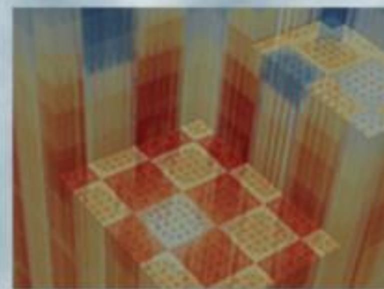
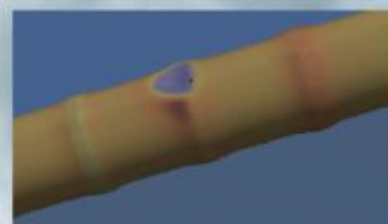
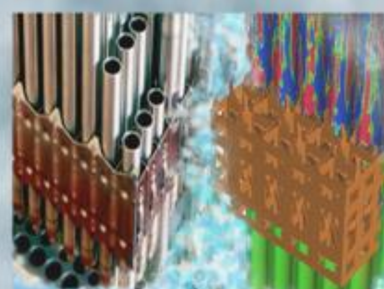
CASL-U-2019-1893-000





VERAView API and Programmer's Guide

Milestone: L3:PHI.NQA-1.P19.03



Ronald W. Lee[1] and Andrew T. Godfrey[1]

[1]Oak Ridge National Laboratory

9 August 2019

Revision Log

Revision	Date	Affected Pages	Revision Description
0	9 August 2019	All	Initial Release

Document pages that are:

Export Controlled:	None
IP/Proprietary/NDA Controlled:	None
Sensitive Controlled:	None
Unlimited:	All

Requested distribution:

To: Benjamin S. Collins, Andrew T. Godfrey, Dave J. Kropaczek, Scott Palmtag

VERAView API and Programmer's Guide

Approvals:

Ron Lee, Author

Date

Scott Palmtag, VERAIO Product Software Manager

Date

Independent Reviewer

Date

EXECUTIVE SUMMARY

VERAView's component-based design facilitates extension and integration into other applications and systems. In addition to describing VERAView's architecture and components, this document provides details on the component application programming interfaces (API), illustrates how to use them independently of VERAView itself, and explains how to extend VERAView with custom visualizations.

(This page intentionally left blank)

CONTENTS

(This page intentionally left blank)

FIGURES

TABLES

LISTINGS

ACRONYMS

2D two-dimensional

3D three-dimensional

CSV comma-separated values

DC device context

GIF Graphics Interchange Format

GUI graphical user interface

HDF5 Hierarchical Data Format 5

HTML Hypertext Markup Language

IO input and output

JSON JavaScript Object Notation

LED light emitting diode

MVC Model-View-Controller

PIL Python Imaging Library

PNG Portable Network Graphics

UML Unified Modeling Language

VERA Virtual Environment for Reactor Applications

(This page intentionally left blank)

1 INTRODUCTION

VERAView is an application commonly used by analysts and engineers to visualize and analyze outputs from Virtual Environment for Reactor Applications (VERA) multi-physics models [1]. Since it is implemented as a collection of Python packages and modules, much of the functionality can be imported and integrated in other Python scripts and applications. Further, the various data displays (called *widgets*) are implemented in a framework which can be extended with custom implementations. The purpose of this guide is to provide the information needed to:

- Reuse VERAView components (i.e., Python packages and modules) in other systems,
- Implement custom widgets, and/or
- Modify the application for desired customizations

1.1 Python Environment

At present VERAView requires a Python-2 (specifically version 2.7) runtime and uses the common or standard modules listed in Table 1. Additional packages required by VERAView are described in Table 2.

1.2 Anaconda

Given the myriad of required Python packages and the desire to support three plaforms (Microsoft Windows, Mac OSX, and Linux), VERAView is developed and distributed to run in the Anaconda-2 Python environment [3]. Although is possible to construct from scratch a Python environment that supports VERAView, Anaconda significantly reduces the effort by making available the required packages with an easy install process.

VERAView installers assume Anaconda is installed and create a virtual environment (named "veraview") in which required packages and VERAView itself are installed. Scripts for creating the required environment on each platform are given below.

1.2.1 Windows

It is expected that Anaconda is installed in an *anaconda*2/ folder in the user's home folder, but this is not strictly necessary. In the Command Prompt script examples below, the variable %*CondaDir*% represents the path to the Anaconda-2 installation.

Create the veraview environment.

```
> path=%CondaDir%\scripts;%CondaDir%;%path%
> conda create -y -n veraview python=2.7.16
```

Install required packages.

```
> path=%CondaDir%\scripts;%CondaDir%;%path%
> activate veraview
> conda install -y -c https://repo.anaconda.com/pkgs/free wxpython=3.0
   mayavi
> conda install -y h5py matplotlib scipy pillow
```

Install VERAView.

After the VERAView source distribution is extracted or `unzipped`, it is installed using the standard Python setup.

```
> path=%CondaDir%\scripts;%CondaDir%;%path%
> activate veraview
```

Table 1. Standard modules used by VERAView.

Module	Description
argparse	command-line argument parsing
bisect	item search in sorted lists
copy	object deep copy
functools	function composition
glob	file matching
hashlib	secure hashes and message digests
inspect	object introspection
json	JavaScript Object Notation (JSON) serialization/deserialization
logging	log file management and logging
math	math operations
numbers	abstract type definitions
os	miscellaneous operating system interfaces
platform	platform indentifying information
pyparsing	parsing grammar definition and execution
re	regular expression processing
shutil	high-level file operations
six	Python-2 and -3 common operations
StringIO	strings as file-like objects
sys	system-specific functions
tempfile	creation and management of temporary files
threading	high-level threading interface
time	miscellaneous date/time functions
timeit	execution time measurement
traceback	stack trace introspection
webbrowser	Web browser launch

Table 2. Packages required by VERAView.

Package	Description
h5py	Python interface to the Hierarchical Data Format 5 (HDF5) libraries. All data processed by VERAView are in HDF5 files, for the most part meeting the VERAOutput specification. [2]
matplotlib	Python two-dimensional (2D) plotting library. All VERAView plots and many other *widgets* (described in more detail below) are based on *matplotlib*.
mayavi	three-dimensional (3D) visualization and plotting. All 3D displays in VERAView use this package.
numpy	Scientific data and array/matrix processing. The *numpy* package is used by *h5py* and most data manipulation in VERAView. The underlying implementation is native code (e.g., C, Fortran) and thus performs well.
pillow	Python Imaging Library (PIL) implementation.
scipy	Math, science, and engineering functions. Like *numpy*, it is implemented in native code.
wxPython	Python interface to the *wxWidgets* library. It is a cross-platform graphical user interface (GUI) toolkit used for the VERAView user interface. Note that although version 4 is available, VERAView currently uses version 3.0.

```
> unzip veraview-3.0.3.zip
> cd veraview-3.0.3
> python setup.py install
```

1.2.2 Mac OSX

It is expected that Anaconda is installed in an *anaconda*2 subdir in the user's home directory, but this is not strictly necessary. In the bash script examples below, the variable $*CondaDir* represents the path to the Anaconda-2 installation.

Create the veraview environment.

```
$ export PATH="${CondaDir}/bin:$PATH"
$ conda create -y -n veraview python=2.7.16
```

Install required packages.

```
$ . "${CondaDir}/etc/profile.d/conda.sh"
$ conda activate veraview
$ conda install -y -c https://repo.anaconda.com/pkgs/free wxpython=3.0
   mayavi
$ conda install -y h5py matplotlib pillow
$ conda install -y -c https://repo.anaconda.com/pkgs/free scipy
```

Install VERAView.

After the VERAView source distribution is extracted it is installed using the standard Python setup.

```
$ . "${CondaDir}/etc/profile.d/conda.sh"
$ conda activate veraview
$ tar xvfz veraview-3.0.3.tar.gz
$ cd veraview-3.0.3
$ python setup.py install
```

1.2.3 Linux

Some Linux distributions might not support *wxPython* version 3.[1] In such cases a Docker image based on CentOS can be used.

On a supported distribution it is expected that Anaconda is installed in an *anaconda*2/ subdir in the user's home directory, but this is not strictly necessary. In the bash script examples below, the variable $\$CondaDir$ represents the path to the Anaconda-2 installation.

Create the veraview environment.

```
$ export PATH="${CondaDir}/bin:$PATH"
$ conda create -y -n veraview python=2.7.16
```

Install required packages.

```
$ . "${CondaDir}/etc/profile.d/conda.sh"
$ conda activate veraview
$ conda install -y -c https://repo.anaconda.com/pkgs/free wxpython=3.0
   mayavi
$ conda install -y h5py matplotlib pillow pango
$ conda install -y -c https://repo.anaconda.com/pkgs/free scipy
```

Install VERAView.

After the VERAView source distribution is extracted it is installed using the standard Python setup.

```
$ . "${CondaDir}/etc/profile.d/conda.sh"
$ conda activate veraview
$ tar xvfz veraview-3.0.3.tar.gz
$ cd veraview-3.0.3
$ python setup.py install
```

1.3 Document Contents

This document is organized as follows. Section 2 provides an overview of the packages and modules comprising VERAView. Next, Section 3 describes VERAView's data management and processing components with examples. Section 4 explains VERAView's widget framework and walks through a custom widget example.

2 PACKAGES AND MODULES

VERAView is organized into Python packages and modules, the latter comprising its components. The top level package, named veraview, contains six subpackages and a couple of additional subdirectories. The package structure and constituent modules are shown in Figure 1. Descriptions of each package and provided in this section after a discussion of naming conventions.

[1] Ubuntu version 18 is one such distribution.

Figure 1. VERAView packages and modules.

2.1 Conventions

2.1.1 Naming Conventions

Python's creator Guido van Rossum and others have developed a Python Style Guide.[4] When dealing with pure Python, following the style guide is rather easy. However, one of Python's strengths versus environments such as Java or .NET is the facilitated integration of native code libraries, such as is the case for packages like *h5py*, *numpy*, *scipy*, and *wxPython*. As often as not, a library being wrapped in a Python package will not adhere to Python naming conventions. Rather, the library's own naming conventions will be realized in the Python package unless the Python package is explicitly coded to provided aliases for method and class names.

The Python Style Guide includes the admonition to "know when to be inconsistent". Unfortunately, the result for VERAView has led to consistent inconsistency. We started out following the style of *wxPython*, which follows a Windows/.NET camel case style with capital first letters for classes and methods. Occasionally, *wxPython* will use the Java approach of lower first letters with camel case for method names. As time has progressed VERAView source code has migrated toward the Python style guide of underscored method names while maintaining camel case for class names. Field names run the gamut, some following Python conventions, others Java style camel case, and some using a Java style guide that begins field/attributes names with a lower case 'f' followed by camel case.

A couple of the data classes include aliases to provide Python style method name aliases for *wxPython* convention methods. Regardless, one will benefit from adopting a position of flexibility when dealing with class, method, property, and field names in VERAView components.

2.1.2 Array and List Indexes

Within VERAView code and components, 0-based indexes are used universally. Note this differs from VERAOutput files where indexes are 1-based due to many VERA models being written in Fortran. For the sake of sanity in the Python code, they are converted to 0-based indexes by VERAView components. For example, `CORE/core_map` values read from a VERAOutput file must be converted from 1-based to 0-based.

2.2 Top Level Package: veraview

The top level VERAView package is named veraview and contains a single module, also named veraview, imported in Python code with `import veraview.veraview`.

2.2.1 User Interface Component Classes

The veraview.veraview module provides the main application GUI component classes:

- VeraViewApp
- VeraViewFrame
- VeraViewFrameAnimatorWidget
- VeraViewFrameDropTarget
- VeraViewGrid

Figure 2 is a Unified Modeling Language (UML) class diagram showing the relationship between these classes. VeraViewApp is the *wxPython* application (wx.App) extension which holds references to all the active frames or top level windows and manages them. It also implements the static main() method invoked when VERAView is launched.

There are a couple of special-purpose classes. VeraViewFrameAnimatorWidget provides a required interface for doing frame-level animations, described in more detail below. VeraViewFrameDropTarget extends wx.TextDropTarget to provide drag-and-drop functionality in VERAView. Most of the functionality in VERAView main windows is provided in the VeraViewFrame and VeraViewGrid classes.

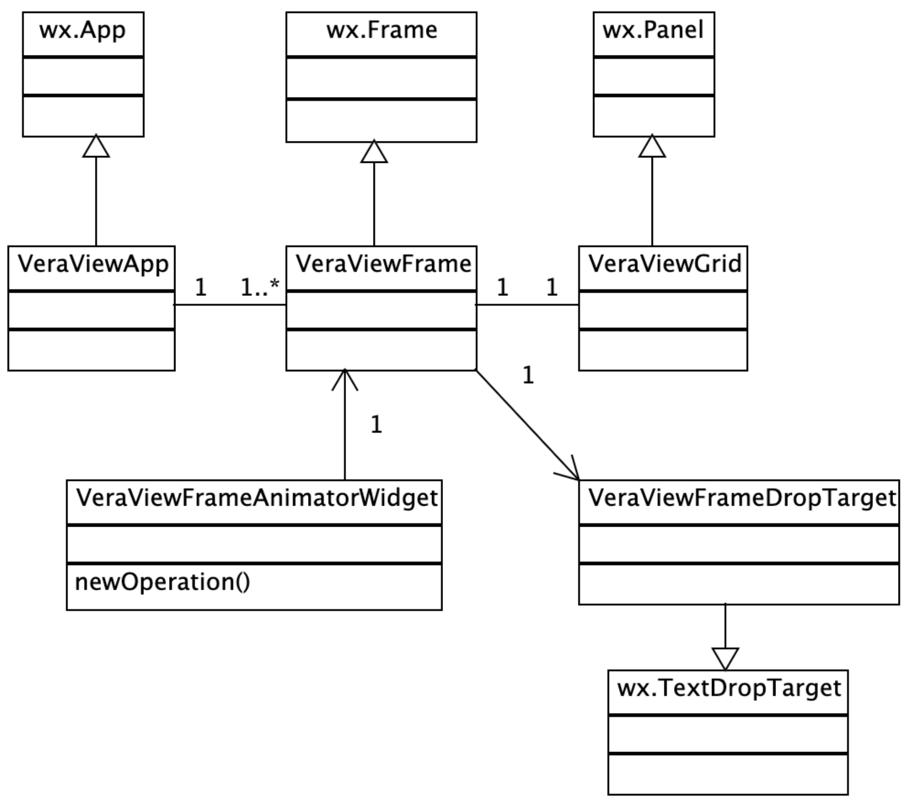


Figure 2. Veraview module class diagram.

As a wx.Frame extension, VeraViewFrame is the main window implementation. It contains the menu bar, the tool bar with icons representing available widgets, sliders for selecting the current time and axial level, a drop target (VeraViewFrameDropTarget) for accepting widgets dragged from other windows, and a VeraViewGrid instance as the widget container.

The center component of the frame is a VeraViewGrid instance in which widgets are contained. It is wx.Panel extension that uses a wx.GridSizer for laying out child widgets in a grid of equal size compartments.

2.3 User Interface Components: veraview.bean

In order to expedite encapsulation and reuse, VERAView follows a component-based design. Implementations of user interface elements for user editing functions that are not specific to widgets reside in modules in this package.[2]

2.3.1 Module: dataset_creator

Four classes are implemented in this module: DataSetCreatorBean, DataSetCreatorDialog, DataSetCreatorMenuWidget, and DataSetCreatorTask. DataSetCreatorMenuWidget and DataSetCreatorTask are used internally to meet interface requirements for a veraview.widget.bean.dataset_menu.DataSetsMenu instance and provide multithreaded operation with GUI feedback.

DataSetCreatorDialog is a wx.Dialog extension that contains an instance of DataSetCreatorBean instance, which extends wx.Panel. The dialog constructor has two parameters, an optional parent component and a veraview.event.state.State instance. Its use in VeraViewFrame follows:

```
dialog = DataSetCreatorDialog( self, self.state )
try:
  dialog.ShowModal()
finally:
  dialog.Destroy()
```

2.3.2 Module: dataset_diff_creator

Three classes are implemented in this module: MenuWidget, DataSetDiffCreatorBean, and DataSetDiffCreatorDialog. Whereas the first is used internally to meet interface requirements for a veraview.widget.bean.dataset_menu.DataSetsMenu instance, the others are used to present the user with options for creating a difference dataset. DataSetDiffCreatorDialog is a wx.Dialog extension that contains an instance of DataSetDiffCreatorBean instance, which extends wx.Panel. The dialog constructor has two parameters, an optional parent component and a veraview.event.state.State instance. Its use in VeraViewFrame follows:

```
dialog = DataSetDiffCreatorDialog( self, self.state )
try:
  dialog.ShowModal()
finally:
  dialog.Destroy()
```

Event handlers in the bean instance will invoke the necessary objects and methods to create difference datasets and store them in the managed data.

2.3.3 Module: filemgr_bean

Two classes are implemented in this module, FileManagerBean (a wx.Panel extension) and FileManagerDialog (a wx.Dialog extension). The dialog contains an instance of the bean, which lists currently open VERAOutput files and offers the ability to open additional files and/or close open files. The dialog construct takes two parameters, an optional parent component and a veraview.data.datamodel_mgr.DataModelMgr instance. A typical use is shown below.

