ParaView Catalyst: Enabling In Situ Data Analysis and Visualization

Utkarsh Ayachit
Kitware, Inc.

Andrew Bauer
Kitware, Inc.

Berk Geveci
Kitware, Inc.

Patrick O'Leary
Kitware, Inc.

Kenneth Moreland
Sandia National Laboratories

Nathan Fabian
Sandia National Laboratories

Jeffrey Mauldin
Sandia National Laboratories

ABSTRACT

Computer simulations are growing in sophistication and producing results of ever greater fidelity. This trend has been enabled by advances in numerical methods and increasing computing power. Yet these advances come with several costs including massive increases in data size, difficulties examining output data, challenges in configuring simulation runs, and difficulty debugging running codes. Interactive visualization tools, like ParaView, have been used for post-processing of simulation results. However, the increasing data sizes, and limited storage and bandwidth make high fidelity post-processing impractical. *In situ* analysis is recognized as one of the ways to address these challenges. *In situ* analysis moves some of the post-processing tasks in line with the simulation code thus short circuiting the need to communicate the data between the simulation and analysis via storage. ParaView Catalyst is a data processing and visualization library that enables *in situ* analysis and visualization. Built on and designed to interoperate with the standard visualization toolkit VTK and the ParaView application, Catalyst enables simulations to intelligently perform analysis, generate relevant output data, and visualize results concurrent with a running simulation. In this paper, we provide an overview of the Catalyst framework and some of the success stories.

1. INTRODUCTION

The challenges for existing scientific workflows for exascale are well documented[9, 16, 4, 1]. The main cause of the challenges often boils down to the fact that while each successive computing generation has seen improvement in all the major components of the HPC infrastructure, the rate of change of different components has been disproportionate. As a result computing is getting cheaper while data transfer, be it in memory or external storage, is getting comparatively expensive. That has necessitated an examination of our existing workflows. Traditional scientific computing workflows can be characterized by three main tasks: preprocessing (preparing input), simulation (execution), and post-processing (analyzing and visualizing the simulation results) with data flowing between each stage through the I/O subsystem. With limited I/O bandwidth and capacity, it is acknowledged that such workflows are not sustainable at exascale. The main bottleneck is the data transfer between simulation and post-processing state. One approach to minimize this bottleneck is through *in situ* analysis: instead of performing the data analysis and visualization as a post-processing step, all those tasks are performed in line with the simulation thus minimizing, and potentially eliminating, the data transfer bottlenecks between the two stages.

ParaView Catalyst (or simply Catalyst) [5, 8] was created as a library to enable such integration between simulation and post-processing stages. It leverages the analysis and visualization capabilities of the familiar post-processing platforms, VTK[19] and ParaView[2], while providing straight forward mechanism to integrate with simulation codes. Researchers can develop analysis pipelines using C++ or Python[18] that are executed along side the simulation run, in the same address space.

The first step to using Catalyst with a simulation code is to *integrate* Catalyst into a simulation (Section 3). Once integrated, Catalyst can be used to deploy various analysis pipelines *in situ* with the simulation. To enable easy analysis pipeline customization, Catalyst supports scripting using Python. Users can write these scripts from scratch or use the ParaView GUI to interactively setup prototype visualization and then export the state as Catalyst scripts. Such Catalyst scripts provide access to a wide range of co-processing capabilities including executing pipelines for producing images, computing statistical quantities, generating plots, extracting derived information such as polygonal data or iso-surfaces for further post-processing. Recent work is adding support to export explorable data artifacts, or *Cinema*[3] database, enabling post-processing using light weight viewer applications. Other efforts are adding support to serialize analysis results using high performance I/O libraries

ISAV2015 November 15-20, Austin, TX, USA

 ISBN 978-1-4503-4003-8/15/11…$15.00

DOI: http://dx.doi.org/10.1145/2828612.2828624

such as ADIOS[11]. While Catalyst is designed to run in the same process space as the simulation, it is possible to connect to a separately running ParaView session for exploring results as they are being produced. Such *live* co-processing also facilitates analysis steering i.e. allows changing of the analysis pipelines interactively, through user feedback.
Once a simulation code has been instrumented to use Catalyst, accessing any of these features (or new ones that are incorporated in the future) is simply a matter of updating the customizable Catalyst pipeline, often described through Python scripts. The Python scripting API makes it easy to customize the pipeline without recompiling the code, however, for advanced users or for use-cases where a Python dependency is not appropriate, a C/C++ API is also available.

