

Scientific visualization with ParaView

Part 1

Alex Razoumov
alex.razoumov@westdri.ca



**Digital Research
Alliance** of Canada

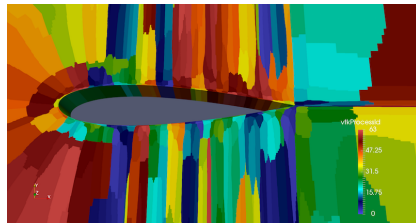


SIMON FRASER
UNIVERSITY

- ✓ slides, data, codes at <https://bit.ly/paraviewzipp>
 - ▶ the link will download a file `paraview.zip` (~36MB)
 - ▶ unpack it to find `codes/`, `data/` and `slides{1,2}.pdf`
 - ▶ command line: `wget https://bit.ly/paraviewzipp -O paraview.zip`
- ✓ install ParaView 5.11.x on your laptop from
<http://www.paraview.org/download>

Workshop outline

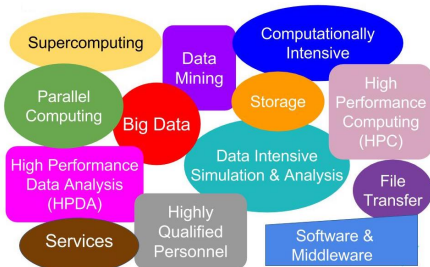
- Introduction to sci-vis: general ideas, tools, plotting vs. multi-dimensional vis.
- Overview of current general-purpose multi-dimensional sci-vis tools
- ParaView's architecture
- Importing data into ParaView: raw binary, VTK data types, NetCDF/HDF5, OpenFOAM
- Basic workflows: filters, creating a pipeline, working with vector fields



-
- Scripting: ways to run scripts, few simple scripts, trace tool, programmable filter/source, camera animations
 - Animation: three approaches, camera animation, one big exercise on scripting/animation, single timeline with many properties
 - Remote visualization
 - ▶ running ParaView in **client-server** mode
 - ▶ opening and load balancing of very large, multi-GB datasets
 - ▶ recommendations on running on cluster GPUs vs. multiple CPUs
 - ▶ writing, debugging, running PV Python scripts as **batch rendering jobs**
 - Not covered today: ParaView Cinema, in-situ visualization with Catalyst

Who are we?

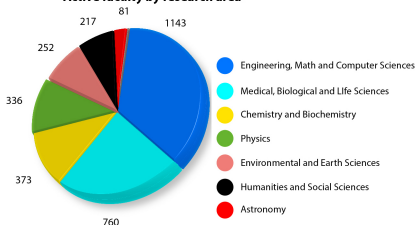
<https://docs.ALLIANCECAN.ca>



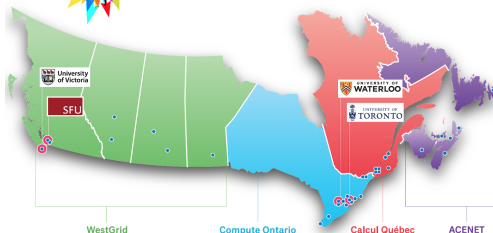
We provide Advanced Research Computing (ARC) infrastructure and services, a.k.a. everything beyond a standard desktop, at *no cost* to researchers

- SUPERCOMPUTERS / HPC
- cloud computing
- data storage / management / transfer
- support, training, visualization

Active faculty by research area



computecanada



Scientific visualization

Usually of spatially defined data

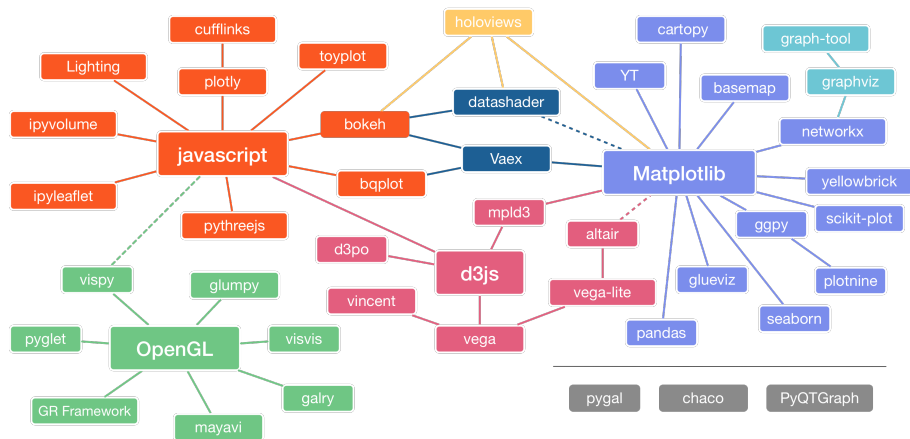
- **Sci-vis is the process of mapping scientific data to VISUAL FORM**
 - ▶ much easier to understand images than a large set of numbers
 - ▶ for interactive data exploration, debugging, communication with peers
- **National visualization team** <https://ccvis.netlify.app>
- *Visualize This* / IEEE SciVis contests (since 2016)
<https://ccvis.netlify.app/contests>

FIELD	VISUALIZATION TYPE
computational fluid dynamics	2D/3D flows, density, temperature, tracers
climate, meteorology, oceanography	fluid dynamics, clouds, chemistry, etc.
astrophysics	2D/3D fluids, particle data, ≤ 6 D radiation field, magnetic fields, gravitational fields
quantum chemistry	wave functions
molecular dynamics (phys, chem, bio)	particle/molecular data
medical imaging	MRI, CT scans, ultrasound
geographic information systems	elevation, rivers, towns, roads, layers, etc.
bioinformatics	networks, trees, sequences
humanities, social sciences, info-vis	abstract data, or any of the above

1D/2D plotting vs. multi-dimensional visualization

- **1D/2D plotting:** plotting functions of one variable, 1D tabulated data
 - ▶ something as simple as gnuplot or pgplot
 - ▶ highly recommend: Python's Matplotlib, Plotly, Bokeh libraries
 - ▶ another excellent option: R's ggplot2 library
- **2D/3D visualization**
 - ▶ displaying multi-dimensional datasets, typically data on structured (uniform and multi-resolution) or unstructured grids (that have some topology in 2D/3D)
 - ▶ rendering often CPU- and/or GPU-intensive
- Whatever you do, may be a good idea to avoid proprietary tools, unless those tools provide a clear advantage (most likely not)
 - ▶ large \$\$
 - ▶ limitations on where you can run them, which machines/platforms, etc.
 - ▶ cannot get help from open-source community, user base usually smaller than for open-source tools
 - ▶ once you start accumulating scripts, you lock yourself into using these tools for a long time, and consequently paying \$\$ on a regular basis

Python visualization landscape broken by renderer

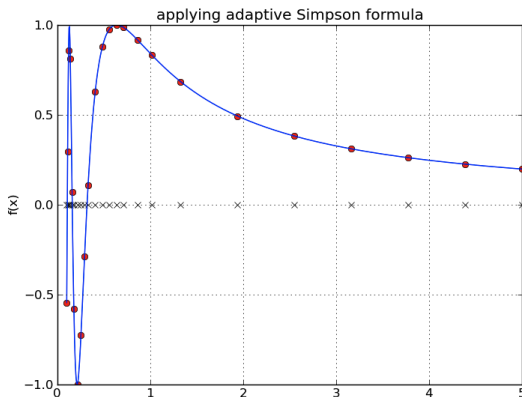


<https://github.com/rougier/python-visualization-landscape> by Nicolas Rougier

This figure is heavily influenced by 1D/2D plotting ... shows only a handful of 3D tools

Matplotlib example: 1D plotting

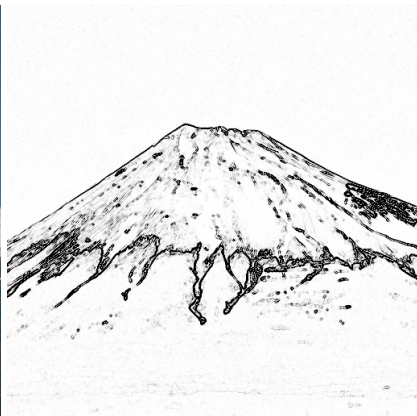
Adaptive Simpson integration



- Simple Python function `simpsonAdaptive(function, a, b, tolerance)` handles both calculation and plotting (~40 lines of code)
- Code in `codes/adaptive.py`

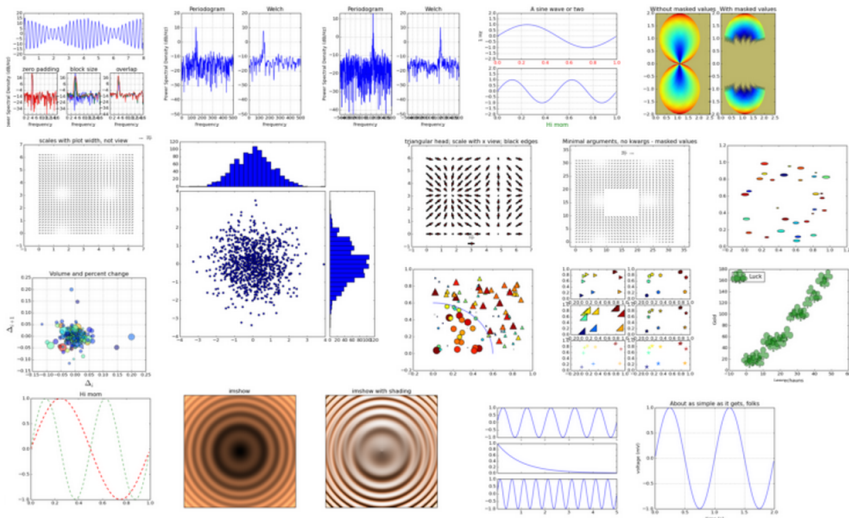
Python Imaging Library (PIL) example: 2D plotting

Edge detection using numerical differentiation



- Simple Python script reading a colour PNG image, calculating gradient of the blue filter, plotting its norm in black/white (20 lines of code total)
- Code in `codes/fuji.py`

Matplotlib gallery contains hundreds of examples



- <http://matplotlib.org/gallery.html> – click on any plot to get its source code
- <http://www.labri.fr/perso/nrougier/teaching/matplotlib> is a really good introduction