[2] The term "bean" is borrowed from the Java world, where a "bean" is an encapsulation of other objects.

```
dialog = FileManagerDialog( self , self.state.dataModelMgr )
try:
  dialog.ShowModal()
finally:
  dialog.Destroy()
```

Event handlers in the bean instance will invoke the necessary objects and methods to open and close VERAOutput file.

2.3.4 Module: grid_sizer_bean

This module provides the user interface for changing the layout (rows and columns) of the VeraViewGrid in a VeraViewFrame. It contains three classes and defines a *wxPython* event:

- GridSizerBean
- GridSizerEvent
- GridSizerForm
- GridSizerGraphic

All the non-event classes extend wx.Panel. The primary class, GridSizerBean, contains instances of GridSizerForm and GridSizerGraphic and generates GridSizerEvents on user changes. It also defines a value property of type tuple representing the number of rows and columns and with associated accessor methods GetValue() and SetValue()

One can instantiate a GridSizerBean and respond to GridSizerEvents or reference the value property. VERAView instantiates a GridSizerDialog which contains a GridSizerBean.

2.3.5 Module: grid_sizer_dialog

This module contains a single class, GridSizerDialog, which extends wx.Dialog and contains a GridSizerBean instance. It has read-only bean and result properties, the latter set from the bean.value property when the dialog is closed affirmatively (i.e., not via the *Cancel* button). Its use in VERAView is shown below.

```
dialog = GridSizerDialog( None )
try:
  dialog.ShowModal( cur_size )
  new_size = dialog.result
finally:
  dialog.Destroy()
```

2.3.6 Module: html_message_dialog

Often it is useful to present messages in Hypertext Markup Language (HTML) instead of simple text to take advantage of advanced formatting and to embed clickable hyperlinks. The two classes in this module exist for this purpose: HtmlMessageDialog and HtmlMessageWindow. The latter extends wx.html.HtmlWindow to set some default styling and call wx.LaunchDefaultBrowser() when links are clicked, and the former extends wx.Dialog to embed the window in a dialog with a *Close* button. HtmlMessageDialog provides static methods CreateDocument() and ShowBox() to simplify usage. CreateDocument() accepts a body content argument and an optional header content argument. ShowBox() takes message, caption, and parent arguments. It is used in multiple places in VeraViewFrame and also in FileManagerBean. An example use is shown below.

```
HtmlMessageDialog.ShowBox(
    HtmlMessageDialog.CreateDocument( html_message ),
    'Cannot Open File(s)', self
    )
```

2.3.7 Module: ignore_range_bean

Although no longer used in VERAView, this module has been retained in case it might prove useful again. It contains two classes intended to be exported and an internal support class. IgnoreRangeBean and IgnoreRangeDialog follow the patter of other (bean, dialog) pairs in this package. The bean displays controls for selecting a dataset and a value range to ignore. The bean and dialog have been superceded by RangeBean and RangeDialog in the veraview.bean.range_bean module.

2.3.8 Module: range_bean

Following the familiar pattern, this module includes RangeBean and RangeDialog classes, with additional MenuWidget class used internally to support a veraview.widget.bean.dataset_menu.DataSetsMenu instance. The bean provides user interface components for specifying an interval of values which are to be displayed for a selected dataset. The interval is specified as an upper or lower value threshold (inclusive or exclusive), with an optional intersection of both. The dialog accepts parent component veraview.event.state.State instance arguments, and event handlers on the bean will invoke necessary objects and methods to register and apply the range or threshold to the dataset. Use in VERAView is shown below.

```
dialog = RangeDialog( self, state = self.state, title = 'Dataset
    Thresholds' )
try:
  dialog.ShowModal()
  self.state.FireStateChange( STATE_CHANGE_forceRedraw )
finally:
  dialog.Destroy()
```

2.3.9 Module: selection_edit_bean

This module provides user interface elements for displaying and changing current selections (assembly, pin address, node address, axial level, time value, etc.). Instead of a dialog class, this module provides SelectionEditFrame, an extension of wx.Frame, in addition to a bean class, SelectionEditBean. Given all selections are stored as properties of a veraview.event.state.State instance, the bean and frame constructors accept a State reference as an argument, posting selection changes to it. SelectionEditBean extends wx.Notebook to provide a tabbed panel user interface, each tab representing a category of selections.

Since the controls are meant to be free floating instead of modal, a frame is used instead of a dialog, and a frame instance is intended to be maintained between showings instead of being disposed after each showing. VeraViewFrame's use of SelectionEditFrame is show below.

```
def \_OnSelections( self, ev ):
  ev.Skip()
  if self.selectionEditFrame is not None and bool( self.
      selectionEditFrame ):
    self.selectionEditFrame.Raise()
  else:
    self.selectionEditFrame = SelectionEditFrame( self, self.state )
    self.selectionEditFrame.Show()
```

2.4 Data Management: veraview.data

Of the 12 modules in this package that are depicted in Figure 1, six contain the classes most essential to processing VERAOutput files. Classes in those six modules are represented in the diagram of Figure 3 and summarized here. A detailed example of using these classes to process VERAOutput files is given in Section 3.

2.4.1 Module: datamodel

The ten classes defined in this model belong in two groups: classes related to VERAOutput HDF5 files, and other encapsulations. Table 3 summarizes classes that aren't tied directly to HDF5 files.

Table 3. Data encapsulation classes in the datamodel module.

Package	Description
AxialValue	An axial level in cm associated with indexes for all the aaxial meshes.
DataSetName	Name of a dataset that includes a reference to the source file.
DataSetResolver	Methods for resolving a dataset's category or type.[1]
FluenceAddress	Indexes representing fluence selections including the dataset name, the theta index, and the radius index.
VesselFluenceMesh	Fluence z, theta, and r meshes.
VesselGeometry	Vessel geometry description defining barrel, line, pad, and other parameters.

[1]Refer to Section 3.3

One additional module, utils, contains support functions for data processing, many residing in a DataUtils utility class. The three remaining modules, Core, DataModel, and DataModelMgr are built from the opened HDF5 file(s).

DataModel DataModel provides most of the data processing functionality in VERAView. It represents and encapsulates a single VERAOutput HDF5 file, containing a Core instance corresponding to the *CORE* HDF5 group (of type hp5y.Group) and a list of State instances corresponding to statepoint groups in the VERAOutput file. A list of DerivedState instances is also created, each one corresponding to a State to hold derived or calculated datasets. The listing below shows the output of `h5ls` on a typical VERAOutput file.

```
CORE                      Group
INPUT                     Group
STATE_0001                Group
STATE_0002                Group
STATE_0003                Group
...
STATE_0016                Group
title                     Dataset {SCALAR}
veraout_version           Dataset {1}
```

In this case the DataModel object would have 16 States and DerivedStates. There are many DataModel methods ranging in function from reading datasets to finding minimum and maximum values to storing axial meshes by dataset category. Refer to Section 3 for more information.

Core, State, DerivedState The contents of the CORE group are encapsulated in the Core class, which has properties representing axial meshes, the core map, and core geometry as well as a reference to the h5py.Group. It includes many support methods and infers much of the required geometry if not specified in datasets. State is a fairly thin wrapper around a corresponding statepoint h5py.Group. DerivedState extends State and exists to distinguish defined versus derived data. Derived datasets are stored in a temporary file.

2.4.2 Module: datamodel_mgr

Whereas veraview.data.datamodel deals with a single VERAOutput HDF5 file, this primary class in this module, DataModelMgr processes multiple open files and is the interface for data processing in VERAView. The other module class, HtmlException is an Exception that adds a property (htmlMessage) for an HTML version of the message.

DataModelMgr attempts to encapsulate DataModel as much as possible, internally storing DataModel instances and invoking them on the caller's behalf. The first file opened via the OpenModel() method (aliased as open_file()) defines the core geometry against which additional files are compared for congruence on open attempts. OpenModel() calls the CheckDataModelIsCompatible() method is to ensure the file contains data. If one or more files have already been opened, it also checks for congruence of the new file with the current geometry. If it is not congruent, an exception is thrown with a tabular HTML message explaining the incompatibility. In VERAView this table is displayed in a message box.

VERAOutput File Congruence The core geometry is defined by the six Core properties summarized in Table 4.

Table 4. Core geometry properties.

Property	Description
coreMap	Assembly index (1-based) in each row and column.
coreSym	Core symmetry, with allowed values of 1 and 4 representing full and quarter symmetry, respectively.
nass	Number of assemblies.
nassx	Number of assembly columns.
nassy	Number of assembly rows.
npinx	Number of pin (fuel rod) columns.
npiny	Number of pin (fuel rod) rows.

Two DataModel instances (and thus two respective VERAOutput files from which they are realized) are considered congruent if these properties match. Otherwise, they are not congruent and cannot be processed together in VERAView. Note axial and other meshes, the number of statepoints, and statepoint time values can differ and will be resolved within VERAView.

Reading Data If one wishes to open only a single VERAOutput file, a DataModel instance will suffice. DataModelMgr can certainly be used with a single file and is required to process multiple files at once. One significant difference is dataset naming. DataModelMgr requires that a *qualified* name be specified as a DataSetName instance or a string from which a DataSetName can be created. Qualified names are constructed as the concatenation of the file name, a vertical bar (|), and the dataset name. DataModelMgr provides a higher level interface, which can be demonstrated in the following snippets.

Suppose we wish to open a VERAOutput file and read the "pin_powers" dataset in the statepoint with "exposure" value 3 using DataModel. The following code will suffice.

```
from veraview.data.datamodel import *
```

```
dm = DataModel( 'l1_all.h5' )
dset = dm.get_state_dataset( 5, 'pin_powers' )
```

One might immediately ask how to determine 5 is the correct statepoint index in the call to get_state_dataset(). With DataModel one would have to call the get_time_value() method with successive statepoint indexes to determine the proper index. For example,

```
[ dm.get_time_value( i, 'exposure' ) for i in range( dm.get_states_count
    () ) ]
[0.0, 0.38405777217249565, 0.7681155443449913, 1.152173316517487,
1.7282599747762306, 2.3043466330349744, 3.0724621773799656,
    3.840577721724957,
4.608693266069948, 6.14492435475993, 7.681155443449913,
    9.217386532139896,
10.753617620829878, 12.28984870951986, 13.826079798209843,
    15.362310886899825]
```

Using DataModelMgr one can specify the time value.

```
from veraview.data.datamodel_mgr import DataModelMgr
dmgr = DataModelMgr()
dm = dmgr.open_file( 'l1_all.h5' )
dmgr.set_time_dataset( 'exposure' )
dset = dmgr.read_dataset( 'l1_all|pin_powers', 3.0 )
```

Nomenclature Philosophy One might ask why the names DataModel and DataModelMgr were chosen instead of, for example, VERAOutputFile and VERAOutputFileMgr, or ReactorData and ReactorDataMgr. The answer is rather simple. In following the Model-View-Controller (MVC) architectural pattern, the original design concept allowed for different types of files to be opened and processed. Although it was established early in VERAView development that all data would be read from HDF5 files adhering to the VERAOutput specification (with possible extensions), the generic names DataModel and DataModelMgr had proliferated enough to make the name change more painful than beneficial.[2]

2.5 Event and State Management: veraview.event

There are two modules in this package, event and state. The former is a generic event processing class which is no longer used in VERAView and will likely be deprecated soon. The state module contains a single State class encapsulating the (you guessed it) "state" of the VERAView application.[3] Properties in the State class represent user selections and other VERAView state data, and methods are provided for associating property values with event (i.e., state change) types and vice versa. State also manages events, associates named events with state changes, and fires changes to registered listeners.

Note two concepts are important in understanding events in VERAView. First, with one exception, an event represents a state change. (STATE_CHANGE_forceRedraw is the exception.) A single event may include multiple state changes. Second, VERAView creates a single State instance in the VeraViewApp.main() method which is stored as a field/property in the VeraViewApp singleton. This refernce is passed to each VeraViewFrame and to each created widget. Were it ever more convenient to do so, copies of State objects could be used, but heretofore it has been more efficient and effective to pass around a single object and store widget-specific values as widget properties.

State changes are codified as bitmask values defined as module fields, shown in Table 5.[4]

Table 6 shows the association of the state change bitmasks, State object properties, and state change method parameters.

[3] This should be distinguished from "state" as a shortened name for a statepoint in reactor model output.

[4] Although these values can be modified, it is assumed coders are good Python citizens and will not change them. A future VERAView implementation may use namedtuple to make these values immutable

Table 5. State change bitmask fields.

Name	Value	Lockable
STATE_CHANGE_noop	0	
STATE_CHANGE_init	$0x1 << 0$ (1)	no
STATE_CHANGE_axialValue	$0x1 << 1$ (2)	yes
STATE_CHANGE_coordinates	$0x1 << 2$ (4)	yes
STATE_CHANGE_curDataSet	$0x1 << 3$ (8)	no
STATE_CHANGE_dataModelMgr	$0x1 << 4$ (16)	no
STATE_CHANGE_forceRedraw	$0x1 << 5$ (32)	no
STATE_CHANGE_scaleMode	$0x01 << 6$ (64)	yes
STATE_CHANGE_fluenceAddr	$0x01 << 8$ (256)	yes
STATE_CHANGE_timeDataSet	$0x01 << 9$ (512)	no
STATE_CHANGE_timeValue	$0x01 << 10$ (1024)	yes
STATE_CHANGE_weightsMode	$0x01 << 11$ (2048)	no
STATE_CHANGE_dataSetAdded	$0x01 << 12$ (4096)	no
STATE_CHANGE_ALL	0xffffffff	

2.5.1 Applying Changes to the State Object

The Change() method should be called to update the State object, which should be followed by a FireStateChange() call to notify other VERAView objects that a change has occurred. Change() takes as a first argument a dictionary of boolean values keyed by state which defaults to None. Keyword arguments follow as described in Table 6, so multiple State properties can be updated at once. The return value from Change() is a reason state change mask, with bits set to represent each state change type that has been applied, which should be passed as the single argument to FireStateChange().

For example, the following snippet demonstrates changing the selected assembly (column 5, row 6), pin (column 10, row 7), and axial value (100 cm), where "self.dmgr" and "self.state" refer to the DataModelMgr and State instances, respectively.

```
reason = self.state.Change(
    assembly_addr = self.dmgr.CreateAssemblyAddr( 5, 6 ),
    axial_value = self.dmgr.GetAxialValue( cm = 100.0 ),
    sub_addr = ( 10, 7 ),
    )
if reason != STATE_CHANGE_noop:
  self.state.FireStateChange( reason )
```

2.5.2 Responding to State Object Changes

State has AddListener() and RemoveListener() methods for registering and unregistering listeners, respectively. A listener can be a callable or an object with either a HandleStateChange() or OnStateChange() method. In any of the three cases a single parameter, the reason mask, is passed in the call. The reason mask can be checked to see what changes need to be saved. The snippet below demonstrates the handler method for an object that only cares about a pin or channel address or the axial value.

```
def OnStateChange( self, reason ):
  if (reason & STATE_CHANGE_coordinates) > 0:
    if self.subAddr != self.state.subAddr:
      self.subAddr = self.state.subAddr
```

Table 6. State change events, properties, and parameters.

Bitmask Name	State Props	Param Name	Param Value
axialValue	axialValue	axial_value	AxialValue instance
coordinates	assemblyAddr	assembly_addr	(index, col, row)
	auxNodeAddrs	aux_node_addrs	list of 0-based indexes
	auxSubAddrs	aux_sub_addrs	list of (col, row) pairs
	nodeAddr	node_addr	0-based index
	subAddr	sub_addr	(col, row) 0-based
curDataSet	curDataSet	cur_dataset	DataSetName instance
dataModelMgr	dataModelMgr	data_model_mgr	DataModelMgr instance
fluenceAddr	fluenceAddr	fluence_addr	FluenceAddress instance
scaleMode	scaleMode	scale_mode	'all' or 'state'
timeValue	timeValue	time_value	time dataset value
weightsMode	weightsMode	weights_mode	'on' or 'off'
dataSetAdded		dataset_added	DataSetName instance

```
  if (reason & STATE_CHANGE_axialValue) > 0:
    if self.axialValue != self.state.axialValue:
      self.axialValue = self.state.axialValue
```

In some situations it is advantageous to get back a dictionary of state parameters and values representing the change, basically the inverse of Change(). This is provided by the CreateUpdateArgs() method of State, which takes the reason mask as an argument and returns a dictionary of corresponding property values. The OnStateChange() example above could be rewritten to use CreateUpdateArgs() as demonstrated below.

```
def OnStateChange( self, reason ):
  update_args = self.state.CreateUpdateArgs( reason )
  if 'axial_value' in update_args and \
      update_args[ 'axial_value' ] != self.axialValue:
    self.axialValue = update_args[ 'axial_value' ]

  if 'sub_addr' in update_args and \
      update_args[ 'sub_addr' ] != self.subAddr:
    self.subAddr = update_args[ 'sub_addr' ]
```

2.5.3 Lockable State Changes

The VERAView widget framework (refer to Section 2.7) includes a mechanism for user control of events that are delivered to and emanate from an individual widget. Table 5 indicates which state changes are lockable. When events are processed in a widget, it can use this mechanism to apply user choices to ignore associated events by passing a dictionary of locks as the first argument to the Change() method. If this argument is not supplied or is None, it is assumed that all state changes are to be processed.

