SANDIA REPORT
SAND2007-6006
Unlimited Release
Printed September 2007

Post-Processing V&V Level II ASC Milestone (2360) Results

David B. Karelitz Elmer (Alfonso) Chavez V. Gregory Weirs Timothy M. Shead
Kenneth D. Moreland Thomas A. Brunner Timothy G. Trucano

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online order:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2007-6006
Unlimited Release
Printed September 2007

Post-Processing V&V Level II ASC Milestone (2360) Results

David B. Karelitz and Elmer (Alfonso) Chavez
Scientific Applications and User Support Department

V. Gregory Weirs
Computational Physics Research and Development Department

Timothy M. Shead and Kenneth D. Moreland
Data Analysis and Visualization Department

Thomas A. Brunner
HEDP Theory Department

Timothy G. Trucano
Optimization and Uncertainty Estimation Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0822

Abstract

The 9/30/2007 ASC Level 2 Post-Processing V&V Milestone (Milestone 2360) contains functionality required by the user community for certain verification and validation tasks. These capabilities include loading of edge and face data on an Exodus mesh, run-time computation of an exact solution to a verification problem, delivery of results data from the server to the client, computation of an integral-based error metric, simultaneous loading of simulation and test data, and comparison of that data using visual and quantitative methods. The capabilities were tested extensively by performing a typical ALEGRA HEDP verification task. In addition, a number of stretch criteria were met including completion of a verification task on a 13 million element mesh.

CONTENTS

FIGURES

1. EXECUTIVE SUMMARY

Sandia has met the requirements of the 9/30/2007 ASC Level 2 Post-Processing V&V Milestone (Milestone 2360). All functionality required by the user community is present and has been tested extensively. Capabilities provided and/or tested by the user community for this milestone include loading of edge and face data on an Exodus mesh, run-time computation of an exact solution to a verification problem, delivery of results data from the server to the client, computation of an integral-based error metric, simultaneous loading of simulation and test data, and comparison of that data using visual and quantitative methods. These capabilities have been tested extensively by performing a typical ALEGRA HEDP verification task. In addition, a number of stretch criteria were met including completion of a verification task on a 13 million element mesh.

2. MILESTONE OBJECTIVE SUCCESS CRITERIA

The Completion Criteria specified in the Milestone were to: Complete a set of V&V analyses that: (1) exercise the delivered data analysis capabilities on a range of application runs and/or data sets; and (2) are applied in direct support of SNL V&V application requirements. The example problem chosen to demonstrate the functionality required for this milestone consisted of running the same calculation against a few datasets, 2 of which are pictured in the results section below, a relatively small 2D dataset, and a large 3D dataset. The results were generated by an application code developer as part of his verification efforts.

From the completion criteria specified above, the following objective success criteria were developed early in the calendar year by the milestone team, consisting of David Karelitz, Tom Brunner, Tim Trucano, Dino Pavlakos, David Rogers, Ken Moreland, Tim Shead, Greg Weirs, and Pat Knupp. To successfully complete the milestone, the following baseline capabilities were developed and tested against HEDP applications to the satisfaction of the customers, as represented by Tim Trucano and Tom Brunner.

1. Provide code verification capabilities for standalone and client <-> single-server scenarios
    a. Load and display simulation results containing data residing on edges and faces of elements
    b. Load and calculate a user-defined function at run-time
    c. Calculate an element integral
    d. Deliver results data from the server to the client

2. Provide validation capabilities
    a. Compare experimental and simulated results using plots
    b. Quantitatively compare experimental and simulated results

3. Provide capabilities allowing for integration with automated testing frameworks

    a. Execute a command-line script that performs a code verification task and returns the results of that task
    b. Execute the same command-line script as above that shows the visual results of executing that script.

3. MILESTONE OBJECTIVE STRETCH GOALS

The following capabilities were identified as desirable features for additional development. These capabilities are over and above the baseline success criteria for the milestone -- they were not required for milestone success.

1. Provide code verification capabilities for client <->multiple-server scenarios
    a. Load and calculate a user-defined function at run-time on a 10 million element dataset
    b. Deliver an aggregated value (sum, min, max) from the parallel server to the client

2. Advanced Integrals - Calculate an element integral using an element volume function written by the user

4. RESULTS

4.1. Milestone Objective Success Criteria

1a. Successful loading and display of simulation results containing data residing on edges and faces of elements

A test file was generated containing the necessary edge and face data. Figure 1 contains three images showing the edge, face, and element data present in the results file.

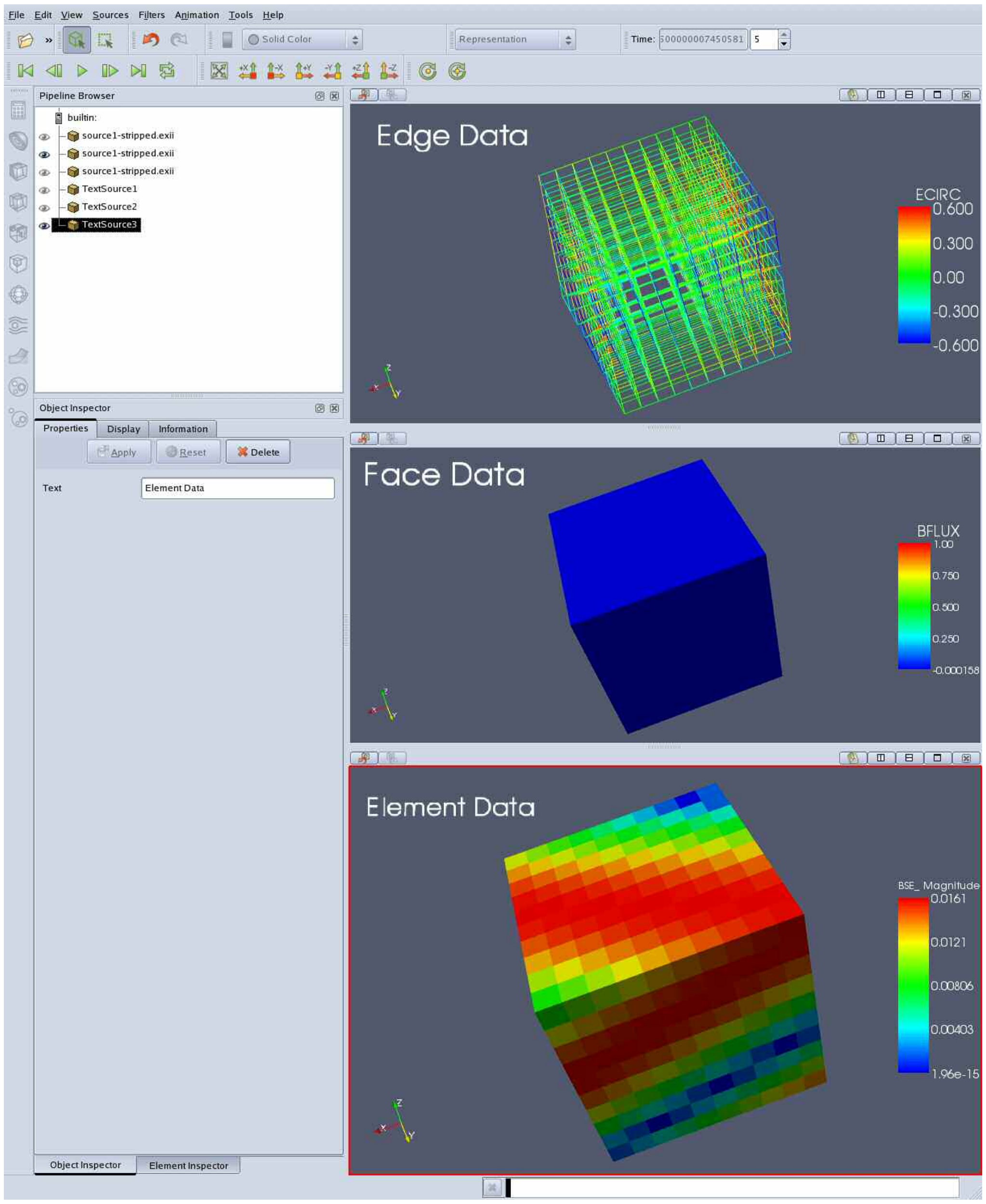


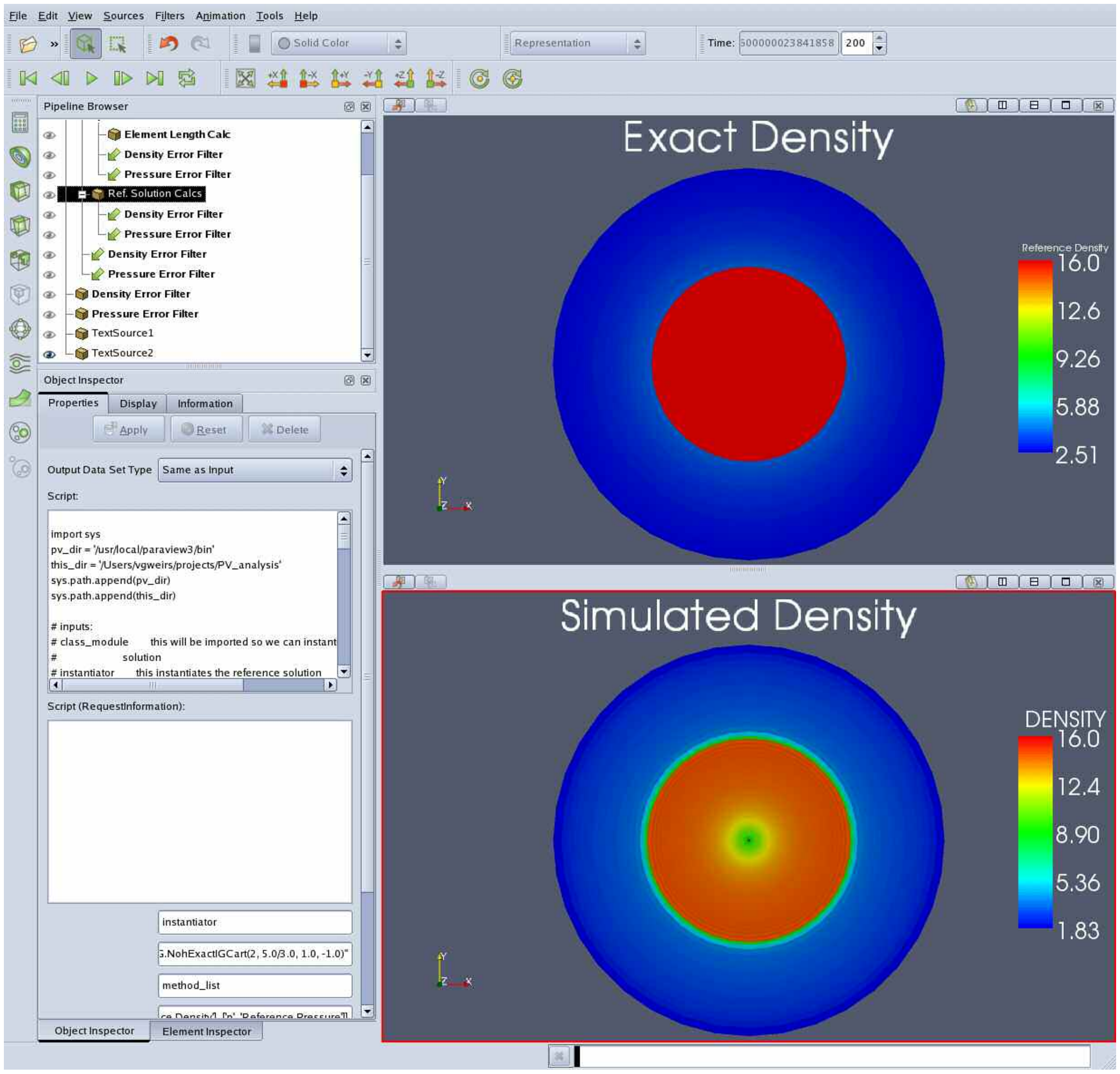
Figure 1. Edge and Face Data.

1b. Successful loading and calculation of a user defined function at run-time

An example HEDP implosion test case, the Noh problem, was used to demonstrate this requirement as well as 1c., 1d., and 3. In this problem, fluid is compressed as it flows inward. At the center the material accumulates causing a shock wave to propagate outward.

In order to compare the simulated results of the Noh test problem with the exact solution, it is necessary to compute the exact solution. The exact solution is a function of time, and location.

Figure 2 shows an image containing the exact computed density on the top and the simulated density (using ALEGRA) on the bottom.