Bokeh gallery



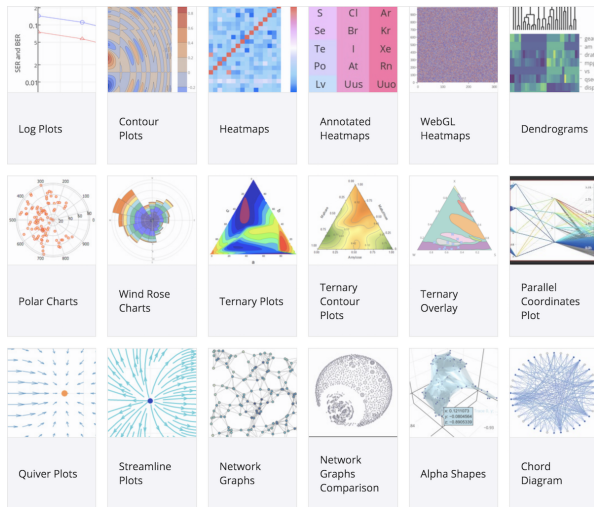
- Open-source project from Continuum Analytics
<https://docs.bokeh.org/en/latest/docs/gallery.html>
- Produces dynamic html5 visualizations for the web
- Basic server-less interactivity packed into a json object; more complex interactions via a Bokeh server

Plotly Python library

- Open-source project from Plot.ly
<https://plot.ly/python>

- Produces dynamic html5 visualizations for the web

- APIs for Python (with/without Jupyter), R, JavaScript, MATLAB



- Can work offline (free) or by sending your data to your account on plot.ly (public plotting is free, paid unlimited private plotting and extra tools)

Other popular sci-vis software

- Other excellent plotting libraries in Python: **Seaborn**, **HoloViews**, **Altair**, **plotnine** (each deserving its own page in this deck)
- Partial list of other sci-vis tools you may find on large HPC clusters:
 - ▶ **Avizo** for general-purpose 3D visualization; the only commercial package on this list
 - ▶ **Tablet** viewer for genome sequence assembly and alignment
 - ▶ **Molden** for visualization of molecular and electronic structure
 - ▶ WebMO web portal for computing/visualization in chemistry
 - ▶ XCrySDen for crystalline and molecular structure visualisation
 - ▶ **GNU Data Language** (GDL) for data analysis and visualization in astronomy, geosciences and medical imaging; open-source implementation of IDL
 - ▶ GDIS for visualization of molecular and periodic structures
 - ▶ Molekel for molecular visualization
 - ▶ Ncview visual browser for NetCDF files
 - ▶ **ParaView** for general-purpose scientific visualization
 - ▶ Rasmol for molecular visualization
 - ▶ **VisIt** for general-purpose scientific visualization
 - ▶ **VTK** = Visualization Toolkit library
 - ▶ **VMD** for visualization of large biomolecular systems

Today's focus: multi-dimensional sci-vis packages

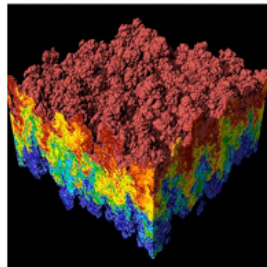
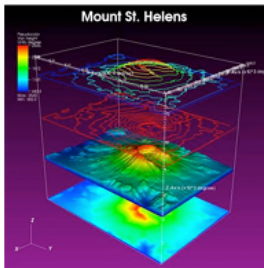
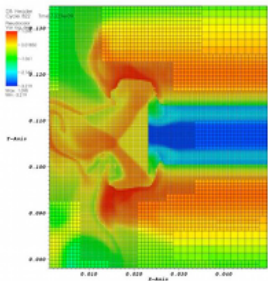
- Open-source + multi-platform + general-purpose + must support the following features:
 - ▶ visualize scalar and vector fields
 - ▶ structured and unstructured meshes in 2D and 3D, particle data, polygonal data, irregular topologies
 - ▶ ability to handle very large datasets (GBs to TBs)
 - ▶ ability to scale to $10^{\sim 12}$ resolution elements
 - ▶ ability to scale to large ($10^3 - 10^5$ cores) computing facilities
 - ▶ interactive manipulation
 - ▶ support for scripting
 - ▶ support for most common data formats, parallel I/O

1. **VisIt** (latest is 3.3)
2. **ParaView** (latest is 5.11)

VisIt

<https://visit-dav.github.io/visit-website>

- Developed by the Department of Energy (DOE) Advanced Simulation and Computing Initiative (ASCI) to visualize results of terascale simulations, first public release in fall 2002
- Available as source and binary for Linux/Mac/Windows
- Over 80 visualization features (contour, mesh, slice, volume, molecule, ...)
- Reads over 110 different file formats; APIs for C++, Python, and Java
- Interactive and Python scripting; full integration with VTK library
- Uses MPI for distributed-memory parallelism on HPC clusters

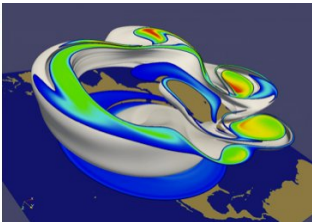
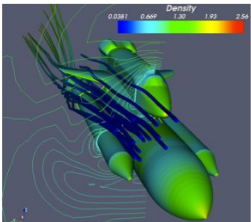
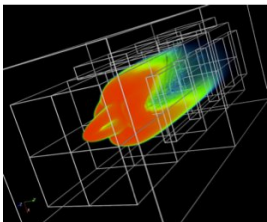


Lawrence Livermore National Laboratory

ParaView

<http://www.paraview.org> and <https://github.com/Kitware/ParaView>

- Started in 2000 as a collaboration between Los Alamos NL and Kitware Inc., later joined by Sandia NL and other partners; first public release in 2002
- Available as source and pre-compiled binary for Linux/Mac/Windows
- To visualize extremely large datasets on distributed memory machines
- Both interactive and Python scripting
- Client-server for remote interactive visualization
- Uses MPI for distributed-memory parallelism on HPC clusters
- ParaView is based on VTK (Visualization Toolkit)
 - ▶ not the only VTK-based open-source scientific renderer, e.g. VisIt, MayaVi (Python + numpy + scipy + VTK), an of course a number of Kitware's own tools besides ParaView are based on VTK
 - ▶ VTK can be used from C++, Python, and now JavaScript as a standalone renderer



Why ParaView for this workshop?

- Back in ~2010, I had to pick one
 - ▶ both ParaView's and VisIt's binaries are widely available, in active development
 - ▶ both can do remote client-server visualization, very good parallel scalability
 - ▶ ParaView and VisIt interfaces are very different
- Tight integration with VTK (developed by the same folks), 130 input formats
- A number of add-on projects
 - ▶ **ParaViewWeb** is a JavaScript library to write web applications that talk to a remote ParaView server; can reproduce full standalone ParaView in a web browser (WebGL + remote processing)
 - ▶ **vtk.js** is a scientific rendering library for the web (standalone WebGL)
 - ▶ **ParaView Glance** is a standalone open-source web app for in-browser 3D sci-vis
 - ▶ **Catalyst** is an open-source *in-situ visualization* library that can be embedded directly into parallel simulation codes; interaction through ParaView scripts
 - ▶ **ParaView Cinema** for interactive visualization from pre-rendered images (rotation, panning, zooming, variables on/off)
- Stereo viewing on 3D hardware; experimental builds for various head-mounted displays (HMDs), Looking Glass displays
- We also use and promote <https://visit-dav.github.io/visit-website>, other open-source packages

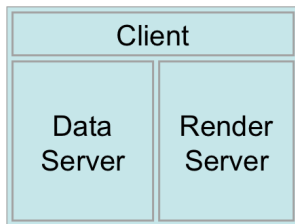
PARAVIEW ARCHITECTURE AND GUI

ParaView's distributed parallel architecture

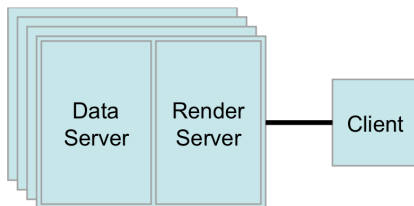
Three logical components inside ParaView – these units can be embedded in the same application on the same computer, but can also run on different machines:

- **Data Server** – The unit responsible for data reading, filtering, and writing. All of the pipeline objects seen in the pipeline browser are contained in the data server. The data server can be parallel.
- **Render Server** – The unit responsible for rendering. The render server can also be parallel, in which case built-in parallel rendering is also enabled.
- **Client** – The unit responsible for establishing visualization. The client controls the object creation, execution, and destruction in the servers, but does not contain any of the data, allowing the servers to scale without bottlenecking on the client. If there is a GUI, that is also in the client. The client is always a serial application.

Two major workflow models



Standalone mode: computations and user interface run on the same machine



Client-server mode: `pvserver` on a multi-core server or distributed cluster

Advantages of remote client-server rendering

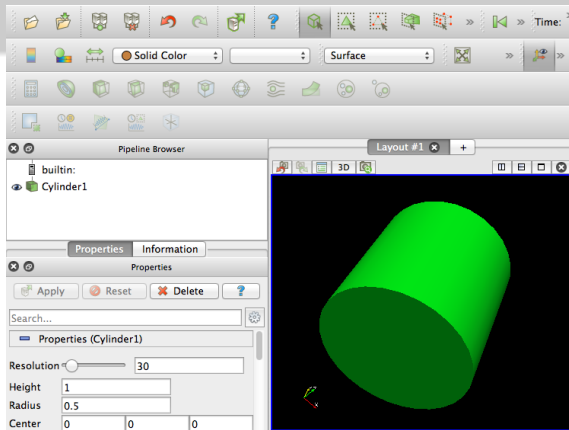
- Standalone ParaView has its limitations: **limited memory**, **limited I/O bandwidth**, and **limited CPU / GPU power**
- For example, on a workstation with 48GB memory works well up to 2048³ single-precision float variable on structured grids stored locally
- Larger / higher-res datasets, more complex grids, or datasets requiring complex filters won't fit
- **Typical problem that won't fit on a 48GB workstation and is too slow to read via sshfs:** simulation of the airflow around a wing on an *unstructured grid* (*.vtu) with 246×10^6 cells (equiv. to 627^3), one variable takes 25GB — however, can do this interactively without problem on 64 cores on a cluster with `pvsolver` taking ~ 120 GB memory
- **We'll study remote ParaView in more detail towards the end of this workshop**

Starting ParaView

- Today we'll do everything in standalone ParaView on your desktop
- **Linux/Unix:** type paraview at the command line
- **Mac:** click paraview in Applications folder
- **Windows:** select paraview from start menu
- ParaView GUI should start up
- The server `pvserver` is run for you in the background