We can modify the example invocation of Change() above to preclude a change to the assembly address.

```
reason = self.state.Change(
    { STATE_CHANGE_coordinates: False },
    assembly_addr = self.dmgr.CreateAssemblyAddr( 5, 6 ),
    axial_value = self.dmgr.GetAxialValue( cm = 100.0 ),
```

```
    sub_addr = ( 10, 7 ),
    )
if reason != STATE_CHANGE_noop:
  self.state.FireStateChange( reason )
```

Similarly, a listener can preclude applying a state change by invoking the ResolveLocks() method of the State object to receive a (possibly) modified reason mask.

```
def OnStateChange( self, reason ):
  locks = dict( STATE_CHANGE_axialValue = False, STATE_CHANGE_coordinates
      = False )
  reason = self.state.ResolveLocks( reason, locks )
  update_args = self.state.CreateUpdateArgs( reason )
  if 'axial_value' in update_args and \
      update_args[ 'axial_value' ] != self.axialValue:
    self.axialValue = update_args[ 'axial_value' ]

  if 'sub_addr' in update_args and \
      update_args[ 'sub_addr' ] != self.subAddr:
    self.subAddr = update_args[ 'sub_addr' ]
```

2.6 3D Graphics: veraview.view3d

This package contains three modules supporting 3D visualizations and widgets in VERAView: env3d, slicer_view, and volume_view. Whereas slicer_view and volume_view are widget implementations, env3d provides methods and properties for loading the *Mayavi* environment used for 3D rendering.

2.6.1 Module: env3d

Two classes are implemented in this module: Environment3D and Environment3DLoader. The latter is used internally by the former. Environment3D is a utility class providing only static methods and internally managed class fields for managing the environment. The single method intended to be used by callers is Environment3D.LoadAndCall(), which takes an optional callable as an argument.

Environment3D.LoadAndCall() performs lazy loading of the required Python packages, checking to see if they are already loaded. If not, an Environment3DLoader instance is created and invoked. Environment3DLoader creates a wx.ProgressDialog to display progress of the load performed in a separate thread. Loading consists of importing the required modules in the following sequence:

```
from traits.api import HasTraits, Instance, Array, on_trait_change
from traitsui.api import View, Item, HGroup, Group
from tvtk.api import tvtk
from tvtk.pyface.scene import Scene
from mayavi import mlab
from mayavi.core.api import PipelineBase, Source
from mayavi.core.ui.api import SceneEditor, MayaviScene, MlabSceneModel
```

If passed to Environment3D.LoadAndCall(), the callable is invoked with two arguments: a boolean indicating the success/failure of the load, and a list of error messages. Note if the environment has already been loaded successfully, the callback arguments will be $(True, None)$.

2.6.2 Widget Modules

The first widget implementation is in the slicer_view module, which contains two classes. Slicer3DView is a veraview.widget.widget.Widget extension that works in the VERAView widget framework. It uses a VolumeSlicer (traits.api.HasTraits extension) instance which creates three slice planes for displaying a cut through the current dataset.

Volume3DView is the widget extension in the volume_view module, which also contains a Volume class to render the data. It does so with a collection of volumes and slice to represent a cut into a volume representing the current dataset.

2.7 Widget Framework and Widget Implementations: veraview.widget

Figure 4 is a diagram of the classes in the various modules in this package. For the most part, each module contains a single class. Whereas details on the widget framework are provided in Section 4, an overview of the current widget implementations is given in this section.

All widgets are derived from the Widget class, and widget instances reside in a WidgetContainer.[5]. WidgetContainer provides capabilities common to all widgets, such as a toolbar with a couple of menu buttons and a close button. The full classpath to the Widget to enclose in the container is based to the WidgetContainer constructor. Thus, WidgetContainer takes care of dynamically loading the widget model and class and constructing the Widget instsance. Widgets with 2D displays are divided into three kinds: plot widgets, raster image widgets, and other.

2.7.1 Plot Widgets

All plot widgets display data via *matplotlib* plots and extend PlotWidget. Table 7 summarizes the current plot widget implementations.

Table 7. Plot widget implementations.

Class	Module	Description
AxialPlot	axial_plot	Plots any dataset with an axial dimension
PinSurfaceMap	pin_surface_map	Displays crud/corrosion data in a heat map
SubPinLengthPlot	subpin_len_plot	Plots crud/corrosion data in a polar plot
TimePlots	time_plots	Plots datasets versus time

2.7.2 Raster Image Widgets

Raster image widgets represent data by rendering to an image that is then displayed. RasterWidget is the base class for these widgets and provides the bookkeeping for caching and reusing images. Originally the PIL was used for creating these images, but poor image quality led to use of the graphics rendering capabilities built into *wxPython*. Typically, a wx.Bitmap is created by calling wx.EmptyBitmapRGBA(), a wx.MemoryDC is created, and the bitmap is passed to the SelectObject() method of the device context (DC). In some cases a wx.GraphicsContext is created from the DC, and drawing methods are invoked (e.g., DrawPath(), DrawRectangle()). In other cases drawing methods are called on the DC itself. (*A future migration to Qt5 will use QPixMap and QPainterinstead*).

Table 8 summarizes the current raster widget implementations.

2.7.3 Other Widgets

In addition to the 3D widget implementations, there are two additional widgets that extend neither PlotWidget nor RasterWidget. DataAnalysisView is a work-in-progress toward providing statistical analysis of dataset contents. TableView provides a tabular view of data values at the current state (e.g., assembly, pin/channel, axial level, time value) for mulitiple datasets.

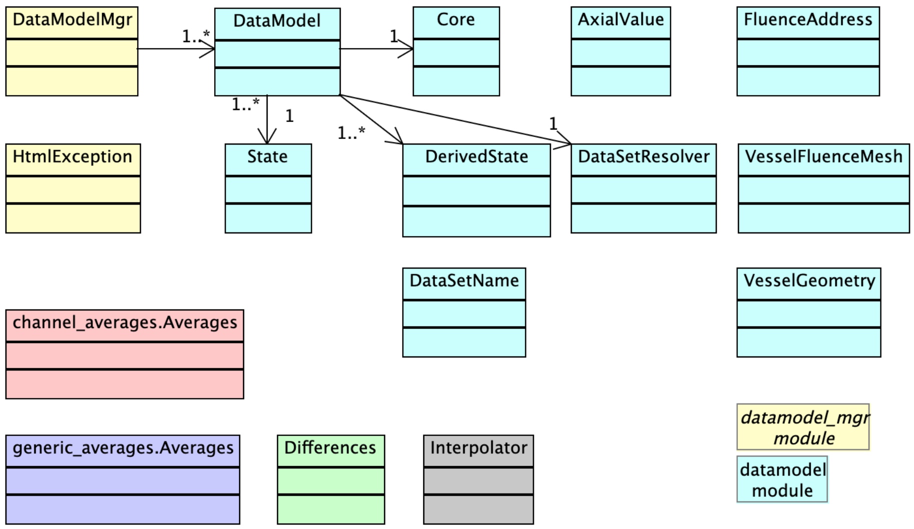


Figure 3. Data Package Class Diagram.

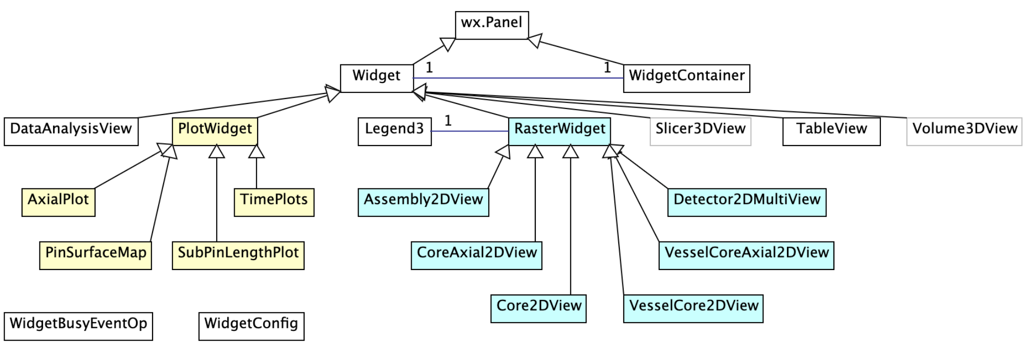


Figure 4. Widget classes.

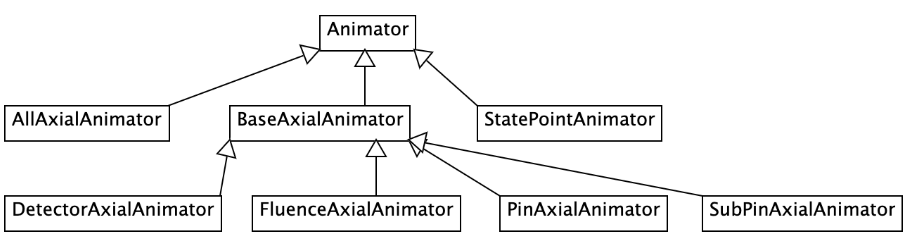


Figure 5. Animator classes.

Table 8. Raster widget implementations.

Class	Module	Description
Assembly2DView	assembly_view	Lattice, single-assembly view of pin/rod/channel values
CoreAxial2DView	core_axial_view	View of the core as a vertical slice along a column or row
Core2DView	core_view	View of the core as a horizontal slice at an axial level
Detector2DMultiView	detector_multi_view	Core view with detector plots or value displays
VesselCoreAxial2DView	vessel_core_axial_view	Core and vessel view displaying fluence data across a radial slice
VesselCore2DView	vessel_core_view	Core and vessel view displaying fluence data across a horizontal slice along an axis

2.7.4 Widget Animation

Figure 5 shows the classes supporting widget animation, all of which are in the animators module. Animation is supported over axial meshes and time via the implementation classes, which are summarized in Table 9.

Table 9. Animation classes.

Class	Description
AllAxialAnimator	Animates over union of all axial meshes
Animator	Base class for all animators
BaseAxialAnimator	Generalization base class for axial animators
DetectorAxialAnimator	Animates over the detector axial mesh
FluenceAxialAnimator	Animates over the fluence axial mesh
PinAxialAnimator	Animators over the pin/channel (default) axial mesh
StatePointAnimator	Animators over time
SubPinAxialAnimator	Animators over the subpin axial mesh

The constructor for each animator class takes a Widget as an argument and an optional callable as a callback. The Widget is the content being animated via calls to its CreatePrintImage() method. Calls to the Widget UpdateState() method are used to advance the display to the next step, and how that is accomplished is specific to the animator implementation.

Animation is accomplished by instantiating an animator and calling the Run() method, passing the path to the animation file as a parameter. Optional Run() arguments are the delay between frames in seconds and a boolean for showing current selections. The following snippet demonstrates an animation invocation.

```
animator = PinAxialAnimator( self.widget )
animator.Run( file_path, 0.1, True )
```

[5] WidgetContainer instances are added to a VeraViewGrid

2.8 Widget User Interface Components: veraview.widget.bean

User interface components used in widgets are implemented in the ten modules in this package, as summarized in Table 10.

2.9 Additional Package Subdirectories

There are two additional subdirectories in the top level veraview package directory: `bin/` and `res/`. Under `bin/` are per-platform subdirs with *gifsicle* executables for each supported platform: `linux64/_, \verb`macos/+, and `win64/`. *Gifsicle* is used to create an animated Graphics Interchange Format (GIF) file from the individual step Portable Network Graphics (PNG) files created by an Animator.

Icons and images used throughout VERAView are stored in `res/` in PNG format. One additional file in this subdir is `logging.conf`, the configuration file for the Python *logging* package.

The contents of these subdirs are referenced by calling the resource_filename() function of the standard Python pkg_resources module, as demonstrated in the following snippet.

```
res_dir = pkg_resources.resource_filename( 'veraview', 'res' )
```

Table 10. Widget bean classes.

Class	Module	Description
AnimateOptionsBean	animate_options_bean	Displays controls for setting frame delay and showing selections
AnimateOptionsDialog		wx.Dialog containing an AnimateOptionsBean
AxialSliderBean	axial_slider	Vertical slider for displaying and selecting the axial level[1]
DataRangeBean	data_range_bean	Controls for setting the display range, colormap, and other widget settings
DataRangeBeanDialog		wx.Dialog containing a DataRangeBean
DataRangeValues		Values edited with a DataRangeBean
DataModelMgrTree	datamodel_mgr_tree	wx.TreeCtrl extension for toggling display of datasets
DataModelMgrTreeDialog		wx.Dialog containing a DataModelMgrTree
DataSetChooserBean	dataset_chooser	Displays controls for selecting datasets[2]
DataSetListBean	dataset_list_bean	Displays a list of dataset names for single or multiple selection, used in place of a pullright menu when there are too many items to display in a menu
DataSetListDialog		wx.Dialog containing a DataSetListBean
DataModelMenu	dataset_menu	wx.Menu extension representing a single DataModel or VERAOutput file
DataSetsMenu		DataModelMenu extension with top level pullrights for each DataModel or open file
EventsChooserBean	events_chooser	List of checkboxes representing lockable state changes
EventChooserDialog		wx.Dialog containing an EventsChooserBean
ExposureSliderBean	exposure_slider	Horizontal slider for displaying and selecting the time value[1]
PlotDataSetPropsBean	plot_dataset_props	Displays control for settings for visible datasets in a plot widget
PlotDataSetDialog		wx.Dialog containing a PlotDataSetPropsBean

[1]Now used in a VeraViewFrame instead of individual widgets
[2]No longer used

3 DATA MANAGEMENT AND PROCESSING

In effect VERAView is a VERAOutput file browser, and thus reading and processing VERAOutput HDF5 files is fundamental to everything in VERAView. VERAOutput processing is encapsulated in modules in the veraview.data package with classes designed to be reused in other applications and systems. Refer to 2.4 for an overview of the modules and their classes. This section describes the essential concepts and mechanisms exposed in VERAView's data handling classes and provides example of how to use them.

3.1 Dataset Factors and Weights

Factors can be used to mask values when displaying datasets, and weights are calculated from factors to normalize results when deriving averages and other aggregations (refer to Section 3.7). The category or type of a dataset dictates how factors and weights are calculated by VERAView. However, VERAView will only calculate factors if they are not specified for a specific dataset or for the file as a whole. When a VERAOutput file is opened, default factors are determined or resolved.

3.1.1 Specified Dataset and Default Factors

Factors for a dataset are defined explicitly if the dataset has an attribute named *factor* or *factors*. For either attribute the value is the name of a dataset in the *CORE* group. Note the attribute must be specified in the first statepoint ($STATE\_0001$), and the referenced *CORE* dataset must have the same shape as the target dataset. For example, if $STATE\_0001/my_data$ is a *pin* dataset with *factor* attribute value of $my_factors$, then $CORE/my_factors$ will be the factors applied for the *my_data* dataset. The same factors may be applied to multiple datasets.

Two *CORE* group dataset names are reserved for specifying default factors for all datasets in a file: $CORE/node\_factors$ and $CORE/pin\_factors$. If they have the correct shapes they will apply to dataset categories *:node* and *pin*, respectively,

3.1.2 Calculated Default Factors

If not specified in the VERAOutput file, VERAView calculates defaults for node and pin factors. Node factors default to all ones with shape $(1, 4, nax, nass)$. Pin factors are calculated from the *pin_powers* dataset in the middle statepoint if it exists. Otherwise, factors are calculated from all ones. Regardless of how pin factors are derived, VERAView calculates weights across various axes for deriving from *pin* datasets. Default channel factors are initialized as all ones.

3.1.3 Resolving Dataset Factors

Factors for a specific dataset are resolved or determined by the DataModel method GetFactors(), aliased as get_factors(). The sequence of resolution steps follows.

1. If *no_factors* is specified in the DATASET_DEFS entry for the dataset's category, there are no factors.
2. Regardless of the dataset's category, explicit factors as described in Section 3.1.1 are used if defined.
3. Otherwise, based on the dataset category:
   (a) *channel*: The default all ones factors are assumed.
   (b) *:chan_radial*: All ones summed across the axial axis are used.
   (c) all others: Factors are calculated from pin factors based on the dataset shape.

3.2 Multiple Open VERAOutput Files

It is important to understand that VERAView is designed to display and process data from multiple open VERAOutput files. Much of the functionality in the DataModelMgr class (veraview.data.datamodel_mgr module) involves resolving multiple files, each of which is encapsulated with a DataModel instance. In VERAView a single DataModelMgr instance is created and stored as the dataModelMgr property of the single veraview.event.state.State instance that is shared by other components.