2. UNDERSTANDING VTK, PARAVIEW, CATALYST AND CATALYST EDITIONS

The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D computer graphics, image processing, and visualization. It consists of a C++ class library and several interpreted interface layers including Tcl/Tk, Java, and Python. VTK provides abstractions for representing data (data model) and rules that govern processing and transformation of the data (execution model). Using VTK, developers and users can develop applications that handle specific visualization and analysis needs. ParaView is an open-source, multi-platform data analysis and visualization application built on top of VTK. ParaView was developed to analyze extremely large datasets using distributed memory computing resources. ParaView builds upon VTK's data and execution model to add control logic for managing distributed (and remote) data processing and rendering. Also, ParaView is an end-user application. It provides a graphical interface to enable users to build analysis and visualization pipelines interactively. Catalyst is a library designed for *in situ* analysis. Catalyst itself depends on the UI independent components of the ParaView application, thus providing access to the data processing and rendering capabilities of the ParaView application, minus the UI. The Catalyst library also provides APIs that initialize, execute, and finalize the co-processing pipelines for a simulation run.
A standard SDK install of ParaView includes the Catalyst library. Thus, when developing code to integrate a new simulation with Catalyst, one can work off a ParaView SDK install. For production runs on systems with limited memory, one common concern is the size of code that Catalyst (through ParaView and its dependencies) brings in. Since ParaView is a general purpose tool, by default, it builds in a wide array of readers, filters, writers etc. Not building components that are not expected to be used during *in situ* analysis not only saves on the code memory needed, but also simplifies the build process. Such reduced-size versions of Catalyst library are called *Catalyst editions*. Using a generator Python script included in the ParaView source code, along with JSON files describing the classes to include in the edition, one can produce custom Catalyst editions from the ParaView source code.

3. PLUGGING IN TO CATALYST

In this section, we will give an overview of the steps involved in instrumenting a simulation code with Catalyst. For detailed developer documentation, the reader is referred to the ParaView Catalyst User's Guide[5].
A key step is developing the interface between the simulation and Catalyst. This interface is called the *adaptor*. The adaptor has two primary roles: map the simulation data structures to Catalyst (the VTK data model) and provide an API that the simulation uses to invoke Catalyst.

3.1 Anatomy of the Adaptor

Developing the Catalyst adaptor is typically the most important part in interfacing with Catalyst and may require significant effort. In most situations, however, the impact on the simulation codebase can be kept minimal by using a well designed adaptor API. This API has three parts:
1. **Initialize** – Catalyst needs to be initialized in order to be put in a proper state. This needs to be done once per simulation run before the first invocation to the `CoProcess` call. For codes using MPI, this must be called after the `MPI_Init()` call. Adaptor implementations can customize this call to allow the simulation to provide global information about the simulation run that doesn't change between timesteps. This initialization step also sets up the analysis pipelines to be run. Pipelines using the C++ API need to be defined at compile time. For Python enabled co-processing, Initialize loads the specified Catalyst script(s).
2. **CoProcess** – This routine is called for each time step as the simulation progresses. The adaptor implementation maps simulation data structures to the Catalyst/VTK data model and passes that along to the Catalyst code for processing. Catalyst provides API here to determine if any analysis is required to be run for the current timestep; if not, any work to map the data can be entirely skipped.
3. **Finalize** – On completion, the simulation code must call this routine to clean up Catalyst state including releasing any allocated memory. This is the counterpart of the *Initialize* call, and must only be called once for the entire run, not per timestep, and is called before *MPI_Finalize()* for MPI enabled codes.
When implementing each of these routines, the adaptor may use several API calls provided by the Catalyst library to assist with each of these stages. The Catalyst library itself is a C++ library. The adaptor, however, can be implemented to provide a C/C++/Fortran or Python interface based on the programming language used by the simulation code itself.

