Mayavi: 3D Visualization of Scientific Data

Mayavi is a general purpose, open source 3D scientific visualization package that is tightly integrated with the rich ecosystem of Python scientific packages. Mayavi provides a continuum of tools for developing scientific applications, ranging from interactive and script-based data visualization in Python to full-blown custom end-user applications.

The Mayavi scientific visualization package seeks to provide easy and interactive tools for data visualization that fit with the scientific user's workflow. It therefore offers several entry points: a full-blown interactive application; a Python library with both a Matlab-like interface, focused on easy scripting, and a feature-rich object hierarchy; widgets associated with these objects for assembling in a domain-specific application; and plugins that work with a general purpose framework for application building.

Here, we offer an overview of Mayavi's various features and insights on design and engineering decisions in implementing Mayavi. We also describe a few novel applications.

What is Mayavi?

Various factors combine to make Mayavi unlike most other visualization tools. First, the Mayavi project avoids domain-specific cases and strives to build reusable, general purpose abstractions. This is important because different research fields often seek solutions to similar problems.

Second, Mayavi exposes tools and objects that fit closely with a naive user's expectations and also lets expert users handle complex visualizations. Finally, and most importantly, the project is more than just a visualization library; it also provides widgets, dialogs, plugins, and an application to accommodate various aspects of the scientific workflow.

A rich interactive graphical application and simple but full-featured scripting is important because visualizing complex datasets is best done interactively so that users can examine the data visually, tweak parameters, and build visualizations to suit their data. To enable batch processing and noninteractive use, Mayavi visualizations can be driven without a user interface.

Because it's a powerful, yet easy-to-learn programming language, Python is central to answering multiple scientific use cases. Thanks to a growing number of high-quality scientific libraries, Python has recently garnered significant mindshare among scientists. Indeed, it lends itself to interactive use with simultaneous plotting, which is ideal for scientific and data-processing development. Mayavi brings powerful 3D scientific data visualization to this tightly integrated environment.

There are several excellent general purpose 3D visualization programs that expose high-quality Python interfaces, including ParaView and VisIt. Unlike Mayavi, these tools support parallel data visualization and 4D datasets. However, Mayavi more tightly integrates with the workflow.
of a typical scientist using Python for numerical computation by offering familiar data structures and exposing all of its GUIs as components.

First, Mayavi operates naturally on NumPy arrays, the core data structure used in key scientific Python projects (see http://numpy.scipy.org). As we describe later, the entire Visualization ToolKit (VTK) API is wrapped with implicit conversion between VTK arrays and NumPy arrays, and Mayavi’s mlab interface provides convenient functions to visualize the 3D data that NumPy arrays describe. Along with the fact that Mayavi integrates well with IPython, this makes it highly convenient for interactive work.

Second, Mayavi relies on a reflexive object model in which each object can be used as a component to create custom dialogs that embed a Mayavi visualization. The Enthought tool suite (ETS) (www.enthought.com/products/ets.php) lets users create rich, interactive scientific applications that support 2D and 3D plotting and require knowledge only of Python and object oriented programming. Thus Mayavi fills a valuable need in the scientific computing ecosystem.

**Powerful Underlying Technologies**

Mayavi is part of the ETS and builds on a powerful stack of existing libraries.

VTK is an open source, tool suite that forms one of the best actively developed general purpose visualization and graphics libraries available.8

The NumPy array structure is the workhorse of scientific computing with Python. Its multidimensional numerical array transforms Python into a high-level array language, similar to Matlab. Most scientific libraries and projects relying on Python use it as a common data container.

The Traits library (http://code.enthought.com/projects/traits) is the ETS cornerstone, extending Python object attributes and providing an elegant mechanism for attribute initialization, validation, delegation, notification (efficient callbacks on attribute modification), and visualization (through dialogs using wxPython or Qt4). Here, we refer to objects with traits as *traited objects*.

The Traits VTK library (TVTK) is an automatically generated combination of VTK and Traits that wraps VTK-Python objects and provides a traits-enabled API with a more “Pythonic” feel. VTK has its own C++-based array structures, and the VTK-Python bindings require tedious manipulations to copy or reference the data from NumPy arrays to VTK arrays. Traits replaces all C++-style getters and setters with a seamless API that dynamically converts VTK arrays from and to NumPy arrays, using a view of the same memory where possible. In addition, users can edit all TVTK object properties through a default dialog. TVTK is thus a powerful object hierarchy that uses simple rules to map to VTK and forms Mayavi’s foundation.