Output Data Set Type: Same as Input

Script:

```
import sys
pv_dir = '/usr/local/paraview3/bin'
this_dir = '/Users/vgweirs/projects/PV_analysis'
sys.path.append(pv_dir)
sys.path.append(this_dir)

# inputs:
# class_module       this will be imported so we can instant
#                    solution
# instantiator       this instantiates the reference solution
```

Script (RequestInformation):

instantiator: G.NohExactIGCart(2, 5.0/3.0, 1.0, -1.0)"

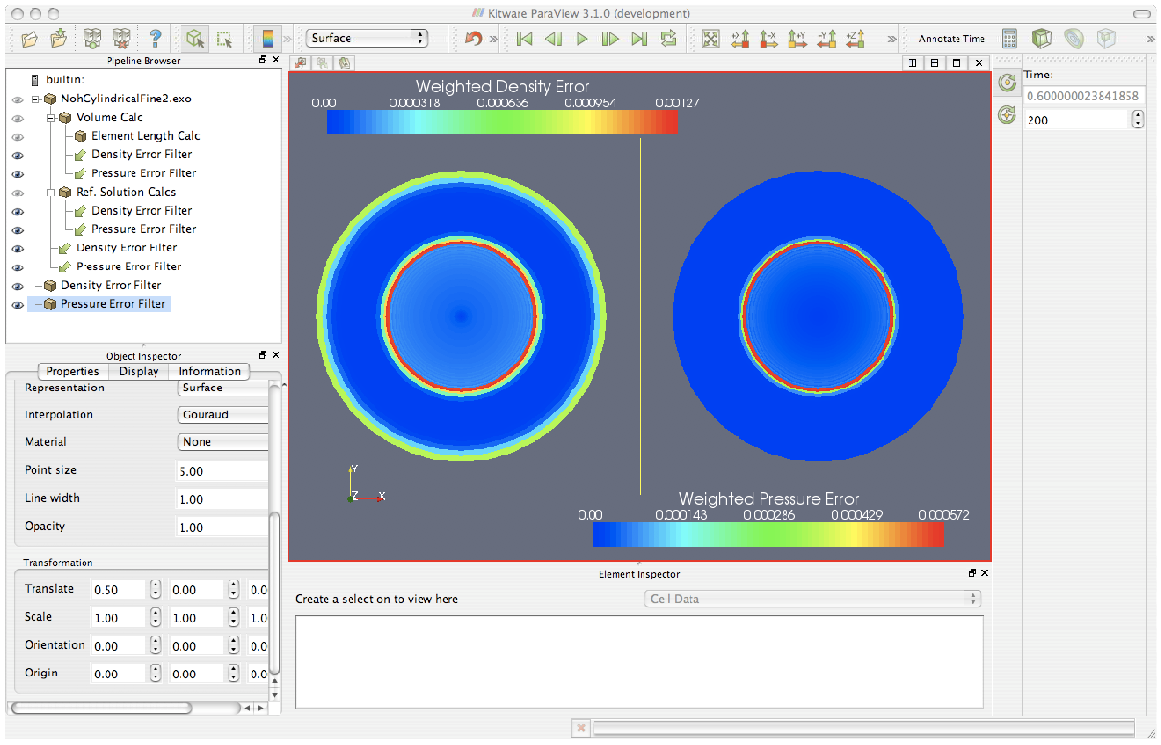
method_list: ce Density'], ['p', 'Reference Pressure']]

Figure 2. Computed Density Versus Simulated Density.

1c. Successful calculation of an element integral

In order to calculate the total error between the simulated and exact solutions, it is necessary to compute a volume-weighted sum of the difference between the simulated and exact solutions.

Figure 3 shows the volume weighted density and pressure error for each cell. Figure 4 shows the sum over all elements in that field, or the integral of the density and pressure error.



```
Representation   Surface
Interpolation    Gouraud
Material         None
Point size       5.00
Line width       1.00
Opacity          1.00

Transformation
Translate    0.50  0.00  0.0
Scale        1.00  1.00  1.0
Orientation  0.00  0.00  0.0
Origin       0.00  0.00  0.0

Time: 0.600000023841858
200
```

Figure 3. Volume Weighted Density and Pressure Error.

```
[dbkarel@s889001 milestone]$ /home/projects/ParaView3-Release/Build/bin/pvpython
proto_script.py
++RESULT        : INFO     Density Error Sum = 0.460152
++RESULT        : INFO     Pressure Error Sum = 0.129899
```

Figure 4. Result of Computing an Element Integral.

1d. Successful delivery of results data from the server to the client

In order to perform client side display and calculations with data, it is necessary to be able to transfer results data from the server to the client. It is also necessary when computing an integral to be able to compute the sum of a field over the entire dataset. ParaView provides a fetch call that can either return the entire dataset, or perform an operation on the data and return the result to the client. It currently supports min, max, and sum collect operations.

Figure 4 shows multiple deliveries of results from the server to the client.

2a. Successful comparison of experimental and simulated results using plots

To show success for criteria 2a and 2b, we compared accelerometer data from a vehicle crash with exodus simulation results of a similar event.

Figure 5 shows the results of the comparison. The graph is plotting Velocity vs. Time, with the simulated results in blue, and the experimental results in red. The simulation results were from an early simulation with different conditions than the experimental result.

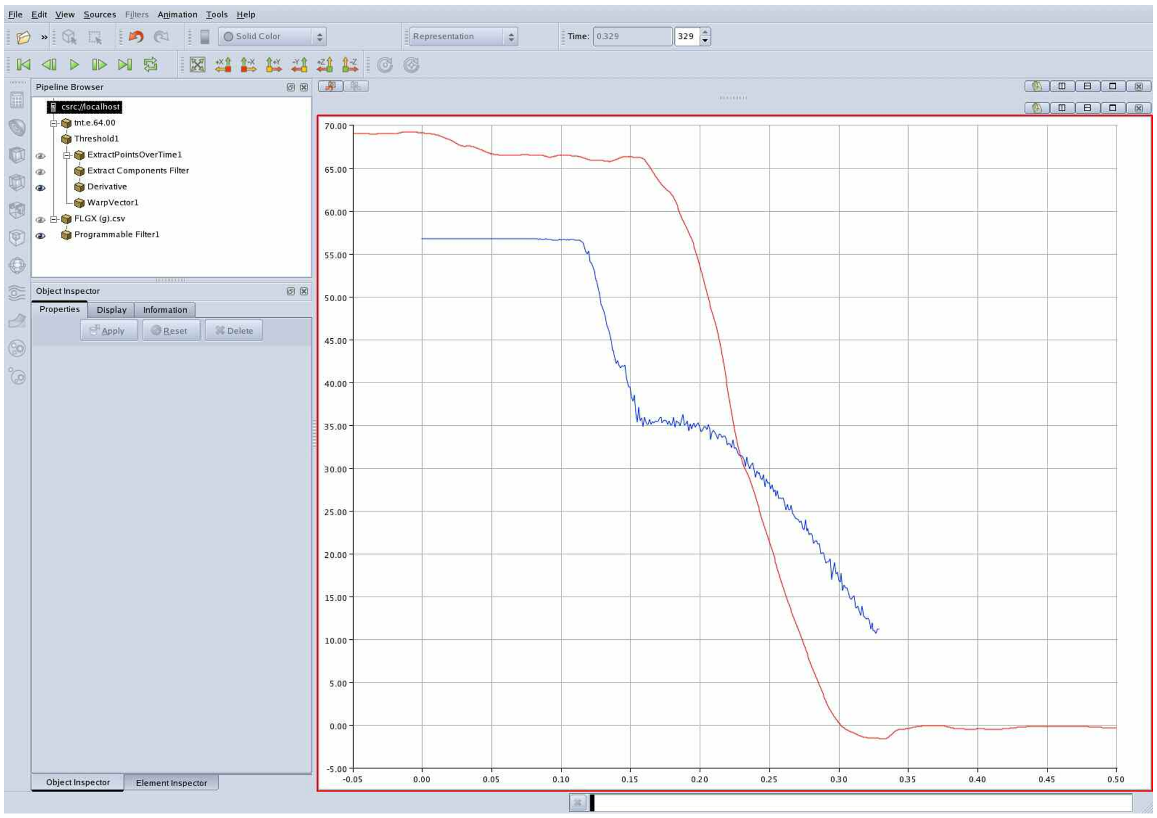
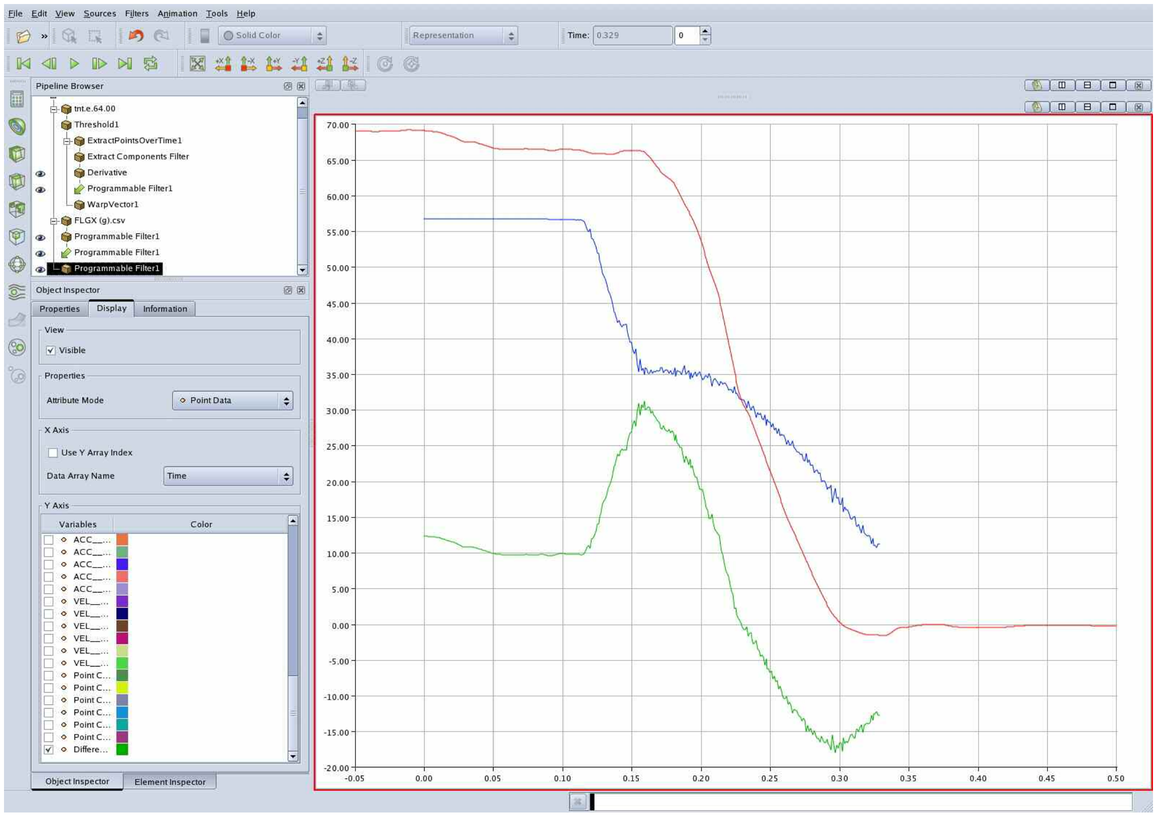


Figure 5. Comparison of Simulated (Blue) and Experimental (Red) Data.

2b. Successful quantitative comparison of experimental and simulated results

Figure 6 shows a similar graph to Figure 5, but it has the addition of a green line showing the difference between the simulated and experimental results. Since the simulated results were at a lower frequency than the experimental results, the experimental result value was linearly interpolated at each timestep in the simulated results data, and the difference computed from that value.



Time: 0.329 | 0

Pipeline Browser:
- tnt.e.64.00
  - Threshold1
    - ExtractPointsOverTime1
      - Extract Components Filter
      - Derivative
      - Programmable Filter1
      - WarpVector1
- FLGX (g).csv
  - Programmable Filter1
  - Programmable Filter1
  - Programmable Filter1

View: [✓ Visible]

Properties: Attribute Mode: Point Data

X Axis: Use Y Array Index ☐; Data Array Name: Time

Y Axis:

Variables	Color
☐ ACC__...	🟧
☐ ACC__...	🟩
☐ ACC__...	🟦
☐ ACC__...	🟥
☐ ACC__...	🟪
☐ VEL__...	🟪
☐ VEL__...	🟦
☐ VEL__...	🟫
☐ VEL__...	🟥
☐ VEL__...	🟩
☐ VEL__...	🟩
☐ Point C...	🟩
☐ Point C...	🟨
☐ Point C...	🟦
☐ Point C...	🟦
☐ Point C...	🟦
☐ Point C...	🟪
✓ Differe...	🟩

Figure 6. Difference (Green) Between Simulated (Blue) and Experimental (Red) Data.

3a. Successful execution of a command-line script that performs a code verification task and returns the results of that task

Figure 7 shows the results of running a code-verification task from command line. The script outputs the results of running that task to a text file.