User interface

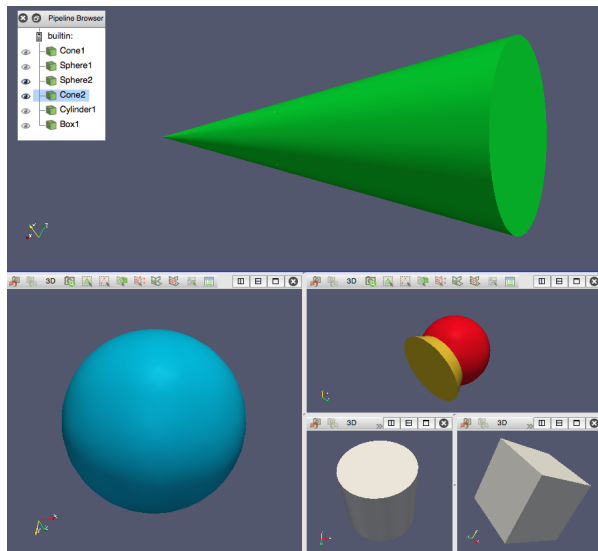
- **Pipeline Browser:** data readers, data filters, can turn visibility of each object on/off
- **Object Inspector:** view and change parameters of the current pipeline object (via tabs properties-display-info or properties-info)
- **View window:** displays the result



1. Find the following in the toolbar: "Connect", "Disconnect", "Toggle Colour Legend Visibility", "Edit Colour Map", "Rescale to Data Range"
2. Load a predefined dataset: in ParaView select Sources → Cylinder
3. Try dragging the cylinder using the left mouse button; also try the same with the right and middle buttons
4. Identify drop-down menus; try changing to a different view (e.g. from Surface to Wireframe) or changing colour via "Edit Colour Map"

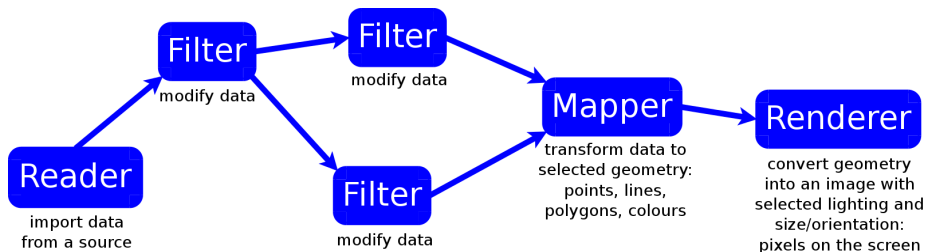
ParaView windows

- Reproduce this image
- Use objects from the “Sources” menu (cone, sphere, cylinder, box), can edit their properties
- Use the icons in the upper right of each window to split the view
- Optionally can link any two views by right-clicking on an image, selecting “Link Camera”, and clicking on a second image



Data flow in VTK

https://vtk.org/Wiki/VTK/Tutorials/VTK_Terminology



- Data goes through **Mapper** which knows how to draw it, places that data into the rendered scene via a VTK **Actor**
 - ▶ `mapper.setInputConnection(object.getOutputPort())`
- **Actor** is an OpenGL object = the part that is rendered
 - ▶ takes data from Mapper: `actor.setMapper(mapper)`
 - ▶ passed to Renderer: `renderer.addActor(actor)`
- **Renderer** can hold multiple actors
- **RenderWindow** (on the screen) can hold multiple renderers

IMPORTING DATA INTO PARAVIEW

Data sources

- Generate data with a *Source* object
- Read data from a file

ADAPT Files (*.nc *.cdf *.elev *.ncd)
 Adaptive cosmo files (*.cosmo)
 ADIOS2 BP3 File (CoreImage) (*.bp)
 ADIOS2 BP4 Directory (CoreImage) (*.bp)
 ADIOS2 BP4 Metadata File (CoreImage) (md.idx)
 AMR Enzo Files (*.boundary *.hierarchy)
 AMR Flash Files (*.Flash *.flash)
 AMR Velodyne Files (*.xamr *.Xamr *.XAMR)
 AMReX/BoxLib plotfiles (grids) (plt*)
 AMReX/BoxLib plotfiles (particles) (plt*)
 ANSYS Files (*.inp)
 AU File Files (*.au)
 AVS UCD Binary/ASCII Files (*.inp)
 BOV Files (*.bov)
 BYU Files (*.g)
 CAM NetCDF (Unstructured) (*.nc *.ncdf)
 Case file for restarted CTH outputs
 CCSM MTSD Files (*.nc *.cdf *.elev *.ncd)
 CCSM STSD Files (*.nc *.cdf *.elev *.ncd)
 CEAcad Files (*.acd *.imp)
 CCNS Files (*.cgn)
 Chombo Files (*.hdf5 *.h5)
 CityGML files (*.gml *.xml)
 Claw Files (*.claw)
 CMAT Files (*.cmat)
 CMI (*.cmi)
 CONVERGE CFD (*.h5)
 Cosmology Files (*.cosmo64 *.cosmo)
 CTRL Files (*.ctrl)
 Curve2D Files (*.curve *.ultra *.alt *.u)
 DDCMD Files (*.ddcmd)
 Delimited Text (*.csv *.tsv *.txt *.CSV *.TSV *.TXT)
 DICOM Files (directory) (*.dcm)
 DICOM Files (single) (*.dcm)
 Digital Elevation Map Files (*.dem)
 Dyna3D Files (*.d3d)
 EnSight Files (*.case *.CASE *.Case)
 EnSight Master Server Files (*.sos *.SOS)
 ENZO AMR Particles (*.boundary *.hierarchy)
 Exodus11 (*.g *.gc *.ex2 *.exo *.gen *.par
 Extruded Vol Files (*.exvol)
 Facet Polygonal Data Files (*.facet)
 Fides Data Model File (JSON) (*.json)
 Fides Files (ADIOS2 BP) (*.bp)
 FLASH AMR Particles Reader (*.Flash *.flash)

FLASH Files (Visit) (*.flash *.f5)
 Fluent Case Files (*.cas)
 Fluent Files (Visit) (*.cas)
 FVCOM MTMD Files (*.nc *.cdf *.elev *.ncd)
 FVCOM MTSD Files (*.nc *.cdf *.elev *.ncd)
 FVCOM Particle Files (*.nc *.cdf *.elev *.ncd)
 FVCOM STSD Files (*.nc *.cdf *.elev *.ncd)
 Gadget Files (*.gadget)
 Gaussian Cube Files (*.cube)
 GDAL Raster (*.tif *.gen *.thf *.adf *.arg *.hls *.xib)
 GDAL Vector (*.shp *.faa *.bmn *.dxf *.csv *.geojson)
 Generic10 files to MultiBlockDataset (*.gio)
 Generic10 files to UnstructuredGrid (*.gio)
 GGCM Files (*.3df *.mer)
 gITF 2.0 Files (*.gltf *.glib)
 GTC Files (*.h5)
 GULP Files (*.trg)
 H5Nimrod Files (*.h5nimrod)
 H5Part particle files (*.h5part)
 HyperTreeGrid (*.h5)
 HyperTreeGrid (partitioned) (*.phtg)
 Image Files (*.pnm *.ppm *.adt *.spe *.imgvol)
 JPEG lma
 LAMMPS Dump Files (*.dump)
 LAMMPS Struct. (*.eam *.meam *.rigid *.lammps)
 Legacy VTK files (*.vtk *.vtk-series)
 Legacy VTK Files (partitioned) (*.pvtk)
 Lines Files (*.lines)
 LODI Files (*.mo *.cdf *.elev *.ncd)
 LODI Particle Files (*.nc *.cdf *.elev *.ncd)
 LSDyna (*.k *.lsdyna *.d3plot *.d3plot)
 M3DC1 Files (*.h5)
 Meta Image Files (*.mhd *.mha)
 Meta-Generic10 files (*.exos)
 Metafile for restarted exodus outputs
 MFIX netcdf Files (*.nc)
 MFIX Res Files (Visit) (*.RES)
 MFIX Unstructured Grid Files (*.RES)
 Mill Files (*.m)
 Miranda Files (*.mir *.raw)
 MM5 Files (*.mm5)
 MotionFX CFG Files (*.cfg)
 MPAS NetCDF (Unstructured) (*.ncdf *.sid)
 MRIC Image Files (*.mnc *.ali *.sid *.nc)
 Multilevel 3D Plasma Files (*.m3d *.h5)

NASTRAN Files (*.nas *.f06)
 Nek5000 (*.nek3d *.nek2d *.nek5d *.nek5000 *.nek)
 netCDF generic and CF conventions (*.ncdf *.nc)
 Nrm Raw Image Files (*.nrm *.nhdr)
 OME TIFF Files (*.ome.tif *.ome.tiff)
 OpenFOAM (*.foam)
 OpenFOAM Files (Visit) (*.controlDict)
 openPMD files (*.pmd)
 OVERFLOW Files (Visit) (*.dat *.save)
 ParaDIS Files (*.prds *.data *.dat)
 ParaDIS Tecplot (Mid *.field *.cyl *.cylinder *.dat)
 Parallel POP Ocean NetCDF (*.pop.ncdf *.pop.nc)
 ParaView Data Files (*.pvd)
 ParaView Ensemble Data (*.pve)
 PATRAN Files (*.neu)
 PFOTRAN Files (*.h5)
 PhaSta Files (*.pht)
 PIO Dump Files (*.pio)
 Poxe Files (*.h5)
 PLOT2D Files (*.p2d)
 PLOT3D Files (*.xyz)
 PLOT3D Meta Files (*.p3d)
 PLY Polygonal File Format (*.ply *.ply-series)
 PNC Image Files (*.png)
 POINT3D Files (*.3D)
 POP Ocean NetCDF Rectilinear (*.pop.nc)
 POP Ocean NetCDF Unstructured (same as prev.)
 prevSTAR Files (*.cd *.vrt)
 Protein Data Bank Files (*.pdb)
 Protein Data Bank Files (Visit) (*.ent *.pdb)
 PTS (Point Cloud) Files (*.pts)
 Radiance HDR file (*.hdr)
 Raw (binary) Files (*.raw)
 RAW Files (*.raw)
 SAMRAI series files (*.samrai)
 SAR Files (*.SAR *.nc)
 SAS Files (*.sasgeom *.sas *.sasdata)
 SEG-Y Files (*.sgy *.seggy)
 SEP file (Plugin) (*.H)
 Silo Files (*.silo *.pdb *.silo-series *.pdb-series)
 SLAC Mesh Files (*.ncdf *.nc)
 SLAC Particle Files (*.ncdf *.netcdf)
 Spherical Files (*.spherical *.sv)
 Spy Plot History Files (*.h5h5 *.h5h5t)
 SpyPlot CTH dataset (*.apcr *.spot*)