The TraitsUI/Envisage end-user application interface relies entirely on Traits and optionally on Envisage, the Eclipse-derived application-building framework. At its core, Envisage is a system for defining, registering, and using plugins. Both Traits and Envisage can use wxPython or PyQt as a backend. Thus, both toolkits can use Mayavi, although the wxPython backend is more developed and thus more mature.

**A Simple Pipeline Model for Visualization**

VTK relies on an elaborate pipeline model that’s assembled to create a visualization. Mayavi exposes users to a basic pipeline interface: data is obtained from a data source and users can add visualization modules to display the data in various ways or add filters to modify the data before visualization.

Mayavi connects various pipeline components and, to an extent, manages components that can be interconnected. For example, a cloud of scattered points can’t be visualized with a grid plane because the points have no fixed grid. The different sources, modules, and filters are traited objects with associated dialogs and methods that expose the underlying TVTK objects as much as possible without requiring detailed VTK knowledge. The Mayavi pipeline collapses together several VTK pipeline elements; for example, Mayavi modules are made of VTK mappers, actors, and eventually widgets or even filters when their use helps represent data. Finally, although pipeline loops are possible, in Mayavi, the pipeline is presented as a tree rather than a graph (see Figure 1) to facilitate manipulation, both interactively (through the user interface) and programmatically.

**Project History**

One of this article’s coauthors (Prabhu Ramachandran) created Mayavi—named after the Sanskrit word for magician—in 2001 as an end-user application for scientific visualization.9 Although appreciated for its ease of use and interactivity, Mayavi wasn’t easy to script from Python.

In 2004, Enthought hired Ramachandran to write TVTK and develop mayavi2, which uses Enthought tools and focuses on reuse and embedding.10 In 2007, the other coauthor (Gaël Varoquaux) joined the project and our collaboration.
began. Mayavi is rapidly gaining features and documentation, and we’re improving usability on the basis of user feedback. In early 2008, TVTK and Mayavi were each awarded Free and Open Source Software (FOSS) India awards.

**Using Mayavi**

Because Mayavi can be used in different ways, it has several entry points.

**The mayavi2 Application**

The interactive mayavi2 is an end-user tool that can be used without any programming knowledge. It provides an interface with menus and several panels to guide users in creating a visualization (see Figure 1). As we described earlier, Mayavi presents a simplified pipeline view of the visualization. Using menus, users can load data from files or use predefined objects such as a layout of Earth’s continents or parametric surfaces defining, for example, a Klein bottle. Users then filter data and add visualization modules to create their visualizations.

Various pipeline components (sources, modules, and filters) appear in a tree view (Figure 1, upper left panel). Users can add more components using menus and dialogs. In particular, contextual menus suggest which filters or modules are applicable to a given data source. Users can also reorganize the pipeline using drag-and-drop operations on the tree nodes. Objects selected on the pipeline view can be edited in the bottom left panel and modifications are immediately applied to the visualization. Although the pipeline in Figure 1 is extremely simple, more complex pipelines are also possible.

Although a raw VTK dataset is a versatile data structure describing data embedded in a 3D space, a Mayavi source tries to expose a simplified interface to users for importing data in Mayavi. Similarly, the Mayavi modules are a single entry point for changing all object properties and gathering all VTK sub-objects in one object and one dialog. An exception to this rule is that modules can share the color maps and legends, which thus are represented as a separate pipeline node.

Through its primary tree-based interface, Mayavi offers only a limited subset of VTK’s filters, and the Mayavi sources can’t cover all possible ways to create VTK datasets. Mayavi thus offers a User Defined filter to plug in almost any VTK filter in the Mayavi pipeline by specifying its name and a VTKDataSource class to create a Mayavi source from any VTK dataset.

The entire visualization can be saved to disk in a Mayavi-specific format; however, this is one aspect of Mayavi that should be more robust. Alternatively, users can save data generated at any point of the pipeline to VTK files.

The mayavi2 application has a few features to help create Python scripts from a visualization. First, the application displays an interactive Python shell, where users can enter Python commands for immediate execution. The scripting API (which we describe later) can be used to create or modify visualizations. Objects dragged from the pipeline tree to the shell appear as Python objects for exploration or modification.