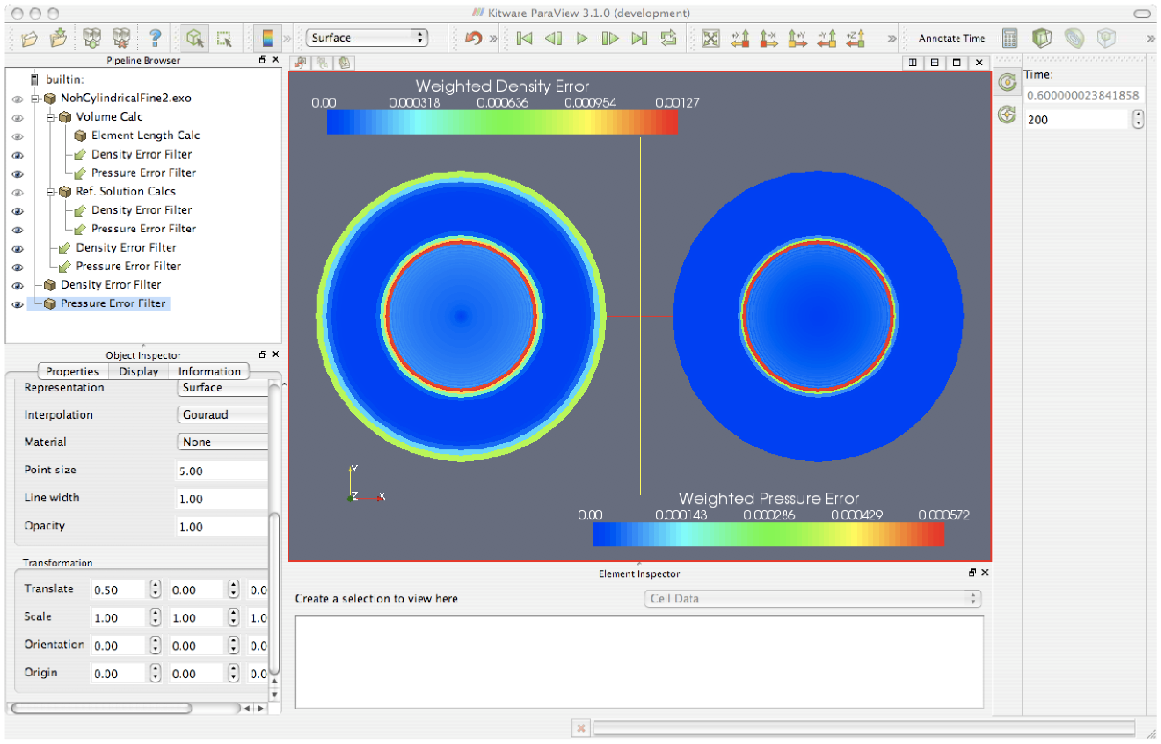
```
++PROTO_SCRIPT: INFO     Opening Connection
++PROTO_SCRIPT: INFO      Running from the python Client; time selection is ENABLED.
++PROTO_SCRIPT: INFO     Reading file
/Users/vgweirs/projects/PV_analysis/newNoh/NohCylindricalFine2.exo
Process PExodusReader [..........]
++PROTO_SCRIPT: INFO     Number of elements = 2880
++PROTO_SCRIPT: INFO     Loading Variables
Process PExodusReader [..........]
++PROTO_SCRIPT: INFO     Running Programmable filter with script volume_field_script_native
Process PExodusReader [..........]
Process PythonProgrammableFilter [..........]
++PROTO_SCRIPT: INFO     Running Programmable filter with script elem_length_field_script
Process PythonProgrammableFilter [..........]
++PROTO_SCRIPT: INFO     Adding TimeSelectionFilter to pipeline, time = 0.600000
Process PExodusReader [..........]
Process PythonProgrammableFilter [..........
++PROTO_SCRIPT: INFO     Running FetchSumFilter
applying operation
]
Process ReductionFilter [..........]
Process MinMax [..........]
Process ReductionFilter [..........]
Process ClientServerMoveData [..........
++RESULT      : INFO     Domain Volume = 0.518073
++PROTO_SCRIPT: INFO     Running FetchSumFilter
applying operation
]
Process ReductionFilter [..........]
Process MinMax [..........]
Process ReductionFilter [..........]
Process ClientServerMoveData [..........
++RESULT      : INFO     Characteristic Length SUM = 31.616480
++RESULT      : INFO     Average Characteristic Length = 0.010978
++PROTO_SCRIPT: INFO     Running Programmable filter with script cell_field_function_script
]
Process PythonProgrammableFilter [..........]
++PROTO_SCRIPT: INFO     Running Weighted Difference filter - DENSITY
Process PythonProgrammableFilter [..........]
++PROTO_SCRIPT: INFO     Running Weighted Difference filter - PRESSURE
Process PythonProgrammableFilter [..........]
++PROTO_SCRIPT: INFO     Adding TimeSelectionFilter to pipeline, time = 0.600000
++PROTO_SCRIPT: INFO     Running FetchSumFilter
applying operation
Process ReductionFilter [..........]
Process MinMax [..........]
Process ReductionFilter [..........]
Process ClientServerMoveData [..........
++RESULT      : INFO     Density Error Sum = 0.460152
++RESULT      : INFO     L1 Density Error = 0.888201
++PROTO_SCRIPT: INFO     Running FetchSumFilter
applying operation
]
Process ReductionFilter [..........]
Process MinMax [..........]
Process ReductionFilter [..........]
Process ClientServerMoveData [..........
++RESULT      : INFO     Pressure Error Sum = 0.129899
++RESULT      : INFO     L1 Pressure Error = 0.250736
```

Figure 7. Results of Running a Code Verification Script from the Command Line.

3b. Successful execution of the same command-line script as above that shows the visual results of executing that script

So that it's easy to find out why a verification task is not being completed, it's necessary to be able to run the same script used for the command line requirement from the GUI and see the visual results of running that script.

Figure 8 shows the GUI results of running the same script as above.



Representation: Surface
Interpolation: Gouraud
Material: None
Point size: 5.00
Line width: 1.00
Opacity: 1.00

Translate	0.50	0.00	0.0
Scale	1.00	1.00	1.0
Orientation	0.00	0.00	0.0
Origin	0.00	0.00	0.0

Time: 0.600000023841858
200

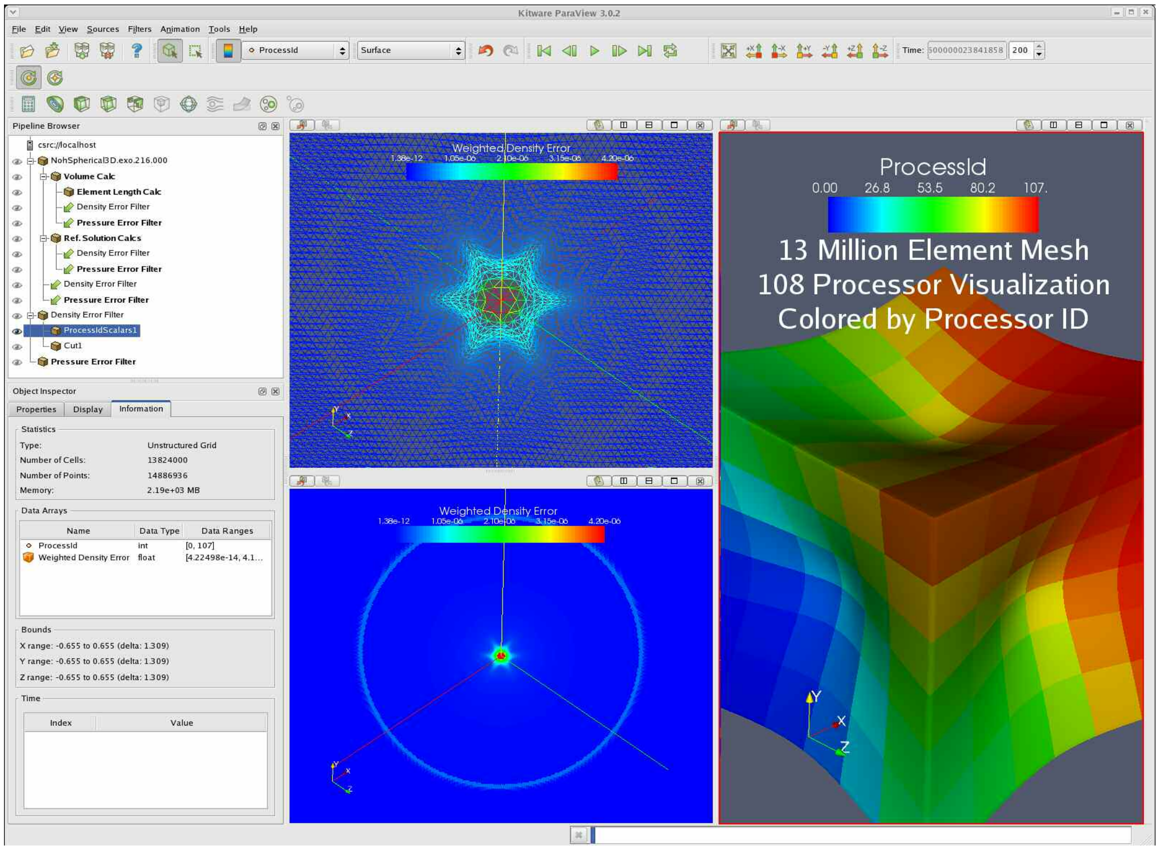
Figure 8. Results of Running a Code Verification Script from the GUI.

4.2. Milestone Objective Stretch Goals

1a. Successful loading and calculation of a user-defined function at run-time on a 10 million element dataset

A 13M element dataset was generated for the Noh Test problem and used for the large dataset stretch goals. The images were generated running client/server from an Engineering Sciences Linux workstation to 54 nodes on the BlackRose cluster. It took less than a minute to calculate the exact solution and the error metrics.

Figure 9 shows the dataset on the right colored by processor ID. The top left shows a wireframe view of a slice through the mesh colored by the difference between the computed and exact solutions, and the lower left shows the same mesh, but as a surface. You can see the number of elements in the dataset (13,824,000) on the left hand side.



Statistics

Type:	Unstructured Grid
Number of Cells:	13824000
Number of Points:	14886936
Memory:	2.19e+03 MB

Data Arrays

Name	Data Type	Data Ranges
ProcessId	int	[0, 107]
Weighted Density Error	float	[4.22498e-14, 4.1...

Bounds

X range: -0.655 to 0.655 (delta: 1.309)
Y range: -0.655 to 0.655 (delta: 1.309)
Z range: -0.655 to 0.655 (delta: 1.309)

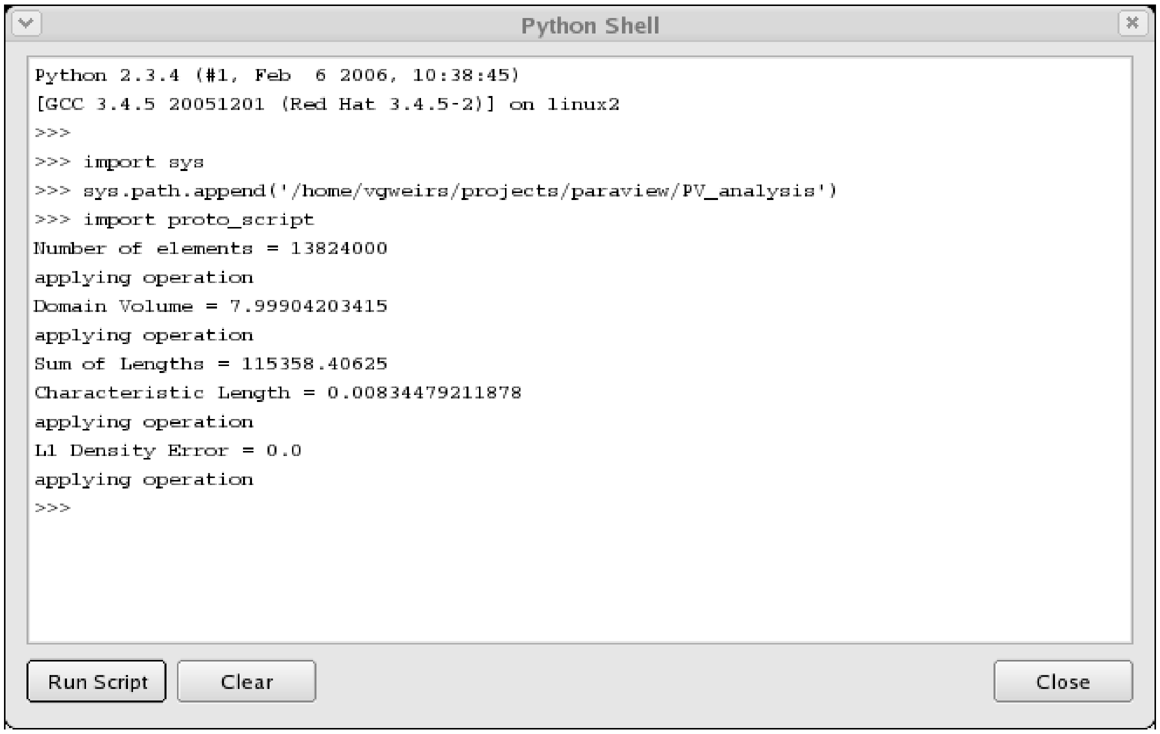
Time

Index	Value

Figure 9. Results of Running a User-Defined Function on a 13M Element Dataset.

1b. Successful delivery of an aggregated value (sum, min, max) from the parallel server to the client

The characteristic length was computed on the 13M element dataset. This requires computing the characteristic length of each element, summing those values, and dividing by the number of elements. The result is shown as “Characteristic Length” in Figure 10.

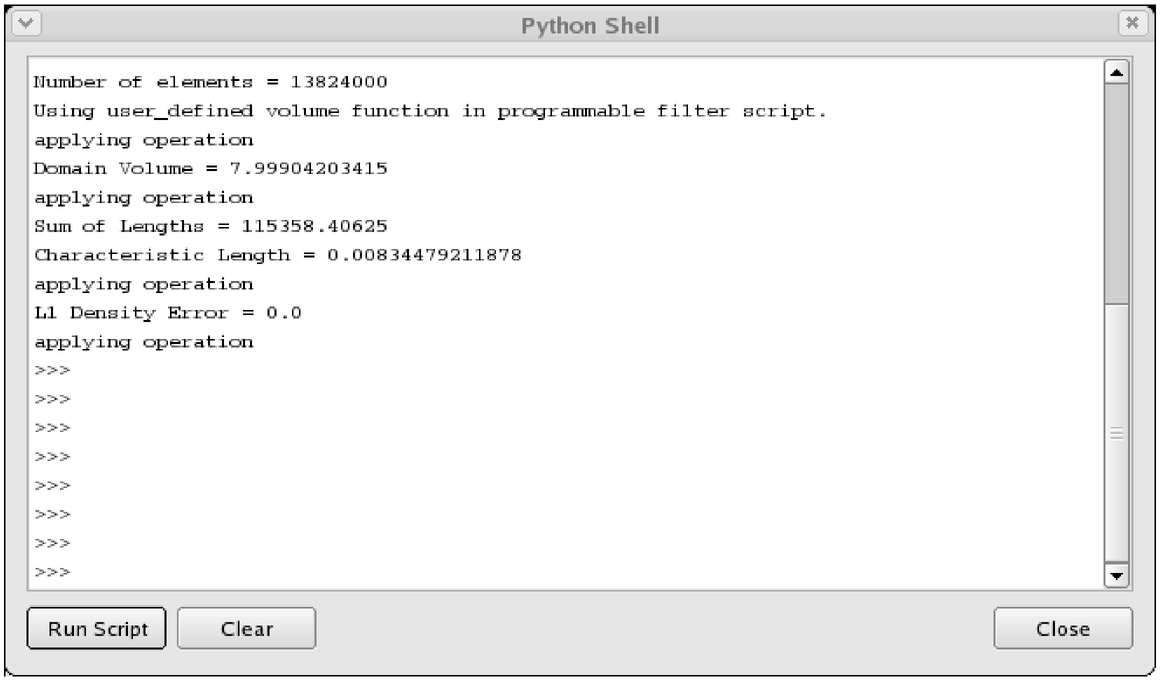


```
Python 2.3.4 (#1, Feb  6 2006, 10:38:45)
[GCC 3.4.5 20051201 (Red Hat 3.4.5-2)] on linux2
>>>
>>> import sys
>>> sys.path.append('/home/vgweirs/projects/paraview/PV_analysis')
>>> import proto_script
Number of elements = 13824000
applying operation
Domain Volume = 7.99904203415
applying operation
Sum of Lengths = 115358.40625
Characteristic Length = 0.00834479211878
applying operation
L1 Density Error = 0.0
applying operation
>>>
```

Figure 10. Client-Side Output from a Calculation Run Using a 13M Element Dataset.