3.2.1 Core Geometry and Congruence

As explained in Section 2.4.2, VERAView imposes the restriction that multiple open files must have the same core geometry. The core property of DataModelMgr is a veraview.data.datamodel.Core instance that encapsulates the common geometry. Several DataModelMgr methods reference the core property, in a couple of cases merely passing the method call to it. Table 11 lists those methods.

Table 11. DataModelMgr methods referencing the core property.

Method	Method Alias	Core References
CreateAssemblyAddr()	create_assembly_addr()	core.coreMap
CreateAssemblyAddrFromIndex()	create_assembly_addr_from_index()	core.coreMap
CreateAssyLabel()	create_assy_label()	core.CreateAssyLabel()
CreateDetectorAddr()	create_detector_addr()	core.detectorMap
CreateDetectorAddrFromIndex()	create_detector_addr_from_index()	core.detectorMap
IsChannelType()	is_channel_type()	core.npinx, core.npiny

3.2.2 Axial Meshes and Mesh Resolution

Although the core geometry must match across open files, the various axial meshes can differ, and DataModelMgr manages global (across all files) meshes. The axial mesh associated with each dataset in a file is tracked in the encapsulating DataModel object and can be specified in one of three ways. Note that for all mesh types except *detector*, mesh centers are also calculated and stored and provide the reference axial level for corresponding dataset values. For example, if *axial_mesh* is defined with three values 1.0, 2.0, and 3.0, the center values will be 1.5 and 2.5. Datasets using the mesh will have two values in the axial dimension with the levels for those dimensions assumed to be the respective center values.

Default Mesh Assignment Table 12 lists the predefined mesh types which are read implicitly from a VERAOutput file. If present, these are applied to a dataset based on the dataset category or type (refer to Section 3.3). For many categories the shape of the mesh is a determinant. By default the *pin* (or *core*) mesh is applied.

Table 12. Predefined axial mesh types.

Type	CORE Dataset	Centers Calculated	Notes
core or pin	`CORE/axial_mesh`	y	required dataset
detector	`CORE/detector_mesh`	n	
fixed_detector	`CORE/fixed_detector_mesh`	y	
fluence	`CORE/VesselFluenceMesh/axial_mesh`	y	
subpin	`CORE/subpin_mesh`	y	

Dataset Attribute If a dataset (instance of h5py.Dataset) in a VERAOutput file has an attribute named *axial_mesh*, the value is assumed to be either an explicit mesh type name or a *CORE*-relative name or path to the mesh dataset. For example, an *axial_mesh* attribute value of "subpin" would dictate that the subpin mesh be applied to the dataset. An *axial_mesh* value of "other/special_mesh" would specify that the `CORE/other/special_mesh` dataset is the mesh for the dataset.

Mesh Resolution Meshes across all open files are resolved by DataModelMgr upon each invocation of open_file(). In addition to predefined mesh types associated with each file and DataModel and explicitly defined meshs referenced by the *axial_mesh* attribute of a datset, DataModelMgr maintains global versions of all predefined mesh types and a global combined mesh given the pseudo type name *all*. Global meshes are unions of all the file meshes. This can best be illustrated with an example. Suppose two files are opened, the first having the following meshes:

```
axial_mesh: [
  10.315, 49.094, 93.218, 129.286, 169.609, 205.677, 241.745, 282.067,
      318.135, 353.274,
  376.075
  ]
detector_mesh: [ 45.0219, 118.369, 191.716, 265.064, 338.411 ]
fixed_detector_mesh: [ 8.34823, 81.6956, 155.043, 228.39, 301.738,
    375.085 ]
```

After this file has been opened, the global meshes are as follows:

```
axial_mesh: [
  10.315, 49.094, 93.218, 129.286, 169.609, 205.677, 241.745, 282.067,
      318.135, 353.274,
  376.075
  ]
detector_mesh: [ 45.0219, 118.369, 191.716, 265.064, 338.411 ]
fixed_detector_mesh: [ 8.34823, 81.6956, 155.043, 228.39, 301.738,
    375.085 ]

all_mesh: [
  8.34823, 10.315, 45.0219, 49.094, 81.6956, 93.218, 118.369, 129.286,
      155.043, 169.609,
  191.716, 205.677, 228.39, 241.745, 265.064, 282.067, 301.738, 318.135,
      338.411, 353.274,
  375.085, 376.075
  ]
```

The second file has the following meshes:

```
axial_mesh: [ 11.951, 73.295, 129.305, 189.570, 249.835, 310.100, 377.711
     ]
detector_mesh: [ 50.0, 100.0, 200.0, 350.0 ]
```

Opening the second file yields the following global meshes:

```
axial_mesh: [
  10.315, 11.951, 49.094, 73.295, 93.218, 129.286, 129.305, 169.609,
      189.57,
  205.677, 241.745, 249.835, 282.067, 310.1, 318.135, 353.274, 376.075,
      377.711
  ]
detector_mesh: [ 45.0219, 50.0, 100.0, 118.369, 191.716, 200.0, 265.064,
    338.411, 350.0 ]
fixed_detector_mesh: [ 8.34823, 81.6956, 155.043, 228.39, 301.738,
    375.085 ]
```

```
all_mesh: [
  8.34823, 10.315, 11.951, 45.0219, 49.094, 50.0, 73.295, 81.6956,
      93.218, 100.0,
  118.369, 129.286, 129.305, 155.043, 169.609, 189.57, 191.716, 200.0,
      205.677, 228.39,
  241.745, 249.835, 265.064, 282.067, 301.738, 310.1, 318.135, 338.411,
      350.0,
  353.274, 375.085, 376.075, 377.711
  ]
```

Mesh Related Methods There are several DataModelMgr methods for retrieving mesh values by dataset or mesh type, one or the other, with the dataset name taking preference since it inherently specifies the mesh type. These are summarized in Table 13.

Table 13. DataModelMgr mesh methods.

Method	Description
GetAxialMesh2() get_axial_mesh()	calls GetAxialMesh() on the DataModelMgr
GetAxialMeshCenters2() get_axial_mesh_centers()	calls GetAxialMeshCenters() on the DataModelMgr
GetAxialMeshCentersIndex() get_axial_mesh_centers_index()	retrieves the index for a mesh value
GetAxialMeshIndex() get_axial_mesh_index()	retrieves the index for a mesh value
GetAxialValue() get_axial_value()	creates an AxialValue instance with properties for all 0-based axial level indexes
GetDetectorMesh() get_detector_mesh()	calls GetAxialMesh() passing type "detector"
GetDetectorMeshIndex() get_detector_mesh_index()	calls GetAxialMeshIndex() passing type "detector"
GetFixedDetectorMesh() get_fixed_detector_mesh()	calls GetAxialMesh() passing type "fixed_detector"
GetFixedDetectorMeshCenters() get_fixed_detector_mesh_centers()	calls GetAxialMeshCenters() passing type "fixed_detector"
GetFixedDetectorMeshCentersIndex() get_fixed_detector_mesh_centers_index()	calls GetAxialMeshCentersIndex() passing type "fixed_detector"
GetFluenceMesh() get_fluence_mesh()	calls GetAxialMesh() passing type "fluence"
GetFluenceMeshCenters() get_fluence_mesh_centers()	calls GetAxialMeshCenters() passing type "fluence"
GetFluenceMeshIndex() get_fluence_mesh_index()	calls GetAxialMeshCentersIndex() passing type "fluence"

Listing 1 is an example of plotting *pin_powers* from two VERAOutput files on their respective axial meshes using *matplotlib*. Additional methods introduced in the example are described below.

3.2.3 Time Datasets and Resolution

Time in VERAOutput files is represented by statepoint groups with names `STATE_nnnn` where *nnnn* is the sequence number or index starting with 0001. In addition to the statepoint index, scalar datasets can represent time. VERAView currently recognizes the following datasets as time datasets, provided they are scalars.[6].

- exposure
- exposure_efpd
- exposure_efpy
- hours
- msecs
- transient_time

Like axial meshes, times are resolved across all open VERAOutput files after each file is opened. The DataModelMgr method resolve_all_available_time_datasets() is called from other VERAView components to determine which of the above datasets are available in **all** open files and thus can be used as a reference time. The pseudo name *state* is always available and represents the 0-based statepoint index.

In the invocations of the read_dataset() method in Listing 1, the second parameter is a time value, passed as 2.0 in the example. The dataset to which this value is compared is the timeDataSet property of the DataModelMgr, with corresponding accessors get_time_dataset() and set_time_dataset(). The default is *state*, but in the example it is explicitly set to *exposure*. Thus the datasets are retrieved are in the first respective statepoint in each file having an *exposure* value ≥ 2.0. Table 14 summarizes the time processing methods in DataModelMgr.

Table 14. DataModelMgr time methods.

Method	Description
GetTimeDataSet() get_time_dataset()	getter for the timeDataSet property
GetTimeIndexValue() get_time_index_value()	calls GetAxialMeshCenters() on the DataModelMgr returns the value of the time dataset at a specified 0-based index
GetTimeValueIndex() get_time_value_index()	retrieves the index of the first statepoint with time dataset value $\geq$ a specified value
GetTimeValues() get_time_values()	returns a list of values for the time dataset

Methods get_time_value_index() and get_time_index_value() take an index or value, respectively, as a first and required parameter. Both take an additional optional parameter specifying the VERAOutput file to reference, which can be a file name (sans the `.h5` extension) or a DataSetName instance. This is also the lone, optional argument to get_time_values(). If not passed or passed as None, the method applies to the global (union-ed) list of time dataset values across all open files. Otherwise, it applies to the specified file.

[6] A scalar is a dataset with a shape of (1,) or ()

Listing 1. Example code to read and plot data

```
from __future__ import print_function
import os, sys
import matplotlib.pyplot as plt
from veraview.data.datamodel_mgr import *

#        Open manager
dmgr = DataModelMgr()

#        Open VERAOutput files
dmgr.open_file( '.../l1_all.h5' )
dmgr.open_file( '.../c1.h5_new.h5' )

#        Set time dataset
print( 'time datasets=', dmgr.resolve_available_time_datasets() )
dmgr.set_time_dataset( 'exposure' )
print( 'at exposure==2.0, l1_all={0:d}, c1.h5_new={1:d}'.
    format(
        dmgr.get_time_value_index( 2.0, 'l1_all' ),
        dmgr.get_time_value_index( 2.0, 'c1.h5_new' )
        )
    )

#        Read data from first statepoint
powers_one = dmgr.read_dataset( 'l1_all|pin_powers', 2.0 )
powers_two = dmgr.read_dataset( 'c1.h5_new|pin_powers', 2.0 )

#        Get axial meshes
ax_one = dmgr.get_axial_mesh_centers( 'l1_all|pin_powers' )
ax_two = dmgr.get_axial_mesh_centers( 'c1.h5_new|pin_powers' )

#        Plot
fig = plt.figure()
ax = plt.subplot( 111 )
ax.plot(
    powers_one[ 7, 8, :, 50 ], ax_one, 'b-',
    label = 'l1_all', linewidth = 2
    )
ax.plot(
    powers_two[ 7, 8, :, 50 ], ax_two, 'r-',
    label = 'c1.h5_new', linewidth = 2
    )

ax.legend()
ax.set_xlabel( 'pin_powers: assy=51,pin=(9,8) @exposure=2.0' )
ax.set_ylabel( 'axial level (cm)' )
plt.title( 'Plot values on axial meshes' )
plt.show()
```

Deduplication and Resolution Although it is common to have the same time dataset value in multiple statepoints, VERAView requires all dataset values to be unique. Thus, the first step in resolving time values across open VERAOutput files is dedepulication, a tricky endeavor. Deduplication is implemented in the static FixDuplicates() method of the DataUtils utility class in the veraview.data.utils module.

When there are duplicate time dataset values, only the first statepoint with a duplicated value in that file will be accessible. The only way to ensure each statepoint is accessed is to set the time dataset to *state*. However, this forces VERAView to make the tacit assumption that statepoints correspond across open files when *state* is the current time dataset. This is best illustrated with an example. Suppose there are two files with *exposure* values as listed below, and the time value is specified as 5.0.

```
one.h5/STATE_0001/exposure = 0.0
one.h5/STATE_0002/exposure = 1.15217
one.h5/STATE_0003/exposure = 3.84058
one.h5/STATE_0004/exposure = 9.21739
one.h5/STATE_0005/exposure = 10.415
one.h5/STATE_0006/exposure = 15.3623

two.h5/STATE_0001/exposure = 0.0
two.h5/STATE_0002/exposure = 0.346
two.h5/STATE_0003/exposure = 1.23
two.h5/STATE_0004/exposure = 1.919
two.h5/STATE_0005/exposure = 4.065
two.h5/STATE_0006/exposure = 8.493
two.h5/STATE_0007/exposure = 10.842
two.h5/STATE_0008/exposure = 15.774
```

With *exposure* as the time dataset STATE_0003 and STATE_0005 would be used for the first and second files, respectively. With *state* as the time dataset, STATE_0005 would be chosen for both files. The corresponding *exposure* values would be 15.3623 and 8.493, respectively.

3.3 Dataset Categories

One of the fundamental design elements of VERAView is the categorization of datasets based on their shape. Defined categories are (in effect) registered in the DATASET_DEFS field of the datamodel module, which is a dict keyed by category name. The entry for the *pin* category is listed below.

```
'pin':
  {
  'assy_axis': 3,
  'axial_axis': 2,
  'axial_index': 'pinIndex',
  'axis_names': ( ( 3, 'assembly' ), ( 2, 'axial' ), ( ( 1, 0 ), 'pin' )
      ),
  'label': 'pin',
  'pin_axes': ( 1, 0 ),
  'shape_expr': '( core.npiny, core.npinx, core.nax, core.nass )',
  'type': 'pin'
  }
```

Given the above definition, any dataset with a shape matching the *shape_expr* value is determined to be a *pin* dataset. Expressions area evaluated after the Core object has been realized with valid property values. In cases where the dataset shape cannot be used, a *match_func* can be specified. It is invoked with the dataset (type h5py.Dataset and the Core core object and returns True for a match. Refer to the following example for the *intrapin_edits* category.

```
'intrapin_edits':
```

```
  {
  'assy_axis': -1,
  'axial_axis': -1,
  'axis_names': ( ( 0, 'index' ), ),
  'match_func': lambda d, c:
len( d.shape ) > 0 and d.shape[ 0 ] > 0 and \
'PinFirstRegionIndexArray' in d.attrs and \
DataUtils.ToString( d.attrs[ 'PinFirstRegionIndexArray' ] ) in d.parent
    and \
'PinNumRegionsArray' in d.attrs and \
DataUtils.ToString( d.attrs[ 'PinNumRegionsArray' ] ) in d.parent,
  'is_axial': True,
  'label': 'intrapin_edits',
  'no_factors': True,
  'shape_expr': '()',
  'type': 'intrapin_edits'
  },
```

Datasets that represent averages of other datasets belong to "derived" categories, indicated by a leading colon in the name. Derived category entries have additional keys defining how they can be calculated by VERAView. Derived datasets are forced into a four-dimensional shape to standardize processing in widgets. Refer to the following example for the *assembly* derived category.

```
':assembly':
  {
  'assy_axis': 3,
  'avg_method':
    {
    'channel': 'calc_channel_assembly_avg',
    'pin': 'calc_pin_assembly_avg'
    },
  'axial_axis': 2,
  'axial_index': 'pinIndex',
  'axis_names': ( ( 3, 'assembly' ), ( 2, 'axial' ) ),
  'copy_expr': '[ 0, 0, :, : ]',
  'copy_shape_expr': '( 1, 1, core.nax, core.nass )',
  'ds_prefix': ( 'asy', 'assembly' ),
  'factors': 'assemblyWeights',
  'label': 'assembly',  # '3D asy'
  'shape_expr': '( core.nax, core.nass )',
  'type': ':assembly'
  }
```

Note the *shape_expr* value is used to detect an assembly averaged dataset in a VERAOutput file. Whether detected in the file or calculated, VERAView makes a four-dimensional copy with the shape specified by *copy_shape_expr* and stores it in the corresponding DerivedState. Table 15 shows all categories and associated shape expressions as defined in DATASET_DEFS. The properties referenced in the shape expressions are defined in Table 16.

Table 15. Dataset categories and shapes.