Second, the pipeline tree view features a record button (the red button on the upper left panel’s toolbar in Figure 1). When record mode is activated, any object addition or modification automatically generates the necessary lines of Python code to reproduce the action. Users
can thus create fully functional Python scripts. However, the generated code is not always the simplest possible code. The record mode is also an extremely valuable learning tool for scripting Mayavi or TVTK objects. Indeed, VTK is a rich visualization library, and Mayavi’s object hierarchy can be deep and complex. As a result, it can be hard to find a given feature’s corresponding method or attribute. The record feature is thus highly convenient even for experienced VTK users.

Mayavi ships with an extensive user manual (http://code.enthought.com/projects/mayavi/), which is also accessible from the application. The user guide is rendered using Sphinx (http://sphinx.pocoo.org) and embeds a search bar and an index.

**Simple Python Scripting**

As we’ve seen, it’s possible even for nonprogrammers to fully interact with Mayavi through the mayavi2 application. However, Mayavi is also usable through a simple yet powerful scripting API, providing a workflow similar to that of Matlab or Mathematica.

Many scientists use Python for their computational work. Visualization, which is a key component of such work, is most effective when used interactively as an exploratory and debugging tool, as in matplotlib and IPython. Mayavi’s mlab scripting interface is a set of Python functions that work with NumPy arrays and draw some inspiration from the Matlab and matplotlib plotting functions. It can be used interactively in IPython, or inside any Python script or application. The example in Figure 2 generates iso-contours of a mathematical function, sampled on a regular grid. Figure 3 shows the resulting visualization.

Simple plotting commands operating on NumPy arrays, such as `mlab.contour3d` (see previous example), build a complete visualization pipeline. In simple use cases, these simple commands hide the pipeline model from users and accept many extra arguments to control the visualization’s properties. Also, as they return the created Mayavi modules, users can change more properties by modifying their attributes.

Alternatively, `mlab` offers users direct control of the Mayavi pipeline by letting them separately create sources, filters, and modules. Thus, they can replace the call to `mlab.contour3d` in the previous example with two commands: one to create a source object from a NumPy array’s regularly spaced volumetric data, and a second to apply an iso-surface module on it:

```python
from enthought.mayavi import mlab
from numpy import ogrid

x, y, z = ogrid[-10:10:100j, -10:10:100j, -10:10:100j]

ctr = mlab.contour3d(0.5*x**2 + y**2 + 2*z**2)
mlab.show()
```

Figure 2. Generating iso-contours of a mathematical function, sampled on a regular grid. Figure 3 shows the resulting visualization.

Figure 3. Working with Mayavi in IPython. The terminal on the background runs the IPython session from which the visualization window on the foreground (right) was created. The pipeline dialog editing the different visualization objects was created by clicking on the button with the Mayavi icon on the toolbar in the left visualization window’s toolbar.

Manually populating the pipeline requires an understanding of the pipeline model. Manual population is also more powerful because it gives access to a wider range of possibilities through custom-made pipelines. The names of the `mlab.pipeline` functions to create objects are lowercase-with-underscore versions of the camel-case names of the classes represented in the default pipeline view. It’s thus easy to go from a pipeline built interactively to a script.

Although the window that displays the visualization is simple, Mayavi’s full power is still available. Clicking the button with the Mayavi logo on the left toolbar displays a dialog containing the
same pipeline tree view as in the Mayavi application (see Figure 1). All of the mayavi2 application’s interactive functionality is accessible in this dialog: the buttons on the toolbar provide access to help, script recording, or object creation. Clicking on the pipeline nodes creates dialogs that let users modify the object properties. The pipeline can be populated and modified by the context menus accessible through a right click on the nodes.