2. Successful calculation of an element integral using an element volume function written by the user

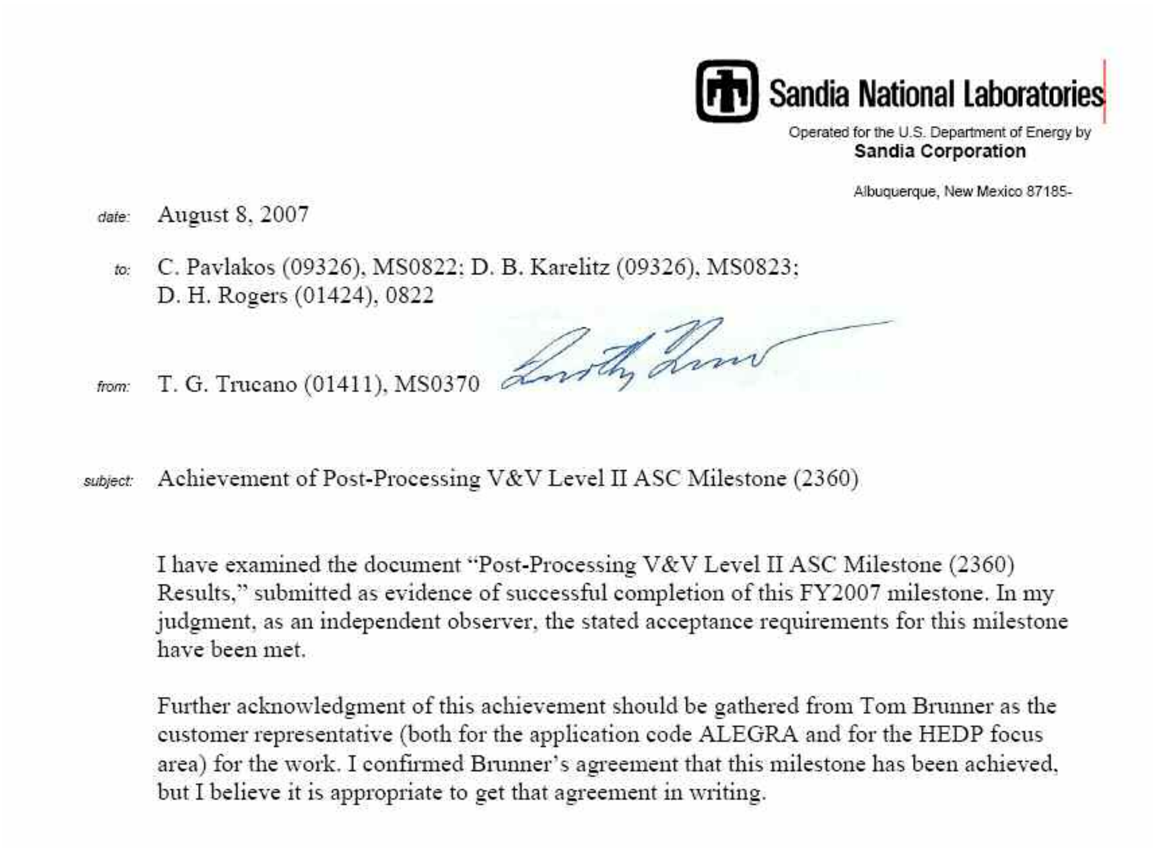
An element integral was performed using a custom volume function. The integral is shown as Domain Volume in Figure 11.



```
Number of elements = 13824000
Using user_defined volume function in programmable filter script.
applying operation
Domain Volume = 7.99904203415
applying operation
Sum of Lengths = 115358.40625
Characteristic Length = 0.00834479211878
applying operation
L1 Density Error = 0.0
applying operation
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```

Figure 11. Integral Using a Custom Volume Function.

Appendix A: Customer Signoff Memos



Sandia National Laboratories

Operated for the U.S. Department of Energy by
Sandia Corporation

Albuquerque, New Mexico 87185-

date: August 8, 2007

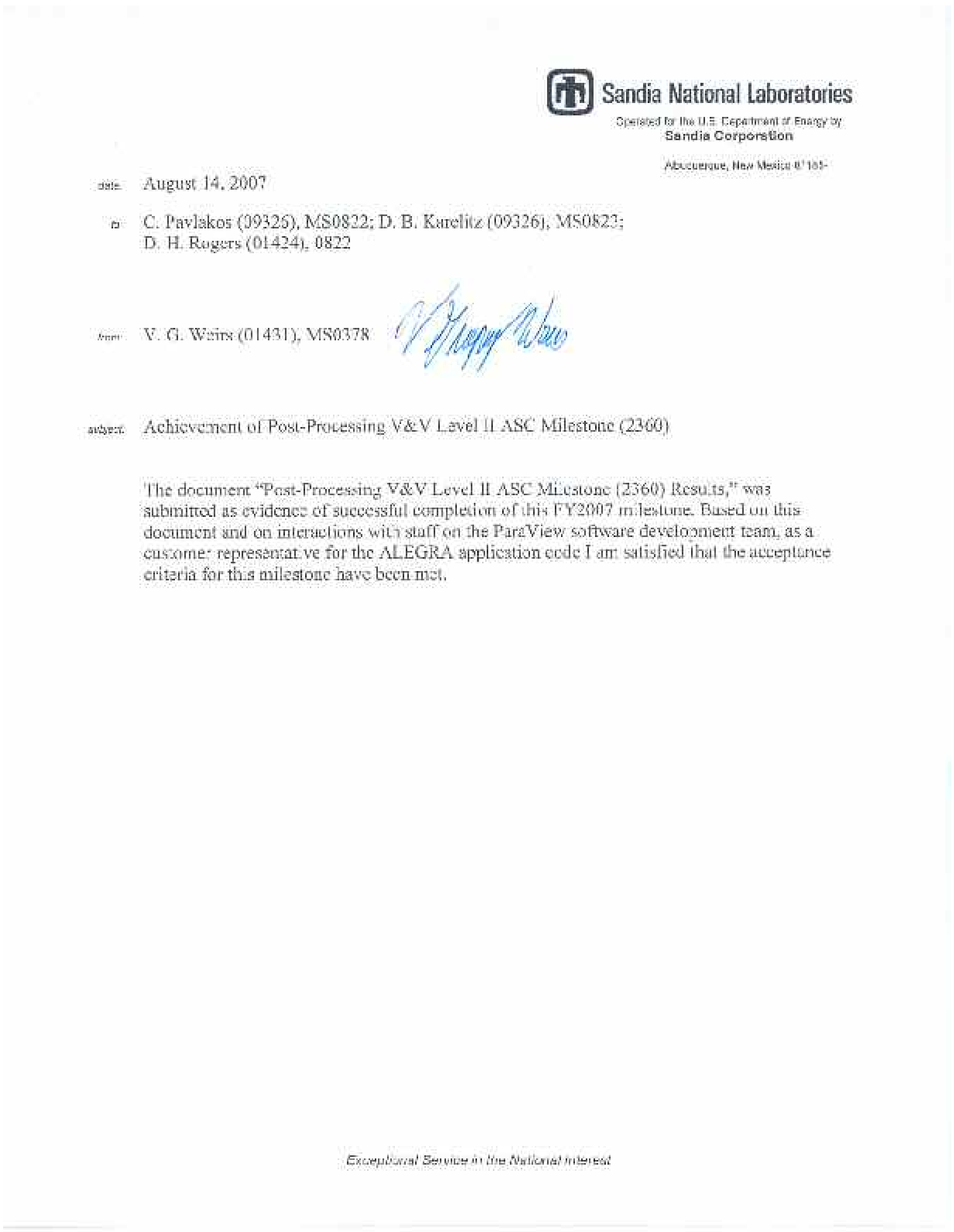
to: C. Pavlakos (09326), MS0822; D. B. Karelitz (09326), MS0823;
D. H. Rogers (01424), 0822

from: T. G. Trucano (01411), MS0370 [illegible]

subject: Achievement of Post-Processing V&V Level II ASC Milestone (2360)

I have examined the document "Post-Processing V&V Level II ASC Milestone (2360) Results," submitted as evidence of successful completion of this FY2007 milestone. In my judgment, as an independent observer, the stated acceptance requirements for this milestone have been met.

Further acknowledgment of this achievement should be gathered from Tom Brunner as the customer representative (both for the application code ALEGRA and for the HEDP focus area) for the work. I confirmed Brunner's agreement that this milestone has been achieved, but I believe it is appropriate to get that agreement in writing.



Sandia National Laboratories

Operated for the U.S. Department of Energy by
Sandia Corporation

Albuquerque, New Mexico 87185-

date: August 14, 2007

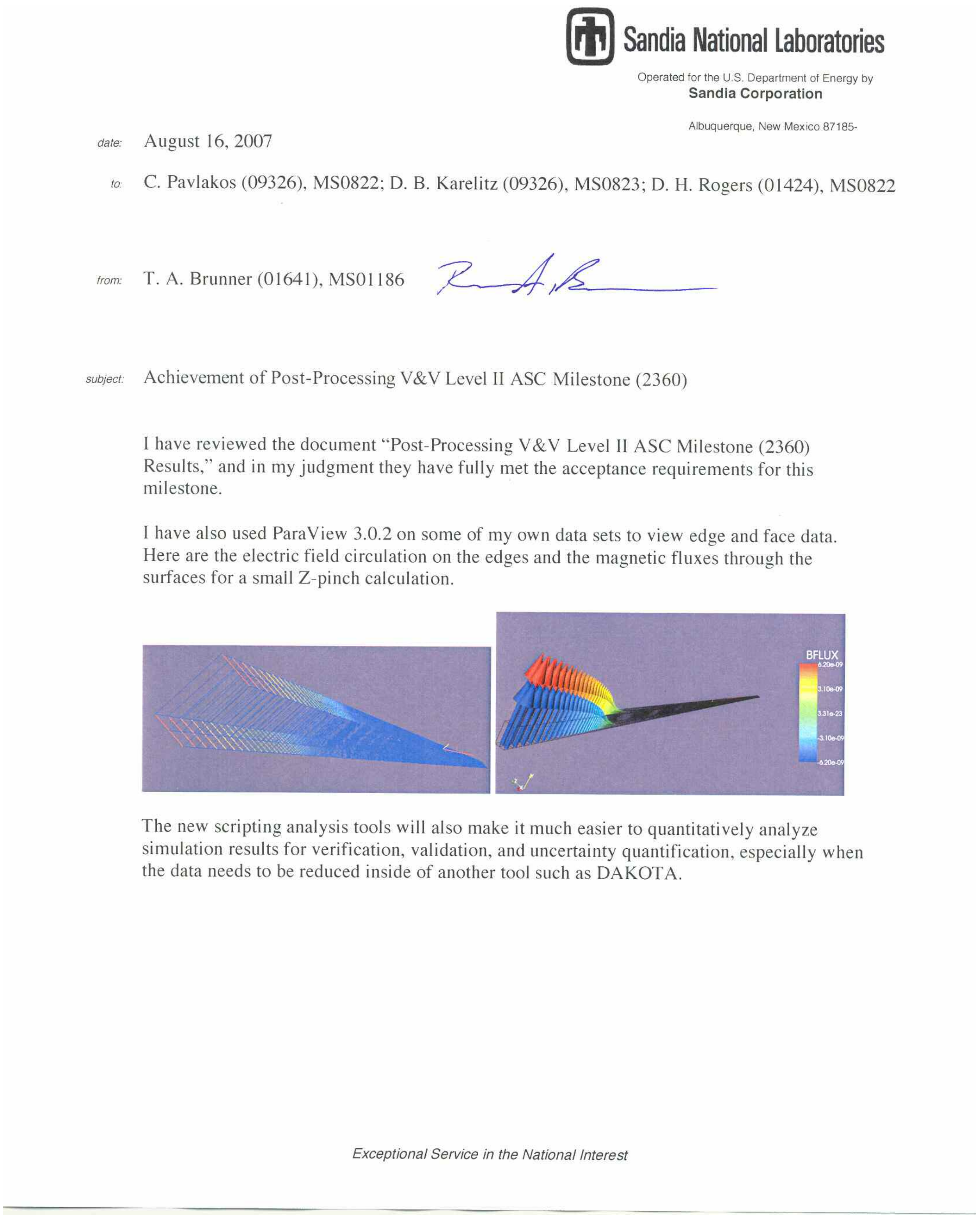
to: C. Pavlakos (09326), MS0822; D. B. Karelitz (09326), MS0822;
D. H. Rogers (01424), 0822

from: V. G. Weirs (01431), MS0378 [illegible]

subject: Achievement of Post-Processing V&V Level II ASC Milestone (2360)

The document "Post-Processing V&V Level II ASC Milestone (2360) Results," was submitted as evidence of successful completion of this FY2007 milestone. Based on this document and on interactions with staff on the ParaView software development team, as a customer representative for the ALEGRA application code I am satisfied that the acceptance criteria for this milestone have been met.

Exceptional Service in the National Interest



Sandia National Laboratories

Operated for the U.S. Department of Energy by
Sandia Corporation

Albuquerque, New Mexico 87185-

date: August 16, 2007

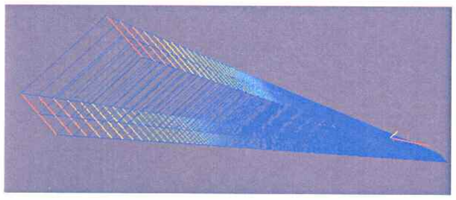
to: C. Pavlakos (09326), MS0822; D. B. Karelitz (09326), MS0823; D. H. Rogers (01424), MS0822

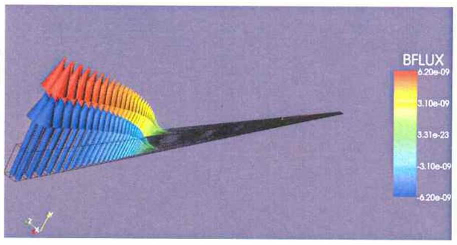
from: T. A. Brunner (01641), MS01186 [illegible]

subject: Achievement of Post-Processing V&V Level II ASC Milestone (2360)

I have reviewed the document "Post-Processing V&V Level II ASC Milestone (2360) Results," and in my judgment they have fully met the acceptance requirements for this milestone.

I have also used ParaView 3.0.2 on some of my own data sets to view edge and face data. Here are the electric field circulation on the edges and the magnetic fluxes through the surfaces for a small Z-pinch calculation.





The new scripting analysis tools will also make it much easier to quantitatively analyze simulation results for verification, validation, and uncertainty quantification, especially when the data needs to be reduced inside of another tool such as DAKOTA.

Exceptional Service in the National Interest

Appendix B: Scripts Used for the Noh Example Problem

proto_script.py

