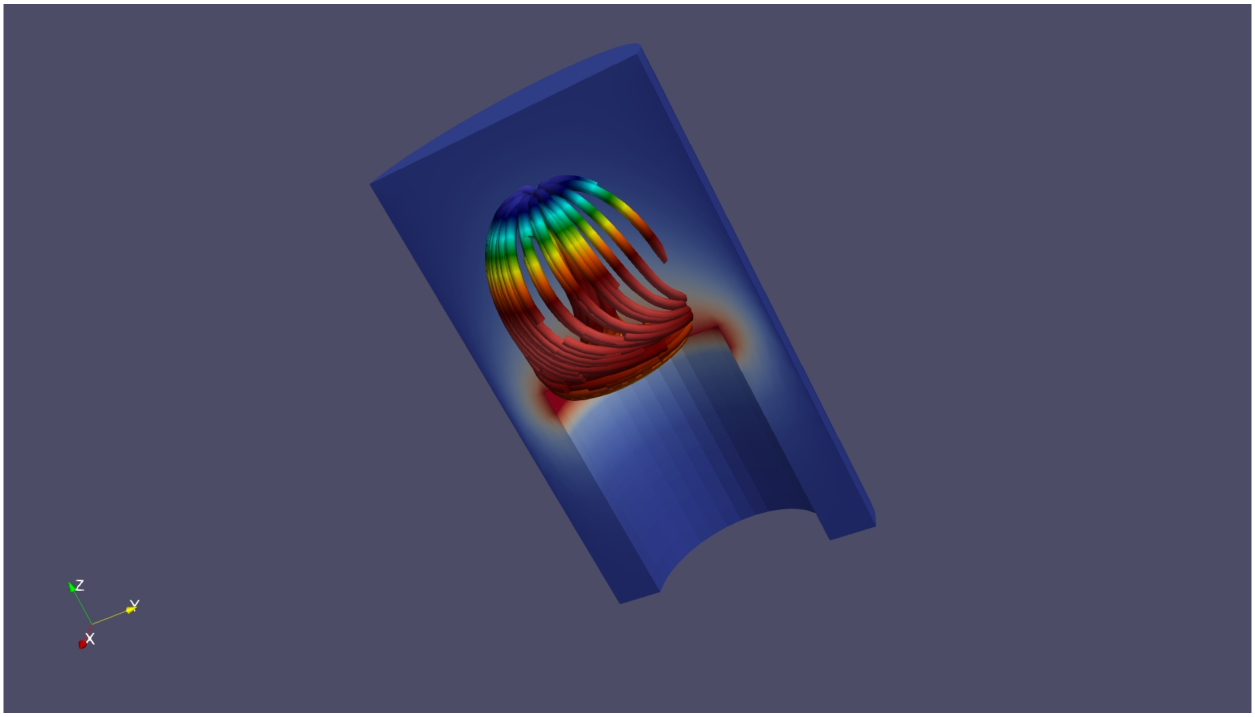


In the upper left corner are menu options. Running from the left side, these icons are:

- Pipeline view.
- Open File.
  - Note that this is wherever the server is located. In this example, it is in New York state.
- Filters.
  - This is a subset of the ParaView Filters. Note that almost any ParaView filter can be exposed, customizing this list to the needs of the user.
- Save Screenshot.
  - This will save locally or remotely, depending on the needs of the user.
- Information.
  - This will give information for whatever filter is selected in the pipeline browser.
- Server settings.
  - Generally speaking, leave this alone.
- Reset View

### Example with disk_out_ref.ex2

- Open disk_out_ref.ex2.
- Select **Filters** → **Clip**. - Sometimes, you need to click on the little paper airplane next to the search window.
- disk_out_ref.ex2 and the clip are both visible. Click the disk to the left of disk_out_ref.ex2 in the pipeline browser. Now you see the clip.
- Paint by Temp.
- Select disk_out_ref.ex2. The original dataset now has focus.
- Select **Filters** → **Stream Tracer**.
- Select **Filters** → **Tube**.
- Paint this filter by **Pres**.
- Change the color map to be something else. I used Rainbow Desaturated.
- In the image below, I also changed the background color to **.3, .3, .4**.



### Example with can.ex2

This is left as an exercise. Try slices. Play this dataset forward in time.