Category	Shape / Copy Shape
channel	(core.npiny + 1, core.npinx + 1, core.nax, core.nass)
detector	(core.ndetax, core.ndet)
fixed_detector	(core.nfdetax, core.ndet)
fluence	(core.fluenceMesh.nz, core.fluenceMesh.ntheta, core.fluenceMesh.nr)
intrapin_edits	uses *match_func*
pin	(core.npiny, core.npinx, core.nax, core.nass)
radial_detector	(core.ndet,)
scalar	(1,)
subpin	(core.nsubtheta, core.nsubr, core.npiny, core.npinx, core.nsubax, core.nass)
subpin_cc	*data_type* dataset attribute
subpin_r	(core.nsubr, core.npiny, core.npinx, core.nsubax, core.nass)
subpin_theta	(core.nsubtheta, core.npiny, core.npinx, core.nsubax, core.nass)
:assembly	(core.nax, core.nass) (1, 1, core.nax, core.nass)
:axial	(core.nax,) (1, 1, core.nax, 1)
:chan_radial	(core.npiny + 1, core.npinx + 1, core.nass) (core.npiny + 1, core.npinx + 1, 1, core.nass)
:core	(1,) (1, 1, 1, 1)
:node	(4, core.nax, core.nass) (1, 4, core.nax, core.nass)
:radial	(core.npiny, core.npinx, core.nass) (core.npiny, core.npinx, 1, core.nass)
:radial_assembly	(core.nass,) (1, 1, 1, core.nass)
:radial_node	(4, core.nass) (1, 4, 1, core.nass)

3.3.1 Resolving Datasets

Dataset resolution is encapsulated in the DataSetResolver class. When a VERAOutput file is opened, either via instantiation of DataModel or a call to open_file() on a DataModelMgr object, the first three statepoints are examined to find all defined datasets. A DataSetResolver is created by DataModel and stored as the resolver property, with the ResolveAll() method invoked to update bookkeeping dictionary properties with all the dataset information. Since these dictionaries and lists are also exposed as DataModel properties, other code shouldn't need to reference the resolver property or even be aware that DataSetResolver exists. Nonetheless, it is help to understand how datasets are resolved and categorized. Properties summarized in Table 17 likely need never be referenced directly, but they underly methods for discovering available datasets and categories described in Section 3.4.2.

A dataset category is a *ds_type* argument to several DataModelMgr methods that reference the properties listed in Table 17, as summarized in Table 18.

Table 16. Shape expression Core properties.

Property	Description
core.fluenceMesh.nz	Number of fluence axial bins.
core.fluenceMesh.nr	Number of fluence radii.
core.fluenceMesh.ntheta	Number of fluence angles.
core.nass	Number of assemblies.
core.nassx	Number of assembly columns.
core.nassy	Number of assembly rows.
core.nax	Number of core/pin axial bins.
core.ndet	Number of detectors.
core.ndetax	Number of detector axial levels.
core.nfdetax	Number of fixed detector axial bins.
core.npinx	Number of pin (fuel rod) columns.
core.npiny	Number of pin (fuel rod) rows.
core.nsubax	Number of subpin axial bins.
core.nsubr	Number of subpin radii.
core.nsubtheta	Number of subpin angles.

Table 17. DataModel dataset category bookkeeping properties.

Property	Key	Value Description
dataSetAxialMesh	dataset name	axial mesh (numpy.ndarray)
dataSetAxialMeshCenters	dataset name	axial mesh centers (numpy.ndarray)
dataSetDefs	dataset category	copy of DATASET_DEFS
dataSetDefsByName	dataset name	DATASET_DEFS entry
dataSetUnitsByName	dataset name	unit string

3.3.2 Special Processing

Inevitably, the best laid plans for canonical processing always encounter special cases and conditions, as is the case when determining the category to which a dataset belongs. As mentioned above, each dataset in the first three statepoints is examined, and each unique dataset is processed. Datasets in the `CORE` group are also examined after the statepoints. Generally, a dataset's category is determined as follows.

1. Skip if the dataset name is in a skip list
2. Check for a *data_type* or *dataset_type* attribute.
3. Check for a special dataset name.
4. Apply the *match_func* for each category definition where it's defined.
5. Apply the *shape_expr* for each category with not *match_func*.

Statepoint Groups The datamodel module field SKIP_DS_NAMES defines names of datasets in statepoint groups that are to be skipped or ignored:

- *comp_ids*
- *exposure_hours*
- *from_matids*
- *mat_units*
- *rlx_xesm*
- *to_matids*

Table 18. DataModelMgr methods accepting a dataset category argument.

Method	Optional?	Description
get_data_model_dataset_names()	yes	Returns a list of dataset names for a DataModel/file and the specified category, or a dict of names by category if the argument is None
get_dataset_qnames()	yes	Returns a list of DataSetName instances for a DataModel/file and the specified category or all categories if the argument is None
get_first_dataset()	no	Returns the DataSetName for the first dataset from any file belonging to the specified category
has_dataset_type()	no	Returns a boolean indicating the existence of the category

After determining a dataset is not to be skipped, the special names *detector*, *fixed_detector*, and *radial_detector* are processed. The reason for the departure from canonical processing is the possibility of a shape "collision". For example, compare the shape expressions for the types *detector* and *:assembly* shown in Table 15. Often core.ndet and core.nass will have the same value. If core.ndetax and core.nax are the same, there is no way to distguish between the two categories. All such ambiguities are avoided when VERAOutput writers include the *data_type* attribute when storing datasets, especially detector-related datasets. However, special processing for the three detector categories accounts for the more common case where no attributes are applied to datasets.

Note that subgroups under a statepoint group are not consulted with one exception. A `QOI` group is scanned to find scalar datasets, each of which is added to the dataset ilst for the *scalar* category.

***CORE* Group** After statepoint groups are processed, the `CORE` is examined for datasets that can be categorized.[7] The module field CORE_SKIP_DS_NAMES specifies the following names that are ignored:

- *comp_ids*
- *exposure_hours*
- *from_matids*
- *mat_units*
- *rlx_xesm*
- *to_matids*
- *apitch*
- *axial_mesh*
- *core_map*
- *coresym*
- *core_sym*
- *detector*
- *detector_map*
- *detector_mesh*
- *detector_response*
- *fixed_detector*
- *fixed_detector_mesh*
- *fixed_detector_response*
- *label_format*
- *nass*
- *nax*
- *npin*
- *npinx*
- *npiny*
- *nsubr*
- *nsubtheta*
- *subpin_axial_mesh*
- *subpin_r*
- *subpin_theta*
- *xlabel*
- *ylabel*

[7] Such datasets are displayed by VERAView widgets as belonging to each statepoint.

3.3.3 Pseudo Categories

Two pseudo categories are defined and used as keys for the DataModel.dataSetDefs properties (refer to Table 16): *axials* and *core*. They can also be used as the *ds_type* argument to the DataModelMgr methods described in Table 18. Whereas *core* simply references datasets found in the `CORE\` group in a VERAOutput file, *axials* references all datasets with an axial component and are thus suitable to be plotted against axial levels. Consequently, a dataset of one of the following categories will also be returned in a request for *axials*:

- channel
- detector
- fixed_detector
- fluence
- intrapin_edits
- pin
- subpin
- subpin_cc
- subpin_r
- subpin_theta
- :assembly
- :axial
- :chan_radial
- :node

3.3.4 Vessel Geometry and Mesh

Vessel fluence datasets require special processing in VERAView, and there are two classes defined in the datamodel module for dealing with them: VesselFluenceMesh and VesselGeometry. VesselFluenceMesh encapsulates the contents of the `CORE/VesselFluenceMesh` group if it exists and contains three datasets: *axial_mesh*, *azimuthal_mesh*, and *radial_mesh*. Respectively, they define the axial levels, the angles, and radii for fluence datasets.

VesselGeometry represents the (you guessed it) geometry of the vessel, necessary for rendering fluence values in widgets. Geometry values are read from the `/CASEID/CORE/MPACT/xmlParams/CASEID/CORE` group as summarized in Table 19.

Table 19. VesselGeometry properties.

Property	Default Value	Source Dataset
coreRadius		
baffleSize	3.0	*baffle_thick*
barrelInner	187.96	*vessel_mats* and *vessel_radii*
barrelOuter	193.68	*vessel_mats* and *vessel_radii*
linerInner	219.15	*vessel_mats* and *vessel_radii*
linerOuter	219.71	*vessel_mats* and *vessel_radii*
padAngles	(45,)	*pad_azi_locs*
padArc	32	*pad_arc*
padInner	194.64	*pad_inner_radius*
padOuter	201.63	*pad_outer_radius*
vesselOuter	241.70	*vessel_mats* and *vessel_radii*

3.4 Processing VERAOutput Files

Section 2.4.2 introduced the DataModelMgr class, which is the interface for accessing data in one or more VERAOutput files. One can open and read from a single VERAOutput file with a DataModel instance, but the additional functionality provided by DataModelMgr makes it the better choice, even when a single VERAOutput file is to be processed. A reference to the DataModel instance encapsulating a single file can be retrieved as the return value of the open_file() method or the get_data_model() method, the argument to which can be a DataSetName instance (refer to Section 2.4.1) or the name of the open file (sans `.h5` extension).

For example, if two VERAOutput files named `l1_all.h5` and `c1.h5_new.h5` have been opened, DataModel objects for each are retrieved as shown in the following snippet:

```
from veraview.data.datamodel_mgr import DataModelMgr
dmgr = DataModelMgr()
dm1 = dmgr.open_file( 'l1_all.h5' )
dm2 = dmgr.open_file( 'c1.h5_new.h5' )

l1_dm = dmgr.get_data_model( 'l1_all' )
c1_dm = dmgr.get_data_model( 'c1.h5_new' )
# dm1 and l1_dm are equivalent
# dm2 and c1_dm are equivalent
```

3.4.1 Dataset Names

The DataSetName class is introduced in Section 2.4.1. Essentially, it provides a unique dataset name by pairing a model or file name and a dataset name, necessary to distinguish datasets of the same name in multiple open files. (Most VERAOutput files will have a *pin_powers* dataset in each statepoint.) DataSetName's constructor accepts three forms of arguments. If two arguments are passed, both are assumed to be strings specifying the model/file name and dataset name. A single argument can be a dictionary with *_modelName* and *_displayName* keys or a string with a vertical bar (|) separating the model and dataset names. The following snippet demonstrates the three ways to specify the same value. Note the vertical bar notation is most common.

```
DataSetName( 'l1_all', 'pin_powers' )
DataSetName( { _modelName: 'l1_all', _displayName: 'pin_powers' } )
DataSetName( 'l1_all|pin_powers' )
```

The static method DataSetName.Resolve() is used throughout VERAView to help polymorphic methods and functions obtain a DataSetName object from optional representations. It's argument can be a DataSetName (immediately returned), a two-tuple containing the model and dataset names, or a single string using the vertical bar notation. DataModelMgr methods with a DataSetName argument allow it to be passed in any of those three ways. The following example shows three ways to make the same call to get_dataset_type().

```
ds_type = dm.get_dataset_type( DataSetName( 'l1_all|pin_powers' ) )
ds_type = dm.get_dataset_type( ( 'l1_all', 'pin_powers' ) )
ds_type = dm.get_dataset_type( 'l1_all|pin_powers' )
```

Note that a DataSetName object can be passed to DataModelMgr methods with a model/file name argument, such as get_data_model() shown in a snippet above. It has three properties, described in Table 20.

Table 20. DataSetName properties.

Property	Description
displayName	dataset name in its file, sans the model name
modelName	model/file name
name	qualified name in vertical bar notation
shortName	qualified name with an abbreviated model/file name

3.4.2 File Introspection

Since VERAView is a general-purpose data browser and has no a priori knowledge about the contents of files that are opened, it must discover the displayable datasets they contain. Thus, VERAView data management modules support various types of introspection on VERAOutput files including: determining axial meshes, discovering datasets, and categorizing them. DataModelMgr methods for discovering axial mesh and time values are listed in Table 13 and Table 14, respectively.

Table 21 summarizes methods related to discovering open files and associated DataModel instances. Most methods involve datasets. Methods for finding information on available datasets and categories are shown in Table 22, and Listing 2 demonstrates their use.

Table 21. DataModelMgr model/file discovery methods.

Method	Description
GetDataModel() get_data_model()	Retrieves the DataModel for the model/file specified with a model name or DataSetName
GetDataModelCount() get_data_model_count()	Returns the number of open models/files
GetDataModelName() get_data_model_names()	Returns a list of open model/file names
GetDataModels() get_data_models()	Returns the dictionary of DataModel objects by model/file name
GetFirstDataModel() get_first_data_model()	Returns the least recently opened model/file

3.4.3 Reading Datasets

Table 23 summarizes DataModelMgr IO methods. There are three primary methods for reading data.

Method read_dataset() This method (aliased as get_h5_dataset(), GetH5DataSet(), and ReadDataSet()) retrieves a dataset from a statepoint as a h5py.Dataset object, in some cases more conveniently than doing so with h5py.File and/or h5py.Group object. It has two arguments, a DataSetName and a floating point time value. As mentioned above, the first argument can be specified as a string using the vertical bar notation for a DataSetName. The second argument is a value in the current time dataset, for which the matching statepoint is determined. The following snippet has two segments with equivalent functionality, although the knowledge that statepoint five is the correct one for an *exposure* value of 2.0 would have to be determined some other way in the second segment.

```
# Using DataModelMgr object dm
dm = DataModelMgr()
dm.set_time_dataset( 'exposure' )
dm.open_file( 'l1_all.h5' )
powers = dm.read_dataset( 'l1_all|pin_powers', 2.0 )

# Using h5py
fp = h5py.File( 'l1_all.h5', 'r' )
powers = fp.get( 'STATE_0005/pin_powers' )
```

Method read_dataset_axial_values() This method provides a generalized way to retrieve data for plotting against an axial mesh for a particular statepoint. The first, required argument is a DataSetName. Other arguments are optional in that a default is assumed:

assembly_index	0-based assembly index, defaulting to 0
detector_index	0-based detector index, defaulting to 0
fluence_addr	optional FluenceAddress instance
node_addrs	optional list of 0-based node indexes
sub_addrs	optional list pin/channel addresses as 0-based pin (col, row) pairs
time_value	value in the current time dataset, defaulting to 0

Listing 2. Example dataset introspection

```
from __future__ import print_function
from veraview.data.datamodel_mgr import *

#        Open manager and VERAOutput files
dm = DataModelMgr()
dm.open_file( '.../c1.h5_new.h5' )
dm.open_file( '.../l1_all.h5' )

#        List open files
print( dm.get_data_models().keys() )
#>>>[u'c1.h5_new', u'l1_all']

#        Check existence of specific dataset categories
print( [ dm.has_dataset_type( t ) for t in ( 'channel', ':nodal', 'pin' ) ] )
#>>>[True, False, True]

#        List dataset names for first file
print( [ d.displayName for d in dm.get_dataset_qnames( 'c1.h5_new' ) ] )
#>>>['axial_offset', 'boron', 'channel_cell_height', 'channel_flow_areas', 'channel_liquid_density', 'channel_liquid_temp', 'channel_mixture_massflux', 'channel_pressure', 'channel_void', 'core_exposure', 'core_mass', 'detector_response', 'exposure', 'exposure_efpd', 'flow', 'initial_mass', 'inlet_dens', 'keff', 'outer_timer', 'outers', 'outlet_dens', 'outlet_temp', 'pin_cladtemps', 'pin_exposures', 'pin_fueltemps', 'pin_moddens', 'pin_modtemps', 'pin_powers', 'pin_steamrate', 'pin_volumes', 'power', 'radial_powers', 'state', 'tinlet']

#        List dataset names for second file
print( [ d.displayName for d in dm.get_dataset_qnames( 'l1_all' ) ] )
#>>>['axial_offset', 'crit_boron', 'exposure', 'keff', 'pin_cladtemps', 'pin_fueltemps', 'pin_moddens', 'pin_modtemps', 'pin_powers', 'pin_volumes', 'state']

#        Get categories for specific datasets in the first file
print([
    dm.get_dataset_type( 'c1.h5_new|' + n )
    for n in ( 'channel_liquid_temp', 'core_mass', 'pin_moddens' )
    ])
#>>>['channel', 'scalar', 'pin']
```

Clearly, some arguments would not be used in combination. For example, one would not pass both *assembly_index* and *detector_index* arguments since no dataset uses both. Similarly, one or the other of *node_addrs* or *sub_addrs* must be passed to retrieve data from dataset with pin/channel dimensions. The return value is a dictionary with *mesh* and *data* keys, the former being a numpy.ndarray with the mesh values, and the latter either a numpy.ndarray (for datasets with no pin/channel or nodal dimension) or a dict of numpy.ndarrays keyed by pin/channel or node address. Listing 3 gives example uses.

Method read_dataset_time_values() Whereas read_dataset_axial_values() reads data for a specific statepoint, a plot versus time requires data from all statepoints to be read, the purpose of this method. Its arguments are a list of specification dictionary objects with the following keys:

assembly_index	optional 0-based assembly index
axial_cm	axial value in cm
detector_index	optional 0-based detector index
qds_name	DataSetName
fluence_addr	optional FluenceAddress instance
node_addrs	optional list of 0-based node indexes
sub_addrs	optional list pin/channel addresses as 0-based pin (col, row) pairs

Returned is a dictionary keyed by DataSetName of dictionaries with two keys: *data* and *times*. The latter is analogous to the *mesh* returned by read_dataset_axial_values() and is the values as a numpy.ndarray. The *data* entry is a numpy.ndarray for datasets with no pin/channel or nodal dimension and a dictionary of pin/channel or node addresses otherwise. Listing 4 gives example uses.