```
# This script computes the L1 error norms for Density and Pressure and
# the characteristic element length using paraview.  Data that will be
# passed in in the future is in proto_data.py. Wrappers for low level
# paraview/vtk commands, which are believed to be broadly useful, are
# in TampaPVTools.py and associated modules; these may one day be
# included in the pvanalysis.py module, which may in turn become part
# of paraview.py. At this time there are still some hard-coded
# sections (esp. paths) which prevent truly general usage.
#
# The pipeline is somewhat complicated. One branch computes the
# characteristic element length with the following sequence: Reader,
# Programmable filter for volume field, ProgrammableFilter for element
# length field, FetchSum filter for sum of element lengths, and
# finally divide the sum by the number of elements for the (desired)
# average characteristic element length. A FetchSum filter is also
# applied to the volume field, which gives the domain volume.
#
# A second branch begins with the same reader. A ProgrammableFilter is
# used to compute the reference density and reference pressure on the
# mesh of the reader. The reference density, simulation density (from
# the reader), and the volume (from the first branch) fields are
# inputs to a ProgrammableFilter which computes the weighted density
# difference field. The weighted density differencs is summed and
# divided by the domain volume (computed in the first branch) to
# produce the L1 norm of the density error. Just as for the density,
# the L1 norm of the pressure error is computed.
#
# Before the Fetch filters are applied, a TimeSelection filter (a time
# suppressor) is used to select the simulation time for the
# calculations. This is disabled if the script is run in through the
# python shell in the paraview GUI client.
#
# V. Gregory Weirs
# 18 June 2007
#


########
# Setup
########
# Find out where we are and append that to the path.
from  proto_data import *
import sys
import os

cwd = os.getcwd()
sys.path.append(cwd)

# Try managing messages with logger module
# Logs for diagnostics and results; both go to the screen and to a file
import logging

log = logging.getLogger('PROTO_SCRIPT')
result_log = logging.getLogger('RESULT')

format = logging.Formatter('++%(name)-12s: %(levelname)-8s %(message)s')

screen_out = logging.StreamHandler()
screen_out.setFormatter(format)

file_out = logging.FileHandler('proto_script.log','w')
file_out.setFormatter(format)
```

```
log.setLevel(logging.DEBUG)
log.addHandler(screen_out)
log.addHandler(file_out)

result_log.setLevel(logging.INFO)
result_log.addHandler(screen_out)
result_log.addHandler(file_out)

# Find our modules and define shorthand references
import paraview
import pvanalysis
import TampaPVTools

PV = paraview
PVA = pvanalysis
TPVT = TampaPVTools

########
# Now start scripting the work.
########
log.info("Opening Connection")
PVA.OpenConnection()

if pvanalysis.GUIClient:
    log.info(' Running from the GUI Client; time selection is SUPPRESSED.')
else:
    log.info(' Running from the python Client; time selection is ENABLED.')

# Get the simulation data from the file.
log.info("Reading file %s",  file )
reader = PVA.OpenExodusFile( file )

reader_info = reader.GetDataInformation()
num_elements = reader_info.GetNumberOfCells()
log.info("Number of elements = %s", num_elements )

log.info("Loading Variables")
TPVT.LoadVariables(reader, var_list)

log.info( "Running Programmable filter with script %s",
          "volume_field_script_native")
vol_calc = TPVT.ProgrammableFilter(reader,
                                   TPVT.volume_field_script_native,
                                   output_filter_name="Volume Calc")

log.info("Running Programmable filter with script %s",
         "elem_length_field_script")
elem_length_calc = TPVT.ProgrammableFilter(vol_calc,
                                           TPVT.elem_length_field_script,
                                           output_filter_name="Element Length Calc")

if (not PVA.GUIClient):
    log.info("Adding TimeSelectionFilter to pipeline, time = %f", test_time)
single_time = TPVT.TimeSelectionFilter(elem_length_calc, test_time, "Chosen Time")


log.info("Running FetchSumFilter")
domain_volume = TPVT.FetchSumFilter(single_time,
                                    "Volume")

result_log.info("Domain Volume = %f", domain_volume)

log.info("Running FetchSumFilter")
length_sum = TPVT.FetchSumFilter(single_time,
                                 "Characteristic Length")

avg_length = length_sum/num_elements
result_log.info("Characteristic Length SUM = %f", length_sum)
result_log.info("Average Characteristic Length = %f", avg_length)
```

```
#
# Now compute the exact solution on the mesh given.
#
log.info("Running Programmable filter with script %s", "cell_field_function_script")
ref_density_calc = TPVT.ProgrammableFilter(reader,
                                           TPVT.cell_field_function_script,
                                           Noh_param_dict,
                                           "Ref. Solution Calcs")

#
# Compute the difference fields
#
log.info("Running Weighted Difference filter - DENSITY")
diff_calc_rho = TPVT.WeightedCellDifferenceFilter(reader, 'DENSITY',
                                                  ref_density_calc, 'Reference Density',
                                                  vol_calc, 'Volume',
                                                  'Weighted Density Error',
                                                  'Density Error Filter')


log.info("Running Weighted Difference filter - PRESSURE")
diff_calc_p = TPVT.WeightedCellDifferenceFilter(reader, 'PRESSURE',
                                                ref_density_calc, 'Reference Pressure',
                                                vol_calc, 'Volume',
                                                'Weighted Pressure Error',
                                                'Pressure Error Filter')


if (not PVA.GUIClient):
    log.info("Adding TimeSelectionFilter to pipeline, time = %f", test_time)
single_time2 = TPVT.TimeSelectionFilter(diff_calc_rho, test_time, "Chosen Time")
single_time3 = TPVT.TimeSelectionFilter(diff_calc_p, test_time, "Chosen Time")

log.info("Running FetchSumFilter")
rhodiff_sum = TPVT.FetchSumFilter(single_time2,
                                 "Weighted Density Error")

result_log.info("Density Error Sum = %f", rhodiff_sum)
result_log.info("L1 Density Error = %f", rhodiff_sum/domain_volume)


log.info("Running FetchSumFilter")
pdiff_sum = TPVT.FetchSumFilter(single_time3,
                                "Weighted Pressure Error")

result_log.info("Pressure Error Sum = %f", pdiff_sum)
result_log.info("L1 Pressure Error = %f", pdiff_sum/domain_volume)
```

proto_data.py

```
# Define inputs that one day will be passed in.
pv_dir = '/usr/local/paraview3/bin'
this_dir = '/Users/vgweirs/projects/PV_analysis/'

FileName = 'newNoh/NohCylindricalFine2.exo'
#FileName = 'NohCylindricalCoarse.exo'

file = this_dir + FileName

var_list = ['DENSITY', 'PRESSURE']

test_time = 0.6

Noh_param_dict={ 'class_module': 'NohExactIG',
                 'instantiator': 'NohExactIG.NohExactIGCart(2, 5.0/3.0, 1.0, -1.0)',
                 'method_list': [['rho', 'Reference Density'],
                                 ['p', 'Reference Pressure']  ],
                 'n_subel':1}
```

ElemData.py

```
#!/usr/bin/python

# Volume calclations are taken directly from element files in alegra/framework

quad4_ndim = 2

def quad4_vol(X):
  vol  = (X[0][0] - X[2][0]) * (X[1][1] - X[3][1])
  vol += (X[1][0] - X[3][0]) * (X[2][1] - X[0][1])
  return 0.5*vol

def quad4_subdiv(elem_coords, intervals):
    """Compute an equispaced subdivision of a quad4 element.

    Quadrature points are equispaced, rather than Gaussian; improved
    order of accuracy of Gaussian quadrature is not achieved for
    discontinuous data.

    Input:
    elem_coords:  coordinates of element's nodes, assuming exodus
                    node order convention - (counter clockwise around
                    the element)
    intervals:    The element will be subdivided into nx*ny equispaced
                    subelements, where nx = ny = intervals
    """

    subelem_coords = []

    for ii in range(intervals):
        for jj in range(intervals):
            i = (0.5 + ii)/intervals
            j = (0.5 + jj)/intervals
            X_pt = (   (1-j)*(1-i)*elem_coords[0][0]
                     + (1-j)*   i *elem_coords[1][0]
                     +    j *   i *elem_coords[2][0]
                     +    j *(1-i)*elem_coords[3][0] )
            Y_pt = (   (1-j)*(1-i)*elem_coords[0][1]
                     + (1-j)*   i *elem_coords[1][1]
                     +    j *   i *elem_coords[2][1]
                     +    j *(1-i)*elem_coords[3][1] )
            subelem_coords.append( [X_pt, Y_pt, 0.0] )

    return subelem_coords

def quad4_ctr(elem_coords):
    """Compute the coordinates of the center of a quad4 element.

    Simple average in physical space.

    The result is the same as for quad4_subdiv with intervals=1.

    Input:
    elem_coords:  coordinates of element's nodes, assuming exodus
                    node order convention - (counter clockwise around
                    the element)
    """
    X_pt = 0.25*(   elem_coords[0][0]
                  + elem_coords[1][0]
                  + elem_coords[2][0]
                  + elem_coords[3][0] )
    Y_pt = 0.25*(   elem_coords[0][1]
                  + elem_coords[1][1]
                  + elem_coords[2][1]
                  + elem_coords[3][1] )
    return [ X_pt, Y_pt, 0.0 ]
```

```
hex8_ndim = 3

def hex8_vol(X):

  x1 = X[0][0]
  x2 = X[1][0]
  x3 = X[2][0]
  x4 = X[3][0]
  x5 = X[4][0]
  x6 = X[5][0]
  x7 = X[6][0]
  x8 = X[7][0]

  y1 = X[0][1]
  y2 = X[1][1]
  y3 = X[2][1]
  y4 = X[3][1]
  y5 = X[4][1]
  y6 = X[5][1]
  y7 = X[6][1]
  y8 = X[7][1]

  z1 = X[0][2]
  z2 = X[1][2]
  z3 = X[2][2]
  z4 = X[3][2]
  z5 = X[4][2]
  z6 = X[5][2]
  z7 = X[6][2]
  z8 = X[7][2]

  rx0 = (  y2*((z6-z3)-(z4-z5))
          +y3*(z2-z4)
          +y4*((z3-z8)-(z5-z2))
          +y5*((z8-z6)-(z2-z4))
          +y6*(z5-z2)
          +y8*(z4-z5) )
  rx1 = (  y3*((z7-z4)-(z1-z6))
          +y4*(z3-z1)
          +y1*((z4-z5)-(z6-z3))
          +y6*((z5-z7)-(z3-z1))
          +y7*(z6-z3)
          +y5*(z1-z6) )
  rx2 = (  y4*((z8-z1)-(z2-z7))
          +y1*(z4-z2)
          +y2*((z1-z6)-(z7-z4))
          +y7*((z6-z8)-(z4-z2))
          +y8*(z7-z4)
          +y6*(z2-z7) )
  rx3 = (  y1*((z5-z2)-(z3-z8))
          +y2*(z1-z3)
          +y3*((z2-z7)-(z8-z1))
          +y8*((z7-z5)-(z1-z3))
          +y5*(z8-z1)
          +y7*(z3-z8) )
  rx4 = (  y8*((z4-z7)-(z6-z1))
          +y7*(z8-z6)
          +y6*((z7-z2)-(z1-z8))
          +y1*((z2-z4)-(z8-z6))
          +y4*(z1-z8)
          +y2*(z6-z1) )
  rx5 = (  y5*((z1-z8)-(z7-z2))
          +y8*(z5-z7)
          +y7*((z8-z3)-(z2-z5))
          +y2*((z3-z1)-(z5-z7))
          +y1*(z2-z5)
          +y3*(z7-z2) )
  rx6 = (  y6*((z2-z5)-(z8-z3))
          +y5*(z6-z8)
          +y8*((z5-z4)-(z3-z6))
          +y3*((z4-z2)-(z6-z8))
          +y2*(z3-z6)
          +y4*(z8-z3) )
  rx7 = (  y7*((z3-z6)-(z5-z4))
          +y6*(z7-z5)
          +y5*((z6-z1)-(z4-z7))
          +y4*((z1-z3)-(z7-z5))
          +y3*(z4-z7)
          +y1*(z5-z4) )

  vol = (   x1*rx0
          + x2*rx1
          + x3*rx2
          + x4*rx3
          + x5*rx4
          + x6*rx5
          + x7*rx6
          + x8*rx7 ) / 12.0

  return vol



def hex8_subdiv(elem_coords, intervals):
    """Compute an equispaced subdivision of a hex8 element.

    Quadrature points are equispaced, rather than Gaussian; improved
    order of accuracy of Gaussian quadrature is not achieved for
    discontinuous data.