3.2 Implementing the Adaptor

Let's take a closer look at an adaptor implementation using a simple example. For this example, consider a simulation written in C that computes time varying quantities such as pressure, temperature, and density for a 3D volume defined by global dimensions and local dimensions per rank, with dimensions remaining fixed during the simulation run.
In this case, the adaptor (written in C++ with C interface) provides a header that the simulation code can include to use the API, defined as follows:

```
#ifdef __cplusplus
extern "C" {
#endif
void coprocessorinitialize(const char* py_scriptname);

void coprocess(int rank,
  int g_dx, int g_dy, int g_dz,
  int l_dx, int l_dy, int l_dz,
  int offset_x, int offset_y, int offset_z,
```

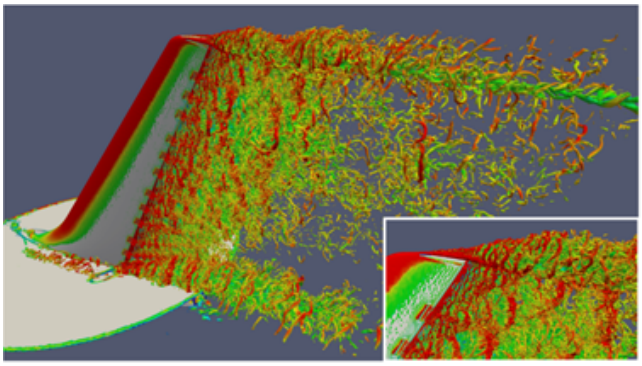

Figure 1: *PHASTA*: Results from 256K MPI rank simulation run instrumented with Catalyst for *in situ* visualization.

```
  double* pressure, double* temperature,
  double* density);

void coprocessorfinalize();
#ifdef __cplusplus
} /* extern "C" */
#endif
```

In `coprocessorinitialize`, we initialize Catalyst and create a Catalyst pipeline that executes the given Python script. As mentioned earlier, such Python scripts can be exported from the ParaView UI.

```
void coprocessorinitialize(const char* py_scriptname)
{
  // Initialize catalyst
  vtkCPAdaptorAPI::CoProcessorInitialize();

  // Register the Python script with the co-processor.
  // Multiple scripts can be added by creating multiple
  // vtkCPPythonScriptPipeline objects.
  vtkNew<vtkCPPythonScriptPipeline> pipeline;
  pipeline->Initialize(py_scriptname);
  vtkCPAdaptorAPI::GetCoProcessor()->AddPipeline(
    pipeline.GetPointer());
}
```

`coprocess` implements the crux of the code to map the datastructures from our simulation to Catalyst that simply uses VTK's data model. In this case, things are straight forward. A 3D volume is represented as a `vtkImageData` in VTK. The scalar arrays for pressure, temperature, etc. can be directly passed to VTK (without deep copying) as *point-data* arrays. Once the `vtkImageData` is prepared, it is passed to the Catalyst pipeline as follows:

```
void coprocess(...)
{
  // Create vtkImageData as a container for the scalars
  vtkNew<vtkImageData> imageData;
  ...

  // Pass the data to Catalyst.
  vtkCPAdaptorAPI::GetCoProcessorData()
    ->GetInputDescriptionByName("input")
    ->SetGrid(imageData.GetPointer());

  // Now, execute the Catalyst pipelines.
  vtkCPAdaptorAPI::CoProcess();
}
```

Catalyst provides API to check if any the processing pipelines are indeed going to execute for the current timestep. Such information can be used to avoid the data-mapping for timesteps that are skipped by the co-processing stages. Of course, the simulation can also skip processing on any timestep by simply not invoking the `coprocess` function.