Stereo Lithography (*.stl *.stl-series)
 Tecplot Binary Files (Visit) (*.plt)
 Tecplot Files (*.tec *.TTC *.Tec *.ap *.TPT)
 Tecplot Files (Visit) (*.tec *.TTC *.Tec *.ap *.TPT)
 Tecplot Table (*.dat *.DAT)
 Tetrad Files (*.hdf5 *.h5)
 TFF Files (*.tff *.tif)
 TIFF Image Files (*.tif *.tiff)
 TRUCHAS dataset (*.hdf5 *.h5)
 Tsarf Files (*.ts_deg3)
 UNIC Files (*.h5)
 VASP Animation Files (*.out)
 VASP CHGCA Files (*.CHG*)
 VASP OUT Files (*.OUT*)
 VASP POSCAR Files (*.POS*)
 VASP Tesselation Files (*.out)
 Velodyne Files (*.vld *.nvd)
 Visit MetaPLOT3D Files (Visit) (*.vp3d)
 VizSchema Files (*.h5 *.vsh5)
 VPIC Files (*.vpc)
 VRML 2 Files (*.vrl *.vrml)
 VTK Hierarchical Box Data Files (*.vthb)
 VTK ImageData Files (*.vti *.vti-series)
 VTK ImageData Files (partitioned) (*.pvti)
 VTK Multiblock Data Files (*.vtm *.vtmb)
 VTK Particle Files (*.particles)
 VTK Partitioned Dataset Collection Files (*.vtpc)
 VTK Partitioned Dataset Files (*.vtpd *.vtpd-series)
 VTK PolyData Files (*.vtp *.vtp-series)
 VTK PolyData Files (partitioned) (*.pvtp)
 VTK RectilinearGrid Files (*.vtr *.vtr-series)
 VTK RectilinearGrid Files (partitioned) (*.pvtr)
 VTK StructuredGrid Files (*.vts *.vts-series)
 VTK StructuredGrid Files (partitioned) (*.pvts)
 VTK Table (partitioned) (*.pvtt *.pvtt-series)
 VTK Table Files (*.vtt *.vtt-series)
 VTK UnstructuredGrid Files (*.vtu *.vtu-series)
 VTK UnstructuredGrid Files (partitioned) (*.pvту)
 VTX reader: ADIOS2 BP3 File (*.bp)
 VTX reader: ADIOS2 BP4 Directory (*.bp *.bp4)
 Waveform OBJ Files (*.obj)
 WindFlade Data (*.wind)
 Xdmf Reader (*.xmf *.xmf *.xmf2 *.xmf2)
 Xdmf3 Reader (*.xmf *.xmf *.xmf3 *.xmf3)
 Xdmf3 Reader (Top Level Partition) (*.xmf *.xmf)
 Xmdv Files (*.okc)
 Xmol Molecule Files (*.xyz)
 XYZ Files (*.xyz)

Example: reading raw (binary) data

Show $f(x, y, z) = (1 - z) [(1 - y) \sin(\pi x) + y \sin^2(2\pi x)]$
 $+ z [(1 - x) \sin(\pi y) + x \sin^2(2\pi y)]$ in $x, y, z \in [0, 1]$ sampled at 16^3

1. File: data/simpleData.raw – load it as RAW BINARY

Update: starting with ParaView 5.9, this no longer works!

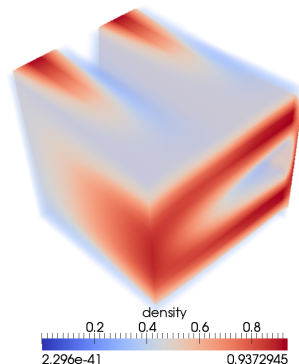
2. Describe the dataset in properties:

- ▶ Data Scalar Type = float
- ▶ Data Byte Order = Little Endian
- ▶ File Dimensionality = 3
- ▶ Data Extent: 0 to 15 in each dimension (1 to 16 will crash recent ParaView versions)
- ▶ Scalar Array Name = density

3. Try different views: Outline, Points, Wireframe, Volume

4. Depending on the view, **can edit the colour map**

5. Try saving data as paraview data type (*.pvd), deleting the object, and reading back from *.pvd – file now contains full description of dataset



VTK = Visualization Toolkit

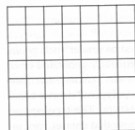
- Open-source software system for 3D computer graphics, image processing and visualization
- Bindings to C++, Tcl, Java, Python
- ParaView is based on VTK \Rightarrow supports all standard VTK file formats
- VTK file formats

<http://www.vtk.org/VTK/img/file-formats.pdf>

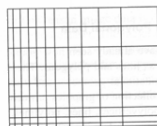
- ▶ legacy serial format (*.vtk): **ASCII header lines** + **ASCII/binary data**
- ▶ XML formats (extension depends on VTK data type): **XML tags** + **ASCII/binary/compressed data**
 - newer, much preferred to legacy VTK
 - supports **parallel file I/O**, compression, portable binary encoding (big/little endian byte order), random access, etc.

VTK 3D data: 6 major dataset (discretization) types

- **Image Data/Structured Points:** *.vti, points on a regular rectangular lattice, scalars or vectors at each point
- **Rectilinear Grid:** *.vtr, same as Image Data, but spacing between points may vary, need to provide steps along the coordinate axes, not coordinates of each point
- **Structured Grid:** *.vts, regular topology and irregular geometry, need to indicate coordinates of each point



(a) Image Data



(b) Rectilinear Grid



(c) Structured Grid

VTK 3D data: 6 major dataset (discretization) types

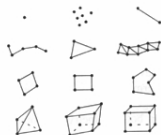
- **Particles/Unstructured Points:** *.particles
- **Polygonal Data:** *.vtp, unstructured topology and geometry, point coordinates, 2D cells only (i.e. no polyhedra), suited for maps
- **Unstructured Grid:** *.vtu, irregular in both topology and geometry, point coordinates, 2D/3D cells, suited for finite element analysis, structural design



(d) Unstructured Points



(e) Polygonal Data



(f) Unstructured Grid

VTK 3D data: dataset attributes

A VTK file can store a number of datasets, each could be of one of the following types:

- Scalars: single valued, e.g. density, temperature, pressure
- Vectors: magnitude and direction, e.g. velocity
- Normals: direction vectors ($|\mathbf{n}| = 1$) used for shading
- LookupTable: each entry in the lookup table is a red-green-blue-alpha array (alpha is opacity: alpha=0 is transparent); if the file format is ASCII, the lookup table values must be float values in the range [0,1]
- TextureCoordinates: used for texture mapping
- Tensors: 3×3 real-valued symmetric tensors, e.g. stress tensor
- FieldData: array of data arrays

Example: reading legacy VTK

Caution: storing large datasets in ASCII is not a very good idea – here we look at text-based VTK files for instructional purposes

1. File: `data/volume.vtk`
 - ▶ simple example (Structured Points): $3 \times 4 \times 6$ dataset, one scalar field, one vector field
2. File: `data/density.vtk`
 - ▶ another simple example (Structured Grid): $2 \times 2 \times 2$ dataset, one scalar field
3. File: `data/cube.vtk`
 - ▶ more complex example (Polygonal Data): cube represented by six polygonal faces. A single-component scalar, normals, and field data are defined on all six faces (CELL_DATA). There are scalar data associated with the eight vertices (POINT_DATA). A lookup table of eight colours, associated with the point scalars, is also defined.

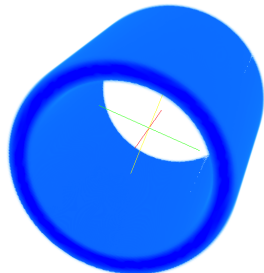
Exercise: visualizing 3D data with legacy VTK

- Visualize a 3D “cylinder” function

$$f(x, y, z) = e^{-|r-0.4|}$$

where $r = \sqrt{(x - 0.5)^2 + (y - 0.5)^2}$,
inside a unit cube ($x, y, z \in [0, 1]$)

➡ reproduce the view on the right



- ASCII data in `data/cylinder.dat` (discretized on a 30^3 Cartesian mesh)
- Add an appropriate header to create a VTK file using `data/volume.vtk` as template

Writing XML VTK from C++

Let's turn to **larger datasets (MB, GB)** – we should store them as binary

- A good option is to use XML VTK format with binary data and XML metadata, calling VTK library functions from C++ / Java / Python to write data
- Here is an example: `codes/SGrid.cpp` and `codes/Makefile`, generates the file `data/halfCylinder.vts`

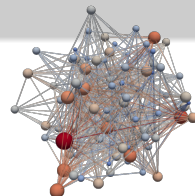
This example shows how to create a Structured Grid, set grid coordinates, fill the grid with a scalar and a vector, and write it in XML VTK to a *.vts file.