**Animating Data and Building Interactive Dialogs**

Objects created using the mlab functions expose an mlab_source attribute, which offers access to the NumPy arrays used to create the dataset. Assigning new arrays to the mlab_source triggers a visualization update. The name of the mlab_source object attribute that you need to modify is the name given to the corresponding argument in the function signature documentation—the mlab.contour3d function signature, for example, is contour3d(scalars, ...), where scalars is a 3D array. Thus, we can animate the previously created contour object by modifying the scalars it represents in place:

```python
from time import sleep
for i in range(1, 10):
    sleep(1)
    ctr.mlab_source.scalars = (0.5*x**2 + y**2 + x**2)
```

In-place modifications are also useful when embedding a visualization in an interactive application. A Mayavi scene can be displayed as part of a Traits-based user interface. The following example displays a dialog visualizing a 1D parametric function embedded in a 3D space as a curved line. The mathematical curve, defined by the curve function, takes one parameter—the number of minor rotations in the transverse direction. Through the dialog (Figure 4) you modify this parameter, with an immediate visualization of the results (Figure 5).

In Figure 5’s code example, the visualization class defines a few traits, including the scene trait, which is an instance of MlabSceneModel. The configure_traits() call at the end creates a dialog representing the object, the layout of which is given by the view defined at the end of the class. This view exposes the scene trait using a SceneEditor in the dialog, and the n_turns attribute as a slider. When the Visualization object is created, the curve is plotted in the embedded scene with the `plot3d` mlab call. When the n_turns attribute is modified, the `update_plot` method is called, the curve data is recomputed, and the plot object is modified using the mlab_source attribute.

Generally, users can modify the different properties of visualization objects in a script (including sources, filters, modules, and even scenes) by simply setting the corresponding trait. The scene will then update, either instantaneously or following a delay. Moreover, the different dialogs that form Mayavi, such as the tree view and the dialog editing an object, can be embedded in a user’s application. Callbacks between different dialogs and the scene are already wired. As such, Mayavi forms more than a visualization library; it can be used as a set of interactive components to provide live visualization to a domain-specific application, requiring little knowledge of GUI programming or event loops.

**Embedding Mayavi in Existing Applications**

Although Traits is a powerful tool for developing interactive applications, most existing applications are developed using a raw GUI toolkit. It’s thus important to integrate the dialogs produced from the code example in a GUI that isn’t Traits-aware. Traits has a wxPython and a Qt4 backend. Although the configure_traits method creates the dialog by creating a full wxPython application and starts the main event loop, the HasTraits class also provides an edit_traits method that creates and returns only a panel or dialog. The following example shows how to embed the previously defined Visualization

---

**Figure 4.** A dialog visualizing a 1D parametric function embedded in a 3D space as a curved line. In addition to interacting with the visualization, users can drag the slider bar to interactively modify the visualization.
and its corresponding dialog in a wxPython application:

```python
import wx

class MainWindow(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Mayavi in Wx')
        self.visualization = Visualization()
        self.control = self.visualization.edit_traits(parent=self, kind='subpanel').control
        self.Show()

app = wx.PySimpleApp()
frame = MainWindow(None, wx.ID_ANY)
app.MainLoop()
```

In the earlier example, the `edit_traits` method is given as an argument the wxPython frame into which the dialog is embedded. The control trait of the object produced by the `edit_traits` call is the wxPython object containing the widget. Similarly, dialogs can be embedded in a PyQt application, as detailed in the user guide.

Any Mayavi dialog can be embedded similarly in more complex applications. For example, the various Mayavi pipeline objects also provide an `edit_traits` method to edit their properties. Thus, the work invested in developing powerful widgets for the Mayavi application, such as the pipeline tree view, is readily available to the application builder.

### Extending mayavi2

Rather than create a new application, users can extend the already-powerful mayavi2 application by adding custom functionality or domain-specific elements. The application is built by integrating Mayavi’s 3D visualization with other functionality, such as a Python shell, via Envisage plugins. Using the same mechanism, you can assemble...
from enthought.mayavi.core.api import registry, SourceMetadata, PipelineInfo
from enthought.mayavi.sources.api import ArraySource
import numpy as np

def array_reader(fname, engine):
    return ArraySource(scalar_data=np.loadtxt(fname))

registry.sources.append(SourceMetadata(
    factory     = __name__ + '.array_reader',
    menu_name   = "Array text files",
    extensions  = ['txt'],
    wildcards   = 'TXT files (*.txt)|*.txt',
    output_info = PipelineInfo(datasets=['image_data'],
                                attribute_types=['any'],
                                attributes=['any'])),

Figure 6. Adding a domain-specific file reader. Placing this code in a module imported in ~/.mayavi2/user_mayavi.py defines a reader that loads arrays stored in a text file.