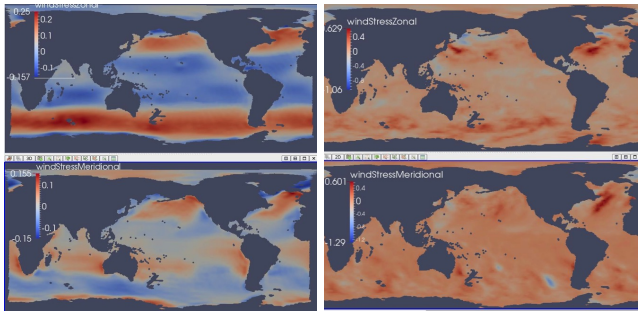

Figure 2: *MPAS-O*: Expected wind stress fields (left), visualization of results from Catalyst (right) used to diagnose simulation setup errors.

Finally, `coprocessorfinalize` simply cleans up the Catalyst state as follows:

```
void coprocessorfinalize()
{
  vtkCPAdaptorAPI::CoProcessorFinalize();
}
```

This example depends on Python for scripting the analysis. For cases where Python support is not feasible, one can develop Catalyst pipelines in C++, by subclassing `vtkCPPipeline`. In that case, the `coprocessorinitialize()` call should include creating and initializing custom pipelines.

The adaptor can get complicated, especially when dealing with simulation codes with complex data structures and meshes. The most challenging part often is mapping simulation data structures to VTK data model. But once an adaptor is implemented, one can have access to the visualization capabilities and the flexibility of the ParaView application *in situ*. Besides traditional visualization and data analysis, ongoing efforts will enable exporting to a Cinema database that enables fast exploration at the post-processing state through lightweight Cinema viewers[3].

4. CATALYST IN ACTION

Catalyst adapters have been implemented for several simulation codes. In this section we look at some of the current successes.

Helios: CREATE-AV^{TM} Helios is a simulation framework focused on rotorcraft analysis [20]. The Helios user base utilizes a variety of post-processing packages and many of them are unfamiliar with using ParaView. Because of this, instrumenting Catalyst with Helios also involved customization to support several "canned" co-processing outputs that are specified in the simulation input file. These canned outputs currently include extracts for internal boundary surfaces, plane slices, contours, streamlines, sensors/taps and particle paths. Although most of the co-processing output from Helios is time-independent, there is a strong need to include particle path tracking in the simulations to fully understand the flow. This is due to complex flow patterns from rotorcraft having to fly in their own wake.

PHASTA: The largest scale run to date of a Catalyst instrumented simulation code was done with PHASTA. Short for Parallel Hierarchic Adaptive Stabilized Transient Analysis, PHASTA is a highly scalable CFD code led by Kenneth

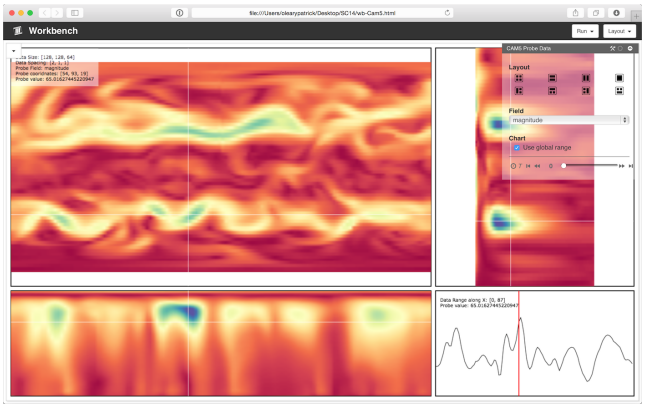

Figure 3: *CAM 5*: Cinema database from CAM 5 being visualized in the *Workbench* for the Velocity magnitude. The database was populated using a Cinema-capable Catalyst pipeline embedded in the simulation run.

Jansen at UC Boulder. In their scaling studies on Argonne National Laboratory's BlueGene/Q Mira machine [15], they were simulating active flow control on a complex wing design. Part of the work included a Catalyst instrumented run using 256K MPI processes. Catalyst was used to contour two separate quantities, Q criterion and wall distance, and generated 1920x1200 png images pseudo-coloring the Q criterion iso-surfaces by velocity magnitude. Figure 1 shows sample output from the simulation run along with an inset zoomed view of the wing tip.