- To run it, you need the VTK C++ library installed (either standalone or pulled from ParaView); check `codes/Makefile` to see the required library files

```
cd codes
make SGrid
(on Linux: export LD_LIBRARY_PATH=/path/to/vtk/lib:$LD_LIBRARY_PATH)
(on a Mac: export DYLD_LIBRARY_PATH=$HOME/Documents/local/vtk/lib)
./SGrid
```

- Many more examples included with the VTK source code or at <http://www.vtk.org/Wiki/VTK/Examples/Cxx>

Writing XML VTK from Python



```
$ pip install vtk networkx
```

Check out codes/{writeNodesEdges, randomGraph}.py

```
def writeObjects(nodeCoords,
                 edges = [],
                 scalar = [], name = '', power = 1,
                 scalar2 = [], name2 = '', power2 = 1,
                 nodeLabel = [],
                 method = 'vtkPolyData',
                 fileout = 'test'):

    """
    Store points and/or graphs as vtkPolyData or vtkUnstructuredGrid.
    Required argument:
    - nodeCoords is a list of node coordinates in the format [x,y,z]
    Optional arguments:
    - edges is a list of edges in the format [nodeID1,nodeID2]
    - scalar/scalar2 is the list of scalars for each node
    - name/name2 is the scalar's name
    - power/power2 = 1 for r-scalars, 0.333 for V-scalars
    - nodeLabel is a list of node labels
    - method = 'vtkPolyData' or 'vtkUnstructuredGrid'
    - fileout is the output file name (will be given .vtp or .vtu extension)
    """
```

```
import networkx as nx
from writeNodesEdges import writeObjects

numberNodes, numberEdges = 100, 500
H = nx.gnm_random_graph(numberNodes,numberEdges)
print('nodes:', H.nodes())
print('edges:', H.edges())

# return a dictionary of positions keyed by node
pos = nx.random_layout(H,dim=3)
# convert to list of positions (each is a list)
xyz = [list(pos[i]) for i in pos]

degree = [d for i,d in H.degree()]
writeObjects(xyz, edges=H.edges(), scalar=degree,
             name='degree', fileout='network')
```

Another option for writing XML VTK from Python

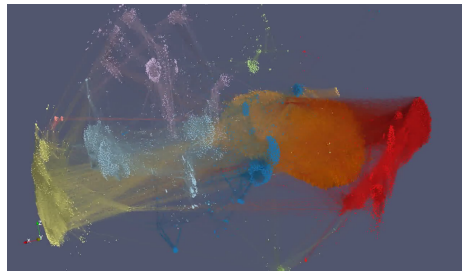
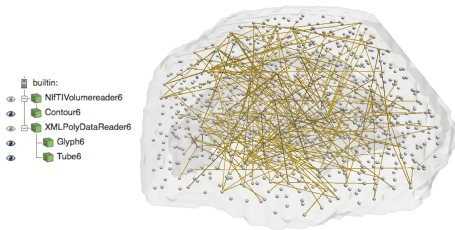
PyEVTk library <https://bitbucket.org/pauloh/pyevtk>

```
$ pip install pyevtk
```

- Works in both Python 2 and Python 3
- Many examples in <https://github.com/paulo-herrera/PyEVTk/tree/master/evtk/examples>

```
from pyevtk.hl import imageToVTK
from numpy import zeros
n = 30
data = zeros((n,n,n), dtype=float)
for i in range(n):
    x = ((i+0.5)/float(n)*2. - 1.)*1.2
    for j in range(n):
        y = ((j+0.5)/float(n)*2. - 1.)*1.2
        for k in range(n):
            z = ((k+0.5)/float(n)*2. - 1.)*1.2
            data[i][j][k] = ((x*x+y*y-0.64)**2 + (z*z-1.)**2) * \
                            ((y*y+z*z-0.64)**2 + (x*x-1.)**2) * \
                            ((z*z+x*x-0.64)**2 + (y*y-1.)**2)
imageToVTK("decoCube", pointData={"scalar" : data})
```

Think of ParaView as a GUI front end to VTK classes



hidden/mutOnCtOrbits.mp4 on presenter's laptop

```

vtkPoints *points = vtkPoints::New();
for (i=0; i<1028; i++) points->InsertNextPoint(x[i], y[i], z[i]);
vtkCellArray *lines = vtkCellArray::New();
for (j=0; j<degree; j++) { // line from node to adjacent[j]
    lines->InsertNextCell(2);
    lines->InsertCellPoint(node);
    lines->InsertCellPoint(adjacent[j]); }
vtkPolyData* polyData = vtkPolyData::New();
polyData->SetPoints(points); polyData->SetLines(lines);
vtkSmartPointer<vtkXMLPolyDataWriter> writer = vtkSmartPointer<vtkXMLPolyDataWriter>::New();
writer->SetFileName("output.vtp"); writer->SetInputData(polyData);
writer->Write();

```

NetCDF and HDF5

- VTK is incredibly versatile format, can describe many different data types
- Very often in science one needs to simply store and visualize multi-dimensional arrays
- **Problem: how do you store a 2000^3 array of real numbers (30GB of data)?**
 - ▶ ASCII – forget about it
 - ▶ raw binary – possible, but many problems
 - ▶ VTK – probably an overkill for simple arrays
- Scientific data formats come to rescue, two popular scientific data formats are NetCDF and HDF5
 - ▶ binary (of course!)
 - ▶ self-descriptive (include metadata)
 - ▶ portable (cross-platform): libraries for many OS's, universal datatypes, byte order in a word (little vs. big endian), etc.
 - ▶ support parallel I/O (through MPI-IO)
 - ▶ support compression

NetCDF support in ParaView

- NetCDF is supported natively in ParaView
 - ▶ `codes/writeNetCDF.cpp` (Fortran version `codes/writeNetCDF.f90`) writes a 100^3 volume with a doughnut shape at the centre in NetCDF

C++ example

```
$ icc writeNetCDF.cpp -o writeNetCDF -I/path/to/netcdf/include \
    -L/path/to/netcdf/lib -lnetcdf_c -lnetcdf
$ ./writeNetCDF
```

F90 example

```
$ ifort writeNetCDF.f90 -o writeNetCDF -I/path/to/netcdf/include \
    -L/path/to/netcdf/lib -lnetcdff -lnetcdf
$ ./writeNetCDF
```

- ParaView understands common **NetCDF conventions**, e.g., conventions for CF (Climate and Forecast) metadata (<http://cfconventions.org>): 2D or 3D datasets on a sphere, coordinate axes, fill-in values, etc.
 - ▶ example 1 on presenter's laptop: 2D dataset `hidden/ice.nc`
 - ▶ example 2: snapshot of a 3D dataset `hidden/temp1.png`
 - ▶ example 3: more polished 3D visualization `hidden/tempsalt.mp4`

On the subject of spheres ...

How about mapping topography on top of our visualization?

There is a good resource on this

<https://www.earthmodels.org/date-and-tools/topography>

- **Option 1:** load precomputed topography stored as Polygonal Data
 - ▶ e.g. [this link](#) (downloads ETOPO_10min_Ice.vtp) provides full globe (both land and ocean) at 10 arcmin resolution
 - ▶ or [this link](#) (downloads ETOPO_10min_Ice_only-land.vtp) provides only land at 10'
- **Option 2:** map a bitmap image to the globe; e.g. [this link](#) downloads a 8192×4096 image `texture_land_ocean_ice_8192.png`
 1. create a high-resolution Sphere (from Sources)
 2. apply Texture-Map-to-Sphere filter, make sure to click Apply before you can see Miscellaneous:Texture
 - creates "texture coordinates"
 - [we haven't studied filters yet](#)
 3. in Properties of the filter: under Miscellaneous:Texture use the drop-down menu to load a PNG image, click Apply
 4. in Properties of the filter: uncheck Prevent Seam at the top, again click Apply
 5. still colouring by Solid Color and viewing as Surface

HDF5 support in ParaView

- No native support for HDF5, however, ParaView supports a container format XDMF (eXtensible Data Model and Format) which uses HDF5 for actual data – only briefly mention it, details at <http://www.xdmf.org>
- XDMF = XML for **light** data + HDF5 for **heavy** data
 - ▶ data type (float, integer, etc.), precision, rank, and dimensions completely described in the XML layer (as well as in HDF5)
 - ▶ the actual values in HDF5, potentially can be enormous
- Single XML wrapper can reference multiple HDF5 files (e.g. written by each node on a cluster)
- Don't need HDF5 libraries to perform simple operations
- C++ API is provided to read/write XDMF data
- Can be used from Python, Tcl, Java, Fortran through C++ calls
- In Fortran can generate XDMF files with HDF5 calls + plain text for the XML wrapper http://www.xdmf.org/index.php/Write_from_Fortran
- Also support for a number of file formats generated by third-party software that in turn use HDF5 underneath

Reading OpenFOAM 2.3.x datasets

- ✓ ParaView can read *.controlDict and *.foam files (**File** → **Open**) but these are not present in OpenFOAM's output; can create an empty case.foam file in the case directory and load it into ParaView
 - ▶ **most of the time it works** – when it does not, the error can be traced to VTK/IO/Geometry/vtkOpenFOAMReader.cxx in ParaView's source code
- ✗ Use OpenFOAM's built-in **foamToVTK** utility to convert data from OpenFOAM format to VTK – **works with all versions of ParaView**
- ✗ Use OpenFOAM-supplied ParaView reader module libraries PV4FoamReader and vtkPV4Foam with precompiled ParaView 4.x.x through **paraFoam** launch script
 - ▶ **tested with precompiled binary ParaView 4.x.x**
 - ▶ no need to compile anything (contrary to OpenFOAM documentation!)
 - ▶ not 100% compatible with ParaView Python scripting
- ✗ Deprecated: recompile ParaView with a third-party (not from OpenFOAM or ParaView) **vtkPOFFReader** plugin – crashes newer ParaView; officially the plugin was written for ParaView 3.10 - 3.14
- ✗ Deprecated: use the same OpenFOAM-supplied reader libraries PV4FoamReader and vtkPV4Foam with bundled third-party software pack **ThirdParty-2.3.1** that includes an older ParaView-4.1.0; requires compilation of ParaView with OpenFOAM's unconventional build scripts

Reading OpenFOAM: using foamToVTK utility

- Assuming you have OpenFOAM installed:

```
<submit an interactive job on the cluster and wait for the prompt>
module load application/OpenFOAM/2.3.0      # or similar
source $OPENFOAM_SETUP
<change to the case directory containing system/, constant/,
  processorXXX/, time outputs>
foamToVTK                                # to process everything
foamToVTK -latestTime                    # to process the last frame in the model
foamToVTK -time 2.98:2.99                 # to process a range of timesteps
foamToVTK -time 9.39:                     # to process a range of timesteps
```

- This will create a VTK subdirectory with one main VTK file per timestep containing the 3D volume, and auxiliary VTK files describing the boundaries
- Next simply load the main VTK files into ParaView and script a movie with ParaView's Python (more on scripting/animation later)

Reading OpenFOAM: paraFoam script on MacOS

Step 1: install and configure paraFoam

```
brew install open-mpi scotch cgal
brew install boost --without-single --with-mpi
cd ~/Downloads && mkdir OpenFOAM && cd OpenFOAM
wget http://downloads.sourceforge.net/foam/OpenFOAM-2.3.1.tgz
tar xvfz OpenFOAM-2.3.1.tgz && cd OpenFOAM-2.3.1
wget https://raw.githubusercontent.com/mrklein/openfoam-os-x/master/\
OpenFOAM-2.3.1.patch
patch -p1 < OpenFOAM-2.3.1.patch
```