Mayavi Architecture and Software Design

We now offer a broad overview of Mayavi’s architecture and software design.

Architecture Overview

Figure 7 shows Mayavi’s general software architecture. The tool’s visualization layer relies on TVTK objects. The Mayavi pipeline objects use TVTK objects and have methods that help wire them together and simplify building the VTK pipeline. A central object, the Mayavi Engine, manages all of the visualization’s pipeline objects.

The pipeline objects and the engine form the core Mayavi functionality. The mlab scripting API controls the engine to create visualizations. Because all objects rely on the Traits library, user interface panels or widgets can be created using the TraitsUI package. The different panels, as well as Mayavi’s core functionality, can be combined with other Envisage plugins to create applications such as the mayavi2 application.

The Engine as Pipeline Warden

The Mayavi engine maintains a tree structure of pipeline objects. Each pipeline object maintains references to its parents and children and exposes a list of inputs and outputs. The output list contains the TVTK datasets flowing between the pipeline elements. The input objects are Mayavi pipeline objects. The Mayavi pipeline objects have callbacks to rewire the underlying VTK pipeline if their inputs change. They also feature two events, pipeline_changed and data_changed, which are propagated down the pipeline to update it.

other applications or extend the mayavi2 application. (A discussion of the Envisage application-building framework is beyond this article’s scope.)

Mayavi also has a mechanism to register new types of data sources, filters, or modules, which are automatically added to various menus and the mlab.pipeline interface. A common use case is to add domain-specific file readers or visualization modules. For example, you can place the code in Figure 6 in a module imported in ~/.mayavi2/user_mayavi.py to define a reader using NumPy that loads arrays stored in a text file.

Of course, proper use of Mayavi as a platform for domain-specific applications requires an understanding of the tool’s finer details, which are available elsewhere.11
The engine manages the pipeline objects’ life cycle—that is, it manages their addition and removal.

We can explicitly build the pipeline of the earlier iso-surface example using the code in Figure 8.

Here, the engine manages the connection between the source, the modules and the scene internally—that is, it maintains the visualization context. For example, the `new_scene` method can be overridden in a subclass to create an embedded scene, as in the mayavi2 application, or a separate window, as when using mlab in IPython. Other subclasses for off-screen rendering are also available. In addition, the engine provides context-dependent actions that can be useful to drive an interactive application in a manner similar to spreadsheet scripting. The `add_module` call implicitly adds the `IsoSurface` instance to the `ArraySource` source `src`. It’s also possible to build the pipeline explicitly by selecting and connecting the different objects rather than delegating the task to the engine:

```python
from enthought.mayavi.core.api import Engine
from enthought.mayavi.sources.api import ArraySource
from enthought.mayavi.modules.api import IsoSurface
from enthought.pyface.api import GUI
from numpy import ogrid
x, y, z = ogrid[-10:10:100j, -10:10:100j, -10:10:100j]

engine = Engine()
engine.start()
engine.new_scene()

src = ArraySource(scalar_data=(0.5*x**2 + y**2 + 2*z**2))
engine.add_source(src)
engine.add_module(IsoSurface())
GUI().start_event_loop()
```

Figure 8. Code for building the iso-surface’s pipeline. The engine manages the connection between the source, the modules, and the scene internally.

The engine manages the pipeline objects’ life cycle—that is, it manages their addition and removal.

We can explicitly build the pipeline of the earlier iso-surface example using the code in Figure 8.

Here, the engine manages the connection between the source, the modules and the scene internally—that is, it maintains the visualization context. For example, the `new_scene` method can be overridden in a subclass to create an embedded scene, as in the mayavi2 application, or a separate window, as when using mlab in IPython. Other subclasses for off-screen rendering are also available. In addition, the engine provides context-dependent actions that can be useful to drive an interactive application in a manner similar to spreadsheet scripting. The `add_module` call implicitly adds the `IsoSurface` instance to the `ArraySource` source `src`. It’s also possible to build the pipeline explicitly by selecting and connecting the different objects rather than delegating the task to the engine:

```python
from enthought.mayavi.core.api import Engine
from enthought.mayavi.sources.api import ArraySource
from enthought.mayavi.modules.api import IsoSurface
from enthought.pyface.api import GUI
from numpy import ogrid
x, y, z = ogrid[-10:10:100j, -10:10:100j, -10:10:100j]

engine = Engine()
engine.start()
engine.new_scene()

src = ArraySource(scalar_data=(0.5*x**2 + y**2 + 2*z**2))
engine.add_source(src)
engine.add_module(IsoSurface())
GUI().start_event_loop()
```

Figure 8. Code for building the iso-surface’s pipeline. The engine manages the connection between the source, the modules, and the scene internally.