### How to use a local server

Open a terminal window. This works in Linux, Windows or OS X. List available applications

```
pvpython -m paraview.apps -l
```

Now, lets run visualizer, which we can run from your local ParaView download. Documentation is found at https://kitware.github.io/visualizer/

```
pvpython -m paraview.apps.visualizer --data ~/ --port 1234
```

You will have to open your browser and point it at: http://localhost:1234/

To run divvy, use the following command. Documentation is found at: https://kitware.github.io/divvy/

```
pvpython -m paraview.apps.divvy --data ~/path-to-file/examples/disk_out_ref.ex2 --port␣
↪1235
```

To run lite, use the following command. Documentation can be found at: https://kitware.github.io/paraview-lite/

```
pvpython -m paraview.apps.lite --data ~/ --port 1236
```

# CHAPTER FIVE

# REFERENCES

CHAPTER

SIX

# DOCUMENTATION SOURCE

This documention is generated from source files in the ParaView Documentation project.

# BIBLIOGRAPHY

[ABM+01] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C.C. Law, and M. Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications*, 21(4):34–41, 2001. doi:10.1109/38.933522.

[ALS+00] James Ahrens, Charles Law, Will Schroeder, Ken Martin, Kitware Inc, and Michael Papka. A parallel approach for efficiently visualizing extremely large, time-varying datasets. Technical Report LAUR-00-1620, Los Alamos National Laboratory, 2000.

[ADM+07] James P Ahrens, Nehal Desai, Patrick S McCormick, Ken Martin, and Jonathan Woodring. A modular extensible visualization system architecture for culled prioritized data streaming. In *Visualization and Data Analysis 2007*, volume 6495, 176–187. SPIE, 2007.

[BGM+07] John Biddiscombe, Berk Geveci, Ken Martin, Kenneth Moreland, and David Thompson. Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1376–1383, 2007. doi:10.1109/TVCG.2007.70600.

[BS08] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4 (2nd Edition) (Prentice Hall Open Source Software Development Series)*. Prentice Hall, hardcover edition, 2 2008. ISBN 978-0132354165.

[CGM+06] Andy Cedilnik, Berk Geveci, Kenneth Moreland, James Ahrens, and Jean Favre. Remote Large Data Visualization in the ParaView Framework. In Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2006. doi:10.2312/EGPGV/EGPGV06/163-170.

[dt] The Matplotlib development team. *Writing mathematical expressions - Matplotlib 3.0.2 documentation*. URL: https://matplotlib.org/tutorials/text/mathtext.html.

[KInc10] Kitware and Inc. *VTK User's Guide*. Kitware, Inc., paperback edition, 3 2010. ISBN 978-1930934238.

[MT03] K. Moreland and D. Thompson. From cluster to wall with vtk. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003.*, volume, 25–31. 2003. doi:10.1109/PVGS.2003.1249039.

[MWP01] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics (Cat. No.01EX520)*, volume, 85–154. 2001. doi:10.1109/PVGS.2001.964408.

[Mor13] Kenneth Moreland. A survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):367–378, 2013. doi:10.1109/TVCG.2012.133.

[MAF07] Kenneth Moreland, Lisa Avila, and Lee Ann Fisk. Parallel unstructured volume rendering in ParaView. In Robert F. Erbacher, Jonathan C. Roberts, Matti T. Gröhn, and Katy Börner, editors, *Visualization and Data Analysis 2007*, volume 6495, 144 – 155. International Society for Optics and Photonics, SPIE, 2007. URL: https://doi.org/10.1117/12.704533, doi:10.1117/12.704533.

[SML06] Will Schroeder, Kenneth W. Martin, and William E. Lorensen. *The visualization toolkit: An object-oriented approach to 3D graphics*. Kitware, 2006. ISBN 1-930934-19-X.

[SML96] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th Conference on Visualization '96*, VIS '96, 93–ff. Los Alamitos, CA, USA, 1996. IEEE Computer Society Press. URL: http://dl.acm.org/citation.cfm?id=244979.245018.

[BerkGeveci] Berk Geveci. *A VTK pipeline primer (part 1)*. URL: https://blog.kitware.com/a-vtk-pipeline-primer-part-1.

[KitwareInc] Kitware, Inc. *VTK API documentation*. URL: http://www.vtk.org/doc/nightly/html.

[NumPydevelopers] NumPy developers. NumPy. http://www.numpy.org/.

[PatMarion] Pat Marion. *What is InterpolateScalarsBeforeMapping in VTK?* URL: https://blog.kitware.com/what-is-interpolatescalarsbeforemapping-in-vtk.

[ThePCommunity] The ParaView Community. *Setting up a ParaView server*. URL: http://www.paraview.org/Wiki/Setting_up_a_ParaView_Server.