Step 2: launch paraFoam

```
<change to the case directory containing system/, constant/,
processorXXX/, time outputs>
export FOAM_INST_DIR=~/.Downloads/OpenFOAM
source $FOAM_INST_DIR/OpenFOAM-2.3.1/etc/bashrc
paraFoam      # launches ParaView, points it to a sequence
               # of time step files, loads the first time step
```

Warning: paraFoam script is not entirely compatible with ParaView's Python
(can't use the trace tool to reproduce paraFoam customization)

Reading OpenFOAM: paraFoam script on a cluster

Step 1: install and configure paraFoam (for system-wide or your own ParaView)

```
cd /scratch2/razoumov
wget http://downloads.sourceforge.net/foam/OpenFOAM-2.3.1.tgz
tar xvfz OpenFOAM-2.3.1.tgz && cd OpenFOAM-2.3.1
export FOAM_INST_DIR=/scratch2/razoumov
sed -i -e 's|4.1.0|4.3.1|' $FOAM_INST_DIR/OpenFOAM-2.3.1/etc/config/paraview.sh
source $FOAM_INST_DIR/OpenFOAM-2.3.1/etc/bashrc
mkdir -p $ParaView_DIR && cd $ParaView_DIR
cp -r /global/software/ParaView/ParaView-4.3.1-Linux-64bit/* .
```

Step 2: launch paraFoam **inside a VNC session**

```
<change to the case directory containing system/, constant/,
processorXXX/, time outputs>
export FOAM_INST_DIR=/scratch2/razoumov
source $FOAM_INST_DIR/OpenFOAM-2.3.1/etc/bashrc
vglrun paraFoam -builtin      # launches ParaView, points it to
                                # a sequence of time step files,
                                # loads the first time step
```

Recap of input file formats

- Raw binary data
- VTK legacy format (*.vtk) with ASCII data for small datasets
 - ▶ Structured Points
 - ▶ Structured Grid
 - ▶ Polygonal Data
- VTK XML formats for large datasets: most versatile, use from C++ and Python
 - ▶ Structured Grid (*.vts)
 - ▶ other formats can be written using the respective class, e.g. vtkPolyData, vtkRectilinearGrid, vtkStructuredGrid, vtkUnstructuredGrid
- HDF5 files via XDMF, **native NetCDF**
- Many 3rd-party file formats understood natively by ParaView
- OpenFOAM is doable but need to use the right technique (don't trust the available documentation: a lot of it is wrong!)

WORKING WITH PARAVIEW: FILTERS

Filters

Many interesting features about a dataset cannot be determined by simply looking at its surface: a lot of useful information is on the inside, or can be extracted from a combination of variables

Sometimes a desired view is not available for a given data type, e.g.

- a 2D dataset $f(x, y)$ will be displayed as a 2D dataset even in 3D (try loading `data/2d000.vtk`), but we might want to see it in 3D by displaying the elevation $z = f(x, y)$
- volumetric view – not available for all VTK datasets (available, among others, for Structured Points and for UnstructuredGrid with connectivity provided)

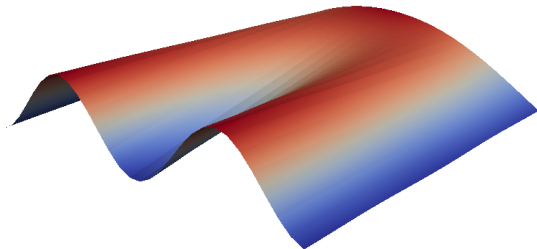
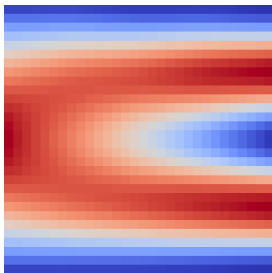
Filters are functional units that process the data to generate, extract, or derive additional features. The filter connections form a **visualization pipeline**

Last time I counted there were 146 filters. One can add new filters with python scripting

- ▶ Check out “Filters” in the menu; some are found in the toolbar
- ▶ List of filters <http://bit.ly/ZX5u2q> with documentation

Simple filter to visualize a 2D dataset in 3D

- Load the file `data/2d000.vtk` that samples the 2D function $f(x, y) = (1 - y) \sin(\pi x) + y \sin^2(2\pi x)$, where $x, y \in [0, 1]$, on a 30^2 grid
- Highlight the dataset in the pipeline browser and apply the WarpByScalar filter
- Change to 3D view, edit the offset factor to **reproduce the 3D view below**



Toolbar filters

- **Calculator** evaluates a user-defined expression on a per-point or per-cell basis.
- **Contour** extracts user-defined points, isocontours/isosurfaces from a scalar field.
- **Clip** removes all geometry on one side of a user-defined plane.
- **Slice** intersects the geometry with a plane. The effect is similar to clipping except that all that remains is the geometry where the plane is located.
- **Threshold** extracts cells that lie within a specified range of a scalar field.
- **Extract Subset** extracts a subset of a grid by defining either a volume of interest or a sampling rate.
- **Glyph** places a glyph on each point in a mesh. The glyphs may be oriented by a vector and scaled by a vector or scalar.
- **Stream Tracer** seeds a vector field with points and then traces those seed points through the steady state vector field.
- **Warp By Vector** displaces each point in a mesh by a given vector field.
- **Group Datasets** combines the output of several pipeline objects into a single multi-block dataset.
- **Extract Level** extracts one or more items from a multi-block dataset.

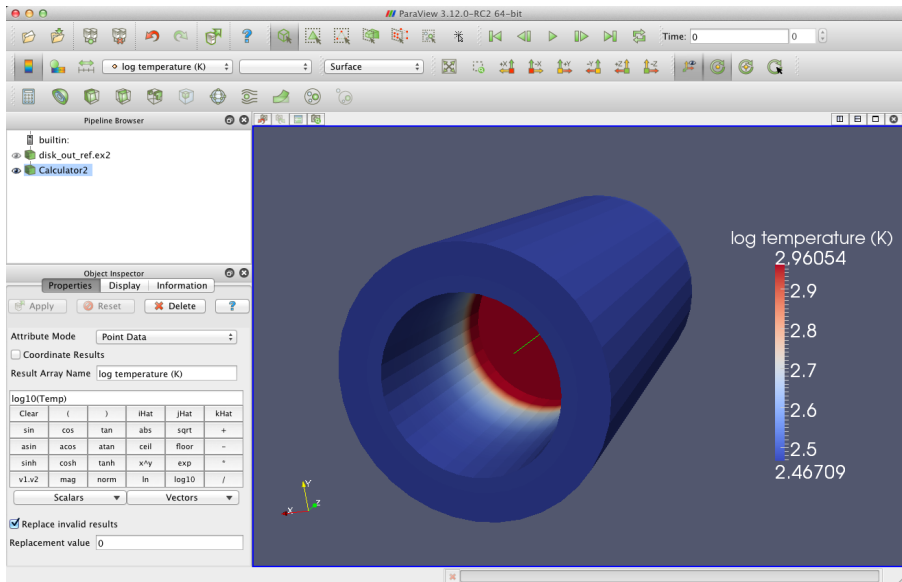
Calculator

- Load one of the datasets, e.g. `data/disk_out_ref.ex2` (load *temperature, velocity, pressure*), and try to visualize individual variables: Pres, Temp, V
- Click on “Toggle Colour Legend Visibility” to see the temperature range
- Now apply **Calculator** filter to display $\log_{10}(\text{Temp})$ – see the next slide
 - ▶ can also try to visualize Pres/Temp, $\text{mag}(\text{V})$
 - ▶ dropdown menus “Scalars” and “Vectors” will help you enter variables
 - ▶ the “?” button is surprisingly useful
- You can change visibility of each object in the pipeline browser by clicking on the eyeball icon next to it

Calculator

- Load one of the datasets, e.g. `data/disk_out_ref.ex2` (load *temperature, velocity, pressure*), and try to visualize individual variables: Pres, Temp, V
- Click on “Toggle Colour Legend Visibility” to see the temperature range
- Now apply **Calculator** filter to display $\log_{10}(\text{Temp})$ – see the next slide
 - ▶ can also try to visualize Pres/Temp, $\text{mag}(\text{V})$
 - ▶ dropdown menus “Scalars” and “Vectors” will help you enter variables
 - ▶ the “?” button is surprisingly useful
- You can change visibility of each object in the pipeline browser by clicking on the eyeball icon next to it

Calculator



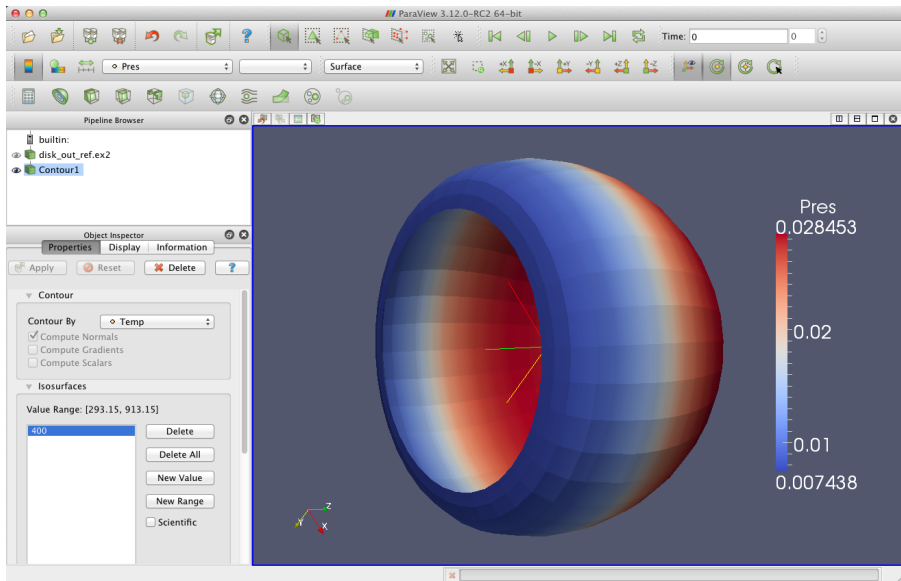
Contour

- Delete **Calculator** from the pipeline browser, load **Contour**
- Create an isosurface where the temperature is 400 K and colour it with pressure – see the next slide
- Now delete the isosurface at 400K and draw two isosurfaces (300K and 800K) and colour them with temperature (add the colour legend to distinguish between the two temperatures)
- Switch to the Wireframe view to see both surfaces clearly