3.4.4 Indexing Datasets

Basic operations for navigating datasets in VERAOutput files include finding the statepoint, axial level, and assembly or detector index. Tables 14 and 13 summarize methods for mesh and time indexing. For example, in order to retrieve values for *pin* and *detector* datasets at a specific time and axial level, the statepoint index and appropriate axial level indexes must be determined. Shown in the following block are the results of `h5ls` and `h5dump` applied to a sample VERAOutput file:

```
$ h5ls maps.h5/CORE
apitch                    Dataset {SCALAR}
axial_mesh                Dataset {2}
core_map                  Dataset {15, 15}
core_sym                  Dataset {SCALAR}
detector_map              Dataset {15, 15}
fixed_detector_mesh       Dataset {6}

$ h5dump -d CORE/axial_mesh maps.h5
DATASET "CORE/axial_mesh" {
...
   DATA {
   (0): 8.34823, 375.085
   }
}

$ h5dump -d CORE/fixed_detector_mesh maps.h5
DATASET "CORE/fixed_detector_mesh" {
...
   DATA {
   (0): 375.085, 301.738, 228.39, 155.043, 81.6956, 8.34823
   }
}
```

Suppose we wish to display *pin* and *fixed_detector* dataset values at an *exposure* value of 0.7, an axial level of 200.0 cm, and for the assembly and detector in row 5 and column 0 (0-based). The snippet below demonstrates the required indexing.

Listing 3. Example code to read over axial meshes

```
from __future__ import print_function
from veraview.data.datamodel_mgr import *

#       Open manager and VERAOutput file, set time dataset
dm = DataModelMgr()
dm.open_file( '.../c1.h5_new.h5' )
dm.set_time_dataset( 'exposure' )

#       Read channel dataset
result = dm.read_dataset_axial_values(
    'c1.h5_new|channel_void',
    assembly_index = dm.core.get_assy_index( 14, 8 ),
    sub_addrs = [ ( 0, 0 ) ],
    time_value = 2.0
    )
print( result )
#>{
#>'mesh': array([
#>     12.568,  19.105,  27.673,    36.2415,  44.81,   53.378,
#>     61.946,  69.7545, 75.184,    81.1215,  89.186,  97.2505,
#>   ...
#>    372.275 ]),
#>'data':
#>  {
#>  (0, 0): array([
#>    1.00074506e-06, 1.00328831e-06, 1.00612069e-06, 1.00886102e
-06,
#>    1.01135508e-06, 1.01353553e-06, 1.01497871e-06, 1.01387548e
-06,
#>    ...
#>    1.20208720e-06])
#>  }
#>}

#       Read detector dataset
result = dm.read_dataset_axial_values(
    'c1.h5_new|detector_response',
    detector_index = dm.core.get_assy_index( 7, 7 ),
    time_value = 2.0
    )
print( result )
#>{
#>'mesh': array([
#>     12.568,   19.105,   27.673,   36.2415,  44.81,    53.378,
#>     61.946,   69.7545,  75.184,   81.1215,  89.186,   97.2505,
#>    ...
#>    372.275 ]),
#>'data': array([
#>    0.30189348, 0.37287396, 0.52348137, 0.66548079, 0.79345165,
#>    0.90648823, 1.00587288, 1.07171036, 1.05739859, 1.1695697 ,
#>    ...
#>    0.47936192, 0.38171453, 0.27888289, 0.20023012])
#>}
```

Listing 4. Example code to read over time

```
from __future__ import print_function
from veraview.data.datamodel_mgr import *

dm = DataModelMgr()
dm.open_file( '.../c1.h5_new.h5' )
dm.set_time_dataset( 'exposure' )

result = dm.read_dataset_time_values(
    dict(
        qds_name = DataSetName( 'c1.h5_new|pin_powers' ),
        axial_cm = 100.0,
        assembly_index = dm.core.get_assy_index( 7, 7 ),
        sub_addrs = [ ( 0, 0 ) ]
        ),
    dict(
        qds_name = DataSetName( 'c1.h5_new|pin_fueltemps' ),
        axial_cm = 100.0,
        assembly_index = dm.core.get_assy_index( 10, 2 ),
        sub_addrs = [ ( 7, 10 ) ]
        )
    )
print( result )
#>{
#>datamodel.DataSetName(c1.h5_new|pin_fueltemps):
#>  {
#>  'data':
#>    {
#>    (7, 10): array([
#>       291.66666962, 537.99253343, 646.04130055, 639.56009215,
#>       645.12436206, 645.9631644 , 646.4708324 , 647.84749895,
#>       ...
#>       648.52747465, 602.63216741])
#>    },
#>  'times': array([
#>        0.         , 0.346      , 1.23       , 1.919      , 2.457      ,
#>       ...
#>       14.33500001, 15.068      , 15.06800002, 15.308      , 15.774      ])
#>  },
#>datamodel.DataSetName(c1.h5_new|pin_powers):
#>  {
#>  'data':
#>    {
#>    (0, 0): array([
#>       1.28031806, 1.44919913, 1.53924774, 1.55390458, 1.55694685,
#>       1.55398506, 1.53781649, 1.52567535, 1.51285773, 1.5023643 ,
#>       ...
#>       1.16464324, 1.15190304, 1.1490925 , 1.15321333, 1.11057501])
#>    },
#>  'times': array([
#>        0.         , 0.346      , 1.23       , 1.919      , 2.457      ,
#>       ...
#>       14.33500001, 15.068      , 15.06800002, 15.308      , 15.774      ])
#>  }
#>}
```

```
dm = DataModelMgr()
dm.open_file( 'maps.h5' )
dm.set_time_dataset( 'exposure' )

axial = dm.get_axial_value( 'maps|fixed_detector_response', cm = 200.0 )
powers = dm.read_dataset( 'maps|pin_powers', 0.7 )
fdr = dm.read_dataset( 'maps|fixed_detector_response', 0.7 )

# axial cut of pin_powers
print( powers[ :, :, axial.pinIndex, dm.core.get_assy_index( 0, 5 ) ] )

# fixed_detector_response value
fdr[ axial.fixedDetectorIndex, dm.core.get_detector_index( 0, 5 ) ]
#>0.5796581503216054
```

3.5 Finding Min and Max Values

DataModelMgr methods for searching for minimum and maximum values in datasets are shown in Table 24. There are methods for searching a single dataset of category *channel*, *fluence*, or *pin* as well as a searching multiple datasets at once for a combined min or max. Datasets of other categories can be searched, but the resulting coordinates dictionary is limited to the keys identified in Table 24. The following snippet demonstrates searching for the maximum value of a *channel* dataset. Both the time and assembly index arguments are passed as -1 to search across all times and assemblies.

```
dm = DataModelMgr()
dm.open_file( '.../c1.h5_new.h5' )
dm.set_time_dataset( 'exposure' )

coords = dm.find_channel_min_max_value( 'max', 'c1.h5_new|
    channel_pressure', -1.0, 1 )
print( coords )
#>{
#>'assembly_addr': (1, 8, 7),
#>'axial_value':
#>  {
#>  'cm': 12.568,
#>  'cm_bin': array([10.315, 14.821]),
#>  'detector': 0,
#>  'fixed_detector': -1,
#>  'fluence': -1,
#>  'pin': 0,
#>  'subpin': 0,
#>  'value': 12.568,
#>  '__jsonclass__': 'veraview.data.datamodel.AxialValue'
#>  },
#>'state_index': 5,
#>'sub_addr': (8, 9),
#>'time_value': 2.994
#>}
```

The resulting coordinates dictionary shows the maximum value occurs at (using 0-based indexes):

- assembly 1 at column 8, row 7
- axial center 12.568, index 0
- channel column 8, row 9
- *exposure* 2.994, or statepoint index 5

The next snippet demonstrates how to use the coords to retrieve the value.

```
axial = results.get( 'axial_value' )
chan_addr = results.get( 'sub_addr' )
data = dm.read_dataset( 'c1.h5_new|channel_pressure', results.get( '
    time_value' ) )
max_value = data[
     chan_addr[ 1 ], chan_addr[ 0 ],
     axial.pinIndex, results.get( 'assembly_addr' )[ 0 ]
     ]
print( 'max_value=', max_value )
#>max_value = 156.41321052691796
```

One can also specify a specific assembly and/or time in which to search, as demonstrated in the following snippet.

```
dm = DataModelMgr()
dm.open_file( '.../c1.h5_new.h5' )
dm.set_time_dataset( 'exposure' )

coords = dm.find_multi_dataset_min_max_value(
     'max', -1.0, -1, None,
     'c1.h5_new|pin_cladtemps',
     'c1.h5_new|pin_fueltemps',
     'c1.h5_new|pin_modtemps'
     )
print( coords )
#>{
#>'assembly_addr': (16, 7, 9)
#>'axial_value':
#>   {
#>   'cm': 12.568,
#>   'cm_bin': array([10.315, 14.821]),
#>   'detector': 0,
#>   'fixed_detector': -1,
#>   'fluence': -1,
#>   'pin': 0,
#>   'subpin': 0,
#>   'value': 12.568,
#>   '__jsonclass__': 'veraview.data.datamodel.AxialValue'
#>   },
#>'state_index': 1,
#>'sub_addr': (8, 9),
#>'time_value': 0.346
#>}
```

A search across multiple datasets is demonstrated in the next snippet.

```
dm = DataModelMgr()
dm.open_file( '.../c1.h5_new.h5' )
dm.set_time_dataset( 'exposure' )

coords = dm.find_multi_dataset_min_max_value(
     'max', -1.0, -1, None,
     'c1.h5_new|pin_cladtemps',
     'c1.h5_new|pin_fueltemps',
     'c1.h5_new|pin_modtemps'
     )
#
print( coords )
#>{
```

```
#>'assembly_addr': (36, 11, 11)
#>'axial_value':
#>  {
#>  'cm': 165.5765,
#>  'cm_bin': array([161.544, 169.609]),
#>  'detector': 20,
#>  'fixed_detector': -1,
#>  'fluence': -1,
#>  'pin': 20,
#>  'subpin': 20,
#>  'value': 165.5765,
#>  '__jsonclass__': 'veraview.data.datamodel.AxialValue'
#>  },
#>'state_index': 2,
#>'sub_addr': (3, 4),
#>'time_value': 1.23
#>}
```

This result indicates the coordinate where the maximum value among the three datasets can be found.

3.6 Creating Difference Datasets

The DataModelMgr CreateDiffDataSet() method (aliased as create_diff_dataset() will create a new dataset representing a difference between two existing datasets.[8] Its arguments are:

ref_qds_name	reference DataSetName, x in $x - y$
comp_qds_name	comparison DataSetName, y in $x - y$
diff_qds_name	name of difference dataset to create
diff_mode	one of *delta*, *pct*, defaulting to the former
interp_mode	one of *linear*, *quad*, *cubic*, defaulting to *linear*
listener	optional callable(message, cur_step, step_count)

The difference dataset is created in the *comp_qds_name* DataModel or file, and a fully-qualified DataSetName instance is returned to represent it. In order to be differenced two datasets must be of the same category and have the same geometry, meaning they have the same shape except for the axial dimension.[9] If the axial meshes differ, the reference dataset mesh will be interpolated onto the comparison dataset mesh using the specified interpolation mode.

Since the difference dataset is stored in the comparison file, it determines time. Reference dataset statepoints are chosen to match the value of the current time dataset in each comparison statepoint. The following snippet demonstrates creating a difference datset.

```
dm = DataModelMgr()
dm.open_file( '.../c1.h5_new.h5' )
dm.open_file( '.../l1_all.h5' )
dm.set_time_dataset( 'exposure' )

diff_qname = dm.create_diff_dataset(
    'l1_all|pin_powers', 'c1.h5_new|pin_powers', 'diff_pin_powers',
    diff_mode = 'delta', interp_mode = 'cubic'
    )
print( 'new name=', diff_qname.name )
#>new name= c1.h5_new|diff_pin_powers
```

[8] Either dataset can be a derived dataset or a difference dataset.
[9] Otherwise, both files could not have been opened simultaneously.

3.7 Creating Derived Datasets

Derived datasets are created by applying some aggregation operation, usually averaging, on a dataset across one or more of its dimensions. Since derivation is applied to a single dataset, the derivation method, CreateDerivedDataSet2() (aliased as create_derived_dataset2()), is defined in the DataModel class instead of DataModelMgr. Note that unless a *factors* attribute is defined for the source dataset, VERAView calculates appropriate weights and factors based on the category of the source dataset for the derivation (refer to Section 3.1). Arguments to create_derived_dataset2() follow:

src_ds_name	name of the source dataset
avg_axis	index or tuple of indexes of axes across which to aggregate
der_ds_name	name of the derived dataset to create
der_method	one of 'avg' (default), 'rms', 'stddev', or 'sum'
use_factors	boolean specifying application of factors, defaults to True
callback	optional callable(int) passed the 0-based state index being processed
return value	name of the newly added dataset

Given the shape of a typical *pin* or *channel* dataset, $(npiny, npinx, nax, nass)$, Table 25 shows the axes over which to aggregate to create common derivations. Note that a derived dataset can be created from another derived dataset. For example, an axial average can be created from an assembly average.

The following snippet demonstrates creation of an axial *pin_powers* average:

```
dm = DataModelMgr()
model = dm.open_file( '.../l1_all.h5' )
dm.set_time_dataset( 'exposure' )

new_name = \
  model.create_derived_dataset2( 'pin_powers', ( 0, 1, 3 ), '
      axial_pin_powers' )
print( 'new_name=', new_name )
#>new_name= axial_pin_powers
dset = dm.read_dataset( 'l1_all|' + new_name, 0.0 )
print( np.reshape( dset, ( 49, ) )[:] )
#>[0.22282568 0.29219183 0.43822772 0.57612515 0.70245444 0.81645842
#> 0.91830121 0.99702987 1.0052433  1.10707051 1.18087765 1.23303103
#> 1.2762625  1.31151611 1.32487887 1.27866    1.35505694 1.38404443
#> 1.39296901 1.39735667 1.39762212 1.3787901  1.31059727 1.36795788
#> 1.37250185 1.3584039  1.34119641 1.32082433 1.28292846 1.20607861
#> 1.24316054 1.22681675 1.19251675 1.1543749  1.1120638  1.0535918
#> 0.97074352 0.97748534 0.93197764 0.86989396 0.80258652 0.72979329
#> 0.64446529 0.55755068 0.52111386 0.43950941 0.34661459 0.24900481
#> 0.18089014]
```

One would calculate core standard deviation *pin_powers* as follows:

```
new_name = model.create_derived_dataset2(
    'pin_powers', ( 0, 1, 2, 3 ), 'core_stddev_pin_powers',
    der_method = 'stddev'
    )
print( 'new_name=', new_name )
#>new_name= core_stddev_pin_powers

dname = DataSetName( 'l1_all', new_name )
result = dm.read_dataset_time_values( dict( qds_name = dname, axial_cm =
    0.0 ) )
pair = result.get( dname )
times = pair.get( 'times' )
data = pair.get( 'data' )
data = np.reshape( data, ( data.shape[ 0 ], ) )
print( 'times: {1}{0}data: {2}'.format( os.linesep, times, data ) )
```

```
#>times: [ 0.      0.384   0.768   1.152   1.728   2.304   3.072   3.841   4.609
     6.145
#>   7.681   9.217  10.754  12.29   13.826  15.362]
#>data:  [0.42670226 0.3967886  0.39625996 0.39724934 0.3900215
    0.38100489
#> 0.36637923 0.35366141 0.34190083 0.32071277 0.30181143 0.28198781
#> 0.26295879 0.24430029 0.22710797 0.21233243]
```

Table 22. DataModelMgr dataset discovery methods.

Method	Description
GetDataModelDataSetNames() get_data_model_dataset_names()	For the specified model/file returns a list dataset names by specified category or a dictionary of names by category
GetDataSetDefByQName() get_dataset_def_by_qname()	Returns the dataset category definition (from DATASET_DEFS for a specified dataset if found
GetDataSetDisplayName() get_dataset_display_name()	Returns an optimal string for representing a dataset given currently open files
GetDataSetHasSubAddr() get_dataset_has_sub_addr()	Returns True if the dataset has pin or channel dimensions
GetDataSetScaleType() get_dataset_scale_type()	Returns the scale type (*linear* or *log*) for a single dataset
GetDataSetScaleTypeAll() get_dataset_scale_type_all()	Returns the scale type appropriate for a lis of datasets
GetDataSetThreshold() get_dataset_threshold()	Returns any assigned threshold range for a specific dataset or the dictionary of all threshold ranges by DataSetName
GetDataSetQNames() get_dataset_qnames()	Returns a list of DataSetNames in a specified category or all categories for a model/file
GetDataSetType() get_dataset_type()	Returns type category name for a specified DataSet-Name if found
GetDataSetTypes() get_dataset_types()	Returns a set of all category names
HasData() has_data()	Returns true if there are open models/files
HasDataSet() has_dataset()	Checks for existing of a DataSetName at a specified time
HasDataSetType() has_dataset_type()	Checks for existence of a dataset of a specified category
HasNodalDataSetType() has_nodal_dataset_type()	Checks for existence of a dataset of category *:node* or *:radial_node*
IsChannelType() is_channel_type()	Checks if a DataSetName has *channel* pin dimensions
IsDerivedDataSet() is_derived_dataset()	Checks if a DataSetName is of a derived category

Table 23. DataModelMgr dataset IO methods.