MPAS-O: MPAS-O is a code designed for the simulation of the ocean system from time scales of months to millenia and spatial scales from sub 1 km to global circulations [17]. Typical of many climate simulations, MPAS-O often is run for many days. Due to the complexity of the required inputs, bad simulation set ups can and do occur. By using Catalyst in their runs they can now easily check for incorrect settings. An example of this is shown in Figure 2 where incorrect wind stresses were specified for the run. These were easily diagnosed through visual inspection of the Catalyst output.

xRage: A significant concern when performing *in situ* analysis and visualization is that the desired result may not be captured. This is of high importance when only images are output from a Catalyst run, especially when the area of interest is moving and/or just a fraction of the entire geometry. For this, xRage [12] was instrumented to use Catalyst. Catalyst was extended to provide automated decision making on when and what to save. Based on statistical analysis, appropriate time steps were chosen to sample the data over. Histograms were used for computing appropriate camera placements [14].

UH3D: Studies[10] using Catalyst for UH3D (a global 3D code) concluded that the co-processing through Catalyst introduced a constant memory overhead per core, and 20-30% computation overhead. When looking from the perspective of saving all the data and then post-processing using ParaView if not using Catalyst, the overheads are modest, thus making *in situ* analysis a viable technique for such codes. The memory overhead was approximately 400 MB for the full ParaView/Catalyst code embedded in the simulation. Catalyst Editions, described in Section 2, can be used to reduce Catalyst footprint to 40 MB for an edition with rendering support[7].

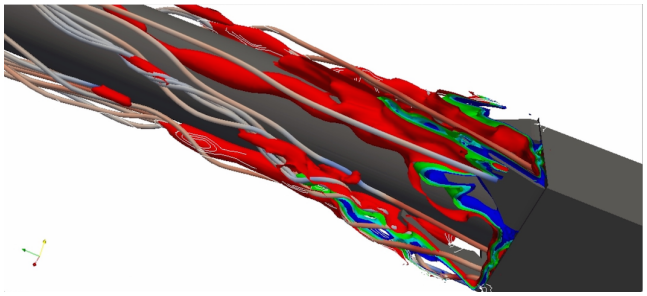

Figure 4: *Hydra-TH*: Visualization of the grid-to-rod frettiung (GTRF) problem – the single rod, spacer and mixing vanes (in grey) with pressure iso-surfaces and velocity streamlines.

CTH: The main goal of *in situ* analysis and visualization is reducing the time to insight for a given workflow. For many simulations, the computational domain and time frame are much larger than the actual domain of interest. This arises for a variety of reasons including difficulty in applying proper initial conditions, avoiding effects of boundary conditions and not knowing *a priori* where the interesting phenomena will occur. Thus, by using *in situ* tools, more appropriate sampling of the data can be performed to reduce file IO and post-processing. This was studied in [13] where a traditional workflow was compared to both an *in transit* and an *in situ* workflow, highlighting the need and benefits of all through approaches for real world analysis.

CAM5: CAM 5.0 is a community resource for studying the atmosphere with a focus on climate applications. There are four dynamic cores: spectral element (default version 5.0), finite-volume (previous default), semi-Lagrangian, and global spectral transformation (Eulerian dynamics). We have demonstrated the current *in situ* visualization implementation can be used to preserve important elements of the CAM 5.0 simulations, significantly reducing the data needed to preserve those elements, and offer the possibility for post-processing exploration. We have implemented the ParaView Catalyst *in situ* analysis and visualization framework in both CAM-FD where we created slice, isosurfaces, and volume visualization pipelines for CAM 5.0. By producing Cinema data products through Catalyst (Figure 3), non-expert users are able to explore the results easily without using complex visualization tools.

Hydra-TH: Hydra-TH[6] was developed as part of the Hydra Toolkit (Hydra) for scalable scientific simulation, led by Dr. Mark A. Christon at Los Alamos National Laboratory. Hydra-TH has been developed for the Consortium for Advanced Simulation of Light-Water Reactors (CASL) to create a computational capability that enables the simulation of the thermal-hydraulics processes inside a nuclear reactor at unprecedented fidelity. These simulations are done by nuclear energy engineers who are not typically computational science experts. Using Catalyst with Hydra-TH has made it simpler to create visualizations for analysis, as shown in Figure 4, by these non-expert users.