Contour

- Delete **Calculator** from the pipeline browser, load **Contour**
- Create an isosurface where the temperature is 400 K and colour it with pressure – see the next slide
- Now delete the isosurface at 400K and draw two isosurfaces (300K and 800K) and colour them with temperature (add the colour legend to distinguish between the two temperatures)
- Switch to the Wireframe view to see both surfaces clearly

Contour



Creating a visualization pipeline

You can apply one filter to the data generated by another filter

Delete all previous filters, start with the original data from `data/disk_out_ref.ex2`, or just press “Disconnect” and reload the data

1. Apply **Clip** filter to the data: rotate, move the clipping plane, select variables to display, make sure there are data points inside the object (easy to see with points/wireframe, uncheck “Show Plane”)
2. Delete **Clip**, now apply Filters → Alphabetical → **Extract Surface**, and then add **Clip** to the result of **Extract Surface** ⇒ the dataset is now hollow (use wireframe/surface)

Creating a visualization pipeline

You can apply one filter to the data generated by another filter

Delete all previous filters, start with the original data from `data/disk_out_ref.ex2`, or just press “Disconnect” and reload the data

1. Apply **Clip** filter to the data: rotate, move the clipping plane, select variables to display, make sure there are data points inside the object (easy to see with points/wireframe, uncheck “Show Plane”)
2. Delete **Clip**, now apply Filters → Alphabetical → **Extract Surface**, and then add **Clip** to the result of **Extract Surface** ⇒ the dataset is now hollow (use wireframe/surface)

Multiview: several variables side by side

- Start with original data (`data/disk_out_ref.ex2`), load all variables
- Add the **Clip** filter, uncheck “Show Plane” in the object inspector, click “Apply”
- Colour the surface by **pressure** by changing the variable chooser in the toolbar from “Solid Colour” to “Pres”
- Press “Split horizontal”, make sure the view in the right is active (has a blue border around it)
- Turn on the visibility of the clipped data by clicking the eyeball next to Clip in the pipeline browser
- Colour the surface by **temperature** by changing the toolbar variable chooser from “Solid Colour” to “Temp” – see the next slide
- To link the two views, right click on one of the views and select “Link Camera...”, click in a second view, and try moving the object in each view
- Can add colourbars to either view by clicking “Toggle Colour Legend Visibility”, try moving colourbars around
- To unlink, go to Tools -> Manage Links, delete the camera link in question

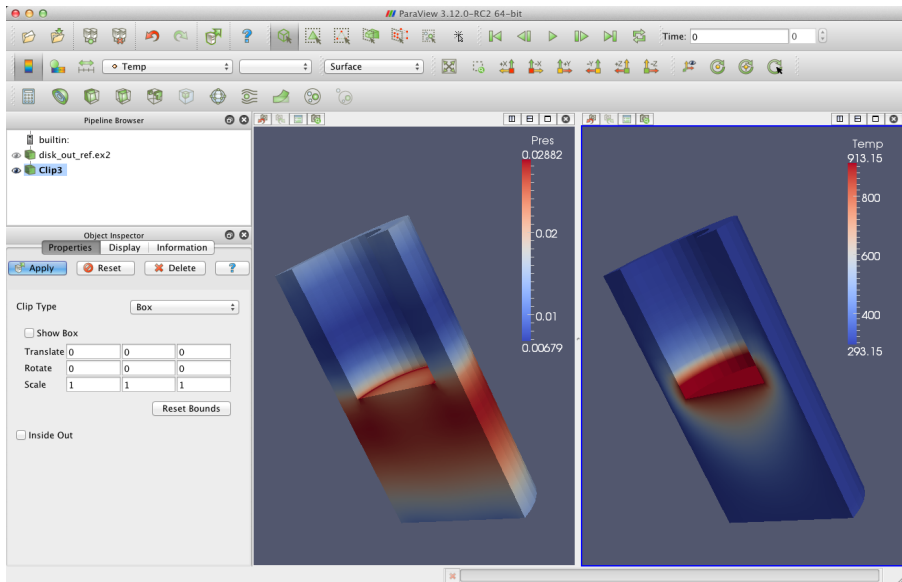
Multiview: several variables side by side

- Start with original data (`data/disk_out_ref.ex2`), load all variables
- Add the **Clip** filter, uncheck “Show Plane” in the object inspector, click “Apply”
- Colour the surface by **pressure** by changing the variable chooser in the toolbar from “Solid Colour” to “Pres”
- Press “Split horizontal”, make sure the view in the right is active (has a blue border around it)
- Turn on the visibility of the clipped data by clicking the eyeball next to Clip in the pipeline browser
- Colour the surface by **temperature** by changing the toolbar variable chooser from “Solid Colour” to “Temp” – see the next slide
- To link the two views, right click on one of the views and select “Link Camera...”, click in a second view, and try moving the object in each view
- Can add colourbars to either view by clicking “Toggle Colour Legend Visibility”, try moving colourbars around
- To unlink, go to Tools -> Manage Links, delete the camera link in question

Multiview: several variables side by side

- Start with original data (`data/disk_out_ref.ex2`), load all variables
- Add the **Clip** filter, uncheck “Show Plane” in the object inspector, click “Apply”
- Colour the surface by **pressure** by changing the variable chooser in the toolbar from “Solid Colour” to “Pres”
- Press “Split horizontal”, make sure the view in the right is active (has a blue border around it)
- Turn on the visibility of the clipped data by clicking the eyeball next to Clip in the pipeline browser
- Colour the surface by **temperature** by changing the toolbar variable chooser from “Solid Colour” to “Temp” – see the next slide
- To link the two views, right click on one of the views and select “Link Camera...”, click in a second view, and try moving the object in each view
- Can add colourbars to either view by clicking “Toggle Colour Legend Visibility”, try moving colourbars around
- To unlink, go to Tools -> Manage Links, delete the camera link in question

Multiview: several variables side by side



Vector visualization: streamlines and glyphs

1. Start with the original data from `data/disk_out_ref.ex2`, load velocity, Temp
2. Add the **Stream Tracer** filter, set Radius = 10 (of sphere with tracer points), play with Number Of Points, Maximum Streamline Length
3. Add shading and depth cues to streamlines: Filters → Alphabetical → **Tube** (could be also called Generate Tubes)
4. Add glyphs to streamlines to show the orientation and magnitude:
 - ▶ select StreamTracer in the pipeline browser
 - ▶ add the **Glyph** filter to StreamTracer
 - ▶ in the object inspector, change the Vectors option (second from the top) to "V"
 - ▶ in the object inspector, change the Glyph Type option (third from the top) to "Cone"
 - ▶ hit "Apply"
 - ▶ colour the glyphs with the "Temp" variable – see the next slide
5. Now try displaying "V" glyphs directly from data, can colour them using different variables ("Temp", "V")

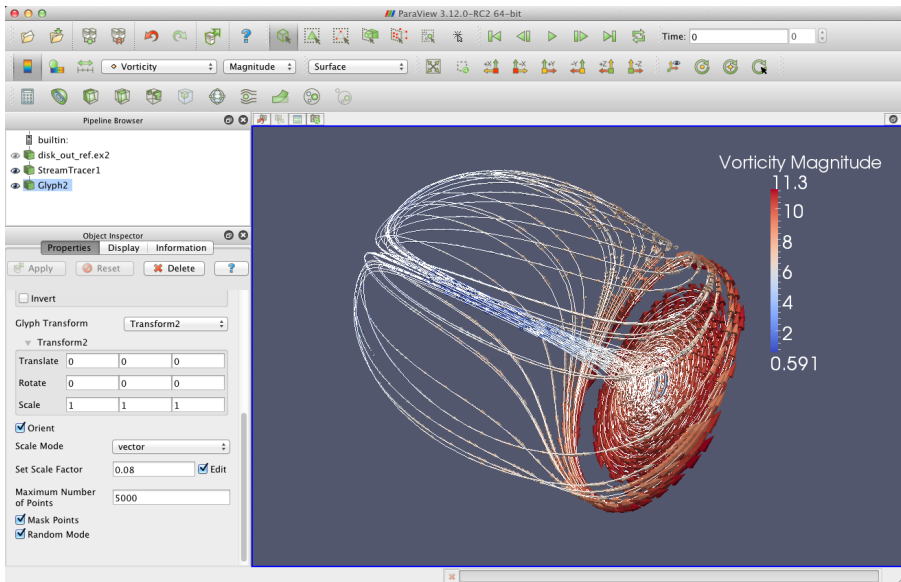
Vector visualization: streamlines and glyphs

1. Start with the original data from `data/disk_out_ref.ex2`, load velocity, Temp
2. Add the **Stream Tracer** filter, set Radius = 10 (of sphere with tracer points), play with Number Of Points, Maximum Streamline Length
3. Add shading and depth cues to streamlines: Filters → Alphabetical → **Tube** (could be also called Generate Tubes)
4. Add glyphs to streamlines to show the orientation and magnitude:
 - ▶ select StreamTracer in the pipeline browser
 - ▶ add the **Glyph** filter to StreamTracer
 - ▶ in the object inspector, change the Vectors option (second from the top) to “V”
 - ▶ in the object inspector, change the Glyph Type option (third from the top) to “Cone”
 - ▶ hit “Apply”
 - ▶ colour the glyphs with the “Temp” variable – see the next slide
5. Now try displaying “V” glyphs directly from data, can colour them using different variables (“Temp”, “V”)

Vector visualization: streamlines and glyphs

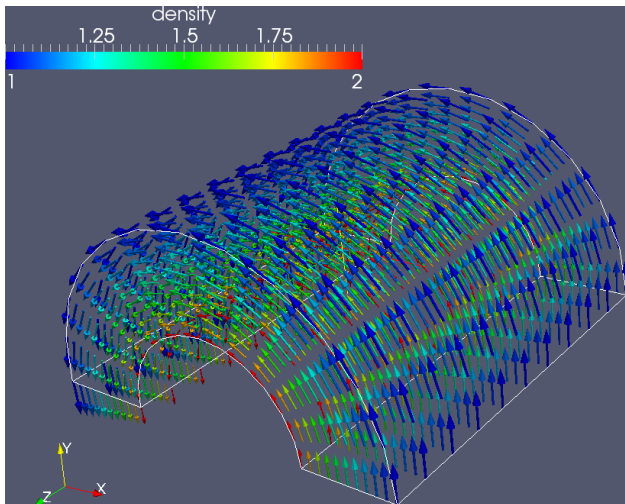
1. Start with the original data from `data/disk_out_ref.ex2`, load velocity, Temp
2. Add the **Stream Tracer** filter, set Radius = 10 (of sphere with tracer points), play with Number Of Points, Maximum Streamline Length
3. Add shading and depth cues to streamlines: Filters → Alphabetical → **Tube** (could be also called Generate Tubes)
4. Add glyphs to streamlines to show the orientation and magnitude:
 - ▶ select StreamTracer in the pipeline browser
 - ▶ add the **Glyph** filter to StreamTracer
 - ▶ in the object inspector, change the Vectors option (second from the top) to “V”
 - ▶ in the object inspector, change the Glyph Type option (third from the top) to “Cone”
 - ▶ hit “Apply”
 - ▶ colour the glyphs with the “Temp” variable – see the next slide
5. Now try displaying “V” glyphs directly from data, can colour them using different variables (“Temp”, “V”)