To avoid side effects in a large application. It also makes Mayavi much easier to reuse and test. For example, if the engine was global, any changes to it in a test suite would influence other tests. However, because you can create as many engines as you want, it’s easy to write tests that avoid unnecessary side effects.

A Central Registry to Avoid Duplication

Mayavi provides a large and growing list of pipeline objects. Such objects are exposed through many different graphical interfaces and programming APIs. To avoid code duplication, all the different sources, filters, and modules are specified in a central registry along with information describing their functionality. The metadata information in the registry is used to generate the different user interface menus, as well as many of the `mlab.pipline` functions, thus enforcing consistency throughout Mayavi.

Model-View Separation

Whenever possible, Mayavi uses a reactive programming style with callbacks on trait modifications. User interfaces are created using representations of the objects’ traits. This programming style permits a clean separation between the model and the view. The model can be fully described by the traits of all the pipeline’s objects. For example, the script-recording functionality we described earlier is implemented in large part by tracking trait modifications. Mayavi’s use of the Model-View-Controller (MVC) design pattern is complete; there’s some mixing of the view and the model. However, while this can be reduced in the future, we believe that the
design is already very reusable, offering most of MVC’s advantages.

The message-passing style that replaces method calls with trait assignments is thread-friendly and can be used to avoid GUI event loops. The code updating the user interface, for example, is hidden from users. Because we primarily rely on traits for views, these are defined in a declarative way (as in the interactive dialog example). Consequently, no application logic exists in the view-related code. This is crucial for a clean scripting API and also enables scripts to run in a headless (off-screen) mode with no modifications.

**Designing Reusable APIs**

In designing Mayavi to be highly reusable in various contexts, one strategy we used was to focus on scriptability of the API as much as possible. We achieved this by driving our design choices via example scripts, but also through unit testing, integration testing, and some test-driven development (TDD). Testing resulted in cleaner APIs and, more reliable code. This experience corroborates with the advantages claimed by TDD practitioners.

Several architectural decisions are key to the overall success of Mayavi’s reusability:

- **Multiple abstraction layers** (see Figure 7) enforce separation of concerns and enables code sharing between different entry points or APIs to address various use cases.
- The engine (a central, well-defined object) coordinates the visualization, provides a context, and thus helps establish a consistent view of the application, both in the Mayavi code and in the scripting APIs. In addition, this central object helps avoid globals. Because any number of engines can be created and used simultaneously, the engine makes data-parallel execution possible.
- The Traits’ powerful object model leads to a good design through reactive programming and strong model-view separation; it also provides multiple-backend user interfaces with little effort.
- Model and view separation and loose coupling are also key. It’s well known that GUI-related code should be well separated from the core application logic. However, we’ve also found that all helper-code unrelated to the core functionality—but that caters to common end-user needs, such as provided by the mlab API—should also be separate from the core.
- Because much of the code and interface is automatically generated, duplication is reduced and thus the interfaces are more consistent and the code easier to maintain. In addition to the TVTK wrapper code, boiler-plate user interface and APIs are auto-generated.
- Creating a simple API greatly improves the developers’ mental representation of the library and the application’s model, and as a result, of its architecture. The API should answer common use-cases and be consistent across various needs. Interaction and feedback from users, whose needs sometimes differ from that of developers, has proven priceless.

As we mentioned earlier, testing and scripting were invaluable in creating a truly reusable tool; such practices are also great for identifying unwanted tight coupling in the object model.

**Real-World Applications**

Researchers use Mayavi in highly diverse settings. Here, we offer two brief examples.

**Weather Visualization**

The FloSolver division at National Aerospace Laboratories (NAL) in Bangalore uses Mayavi to visualize data produced by its weather modeling code, which is primarily used to forecast India’s monsoon. Mayavi is used both as a display device and, more importantly, to help refine the weather models.