    Input:
    elem_coords:  coordinates of element's nodes, assuming exodus
                    node order convention - (counter clockwise around
                    the element)
    intervals:    The element will be subdivided into nx*ny equispaced
                    subelements, where nx = ny = intervals
    """

    subelem_coords = []

    for ii in range(intervals):
        i = (0.5 + ii)/intervals
        for jj in range(intervals):
            j = (0.5 + jj)/intervals
            for kk in range(intervals):
               k = (0.5 + kk)/intervals
               X_pt = (   (1-k)*(1-j)*(1-i)*elem_coords[0][0]
                        + (1-k)*(1-j)*   i *elem_coords[1][0]
                        + (1-k)*   j *   i *elem_coords[2][0]
                        + (1-k)*   j *(1-i)*elem_coords[3][0]
                        +    k *(1-j)*(1-i)*elem_coords[4][0]
                        +    k *(1-j)*   i *elem_coords[5][0]
                        +    k *   j *   i *elem_coords[6][0]
                        +    k *   j *(1-i)*elem_coords[7][0] )
               Y_pt = (   (1-k)*(1-j)*(1-i)*elem_coords[0][1]
                        + (1-k)*(1-j)*   i *elem_coords[1][1]
                        + (1-k)*   j *   i *elem_coords[2][1]
                        + (1-k)*   j *(1-i)*elem_coords[3][1]
                        +    k *(1-j)*(1-i)*elem_coords[4][1]
                        +    k *(1-j)*   i *elem_coords[5][1]
                        +    k *   j *   i *elem_coords[6][1]
                        +    k *   j *(1-i)*elem_coords[7][1] )
               Z_pt = (   (1-k)*(1-j)*(1-i)*elem_coords[0][2]
                        + (1-k)*(1-j)*   i *elem_coords[1][2]
                        + (1-k)*   j *   i *elem_coords[2][2]
                        + (1-k)*   j *(1-i)*elem_coords[3][2]
                        +    k *(1-j)*(1-i)*elem_coords[4][2]
                        +    k *(1-j)*   i *elem_coords[5][2]
                        +    k *   j *   i *elem_coords[6][2]
                        +    k *   j *(1-i)*elem_coords[7][2] )
               subelem_coords.append( [X_pt, Y_pt, Z_pt] )
    return subelem_coords

def hex8_ctr(elem_coords):
    """Compute the coordinates of the center of a hex8 element.

    Simple average in physical space.

    The result is the same as for hex8_subdiv with intervals=1.

    Input:
    elem_coords:  coordinates of element's nodes, assuming exodus
                    node order convention - (counter clockwise around
                    the element)
    """
    X_pt = 0.125*(   elem_coords[0][0]
                   + elem_coords[1][0]
                   + elem_coords[2][0]
                   + elem_coords[3][0]
                   + elem_coords[4][0]
                   + elem_coords[5][0]
                   + elem_coords[6][0]
                   + elem_coords[7][0] )
    Y_pt = 0.125*(   elem_coords[0][1]
                   + elem_coords[1][1]
                   + elem_coords[2][1]
                   + elem_coords[3][1]
                   + elem_coords[4][1]
                   + elem_coords[5][1]
                   + elem_coords[6][1]
                   + elem_coords[7][1] )
    Z_pt = 0.125*(   elem_coords[0][2]
                   + elem_coords[1][2]
                   + elem_coords[2][2]
                   + elem_coords[3][2]
                   + elem_coords[4][2]
                   + elem_coords[5][2]
                   + elem_coords[6][2]
                   + elem_coords[7][2] )
    return [ X_pt, Y_pt, Z_pt ]
```

NohExactIG.py

```
#!/usr/bin/python

import math

class NohExactIG:
  """Exact solution of Noh's problem for an ideal gas.

  The object is the solution defined by the inputs
  nd      = number of dimensions (1 = planar, 2 = cylindrical, 3 = spherical)
  gamma   = ratio of specific heats
  rho_inf = initial density  ( > 0 )
  u_inf   = initial velocity ( < 0 )
  which

  Functions are provided to return point values of rho, p, e,and u,
  given position (radius) and time.
  """


  def __init__(self, nd, gamma, rho_inf, u_inf):
    """Precompute as much as possible (without knowing r and t).

    Then hopefully functions which return field variables will be fast.
    """
    allowed_nd = [ 1, 2, 3]

    if nd not in allowed_nd:
      print 'nd = ', nd
      print 'Invalid number of dimensions!'
      print 'Need nd = 1, 2, or 3.'
    elif rho_inf <= 0:
      print 'rho_inf = ', rho_inf
      print 'Invalid initial density!'
      print 'Need rho_inf > 0.'
    elif u_inf > 0:
      print 'u_inf = ', u_inf
      print 'Invalid initial velocity!'
      print 'Need u_inf < 0.'
    else:
      self.nd = nd
      self.gamma  = gamma
      self.rho_inf = rho_inf
      self.u_inf = u_inf

    self.__gm1 = self.gamma - 1.0
    self.__gp1 = self.gamma + 1.0

    self.shock_speed = -0.5*self.u_inf*(self.__gm1)   # shock speed

    self.u_in = 0.0
    self.ei_in = 0.5*self.rho_inf*self.u_inf*self.u_inf
    self.rho_in = rho_inf*pow( self.__gp1/self.__gm1, self.nd )
    self.p_in = self.ei_in*pow( self.__gp1, nd)/pow( self.__gm1, nd - 1.0 )


  def shock_position(self, t ):
    return t*self.shock_speed

  def rho(self, r, t):
    """Density as a function of position (r) and time (t)
    """

    if r < self.shock_position(t):
      return self.rho_in
    else:
      return self.rho_inf*pow( (r - self.u_inf*t)/r , self.nd - 1)

  def p(self, r, t):
    """Pressure as a function of position (r) and time (t)
    """

    if r < self.shock_position(t):
      return self.p_in
    else:
      return 0.0

  def ei(self, r, t):
    """Internal Energy as a function of position (r) and time (t)
    """

    if r < self.shock_position(t):
      return self.ei_in
    else:
      return 0.0

  def ur(self, r, t):
    """Radial Velocity as a function of position (r) and time (t)
    """

    if r < self.shock_position(t):
      return self.u_in
    else:
      return self.u_inf

class NohExactIGCart( NohExactIG ):
  """Exact solution to Noh problem for an ideal gas,
  written for Cartesian coodinates.  Refer to NohExactIG
  for full documentation.

  Most of the functions are wrappers for equivalents in NohExactIG,
  but take the position as X = [x, y, z].  By default the radius
  is the distance from the origin, but the radius can be measured
  from a point XC = [xc, yc, zc] if this argument is passed during
  the initialization of the solution.

  Additionally, methods are provided for the Cartesian velocity components
  and the Cartesian velocity vector.
  """

  def __init__(self, nd, gamma, rho_inf, u_inf, XC = [0.0, 0.0, 0.0]):
    NohExactIG.__init__(self, nd, gamma, rho_inf, u_inf)
    self.xc = XC[0]
    self.yc = XC[1]
    self.zc = XC[2]

  def r(self, X):
    '''Radius is distance from XC = [xc, yc, zc].'''
    return math.sqrt(   (X[0]-self.xc)**2
                      + (X[1]-self.yc)**2
                      + (X[2]-self.zc)**2  )

  def rho(self, X, t):
    return NohExactIG.rho(self, self.r(X), t)

  def p(self, X, t):
    return NohExactIG.p(self, self.r(X), t)

  def ei(self, X, t):
    return NohExactIG.ei(self, self.r(X), t)

  def ur(self, X, t):
    return NohExactIG.ur(self, self.r(X), t)

  def ux(self, X, t):
    ux = 0.0
    r = self.r(X)
    if r > 0:
      ux = NohExactIG.ur(self, r, t)*(X[0]-self.xc)/r
    return ux

  def uy(self, X, t):
    uy = 0.0
    r = self.r(X)
    if r > 0:
      uy = NohExactIG.ur(self, r, t)*(X[1]-self.yc)/r
    return uy

  def uz(self, X, t):
    uz = 0.0
    r = self.r(X)
    if r > 0:
      uz = NohExactIG.ur(self, r, t)*(X[2]-self.zc)/r
    return uz

  def u(self, X, t):
    ux = 0.0
    uy = 0.0
    uz = 0.0
    r = self.r(X)
    if r > 0:
      ux = NohExactIG.ur(self, r, t)*(X[0]-self.xc)/r
      uy = NohExactIG.ur(self, r, t)*(X[1]-self.yc)/r
      uz = NohExactIG.ur(self, r, t)*(X[2]-self.zc)/r
    return [ux, uy, uz]

class NohExactIGCyl( NohExactIG ):
  """Exact solution to Noh problem for an ideal gas,
  written for 3d cylindrical coodinates.  Refer to NohExactIG
  for full documentation.

  Most of the functions are wrappers for equivalents in NohExactIG,
  but take the position as X = [rcyl, zcyl, tcyl]. Only rcyl and zcyl
  are used; planar, cylindrical and spherical Noh solutions are
  independent of theta.

  If the planar problem is specified (nd=1) the Noh coordinate r
  is aligned with the zcyl direction, r = abs (zcyl).

  If the cylindrical problem is specified (nd=2) the Noh
  coordinate r corresponds to the rcyl coordinate, r = rcyl.

  If the spherical problem is specified (nd=3) the Noh coordinate r
  corresponds to the spherical radius, r = sqrt( rcyl^2 + zcyl^2 ).

  Additionally, methods are provided for the Cylindrical velocity
  components and the Cylindrical velocity vector.  The methods for
  the components are urcyl, uzcyl, and utcyl, and the vector method
  is u (= urcul, uzcul, utcyl).  Note that the Noh velocity is
  provided by ur.
  """

  def __init__(self, nd, gamma, rho_inf, u_inf):
    NohExactIG.__init__(self, nd, gamma, rho_inf, u_inf)

  def r(self, X):
    """Radius as a function of X = [r, z, theta].

    For spherical Noh problem, compute spherical radius
    from cylindrical radius and z.

    For cylindrical Noh problem, just return r.

    For planar Noh problem, just return |z|.
    """
    if self.nd == 3:
      return math.sqrt(X[0]**2 + X[1]**2)
    elif self.nd ==2:
      return X[0]
    elif self.nd == 1:
      return abs(X[1])

  def rho(self, X, t):
    return NohExactIG.rho(self, self.r(X), t)

  def p(self, X, t):
    return NohExactIG.p(self, self.r(X), t)

  def ei(self, X, t):
    return NohExactIG.ei(self, self.r(X), t)

# For some velocities, need to consider nd before computing.
  def ur(self, X, t):
    return NohExactIG.ur(self, self.r(X), t)

  def urcyl(self, X, t):
    ur = 0.0
    if self.nd == 2:
      ur = NohExactIG.ur(self, X[0], t)
    elif self.nd == 3:
      r = self.r(X)
      if r > 0:
        ur = NohExactIG.ur(self, r, t)*X[0]/r
    return ur

  def uzcyl(self, X, t):
    uz = 0.0
    if self.nd == 1:
      uz = NohExactIG.ur(self, abs(X[1]), t)
    elif self.nd == 3:
      r = self.r(X)
      if r > 0:
        uz = NohExactIG.ur(self, r, t)*X[1]/r
    return uz

  def utcyl(self, X, t):
    return 0.0

# Faster (?) than just
#   return[urcyl(self, X, t), uzcyl(self, X, t), utcyl(self, X, t)]
  def u(self, X, t):
    ur = 0.0
    uz = 0.0
    if self.nd == 1:
      uz = NohExactIG.ur(self, abs(X[1]), t)
    elif self.nd == 2:
      ur = NohExactIG.ur(self, X[0], t)
    elif self.nd == 3:
      r = self.r(X)
      if r > 0:
        ur = NohExactIG.ur(self, r, t)*X[0]/r
        uz = NohExactIG.ur(self, r, t)*X[1]/r
    return [ur, uz, 0.0]
```

pvanalysis.py

```
import paraview
import sys
import os.path
import socket
import time

# Assume that if there is already an active connection,
# then we are in the GUI Client; otherwise we are being
# imported in a python script. This just gives a way
# for that python script to figure that out.
if (not paraview.ActiveConnection):
        # False
        GUIClient = 0
else:
        # True
        GUIClient = 1


def OpenConnection(server="builtin",procs=1,minutes=0,account=""):
        # Opens a connection to the specified server.
        # returns 0 if connection was unsuccessful, 1 otherwise


        # If using the builtin server, everything is simple:
        if (server == "builtin"):
                if (not paraview.ActiveConnection):
                        paraview.ActiveConnection = paraview.Connect()
                return 0


def OpenExodusFile(file):
        # Do some path stuff
        #  file can be the file with absolute path
        #              the file with relative path
        #              the file alone, with cwd as the implied path
        # For now require the absolute path in the filename.
        (dir,FileName) = os.path.split(file)

        # Works if absolute filename is given
        absFileName = file

        reader = paraview.CreateProxy("sources", "ExodusReader")
        reader.GetProperty("FileName").SetElement(0,absFileName)
        reader.UpdateVTKObjects()
        paraview.RegisterProxy(reader, FileName)

        return reader

def GetPointVariables(reader):
        """Returns the set of available point-centered variables."""

        reader.UpdatePipeline()
        reader.UpdatePropertyInformation()

        results = []
        variable_list = reader.GetProperty("PointArrayStatus").GetDomain("array_list")
        for i in range(variable_list.GetNumberOfStrings()):
                results.append(variable_list.GetString(i))
        return results;

def GetCellVariables(reader):
        """Returns the set of available cell-centered variables."""