Vector visualization: streamlines and glyphs



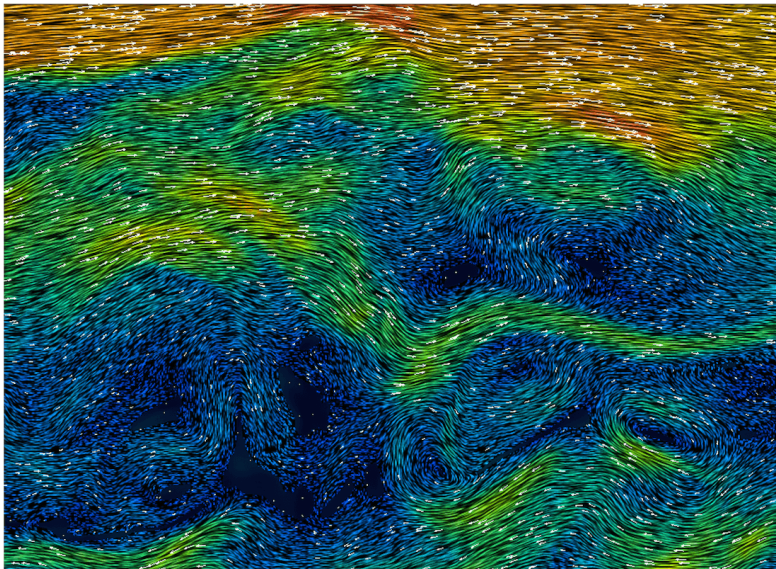
Exercise: vectors

Load `data/halfCylinder.vts` and display the velocity field as arrows, colouring them by density – try to reproduce the view below

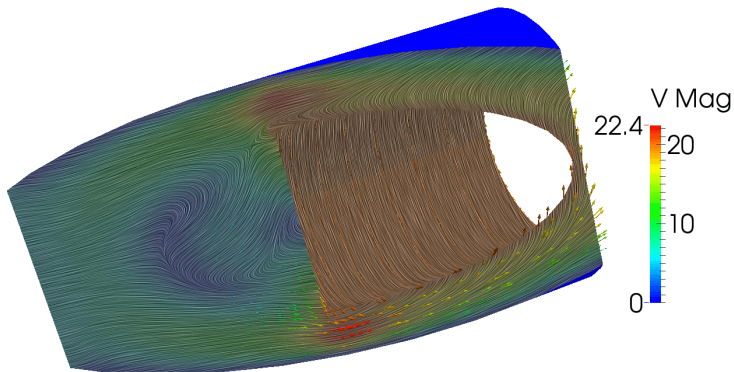


Line Integral Convolution representation

Details at http://www.paraview.org/Wiki/ParaView/Line_Integral_Convolution



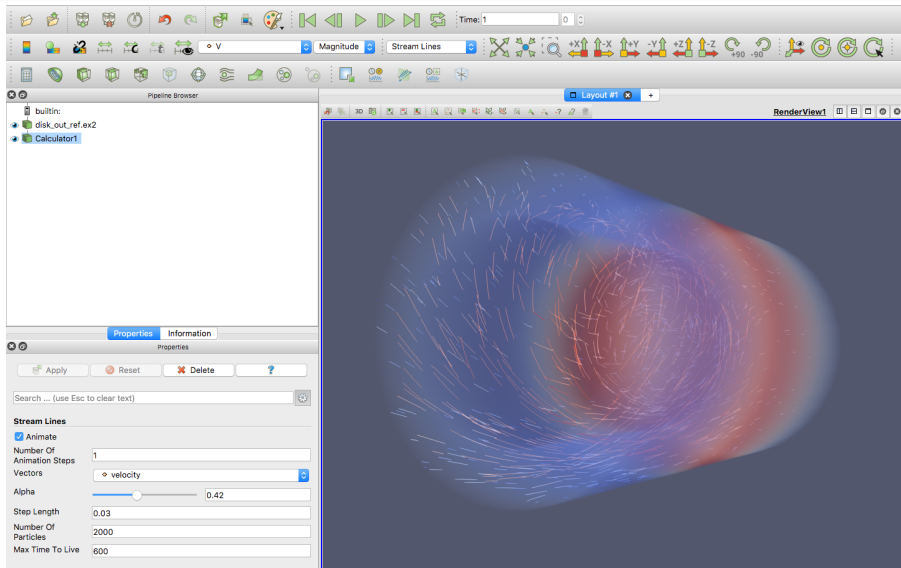
Line Integral Convolution in ParaView



- From Tools → Manage Plugins load *Surface LIC plugin*
- Load `data/disk_out_ref.ex2` or `data/halfCylinder.vts`
- Apply a filter to see its interior (required step for `data/halfCylinder.vts`)
- Switch to *Surface LIC* representation in the drop-down menu
- In properties make sure to select the velocity variable for Surface LIC
- Play with the number of steps and individual step sizes, adjust colour

Stream Lines representation (live drawing)

Details at <http://bit.ly/2NFNcvQ>



Quick and dirty input format: 3D data as columns

- `data/tabulatedPoints.txt` contains 100 random points, with each line storing *x, y, z, scalar* of a point
- Import it into ParaView, apply the **Table To Points** filter, making sure to edit the fields (X/Y/Z Columns)
- Apply the **Glyph** filter to view points as spheres, colour them by *scalar*
- No implied topology here!
- You can optionally pass the points through the **Delaunay 3D** filter, followed by **Extract Edges**, followed by **Tube**

-
- `data/tabulatedGrid.txt` contains 1000 points representing a 10^3 Cartesian mesh, with each line storing *x, y, z, scalar* of a point
 - Import it into ParaView, apply the **Table To Structured Grid** filter, making sure to edit the fields (Whole Extent 0 to 9 in each dimension, X/Y/Z Columns)
 - The data must have some implied topology for this filter to work!

Not recommended for large datasets: waste of disk storage and bandwidth!

- `tabulatedPoints.txt` is 6231 bytes vs. 1600 bytes in single-precision binary
- `tabulatedGrid.txt` is 20,013 bytes vs. 4000 bytes in single-precision binary

Word of caution

- Many visualization filters transform structured grid data into unstructured data (e.g. Clip, Slice)
- Memory footprint and CPU load can grow very quickly, e.g. clipping 400^3 to 150 million cells can take ~ 1 hour on a single CPU \Rightarrow might want to run in distributed mode

Python Calculator filter

https://www.paraview.org/Wiki/Python_Calculator

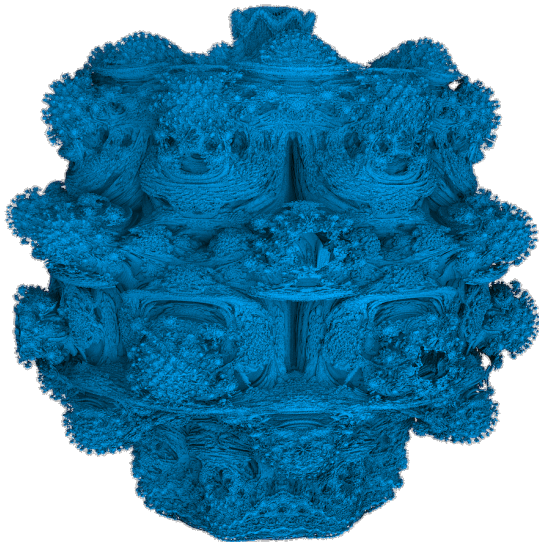
- To calculate vorticity, pick a vector field, enter `curl(V)`, call it **vorticity**
- Supported functions on arrays: `abs()`, `cross()`, `curl()`, `det()`, `dot()`, `eigenvalue()`, `eigenvector()`, `global_mean()`, `global_max()`, `global_min()`, `gradient()`, `inverse()`, `laplacian()`, `ln()`, `log10()`, `max()`, `min()`, `mean()`, `mag()`, `norm()`, `strain()`, `trace()`, `vorticity()`

More filter functionality

- Can merge several existing filters into a *custom filter*
http://www.paraview.org/Wiki/ParaView/Custom_Filters
Tools → **Create Custom Filter** and edit its input, output and properties
- Can script filters in Python
http://www.paraview.org/Wiki/Python_Programmable_Filter
Filters → **Alphabetical** → **Programmable Filter**
(more on general scripting later today)
- Can write new filters as plugins, compile them as shared libraries with the same version of ParaView they are expected to be deployed with
http://www.paraview.org/Wiki/ParaView/Plugin_HowTo

Exercise: try out something more complex

- Power-8 **Mandelbulb** is a 3D fractal
- Load the file `mandelbulb300.nc` (sampled at 300^3)
- If your computer has enough memory, you can try loading `mandelbulb800.nc` (sampled at 800^3)
- Let's do this:
 1. check the grid size and the file size: do these agree?
 2. try slices, volumetric rendering, isosurfaces
 3. try to recreate the picture on the right (at a lower resolution): pay attention to the **lights** and **shadows**



Exercise: 3D optimization

data/stvol.nc contains a discretized scaled variant of the 3D Styblinski-Tang function inside a unit cube ($x_i \in [0, 1]$), built with codes/optimization.c

$$f(x_1, x_2, x_3) = \frac{1}{2} \sum_{i=1}^3 (\xi_i^4 - 16\xi_i^2 + 5\xi_i), \text{ where } \xi_i \equiv 8(x_i - 0.5)$$

Let's answer the following questions:

1. What is the size of the grid? Does it agree with the size of the file?
 2. Find the approximate location of the *global minimum* of $f(x_1, x_2, x_3)$ using visual techniques (slices, isosurfaces, thresholds, volume renderings, etc.)
- Note: you can find the exact coordinates of the global minimum by using Filters -> Statistics -> **Descriptive Statistics**, clicking Apply, and sorting points in order of increasing $f(x,y,z)$