5. CONCLUSION

In situ analysis and visualization is one of the ways of tack-

ling the exascale challenges. ParaView Catalyst is a data processing and visualization library that enables *in situ* analysis and visualization. In this paper, we presented the steps involved in instrumenting a simulation code with Catalyst support along with some of our successes.

6. ACKNOWLEDGMENTS

The work with CAM5 was supported by the DOE Office of Science, Advanced Scientific Computing Research, under award number DE-SC0012387, program manager Lucy Nowell.

7. REFERENCES

[1] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, E. Bethel, H. Childs, et al. Scientific discovery at the exascale. report from the doe ascr 2011 workshop on exascale data management. *Analysis, and Visualization*, 2, 2011.

[2] J. Ahrens, B. Geveci, and C. Law. ParaView: An End-User Tool for Large-Data Visualization. *The Visualization Handbook*, page 717, 2005.

[3] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 424–434, Piscataway, NJ, USA, 2014. IEEE Press.

[4] S. Ashby et al. The opportunities and challenges of exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, Fall 2010.

[5] A. C. Bauer, B. Geveci, and W. Schroeder. *The ParaView Catalyst User's Guide v1.0*. Kitware, Inc., 2013.

[6] M. A. Christon. Hydra-th theory manual. Technical report, 2011.

[7] N. Fabian, J. Moreland, Kenneth Mauldin, B. Boeckel, U. Ayachit, A. C. Bauer, and B. Geveci. Instruction memory overhead of in situ visualization libraries on hpc machines. In *Ultrascale Visualization Workshop*, Nov 2014.

[8] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96, Oct 2011.

[9] C. Johnson and R. Ross. Visualization and knowledge discovery: Report from the DOE/ASCR workshop on visual analysis and data exploration at extreme scale. Technical report, October 2007.

[10] H. Karimabadi, B. Loring, P. O'Leary, A. Majumdar, M. Tatineni, and B. Geveci. In-situ visualization for global hybrid simulations. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, pages 57:1–57:8, New York, NY, USA, 2013. ACM.

[11] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich io methods for portable high performance io. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, May 2009.

[12] R. Menikoff and C. A. Scovel. *xRage: HE initiation models*. Jun 2012.

[13] R. A. Oldfield, K. Moreland, N. Fabian, and D. Rogers. Evaluation of methods to integrate analysis into a large-scale shock shock physics code. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 83–92, New York, NY, USA, 2014. ACM.

[14] J. M. Patchett, J. P. Ahrens, B. Nouanesengsy, P. K. Fasel, P. W. Oleary, C. M. Sewell, J. L. Woodring, C. J. Mitchell, L.-T. Lo, K. L. Myers, J. R. Wendelberger, C. V. Canada, M. G. Daniels, H. M. Abhold, and G. M. Rockfeller. *LANL CSSE L2: Case Study of In Situ Data Analysis in ASC Integrated Codes*. Aug 2013.

[15] M. Rasquin, C. Smith, K. Chitale, E. S. Seol, B. A. Matthews, J. L. Martin, O. Sahni, R. M. Loy, M. S. Shephard, and K. E. Jansen. Scalable implicit flow solver for realistic wing simulations with flow control. *Computing in Science and Engineering*, 16(6):13–21, Nov.-Dec. 2014.

[16] M. Richards et al. Exascale software study: Software challenges in extreme scale systems. Technical report, DARPA Information Processing Techniques Office (IPTO), September 2009.

[17] T. Ringler, M. Petersen, R. L. Higdon, D. Jacobsen, P. W. Jones, and M. Maltrud. A multi-resolution approach to global ocean modeling. *Ocean Modelling*, 69:211 – 232, 2013.

[18] G. Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

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

[20] A. Wissink, V. Sankaran, B. Jayaraman, A. Datta, J. Sitaraman, M. Potsdam, S. Kamkar, D. Mavriplis, Z. Yang, R. Jain, J. Lim, and R. Strawn. Capability enhancements in version 3 of the helios high-fidelity rotorcraft simulation code. In *AIAA-2012-0713, AIAA 50th Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, January 2012.