        reader.UpdateVTKObjects()
        reader.UpdatePipeline()
        reader.UpdatePropertyInformation()

        results = []
        variable_list = reader.GetProperty("CellArrayStatus").GetDomain("array_list")
        for i in range(variable_list.GetNumberOfStrings()):
                results.append(variable_list.GetString(i))
        return results;

def EnablePointVariables(reader, enabled_variables):
        reader.UpdatePipeline()
        reader.UpdatePropertyInformation()

        variable_status = reader.GetProperty("PointArrayStatus")
        variable_list = variable_status.GetDomain("array_list")
        for i in range(variable_list.GetNumberOfStrings()):
                variable = variable_list.GetString(i)
                variable_status.SetElement(i * 2, variable)
                if variable in enabled_variables:
                        variable_status.SetElement(i * 2 + 1, "1")
                else:
                        variable_status.SetElement(i * 2 + 1, "0")

        reader.UpdateVTKObjects()

def EnableCellVariables(reader, enabled_variables):
        reader.UpdatePipeline()
        reader.UpdatePropertyInformation()

        variable_status = reader.GetProperty("CellArrayStatus")
        variable_list = variable_status.GetDomain("array_list")
        for i in range(variable_list.GetNumberOfStrings()):
                variable = variable_list.GetString(i)
                variable_status.SetElement(i * 2, variable)
                if variable in enabled_variables:
                        variable_status.SetElement(i * 2 + 1, "1")
                else:
                        variable_status.SetElement(i * 2 + 1, "0")

        reader.UpdateVTKObjects()


def GetPipelineTime(data):
        return data.GetInformation().Get(data.DATA_TIME_STEPS(), 0)

class PointAdapter:
        def __init__(self, data):
                self.data = data

        def __len__(self):
                return self.data.GetNumberOfPoints()

        def __getitem__(self, key):
                if key >= self.data.GetNumberOfPoints():
                        raise IndexError

                return self.data.GetPoint(key)

        def __setitem__(self, key, value):
                if key >= self.data.GetNumberOfPoints():
                        raise IndexError

                self.data.SetPoint(key, value)

class PointVariable1Adapter:
        def __init__(self, data, array):
                self.array = data.GetPointData().GetArray(array)
                assert self.array
                assert self.array.GetNumberOfComponents() == 1

        def __len__(self):
                return self.array.GetNumberOfTuples()

        def __getitem__(self, key):
                if key >= self.array.GetNumberOfTuples():
                        raise IndexError

                return self.array.GetTuple1(key)

        def __setitem__(self, key, value):
                if key >= self.array.GetNumberOfTuples():
                        raise IndexError

                self.array.SetValue(key, value);

class PointVariable3Adapter:
        def __init__(self, data, array):
                self.array = data.GetPointData().GetArray(array)
                assert self.array
                assert self.array.GetNumberOfComponents() == 3

        def __len__(self):
                return self.array.GetNumberOfTuples()

        def __getitem__(self, key):
                if key >= self.array.GetNumberOfTuples():
                        raise IndexError

                return self.array.GetTuple3(key)

        def __setitem__(self, key, value):
                if key >= self.array.GetNumberOfTuples():
                        raise IndexError

                self.array.SetValue(key, value);

class CellVariable1Adapter:
        def __init__(self, data, array):
                self.array = data.GetCellData().GetArray(array)
                assert self.array
                assert self.array.GetNumberOfComponents() == 1

        def __len__(self):
                return self.array.GetNumberOfTuples()

        def __getitem__(self, key):
                if key >= self.array.GetNumberOfTuples():
                        raise IndexError

                return self.array.GetTuple1(key)

        def __setitem__(self, key, value):
                if key >= self.array.GetNumberOfTuples():
                        raise IndexError

                self.array.SetValue(key, value);

class CellVariable3Adapter:
        def __init__(self, data, array):
                self.array = data.GetCellData().GetArray(array)
                assert self.array
                assert self.array.GetNumberOfComponents() == 3

        def __len__(self):
                return self.array.GetNumberOfTuples()

        def __getitem__(self, key):
                if key >= self.array.GetNumberOfTuples():
                        raise IndexError

                return self.array.GetTuple3(key)

        def __setitem__(self, key, value):
                if key >= self.array.GetNumberOfTuples():
                        raise IndexError

                self.array.SetValue(key, value);

class PointCalculationAdapter:
        def __init__(self, data, functor):
                self.data = data
                self.functor = functor

        def __len__(self):
                return self.data.GetNumberOfPoints()

        def __getitem__(self, key):
                if key >= self.data.GetNumberOfPoints():
                        raise IndexError

                return self.functor(self.data.GetPoint(key))

class PointTimeCalculationAdapter:
        def __init__(self, data, functor):
                self.data = data
                self.functor = functor
                self.time = GetPipelineTime(data)

        def __len__(self):
                return self.data.GetNumberOfPoints()

        def __getitem__(self, key):
                if key >= self.data.GetNumberOfPoints():
                        raise IndexError

                return self.functor(self.data.GetPoint(key), self.time)


def AddPointFloatVariable1(data, name):
        array = paraview.vtkFloatArray()
        array.SetNumberOfComponents(1)
        array.SetNumberOfTuples(data.GetNumberOfPoints())
        array.SetName(name)
        data.GetPointData().AddArray(array)

        return PointVariable1Adapter(data, name)

def AddCellFloatVariable1(data, name):
        array = paraview.vtkFloatArray()
        array.SetNumberOfComponents(1)
        array.SetNumberOfTuples(data.GetNumberOfCells())
        array.SetName(name)
        data.GetCellData().AddArray(array)

        return CellVariable1Adapter(data, name)
```

script_library.py

```
###

cell_field_function_script = """
import sys
pv_dir = '/usr/local/paraview3/bin'
this_dir = '/Users/vgweirs/projects/PV_analysis'
sys.path.append(pv_dir)
sys.path.append(this_dir)

# inputs:
# class_module        this will be imported so we can instantiate the reference
#                        solution
# instantiator        this instantiates the reference solution
# method_list         the name of the method to invoke and an associated
#                        paraview variable name
# subdiv              the number of subdivisions (for computing cell averages)

import paraview
import pvanalysis
import math
import TampaPVTools

# Connect the filter to its input and output
input = self.GetUnstructuredGridInput()
numCells = input.GetNumberOfCells()

output = self.GetUnstructuredGridOutput()
output.ShallowCopy(input)

test_time = pvanalysis.GetPipelineTime(input)

# Import the module with the exact solution
exec ('import ' + class_module )

# Instantiate (initialize) the exact solution
exact = eval( instantiator )

if (n_subel > 1):                 # Subdivide the element to get average value

  for pair in method_list:

    exact_func = getattr(exact, pair[0] )

    exact_field = paraview.vtkFloatArray()
    exact_field.SetName( pair[1] )

    for i in range(0,numCells):

      cell = input.GetCell(i)
      cell_type_index = cell.GetCellType()
      cell_type = paraview.vtkCellTypes.GetClassNameFromTypeId( cell_type_index )

      etype_info = TampaPVTools.GetEtypeInfo( cell_type )
      subdiv = etype_info['subdiv_func']

      elem_coords = []
      for j in range(0,cell.GetNumberOfPoints()):
        elem_coords.append(cell.GetPoints().GetPoint(j))

      subel_coords = subdiv(elem_coords, n_subel)

      subel_exact = map((lambda x:exact_func(x,test_time)), subel_coords)

      elem_avg = sum( subel_exact) / len( subel_coords )

      exact_field.InsertNextValue( elem_avg )

    # Add exact_field array to output
    output.GetCellData().AddArray(exact_field)

else:                    #  Get the value at the center of the element

  for pair in method_list:

    exact_func = getattr(exact, pair[0] )

    exact_field = paraview.vtkFloatArray()
    exact_field.SetName( pair[1] )

    for i in range(0,numCells):

      cell = input.GetCell(i)
      cell_type_index = cell.GetCellType()
      cell_type = paraview.vtkCellTypes.GetClassNameFromTypeId( cell_type_index )

      etype_info = TampaPVTools.GetEtypeInfo( cell_type )
      ctr = etype_info['ctr_func']

      elem_coords = []
      for j in range(0,cell.GetNumberOfPoints()):
        elem_coords.append(cell.GetPoints().GetPoint(j))

      x = ctr( elem_coords )

      elem_val = exact_func(x, test_time)

      exact_field.InsertNextValue( elem_val )


    # Add exact_field array to output
    output.GetCellData().AddArray(exact_field)

"""


volume_field_script_native = """
import sys
pv_dir = '/usr/local/paraview3/bin'
this_dir = '/Users/vgweirs/projects/PV_analysis'
sys.path.append(pv_dir)
sys.path.append(this_dir)

#import pvanalysis
#import paraview
#import math
import TampaPVTools
import ElemData

cell_volume = paraview.vtkFloatArray()
cell_volume.SetName("Volume")

input = self.GetUnstructuredGridInput()
numCells = input.GetNumberOfCells()

for i in range(0,numCells):

  cell = input.GetCell(i)
  cellVol = paraview.IntegrateCell(input, i)
  cell_volume.InsertNextValue( cellVol )

output = self.GetUnstructuredGridOutput()
output.ShallowCopy(input)
output.GetCellData().AddArray(cell_volume)
"""

volume_field_script = """
import sys
pv_dir = '/usr/local/paraview3/bin'
this_dir = '/Users/vgweirs/projects/PV_analysis'
sys.path.append(pv_dir)
sys.path.append(this_dir)

#import pvanalysis
#import paraview
#import math
import TampaPVTools
import ElemData

cell_volume = paraview.vtkFloatArray()
cell_volume.SetName("Volume")

input = self.GetUnstructuredGridInput()
numCells = input.GetNumberOfCells()

for i in range(0,numCells):

  cell = input.GetCell(i)
  cell_type_index = cell.GetCellType()
  cell_type = paraview.vtkCellTypes.GetClassNameFromTypeId( cell_type_index )

  etype_info = TampaPVTools.GetEtypeInfo( cell_type )

  elem_coords = []
  for j in range(0,cell.GetNumberOfPoints()):
    elem_coords.append(cell.GetPoints().GetPoint(j))

  cell_volume.InsertNextValue( etype_info['vol_func'](elem_coords) )

output = self.GetUnstructuredGridOutput()
output.ShallowCopy(input)
output.GetCellData().AddArray(cell_volume)
"""


elem_length_field_script = """
import sys
pv_dir = '/usr/local/paraview3/bin'
this_dir = '/Users/vgweirs/projects/PV_analysis'
sys.path.append(pv_dir)
sys.path.append(this_dir)

#import paraview
#import pvanalysis
import math

characteristic_length = paraview.vtkFloatArray()
characteristic_length.SetName('Characteristic Length')

input = self.GetUnstructuredGridInput()
numCells = input.GetNumberOfCells()
# Error check if Volume is not found?
cell_volume = input.GetCellData().GetArray('Volume')

for i in range(0,numCells):
   cell_dim = input.GetCell(i).GetCellDimension()
   characteristic_length.InsertNextValue( math.pow(cell_volume.GetValue(i),
                                          1.0/cell_dim ) )

output = self.GetUnstructuredGridOutput()
```

```
output.ShallowCopy(input)
output.GetCellData().AddArray(characteristic_length)
"""



point_function_script = """
import sys
pv_dir = '/usr/local/paraview3/bin'
this_dir = '/Users/vgweirs/projects/PV_analysis'
sys.path.append(pv_dir)
sys.path.append(this_dir)

# inputs
# class_module
# instantiator
# method_list         the name of the method to invoke and an associated paraview variable name
# subdiv              the number of subdivisions (for computing cell averages)

print 'Hello World from script_library!'

print class_module
print instantiator
print method_list
print n_subel

import paraview
import pvanalysis
import math
import TampaPVTools


# Connect the filter to its input and output
input = self.GetUnstructuredGridInput()
numCells = input.GetNumberOfCells()

output = self.GetUnstructuredGridOutput()
output.ShallowCopy(input)

test_time = pvanalysis.GetPipelineTime(input)

# Import the module with the exact solution
exec ('import ' + class_module )

# Instantiate (initialize) the exact solution
exact = eval( instantiator )

if (n_subel > 1):                 # Subdivide the element to get average value

  for tuple in method_list:

    exact_func = getattr(exact, tuple[0] )

    exact_field = paraview.vtkFloatArray()
    exact_field.SetName( tuple[1] )

    for i in range(0,numCells):

      cell = input.GetCell(i)
      cell_type_index = cell.GetCellType()
      cell_type = paraview.vtkCellTypes.GetClassNameFromTypeId( cell_type_index )

      etype_info = TampaPVTools.GetEtypeInfo( cell_type )
      subdiv = etype_info['subdiv_func']

      elem_coords = []
      for j in range(0,cell.GetNumberOfPoints()):
        elem_coords.append(cell.GetPoints().GetPoint(j))

      subel_coords = subdiv(elem_coords, n_subel)
      subel_exact = map((lambda x:exact_func(x,test_time)), subel_coords)

      elem_avg = sum( subel_exact) / len( subel_coords )

      exact_field.InsertNextValue( elem_avg )

    # Add exact_field array to output
    output.GetCellData().AddArray(exact_field)

else:                    #  Get the value at the center of the element

  for tuple in method_list:

    exact_func = getattr(exact, tuple[0] )

    exact_field = paraview.vtkFloatArray()
    exact_field.SetName( tuple[1] )

    for i in range(0,numCells):

      cell = input.GetCell(i)
      cell_type_index = cell.GetCellType()
      cell_type = paraview.vtkCellTypes.GetClassNameFromTypeId( cell_type_index )

      etype_info = TampaPVTools.GetEtypeInfo( cell_type )
      ctr = etype_info['ctr_func']

      elem_coords = []
      for j in range(0,cell.GetNumberOfPoints()):
        elem_coords.append(cell.GetPoints().GetPoint(j))

      x = ctr( elem_coords )

      elem_val = exact_func(x, test_time)

      exact_field.InsertNextValue( elem_val )


    # Add exact_field array to output
    output.GetCellData().AddArray(exact_field)

"""


NohExact_script = """
import sys
pv_dir = '/usr/local/paraview3/bin'
this_dir = '/Users/vgweirs/projects/PV_analysis'
sys.path.append(pv_dir)
sys.path.append(this_dir)


import paraview
import pvanalysis
import math
import TampaPVTools
import NohExactIG

print ' ->NohExact_script from proto_data.py <-'
exact = NohExactIG.NohExactIGCart(2, 5.0/3.0, 1.0, -1.0)
exact_func = getattr(exact, 'rho' )
n_subel = 5

exact_density = paraview.vtkFloatArray()
exact_density.SetName("Reference Density")

input = self.GetUnstructuredGridInput()

numCells = input.GetNumberOfCells()

test_time = pvanalysis.GetPipelineTime(input)
```

```
for i in range(0,numCells):

  cell = input.GetCell(i)
  cell_type_index = cell.GetCellType()
  cell_type = paraview.vtkCellTypes.GetClassNameFromTypeId( cell_type_index )

  etype_info = TampaPVTools.GetEtypeInfo( cell_type )
  subdiv = etype_info['subdiv_func']

  elem_coords = []
  for j in range(0,cell.GetNumberOfPoints()):
    elem_coords.append(cell.GetPoints().GetPoint(j))

  subel_coords = subdiv(elem_coords, n_subel)

  subel_exact = map((lambda x:exact_func(x,test_time)), subel_coords)

  elem_avg = sum( subel_exact) / len( subel_coords )

  exact_density.InsertNextValue( elem_avg )

output = self.GetUnstructuredGridOutput()
output.ShallowCopy(input)
output.GetCellData().AddArray(exact_density)
"""
```

TampaPVTools.py

```
import paraview
import pvanalysis
import ElemData
from script_library import *

def LoadVariables(reader, var_list = 'None'):
    """This is mostly a wrapper to functions in pvanalysis.py.