Method	Description
GetFirstDataSet() get_first_dataset()	Returns the first dataset found belonging to a specified category
GetH5DataSet() get_h5_dataset() read_dataset() ReadDataSet()	Retrieves the h5py.Dataset specified by DataSetName and time value (using the current time dataset)
GetRange() get_range()	Retrieves the range for a DataSetName, calculating it if necessary
GetRangeAll() get_range_all()	Calculates the range for a list of DataSetNames
ReadDataSetAxialValues() read_dataset_axial_values()	Retrieves dataset axial slices over other dimensions
ReadDataSetTimeValues() read_dataset_Time_values()	Assembles arrays of values by time for specified datasets and indexes
SetDataSetThreshold() set_dataset_threshold()	Assigns a threshold range to a DataSetName

Table 24. DataModelMgr dataset min max search methods.

Method	Description
FindChannelMinMaxValue() find_channel_min_max_value()	Finds the coordinates[1] of the channel with the minimum or maximum value
FindFluenceMinMaxValue() find_fluence_min_max_value()	Finds the coordinates[2] of the fluence with the minimum or maximum value
FindMultiDataSetMinMaxValue() find_multi_dataset_min_max_value()	Finds the coordinates[1] of the minimum or maximum value of one or more datasets
FindPinMinMaxValue() find_pin_min_max_value()	Finds the coordinates[1] of the pin with the minimum or maximum value

[1] A dictionary with one or more of key *assembly_addr*, *axial_value*, *sub_addr*, *time_value*
[2] A dictionary with one or more of key *axial_value*, *fluence_addr*, *time_value*

Table 25. Common derivation axes.

Derived Name	Axes	Dataset Category	Result Shape
assembly	(0, 1)	*:assembly*	(1, 1, :, :)
axial	(0, 1, 3)	*:axial*	(1, 1, :, 1)
core	(0, 1, 2, 3)	*:core*	(1, 1, 1, 1)
radial	(2,)	*:radial*	(:, :, 1, :)
radial assembly	(2, 3)	*:radial_assembly*	(:, :, 1, 1)

4 WIDGET FRAMEWORK

Each panel in a VERAView window is a *widget*, a wx.Panel extension responsible for data display and/or user selections. A variety of widgets have been implemented for VERAView as introduced in Sections 2.6 and 2.6.2. In effect a VeraViewFrame is a container for widgets, which are merely components that adhere to a framework. Software patterns employed in VERAView's widget framework include dependency injection and the factory and prototype patterns.

4.1 Widget Registration

At present, registration of widget implementation classes and their representation in the VERAView tool bar is encoded in two veraview module variables: TOOLBAR_ITEMS and WIDGET_MAP. The former is an array of dictionaries (i.e., Python dict instances) specifying properties of each widget, and the latter is a dict mapping widget names to implementation classes.

> Note the registration of widget classes should be placed in a configuration file and likely will be in the future. Then the two variables would be read from the configure file. However, given the ease of modifying Python files comprising an application, this change has very low priority in the list of new feature requests.

Table 26. TOOLBAR_ITEMS keys.

Key	Description
func	optional function defining availability of the widget
icon	path to icon file representing availability of the widget
iconDisable	path to icon file representing unavailability of the widget
type	dataset category or type processed by the widget[1]
widget	widget alias

[1]Refer to Section 3.3

Table 26 describes the keys in a TOOLBAR_ITEMS entry. Icon files are assumed to reside in the `res/` subdirectory, which exists under the top level veraview package directory. Refer to Section 2.9.

Whereas *type* simply specifies a dataset category or type, *func* specifies a function with a single parameter of type veraview.data.datamodel_mgr.DataModelMgr. If *type* is specified, the availability of the widget is determined by existence of a dataset of that type. If *func* is specified, it overrides *type* and is invoked with the DataModelMgr. A return value of True makes the widget available. (Details on the data processing classes, including DataModelMgr are provided in Section 3.)

The *Pin Surface Map* definition, listed below, illustrates the simpler form.

```
{
'widget': 'Pin Surface Map',
'icon': 'PinSurfaceMap.1.32.png',
'iconDisabled': 'PinSurfaceMap.1.disabled.32.png',
'type': 'subpin_cc'
}
```

An example using *func* is the *Core 2D View* entry, listed below.

```
{
'widget': 'Core 2D View',
'icon': 'Core2DView.1.32.png',
'iconDisabled': 'Core2DView.disabled.1.32.png',
'type': '',
'func': lambda d: d.core is not None and
    d.core.coreMap is not None and
    d.core.coreMap.shape[ 0 ] > 1 and
```

```
    d.core.coreMap.shape[ 1 ] > 1 and
    (d.HasDataSetType( 'channel' ) or d.HasDataSetType( 'pin' ) or
     d.HasDataSetType( ':assembly' ) or
     d.HasDataSetType( ':chan_radial' ) or
     d.HasDataSetType( ':node' ) or
     d.HasDataSetType( ':radial' ) or
     d.HasDataSetType( ':radial_assembly' ) or
     d.HasDataSetType( ':radial_node' ))
}
```

Keys in the WIDGET_MAP variable match the *widget* values in TOOLBAR_ITEMS entries, and values are full class paths to the widget implementation. WIDGET_MAP is listed below.

```
WIDGET_MAP = \
  {
  'Assembly 2D View': 'veraview.widget.assembly_view.Assembly2DView',
  'Axial Plots': 'veraview.widget.axial_plot.AxialPlot',
  'Core 2D View': 'veraview.widget.core_view.Core2DView',
  'Core Axial 2D View': 'veraview.widget.core_axial_view.CoreAxial2DView
      ',
  'Data Analysis View': 'veraview.widget.data_analysis_view.
      DataAnalysisView',
  'Detector 2D Multi View': 'veraview.widget.detector_multi_view.
      Detector2DMultiView',
  'Intrapin Edits Assembly 2D View': 'veraview.widget.
      intrapin_edits_assembly_view.IntraPinEditsAssembly2DView',
  'Intrapin Edits Plot': 'veraview.widget.intrapin_edits_plot.
      IntraPinEditsPlot',
  'Pin Surface Map': 'veraview.widget.pin_surface_map.PinSurfaceMap',
  'Subpin Length Plot': 'veraview.widget.subpin_len_plot.SubPinLengthPlot
      ',
  'Pin Surface Map': 'veraview.widget.pin_surface_map.PinSurfaceMap',
  'Table View': 'veraview.widget.table_view.TableView',
  'Time Plots': 'veraview.widget.time_plots.TimePlots',
  'Vessel Core 2D View': 'veraview.widget.vessel_core_view.
      VesselCore2DView',
  'Vessel Core Axial 2D View': 'veraview.widget.vessel_core_axial_view.
      VesselCoreAxial2DView',
  'Volume 3D View': 'veraview.view3d.volume_view.Volume3DView',
  'Volume Slicer 3D View': 'veraview.view3d.slicer_view.Slicer3DView'
  }
```

For example, the path to the class implementing the *Core 2D View* widget is veraview.widget.core_view.Core2DView, which means the Core2DView class in the core_view module of the veraview.widget package. Note a widget may be defined in WIDGET_MAP, but without a corresponding entry in TOOLBAR_ITEMS it will never be available.

4.2 Widget Classes

Figure 4 illustrates the relationships between widget classes in the veraview.widget package, three of which are building blocks of the widget framework:

- Widget (veraview.widget.widget module)
- PlotWidget (veraview.widget.plot_widget module)
- RasterWidget (veraview.widget.raster_widget module)

Much of the widget framework is defined in the Widget class. Widget is the "interface" or "abstract base class" for all widgets. [10] Although Python's dynamic typing precludes any type checking, other VERAView components assume widgets provide the methods defined in Widget, and Widget provides enough default processing to make it much easier to extend it rather then implement the implied interface from scratch. One of the other components that assumes the Widget interface is WidgetContainer (veraview.widget.widgetcontainer module), which wraps Widget instances and provides things like dataset selection, a widget menu, and a close button.

Two classes extend Widget directly: PlotWidget and RasterWidget. They serve as base classes for two types of graphical widgets. Whereas PlotWidget assumes *matplotlib* provides the data representation, RasterWidget assumes the visualization is created as an image or bitmap. Figure 4 shows the implementation classes that extend the two respective base classes. However, a widget implementation can extend wx.Widget directly if is not graphical or will create a data representation another way. There are such widget implementations shown in Figure 4.

Widgets are instantiated by a WidgetContainer, which calls the Init() method to initialize the widget. Widget's constructor (__init__() method) initializes fields and properties and invokes _InitUI(), which must be overridden by extensions to create needed GUI components. Widget's Init() implementation invokes HandleStateChange() passing STATE_CHANGE_init as the argument. The default HandleStateChange() implementation invokes _LoadDataModel() and then calls wx.CallAfter() with self.UpdateState() as the callable to be invoked later.

4.3 Anatomy of a Widget

There are some necessary pieces for a widget to be usable in VERAView, starting with icons for the toolbar. Two icons with a width and height of 32 pixels representing the enabled and disabled state of the icon, respectively, must exist. These are specified as the *icon* and *iconDisabled* keys of a TOOLBAR_ITEMS entry. The value of the keys is a path to the file. Full paths may be specified, otherwise the path is relative to the `res/` directory (refer to Section 2.9).

Next, a widget implementation class must be provided and specified as the value for the widget name (or title) key in WIDGET_MAP. The following snippet demonstrates the registration of ExampleWidget, implemented in the example_widget module in the veraview.widget package.

```
{
TOOLBAR_ITEMS = \
  [
  ...
    {
    'widget': 'Example Widget',
    'icon': 'ExampleWidget.32.png',
    'iconDisabled': 'ExampleWidget.disabled.32.png',
    'type': 'pin'
    }
  ]
...
WIDGET_MAP = \
  {
  ...
  'Example Widget': 'veraview.widget.example_widget.ExampleWidget'
  }
```

The source for ExampleWidget is given in Listing 5. With the exception of the _CreateClipboardImage() method override, it's close to the proverbial "hello world" example. It simply displays some random Latin words in a wx.TextCtrl and does not respond to any VERAView events.

Listing 5. Simple widget implementation

```
import random
from .widget import *

```

[10] Python has no such constructs, but exceptions are raised in methods which extensions must override, effectively enforcing an abstract base class.

```
WORDS = [
  'angolo', 'controllo', 'cura', 'decimale', 'esatto',
  'generale', 'insieme', 'notte', 'ottenuto', 'passato',
  'recedere', 'recedo', 'recessi', 'recessum', 'reliquus',
  'sinere', 'sino', 'situm', 'sivi', 'stella',
  'tagliente', 'tale', 'tardus', 'temperatura', 'tratto',
  'tubo'
  ]


class ExampleWidget( Widget ):


  def __init__( self, container, id = -1, **kwargs ):
    self._value = 'Just starting...'
    self._text_ctrl = None

    super( ExampleWidget, self ).__init__( container, id )
  #end __init__


  def _CreateClipboardData( self, mode = 'displayed' ):
    return  self._text_ctrl.GetText()
  #end _CreateClipboardData


  def _CreateClipboardImage( self ):
    text = self._value
    font = self._text_ctrl.GetFont()

    bmap = wx.EmptyBitmapRGBA( 400, 200 )
    dc = wx.MemoryDC()
    dc.SelectObject( bmap )
    dc.SetFont( font )

    vert_pad = 2
    horz_pad = 5
    row_size = dc.GetTextExtent( 'X' * 40 )
    im_wd = (horz_pad << 1) + row_size[ 0 ]
    im_ht = row_size[ 1 ] * ((len( text ) + 39) / 40) + (vert_pad << 1)

    dc.SelectObject( wx.NullBitmap )
    del dc
    del bmap

    bmap, dc = self._CreateEmptyBitmapAndDC( im_wd, im_ht )
    gc = self._CreateGraphicsContext( dc )
    gc.SetFont( font, wx.Colour( 0, 0, 0, 255 ) )
    trans_brush = gc.CreateBrush( wx.TheBrushList.FindOrCreateBrush(
        wx.WHITE, wx.TRANSPARENT
        ) )

    y = row_size[ 1 ] + vert_pad
    while len( text ) > 0:
      gc.DrawText( text[ 0 : 40 ], horz_pad, y, trans_brush )
      text = text[ 41 : ]
      y += row_size[ 1 ]

```

```
    dc.SelectObject( wx.NullBitmap )
    return  bmap
  #end _CreateClipboardImage


  def GetAnimationIndexes( self ):
    return  ()
  #end GetAnimationIndexes


  def GetDataSetTypes( self ):
    return  ( 'pin', )
  #end GetDataSetTypes


  def GetDataSetDisplayMode( self ):
    return  'selected'
  #end GetDataSetDisplayMode


  def GetEventLockSet( self ):
    return  set([])
  #end GetEventLockSet


  def GetTitle( self ):
    return  'Example Widget'
  #end GetTitle


  def GetUsesScaleAndCmap( self ):
    return  False
  #end GetUsesScaleAndCmap


  def _InitUI( self ):
    self._text_ctrl = wx.TextCtrl(
        self, wx.ID_ANY,
        value = self._value,
        size = ( 200, 100 ),
        style = wx.TE_MULTILINE | wx.TE_READONLY | wx.TE_WORDWRAP
        )

    sizer = wx.BoxSizer( wx.VERTICAL )
    sizer.Add(
        self._text_ctrl, 1,
        wx.ALIGN_LEFT | wx.ALIGN_TOP | wx.ALL | wx.EXPAND, 4
        )
    self.SetAutoLayout( True )
    self.SetSizer( sizer )

    self.Bind( wx.EVT_CONTEXT_MENU, self._OnContextMenu )
  #end _InitUI


  def IsDataSetScaleCapable( self ):
    return  False
  #end IsDataSetScaleCapable

```

```
  def _LoadDataModel( self, reason ):
    if not self.isLoading:
      wx.CallAfter( self.UpdateState, rebuild = True )
  #end _LoadDataModel


  def LoadProps( self, props_dict ):
    for k in ( 'value', ):
      if k in props_dict:
        setattr( self, '_' + k, props_dict[ k ] )

    super( ExampleWidget, self ).LoadProps( props_dict )
    self.container.dataSetMenu.UpdateAllMenus()
    wx.CallAfter( self.UpdateState, rebuild = True )
  #end LoadProps


  def _OnSize( self, ev ):
    self.Redraw()
  #end _OnSize


  def Redraw( self ):
    self._BusyDoOp( self.UpdateState, rebuild = True )
  #end Redraw


  def SaveProps( self, props_dict, for_drag = False ):
    super( ExampleWidget, self ).SaveProps( props_dict, for_drag = for_drag )

    for k in ( 'value', ):
      props_dict[ k ] = getattr( self, '_' + k )
  #end SaveProps


  def UpdateState( self, **kwargs ):
    if bool( self ):
      kwargs = self._UpdateStateValues( **kwargs )
      if kwargs.get( 'rebuild', False ):
        self._text_ctrl.ChangeValue( self._value )
        self._text_ctrl.Refresh()
  #end UpdateState


  def _UpdateStateValues( self, **kwargs ):
    rebuild = kwargs.get( 'rebuild', kwargs.get( 'force_redraw', False ) )
    if rebuild:
      self._value = ' '.join( random.sample( WORDS, 10 ) )
      kwargs[ 'rebuild' ] = True

    return  kwargs
  #end _UpdateStateValues

#end ExampleWidget
```

4.3.1 Properties

Properties defined by Widget and inherited by all extensions are described in Table 27. Of particular importance are dmgr and state. The former is also accessible as the dataModelMgr property of state and is the DataModelMgr instance used throughout VERAView. Similarly, state is a State instance shared across all widgets and components in VERAView (refer to Section 2.5). Widgets manage local state values as instance properties, as will be seen in Widget subclasses described below.

Table 27. Widget object properties.

Property	Description
busy	boolean indicating whether or not a busy operation is underway
container	reference to the owning WidgetContainer
dataRangeDialog	DataRangeDialog reference, created on demand
dataRangeValues	DataRangeValues object with user range settings
dmgr	DataModelMgr instance
formatter	RangeScaler instance
isLoading	boolean indicating LoadProps() is on the call stack
logger	logging.Logger instance to use for logging
menuDef	list of menu item definition dicts
popupMenu	wx.Menu for popups, lazily created
state	State instance with current selections

4.3.2 Framework Methods

In effect the prototype pattern is the basis for VERAView's widget framework, with dependency injection used in the relationship between WidgetContainer instances and the Widgets they enclose, as shown in Figure 6. As such, several methods are defined for the Widget class. Somewhat inconsistently, methods are named with an underscore prefix to indicate private versus public visibility, similar to *private* or *protected* versus *public* access modifiers in Java. Since Python enforces no access control for methods, the intent is to be informative rather then prescriptive. Whereas public methods are intended to be called by objects of classes unrelated to a widget, private methods are used within a widget's class hierarchy.