Data generated from weather simulations is rendered interactively using Mayavi scripted from Python to automate many mundane tasks. Mayavi’s automatic scripting mode records user interface actions and generates human-readable Python code. This code is then manually edited to produce weather data visualizations. Thus, without a direct knowledge of Mayavi’s internals or even much knowledge of Python, NAL scientists can generate fully functional Python scripts and tailor them to their needs.

To interactively show different atmospheric fields, each field is displayed on a separate computer running a Mayavi script. The camera position of each Mayavi application is controlled by an in-house OpenGL application used to display cloud data obtained from satellite images. To do this, we wrote a simple Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) server program that lets users send Python statements across the network; these statements are then interpreted by the running Mayavi application. We used the excellent Twisted (http://twistedmatrix.com/trac) library for this.
The resulting server module size was about 90 lines of code (without the documentation and comments). Using the server required two additional lines of code in the existing scripts at NAL. Figure 9 shows NAL's hardware setup. The left screen shows the application, which controls the view of the four other Mayavi applications via the network. The right side visualization wall consists of four separate LCD panels put together as one. This is possible only because of Python's powerful libraries.

**Web Browser Application**

In a recent development, Ondrej Certik and Ramaachandran worked together to setup a Sage (www.sagemath.org) notebook that uses Mayavi to perform visualizations on the Web. Among other things, Sage offers a powerful environment for doing mathematics on the Web through a notebook interface. This interface essentially provides a powerful Python-capable webpage where users can interact with a Python interpreter and embed results, along with images and text, seamlessly from a browser window.

For Mayavi to work in the Sage notebook, we built VTK with support for Mesa’s OSMesa library for pure off-screen rendering (see www.mesa3d.org). Then, using Mayavi’s existing support for off-screen rendering, we rendered images and generated X3D files, displayed interactively by the Sage notebook. The webpage http://lab.femhub.org/home/pub/39 demonstrates the resulting Sage notebook. The page uses Mayavi’s mlab interface to generate visualizations and produce X3D files that can be visualized interactively using a browser plugin.

Scientists and engineers are increasingly moving their computational code to high-level environments, and Python is a popular language of choice because it can unite various modules into a homogeneous environment. Mayavi provides a rich and powerful 3D data visualization package for this environment, tightly integrating various aspects of scientific computing and application development in Python.

**References**


---

**Figure 9.** The hardware for the weather visualization application at the National Aerospace Laboratories (NAL) in Bangalore. The application (left) controls four Mayavi applications via the network from a central program, providing a synchronized view. The visualization wall (right) features four separate LCD panels put together as one.
12. E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.

Prabhu Ramachandran is a faculty member in the Department of Aerospace Engineering at IIT Bombay. His research interests include particle methods and applied scientific computing. He is the creator, author, and lead developer of the Mayavi and TVTK Python packages and contributed to the Python wrappers in the visualization toolkit. Ramachandran has a PhD in aerospace engineering from IIT Madras. He is a member of the SciPy Community and the Society for Industrial and Applied Mathematics, and a nominated member of the Python Software Foundation. Contact him at prabhu@aero.iitb.ac.in.

Gaël Varoquaux is a computational science research fellow in the Neurospin Research Institute at the French National Institute for Research in Computer Science and Control (INRIA) in Paris. His research interests include statistical and computational techniques for probabilistic modeling of intrinsic brain activity using functional imaging, and making advanced data-processing techniques available across new scientific fields. Varoquaux has a PhD in quantum physics from the Université Paris-Sud, Orsay, France, and is a member of IEEE. Contact him at gael.varoquaux@normalesup.org.
Why YOU belong as a Member of IEEE Computer Society

Need to keep up with developments in computing and IT? 
Looking to enhance your knowledge and skills? 
Want to shape the future of your profession?

If you answered “yes” to any of these questions, IEEE Computer Society membership is definitely for you! With benefits that include:

- **Access to 600 titles from Safari® Books Online**, featuring the top technical and business online books from leading publishers such as O’Reilly Media.
- **Access to 3,500 online technical and professional development online courses**, provided by Element K.
- **Access to the newest emerging technologies** through your monthly subscription to COMPUTER magazine.
- **Access to conferences, publications, and certification credentials** at exclusive member-only savings.

Discover even more benefits and become an IEEE Computer Society Member today at www.computer.org