    It adds handling of an empty var_list (get all variables)
    and a string var_list (treat the string as the name of the variable.)

    If given a list of variables, report if any are not found.
    """
    reader.UpdatePipeline()
    reader.UpdatePropertyInformation()

    # Point and cell variables separately
    pArray_list = pvanalysis.GetPointVariables(reader)
    cArray_list = pvanalysis.GetCellVariables(reader)

    if type(var_list)== type(" "):
        # By default read all the variables
        if var_list == "None":
            pvanalysis.EnablePointVariables(reader, pArray_list)
            pvanalysis.EnableCellVariables(reader, cArray_list)
        # If a string is input, assume it is a (single) variable name
        # Make it a list and handle it below.
        else:
            var_name = var_list
            var_list = [ ]
            var_list.append(var_name)

    # If we get a list, turn on the variables one by one.
    if type(var_list) == type([ ]):
        pvanalysis.EnablePointVariables(reader, var_list)
        pvanalysis.EnableCellVariables(reader, var_list)
        # Error checking part
        var_list_found = []
        for i in pArray_list:
            if i in var_list:
                var_list_found.append(i)
        for i in cArray_list:
            if i in var_list:
                var_list_found.append(i)
        if len(var_list_found) < len(var_list):
            for var in var_list:
                if var not in var_list_found:
                    print "  Variable " + var + " not found in dataset!"
    else:
        print "  var_list is not an expected type!"

    reader.UpdateVTKObjects()

################################################################################

def ProgrammableFilter(input_filter,
                       pf_script,
                       parameter_dictionary = 'None',
                       output_filter_name='Programmable Filter'):
    """This creates a programmable filter and connects it to the pipeline.

    The filter takes as inputs:
    input_filter          The filter this one will follow
    pf_script             The script the programmable filter is to run;
                            it is a string
    parameter_dictionary  The dictionary containing the names and values
                            of the parameters in pf_script
    output_filter_name    The name used to refer to this filter.
    """

    # Hook up input
    filter = paraview.CreateProxy("filters", "Programmable Filter")
    filter.GetProperty("Script").SetElement(0, pf_script)
    filter.GetProperty("Input").AddProxy(input_filter.SMProxy, 0)

    # Handle parameters to be set in script, if any
    if (not parameter_dictionary == 'None'):
        # Make sure we get a dictionary that holds the parameters
        if type(parameter_dictionary) == type({}):
            # Define a quote padding function for strings
            def qpad( unpadded_string):
                return '\"' + unpadded_string + '\"'
            # Loop over parameter name, value pairs
            i=0
            for key in parameter_dictionary.keys():
                # Set parameter name
                filter.GetProperty('Parameters').SetElement(i, key)
                # Set parameter value, checking for type
                if type(parameter_dictionary[key]) == type(''):
                    filter.GetProperty('Parameters').SetElement(i+1, qpad(
parameter_dictionary[key] ) )
                else:
                    filter.GetProperty('Parameters').SetElement(i+1, str(
parameter_dictionary[key] ) )
                i = i+2

    # Hook up output
    filter.UpdateVTKObjects()
    paraview.RegisterProxy(filter, output_filter_name)
    filter.UpdatePipeline()

    return filter

def WeightedCellDifferenceFilter(input_filter1, cell_var1,
                       input_filter2, cell_var2,
                       input_filter3, cell_weight_var,
                       difference_var_name = 'Weighted Difference Variable',
                       output_filter_name='Weighted Difference Filter'):
    """This takes a cell variable from filter 2 and subtracts it from
    a cell variable from filter 1, takes the absolute value, then
    multiplies by a cell variable from filter 3.

    output_filter_name  the name used to refer to this filter.
    """

    # Establish the programmable filter with three inputs
    filter = paraview.CreateProxy("filters", "Programmable Filter")
    filter.GetProperty("Input").AddProxy(input_filter1.SMProxy, 0)
    filter.GetProperty("Input").AddProxy(input_filter2.SMProxy, 1)
    filter.GetProperty("Input").AddProxy(input_filter3.SMProxy, 2)

    filter.GetProperty("Script").SetElement(0, weighted_cell_difference_script)

    # Define a quote padding function for strings
    def qpad( unpadded_string):
        return '\"' + unpadded_string + '\"'

    # Now set the parameter values for the weighted_cell_difference_script
    filter.SetParameters("variable1", qpad( cell_var1 ),
                         "variable2", qpad( cell_var2 ),
                         "weight", qpad( cell_weight_var ),
                         "diff_var_name", qpad( difference_var_name ) )

    # Hook up output and update.
    filter.UpdateVTKObjects()
    paraview.RegisterProxy(filter, output_filter_name)
    filter.UpdatePipeline()

    return filter

def PointDifferenceFilter(input_filter1, point_var1,
                          input_filter2, point_var2,
                          difference_var_name = 'Difference Variable',
                          output_filter_name='Difference Filter'):
    """This takes a cell variable from filter 2 and subtracts it from
    a cell variable from filter 1, then takes the absolute value

    output_filter_name  the name used to refer to this filter.
    """

    # Establish the programmable filter with two inputs
    filter = paraview.CreateProxy("filters", "Programmable Filter")
    filter.GetProperty("Input").AddProxy(input_filter1.SMProxy, 0)
    filter.GetProperty("Input").AddProxy(input_filter2.SMProxy, 1)

    filter.GetProperty("Script").SetElement(0, point_difference_script)

    # Define a quote padding function for strings
    def qpad( unpadded_string):
        return '\"' + unpadded_string + '\"'

    # Now set the parameter values for point_difference_script
    filter.SetParameters("variable1", qpad( point_var1 ),
                         "variable2", qpad( point_var2 ),
                         "diff_var_name", qpad( difference_var_name ) )

    # Hook up output and update.
    filter.UpdateVTKObjects()
    paraview.RegisterProxy(filter, output_filter_name)
    filter.UpdatePipeline()

    return filter

def TimeSelectionFilter(input_filter,
                        time,
                        output_filter_name):
    """Select a solution time for the pipeline; essentially
    picks one timestep of data to apply the pipeline commands to.

    Note that this filter uses a suppressor, and as such should
    be at the downstream end of the pipeline commands.
    Also, it is disabled if the GUIClient is detected.
    """
    if (not pvanalysis.GUIClient):
        filter = paraview.CreateProxy("filters", "UpdateSuppressor")
        filter.GetProperty("UpdateTime").SetElement(0, time)
        filter.GetProperty("Input").AddProxy(input_filter.SMProxy, 0)
        filter.UpdateVTKObjects()
        filter.GetProperty("ForceUpdate").SetImmediateUpdate(1)
        paraview.RegisterProxy(filter, output_filter_name)
        return filter
    else:
        return input_filter


################################################################################

def FetchFilter(input_filter, operation, variable):
    """
    input_filter has the variable field
    operation is the reduction operation, one of 'min', 'max', or 'sum'
    variable is the (string) name of the variable to which the reduction
      operation should be applied
    """

    if type(operation) == type(''):
        red_op = operation.upper()
        print red_op
        if (not (   (red_op == 'MIN')
                 or (red_op == 'MAX')
                 or (red_op == 'SUM') )):
            print operation, " is not a known reduction operation!"
    else:
        print operation, " is not a known reduction operation!"

    red = paraview.CreateProxy('filters', 'MinMax')
    red.SetOperation(red_op)
    red.UpdateVTKObjects()

    fetch_out = paraview.Fetch(input_filter,red)
    red_val=None
    red_val_array = fetch_out.GetCellData().GetArray( variable )
    if (red_val_array!=None):
        red_val = red_val_array.GetValue(0)
    else:
        red_val_array = fetch_out.GetPointData().GetArray( variable )
        if (red_val_array!=None):
            red_val = red_val_array.GetValue(0)
        else:
            print variable, " not found!"

    return red_val

def FetchMinFilter(input_filter, variable):
    """
    input_filter has the variable field
    variable is the (string) name of the variable that the min
      should be taken over
    """

    min = paraview.CreateProxy('filters', 'MinMax')
    min.SetOperation('MIN')
    min.UpdateVTKObjects()

    fetch_out = paraview.Fetch(input_filter,min)
    min_val=None
    min_val_array = fetch_out.GetCellData().GetArray( variable )
    if (min_val_array!=None):
        min_val = min_val_array.GetValue(0)
    else:
        min_val_array = fetch_out.GetPointData().GetArray( variable )
        if (min_val_array!=None):
            min_val = min_val_array.GetValue(0)

    return min_val

def FetchMaxFilter(input_filter, variable):
    """
    input_filter has the variable field
    variable is the (string) name of the variable that the max
      should be taken over
    """

    max = paraview.CreateProxy('filters', 'MinMax')
    max.SetOperation('MAX')
    max.UpdateVTKObjects()

    fetch_out = paraview.Fetch(input_filter,max)
    max_val=None
    max_val_array = fetch_out.GetCellData().GetArray( variable )
    if (max_val_array!=None):
        max_val = max_val_array.GetValue(0)
    else:
        max_val_array = fetch_out.GetPointData().GetArray( variable )
        if (max_val_array!=None):
            max_val = max_val_array.GetValue(0)

    return max_val

def FetchSumFilter(input_filter, variable):
    """
    input_filter has the variable field
    variable is the (string) name of the variable that the sum
      should be taken over
    """

    sum = paraview.CreateProxy('filters', 'MinMax')
    sum.SetOperation('SUM')
    sum.UpdateVTKObjects()

    fetch_out = paraview.Fetch(input_filter,sum)
    sum_val=None
    sum_val_array = fetch_out.GetCellData().GetArray( variable )
    if (sum_val_array!=None):
        sum_val = sum_val_array.GetValue(0)
    else:
        sum_val_array = fetch_out.GetPointData().GetArray( variable )
        if (sum_val_array!=None):
            sum_val = sum_val_array.GetValue(0)

    return sum_val


################################################################################

def GetEtypeInfo(vtkCellType):
    """Determine the element type and return the appropriate
    function to compute the volume.

    """

# A dictionary providing the data for each element type
    supported_elements = {'vtkQuad':
                            {'ndim': ElemData.quad4_ndim,
                             'vol_func': ElemData.quad4_vol,
                             'subdiv_func': ElemData.quad4_subdiv,
                             'ctr_func': ElemData.quad4_ctr},
                          'vtkHexahedron':
                            {'ndim': ElemData.hex8_ndim,
                             'vol_func': ElemData.hex8_vol,
                             'subdiv_func': ElemData.hex8_subdiv,
                             'ctr_func': ElemData.hex8_ctr}  }
    etype_info = supported_elements.get(vtkCellType, 'fail')

    if etype_info == 'fail':
        print 'Element Data not found for element type ', vtkCellType

    return etype_info

################################################################################
# The following scripts are viewed as internal to TampaPVTools; they are
# 'hardcoded' into particular programmable filters.
# Other scripts which are generally useful and are often passed into generic
# programmable filters are in script_library.py, and TampaPVTools is aware
# of them (can import them.) Examples: volume_field_script,
# element_length_field_script, cell_field_function_script
# User specific scripts should not be added here or in the script_library,
# but should be at the level of the script that calls TampaPVTools functions.
# Examples: NohExact_script

weighted_cell_difference_script = """
input1 = self.GetInputDataObject(0,0)
input2 = self.GetInputDataObject(0,1)
input3 = self.GetInputDataObject(0,2)

# Need better error checking to determine if we can really
# subtract componentwise...
numCells1 = input1.GetNumberOfCells()
numCells2 = input2.GetNumberOfCells()
numCells3 = input3.GetNumberOfCells()

if ( (not numCells1 == numCells2) or
     (not numCells2 == numCells3)    ):
  print "Warning! The number of cells is different in each input!"
var1 = input1.GetCellData().GetArray( variable1 )
var2 = input2.GetCellData().GetArray( variable2 )
wvar = input3.GetCellData().GetArray( weight )

DiffVar = paraview.vtkFloatArray()
DiffVar.SetName( diff_var_name  )

for i in range(0, numCells1):
  DiffVar.InsertNextValue( wvar.GetValue(i)*
                           abs( var1.GetValue(i) - var2.GetValue(i) ) )

output = self.GetOutput()
output.GetCellData().AddArray(DiffVar)
"""

point_difference_script = """
input1 = self.GetInputDataObject(0,0)
input2 = self.GetInputDataObject(0,1)

# Need better error checking to determine if we can really
# subtract componentwise...
numCells1 = input1.GetNumberOfCells()
numCells2 = input2.GetNumberOfCells()

if ( not numCells1 == numCells2 ):
  print "Warning! The number of cells is different in each input!"
var1 = input1.GetCellData().GetArray( variable1 )
var2 = input2.GetCellData().GetArray( variable2 )

DiffVar = paraview.vtkFloatArray()
DiffVar.SetName( diff_var_name )

for i in range(0, numCells1):
  DiffVar.InsertNextValue(  abs( var1.GetValue(i) - var2.GetValue(i) ) )

output = self.GetOutput()
output.GetCellData().AddArray(DiffVar)
"""
```

DISTRIBUTION

1	MS0104	T. C. Bickel	01200
1	MS0139	P. Yarrington	01220
1	MS0139	M. Pilch	01221
1	MS0321	J. S. Peery	01400
1	MS0370	T. G. Trucano	01411
1	MS0378	R. M. Summers	01431
1	MS0378	V. G. Weirs	01431
1	MS0384	A. C. Ratzel	01500
1	MS0801	R. W. Leland	09300
1	MS0807	J. P. Noe	09328
1	MS0822	E. A. Chavez	09326
1	MS0822	D. B. Karelitz	09326
1	MS0822	C. J. Pavlakos	09326
1	MS0823	J. D. Zepper	09320
1	MS0828	A. A. Giunta	01544
1	MS1181	T. A. Melhorn	01640
1	MS1186	T. A. Brunner	01641
1	MS1323	K. D. Moreland	01424
1	MS1323	D. Rogers	01424
1	MS1323	T. M. Shead	01424
1	MS9018	Central Technical Files	08944 (electronic copy)
1	MS0899	Technical Library	09536 (electronic copy)

Sandia National Laboratories