Table 28 summarizes private Widget methods, with the "Noop" column indicating whether or not the base implementation is empty and must be overridden by subclasses. Public methods are listed in Table 29, with the "Injection Call" column indicating it is a method called by WidgetContainer on its Widget instance. Finally Table 30 lists support methods intended to assist widget implementations.

Basically, there are three kinds or types of widgets anticipated for VERAView: plots, raster images, and other. These are described in the following Sections with references to examples in the VERAView implementation.

4.4 Plot Widgets

PlotWidget plays the role of abstract base class for widgets using *matplotlib*. It defines a number of properties (refer to Table 31) and framework methods (refer to Table 32). If an example is worth the proverbial one thousand words of documentation, AxialPlot (veraview.widget.axial_plot module) and TimePlots (veraview.widget.time_plots module) provide the best basis for understanding how plot widgets work. They have similar implementations. Both introduce properties for managing plot data and user selections. The dataSetValues property is a dictionary storing the plot data as derived from visible datasets. User selections are managed with another dictionary property, dataSetSelections. These classes also demonstrate storing widget instance state values with the following properties:

Table 28. Widget framework private methods.

Method	Noop	Description
_CreateClipboardData()	Y	Creates a textual, usually comma-separated values (CSV) representation of the data displayed in the widget
_CreateClipboardImage()	Y	Creates a bitmap representation of the widget
_CreateLegendBitmap()	N	Creates a legend bitmap; the default implementation uses a Legend3 instance
_CreateMenuDef()	N	Creates a hierarchical dict suitable for WidgetContainer._CreateMenuFromDef()
_CreatePopupMenu()	N	Creates a menu by invoking WidgetContainer._CreateMenuFromDef()
_GetCurAssemblyIndex()	N	Reflects to find a assemblyAddr or then a detectorAddr field or property and return the index
_InitUI()	Y[1]	Create widget GUI components
_LoadDataModel()	Y	Initialize widget properties on initialization of the DataModelMgr or when a DataModel is opened or closed
_OnContextMenu()	N	Shows the popupMenu
_OnCopyData()	N	Handles a clipboard copy operation
_OnCopyImage()	N	Handles a clipboard copy operation
_OnCustomScale()	N	Shows up the dataRangeDialog
_OnFindMinMax()	Y	Invoked on a min/max operation
_OnSize()	Y	Invoked on a resize

[1]The default implementation raises an exception, so subclasses must override.

- assemblyAddr
- auxNodeAddrs
- auxSubAddrs
- axialValue
- fluenceAddr
- nodeAddr
- subAddr

Prominent Widget methods overridden in PlotWidget include:

- _CreateMenuDef()
- _DoUpdatePlot()
- _LoadDataModelValues()
- _OnEditDataSetProps()
- _UpdateDataSetValues()
- _UpdateStateValues()

Note _DoUpdatePlot() and _UpdateDataSetValues() are added to the framework by PlotWidget specifically for widgets based on *matplotlib*. _CreateMenuDef() adds items to the widget menu. _UpdateStateValues() is invoked from _UpdateState() on state changes and updates the object state value properties. _UpdateState() then calls _UpdateDataSetValues() to rebuild dataSetValues from dataSetSelections and data read via dmgr. _DoUpdatePlot() is called later and uses dataSetValues to update the matplotlib.axes.Axes objects ((ax and ax2), set labels and titles, and call plot() on the axes. _OnEditDataSetProps() is the callback for the *Edit Dataset Properties* widget menu item. It lazily creates the dataSetDialog property (PlotDataSetPropsDialog instance) to allow the user to arrange datasets on axes.

4.5 Raster Widgets

RasterWidget plays the role of abstract base class for widgets that create an image or bitmap to represent the data and provides the functionality to manage a cache of bitmaps and invoke re-rendering only when necessary. Two examples worth reviewing are Core2DView (veraview.widget.core_view module) and Assembly2DView (veraview.widget.assembly_view module). RasterWidget also defines properties and framework methods, listed in Table 33 and 34, respectively. Other RasterWidget properties are used internally to manage the bitmap cache, handle zoom/unzoom events and do the associated bookkeeping, and other tasks. Table 35 lists support methods implemented in RasterWidget that are inherited by subclasses.

In addition to the size of the drawing region, bitmap caching and management is based on a widget-specific "state tuple", which includes current state properties (i.e., user selections) that matter to the widget. If the state tuple changes, RasterWidget looks in the cache to see if the bitmap for that tuple has already been created. If so, that becomes the displayed bitmap. Otherwise, the bitmap is created and stored in the cache with the tuple as the key. If the drawing size changes, all cached bitmaps are discarded, and a new bitmap for the current tuple must be recreated.

Bitmaps are created in a sequence of two calls: CreateDrawConfig() and CreateRasterImage(). The former creates the dictionary stored in the config property and then used by the latter. The configuration includes calculated pixel sizes for drawing elements such as pins and assemblies.

4.6 Other Widgets

It is not necessary for a widget to use *matplotlib* or create a bitmap, such as when a widget is not graphical at all. A reference example of such a widget implementation is the TableView widget in the veraview.widget.table_view module. It uses a wx.SplitterWindow and UltimateListCtrl (wx.lib.agw.ultimatelistctrl module) to show values of multiple datasets at current state property indexes. As a Widget extension it adheres to the widget framework and thus overrides the following framework methods:

- _CreateClipboardData()
- _CreateClipboardImage()
- _CreateMenuDef()
- _CreatePrintImage()
- _GetAnimationIndexes()
- _GetDataSetDisplayMode()
- _GetDataSetTypes()
- _GetEventLockSet()
- _GetTitle()
- _GetUsesScaleAndCmap()
- _GetVisibleDataSets()
- _InitUI()
- _IsDataSetVisible()
- _LoadDataModel()
- _LoadProps()
- _Redraw()
- _SaveProps()
- _SetVisibleDataSets()
- _ToggleDataSetVisible()
- _UpdateState()

4.7 Dataset Menus

One of the most reused VERAView components is the DataSetsMenu class in the veraview.widget.bean.dataset_menu module. Every menu displaying datasets for selection is an instance of DataSetsMenu. Each widget has a one. It extends DataModelMenu which in turns extends wx.Menu. Whereas DataModelMenu pertains to a single DataModel file, DataSetsMenu can processing multiple DataModels and thus represents a DataModelMgr. DataSetsMenu listens to State and DataModelMgr events and dynamically rebuilds the menu as necessary to represent available files and datasets within the files.

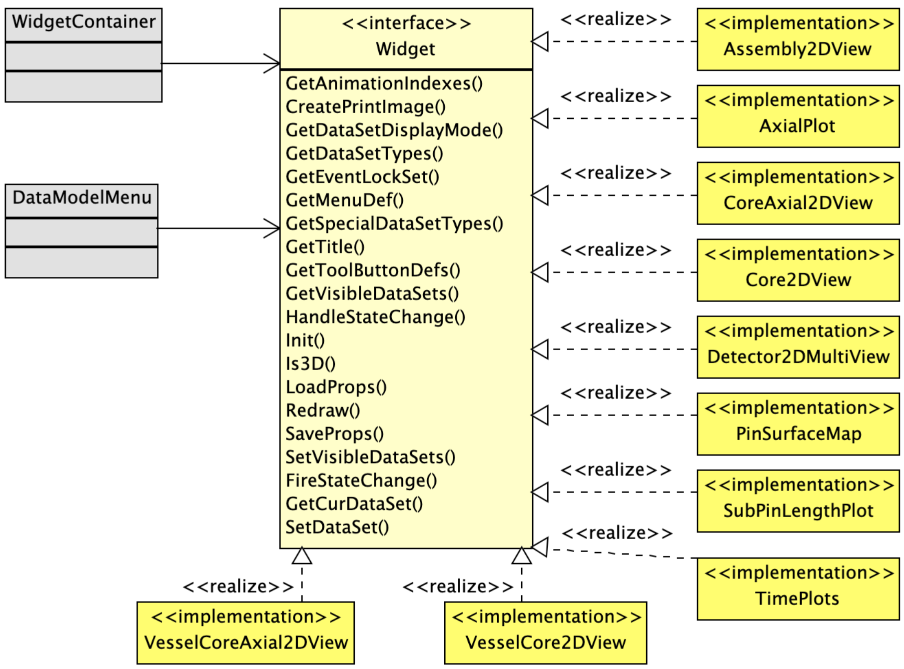


Figure 6. Widget dependency injection.

Table 29. Widget framework public methods.

Method	Injection Call	Description
FireStateChange()	Y	Fires a state change event via the State object
GetAnimationIndexes()	Y	Lists indexes over which the widget supports animation
CreatePrintImage()	Y	Creates a full resolution image representation of the widget and saves it to a file
GetCurDataSet()	Y	Reflects to find a fluenceAddr or curDataSet property
GetDataSetDisplayMode()	Y	Indicates if the widget displays multiple or a single dataset
GetDataSetTypes()	Y	Lists dataset categories supported by the widget
GetEventLockSet()	Y	Returns the set of events which can be locked for the widget
GetMenuDef()	Y	Returns the menu definition (see _CreateMenuDef())
GetPopupMenu()	N	Lazily creates and returns the popupMenu property
GetSpecialDataSetTypes()	Y	Lists special dataset categories requiring a separate menu[2]
GetTitle()	Y	Returns the widget's title
GetToolButtonDefs()	Y	Returns a list of tuples defining additional buttons to add to the widget toolbar
GetVisibleDataSets()	Y	Returns a set of DataSetNames representing datasets visible in the widget[1]
HandleStateChange()	Y	Processes state change events
Init()	Y	Initializes the widget
Is3D()	Y	Returns true if the widget is 3D
IsBusy()	N	True if the widget is in the busy state
IsDataSetVisible()	N	Returns True if a dataset is currently visible[1]
LoadProps()	Y	Deserializes from JSON
Redraw()	Y	Re-renders the widget
SaveProps()	Y	Serializes to JSON
SetDataSet()	Y	Makes the dataset current for the widget
SetVisibleDataSets()	Y	Makes visible in the widget a set of DataSetNames[1]
ToggleDataSetVisible()	N	Toggles visibility of a dataset in the widget[1]
UpdateState()	N	Applies a state change event

[1]Applied when the widget supports multiple datasets [2]Not currently used by any VERAView widgets

Table 30. Widget framework support methods.

Method	Description
_BusyBegin()	Set the light emitting diode (LED) indicator icon to busy
_BusyBeginOp()	Set the busy LED indicator and launch a function a thread
_BusyDoOp()	Set the busy LED indicator, launch a thread, and reset the indicator icon when the thread completes
_BusyEnd()	Set the LED indicator icon to idle
_CalcFontSize()	Determine an effective font size given a screen size
_CreateEmptyBitmapAndDC()	Creates a wx.Bitmap and corresonding wx.MemoryDC with the bitmap selected
_CreateGraphicsContext()	Creates a wx.GraphicsContext from a wx.DC and sets antialiasing
_CreateValueString()	Represents a number with a specified precision, number of digits, or custom format
_FindFirstDataSet()	Finds the first available dataset of a specified category
IsAuxiliaryEvent()	Checks a wx.Event for control or meta modifiers
_OnFindMinMaxChannel()	Finds the min or max for a *channel* dataset
_OnFindMinMaxFluence()	Finds the min or max for a *fluence* dataset
_OnFindMinMaxMultiDataSets()	Finds the min or max across a list of datasets
_OnFindMinMaxPin()	Finds the min or max for a *pin* dataset
_ResolveDataRange()	Determines the value range for a dataset, reading from the DataModel if necessary and applying any user-specified range
_ResolveScaleType()	Checks for the user-specified scale type or one specified as a dataset attribute
_UpdateMenuItem()	Changes menu item labels
_UpdateVisibilityMenuItems()	Changes "Show"/"Hide" labels in menu iems

Table 31. **PlotWidget** object properties.

Property	Description
ax	primary matplotlib.axes.Axes instance
axline	horizontal or vertical matplotlib.lines.Line2D instance used to indicate the current event state
canvas	FigureCanvasWxAgg in which fig is rendered
cursor	plot location following the mouse
cursorline	horizontal or vertical matplotlib.lines.Line2D instance used to follow the mouse
fig	matplotlib.figure.Figure instance
refAxis	reference axis on which values are plotted, either y or x
timeValue	value in the current time dataset
timer	wx.Timer used to ignore intermediate resize events
toolbar	NavigationToolbar2Wx
titleFontSize	point size for title font

Table 32. **PlotWidget** framework private methods.

Method	Noop	Description
_CreateToolTipText()	Y	Creates tooltip text based showing values at the current mouse/cursor location
_DoUpdatePlot()	N	Creates the plot, titles, and labels and sets axline
_DoUpdateRedraw()	Y	Updates for a redraw only
_InitAxes()	N	Creates ax and optionally ax2
_InitUI()	N	Create fig, canvas, and toolbar, calls _InitAxes(), and registers event handlers
_IsTimeReplot()	Y	True if the widget replots on a time change, False otherwise
_LoadDataModelValues()	Y	Applies initial state changes to instance properties
_OnClose()	N	Handler for widget (wx.EVT_CLOSE) close event
_OnMplMouseMotion()	N	Handles mouse motion to update cursor lines and tooltip text
_OnMplMouseRelease()	N	Handles mouse release events
_OnSize()	N	Handles resize events
_OnTimer()	N	Handles resize timer events
_OnTootleToolBar()	N	Toggles toolbar display
_UpdateDataSetValues()	Y	Hook for bookkeeping and management of plot data
_UpdatePlot()	N	Wraps _UpdatePlotImpl() in _BusyDoOp()
_UpdatePlotImpl()	N	Clears fig and calls _DoUpdatePlot() and _DoUpdateRedraw() and draws canvas
_UpdateStateValues()	Y	Applies state changes to instance properties

Table 33. RasterWidget object properties.

Property	Description
axialValue	AxialValue instance (state property)
bitmapCtrl	wx.StaticBitmap containing the image
cellRange	range of cells displayed based on current zoom
cellRangeStack	zoom stack
config	dictionary of drawing properties and calculated values used in rendering the image
curDataSet	DataSetName for the dataset to represent
dragStartCell	one corner of drag cell range
fitMode	indicates scaling by the width or height of the available drawing region
showLabels	flag to show or hide row and column labels
showLegend	flag to show or hide a legend
showValues	flag to show or hide numeric values
stateIndex	0-based statepoint index
timeValue	value in current time dataset (state property)

Table 34. **RasterWidget framework private methods.**

Method	Noop	Description
_CreateAdditionalUIControls()	Y	Creates tooltip text based showing values at the current mouse/cursor location
_CreateDrawConfig()	Y	Calculates drawing and rendering properties (stored in config property)
_CreateRasterImage()	Y	Creates and renders into the image
_CreateStateTuple()	Y	Create tuple of indexes representing state properties of import to the widget
_FindCell()	Y	Determines the cell (e.g., assembly, pin) from a location in the image
_GetPrintFontScale()	N	Injection method specifying a scaling factor for fonts when creating a printabale image
_HiliteBitmap()	Y	Render current state properties and user selections on top of the image
_InitUI()	N	Creates GUI components and initializes event handlers
_IsAssemblyAware()	N	Injection method indicating the widget cares about assemblies
_IsValueDisplay()	N	Injection method indicating the widget can display numeric values
_LoadDataModelUI()	Y	Hook for the widget to initialize any GUI components that depend on an initial data load
_LoadDataModelValues()	Y	Applies initial state changes to instance properties
_OnClick()	Y	Single-click (not drag) event handler
_OnDragFinished()	Y	Called after a drag operation finishes
_OnSize()	N	Handles resize events
_OnTimer()	N	Handles resize timer events
_OnToggleFit()	N	Handles user changes to the fit mode
_OnValues()	N	Handles user toggles of value display
_OnUnzoom()	N	Handles user unzooms
_UpdateDataSetStateValues()	Y	Hook for widget after curDataSet has been set
_UpdateStateValues()	N	Implemented in RasterWidget but overridden with super call in subclasses

Table 35. RasterWidget framework support methods.

Method	Description
_CreateBaseDrawConfig()	Called for common configuration processing (legend creation, calculating font sizes, etc.)
_CreateTitleFormat() _CreateTitleString() _CreateTitleTemplate() _CreateTitleTemplate2()	Create title formats and strings
_CreateValueDisplay() _DrawStringsWx() _DrawValuesWx()	Create and render numeric value strings
_InitEventHandlers()	Binds mouse event listeners

Acknowledgments

Oak Ridge National Laboratory
Ben Collins
Travis Lange

References

[1] Andrew Godfrey and Ronald Lee. *VERAView User's Guide*. Tech. rep. CASL-U-2016-1058-001. Consortium for Advanced Simulation of Light Water Reactors, 2017.

[2] Andrew Godfrey Greg Davidson Ben Collins Scott Palmtag. *VERAOUT - VERA HDF5 Output Specification*. Tech. rep. CASL-U-2014-0043-001. Consortium for Advanced Simulation of Light Water Reactors, 2014.

[3] *Anaconda Distribution*. 2019. URL: https://www.anaconda.com/distribution.

[4] *PEP 8 – Style Guide for Python Code*. 2013. URL: https://www.python.org/dev/peps/pep-0008/.