

Scientific visualization with ParaView

Part 2

Alex Razoumov
alex.razoumov@westdri.ca



**Digital Research
Alliance** of Canada



slides, data, codes at <https://folio.vastcloud.org/winterseries>

- the link will download a file `paraview.zip` (~30MB)
- unpack it to find `codes/`, `data/` and `slides{1,2}.pdf`
- command line: `wget <link> -O paraview.zip`



install ParaView 6.0.x on your laptop from <http://www.paraview.org/download>

EXPORTING SCENES (PRE-COMPUTED POLYGONS)

ParaView Glance

<https://kitware.github.io/paraview-glance>

PV Glance is an open-source **standalone** web app for **in-browser 3D sci-vis**

- very easy to use, ideal for sharing pre-built 3D scenes via the web
- no server ⇒ up to medium-size data (server support planned in future versions)
- interactive manipulation of pre-computed polygons
 - volumetric images, molecular structures, geometric objects, point clouds
- written in JavaScript and vtk.js + can be further customized with vtk.js and ParaViewWeb for custom web and desktop apps
- source and installation instructions <https://github.com/kitware/paraview-glance>

-
1. Create a visualization with several layers, make **all layers visible in the pipeline**
 2. Many options in File → Export Scene... ⇒ save as VTKJS to your laptop
 3. Open <https://kitware.github.io/paraview-glance/app>
 4. Drag the newly saved file to the dropzone on the website
 5. Interact with individual layers in 3D: **rotate and zoom, change visibility, representation, variable, colourmap, opacity**

Automatically load a visualisation into Glance

<https://discourse.paraview.org/t/customise-pv-glance/2831>

- Use the query syntax `GLANCEAPPURL?name=FILENAME&url=FILEURL` to pass `name` and `url` to the web server
- E.g. using ParaView Glance website
`https://kitware.github.io/paraview-glance/app?name=sineEnvelope.vtkjs&url=https://raw.githubusercontent.com/razoumov/publish/master/data/sineEnvelope.vtkjs`
 - shortened to `https://bit.ly/2KtPWNf`
- You can parse long strings with JavaScript (next slide)

Embed into a website with an iframe (embed.html)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sine envelope function</title>
  </head>
  <body>
    <h1>3D sine envelope function</h1>

    <script>
      var app = "https://kitware.github.io/paraview-glance/app";
      var dir = "https://raw.githubusercontent.com/razoumov/publish/master/data/";
      var file = "sineEnvelope.vtkjs";
      document.write("<iframe src='" + app + "?name=" + file + "&url=" +
        dir + file +
        "' id='iframe' width='1100' height='900'></iframe>");
    </script>

    <p>More stuff in here</p>
  </body>
</html>
```

- JavaScript here only to parse long strings

ANIMATION IN PARAVIEW

Animation methods

1. **Keyframe animation: animate any property of any pipeline object**
 - easily create snazzy animations, somewhat limited in what you can do
 - in Time Manager: select object, select property, create a new track with “+”, double-click the track to edit it, press “▶”
 - can combine multiple tracks / timelines
 - examples: load the file `data/sineEnvelope.nc`
 - 1.1 animate a slice
 - 1.2 animate a contour

Animation methods

1. **Keyframe animation: animate any property of any pipeline object**
 - easily create snazzy animations, somewhat limited in what you can do
 - in Time Manager: select object, select property, create a new track with "+", double-click the track to edit it, press "▶"
 - can combine multiple tracks / timelines
 - examples: load the file `data/sineEnvelope.nc`
 - 1.1 animate a slice
 - 1.2 animate a contour
2. **Use ParaView's ability to recognize a sequence of similar files**
 - time animation only, but very convenient
 - try loading `data/2d*.vtk` sequence and animating it: visualize one frame and then press "▶"

Animation methods

1. **Keyframe animation: animate any property of any pipeline object**
 - easily create snazzy animations, somewhat limited in what you can do
 - in Time Manager: select object, select property, create a new track with "+", double-click the track to edit it, press "▶"
 - can combine multiple tracks / timelines
 - examples: load the file `data/sineEnvelope.nc`
 - 1.1 animate a slice
 - 1.2 animate a contour
2. **Use ParaView's ability to recognize a sequence of similar files**
 - time animation only, but very convenient
 - try loading `data/2d*.vtk` sequence and animating it: visualize one frame and then press "▶"
3. **Script your animation in Python (covered in the Scripting section)**
 - steep learning curve, very powerful, can do anything you can do in the GUI
 - typical usage scenario: generate one frame per input file
 - a simpler exercise without input files: see next slide

Exercise: animating function growth

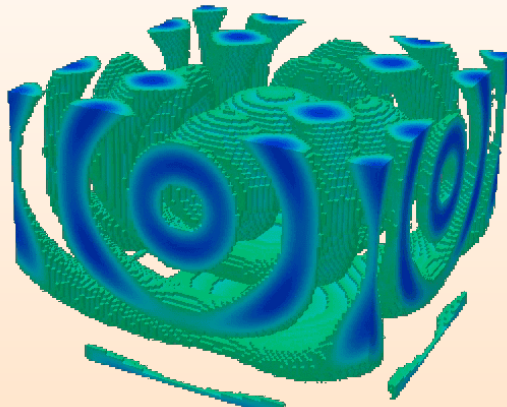
- ➡ 3D sine envelope wave function defined inside a unit cube ($x_i \in [0, 1]$)

$$f(x_1, x_2, x_3) = \sum_{i=1}^2 \left[\frac{\sin^2 \left(\sqrt{\xi_{i+1}^2 + \xi_i^2} \right) - 0.5}{\left[0.001(\xi_{i+1}^2 + \xi_i^2) + 1 \right]^2} + 0.5 \right], \text{ where } \xi_i \equiv 15(x_i - 0.5)$$

- ➡ Reproduce the movie on the screen

<https://youtu.be/Oc3DAJvArIU>

or `hidden/growth.mp4` on presenter's laptop



Exercise: animating function growth (cont.)

To visualize a single frame of the movie:

1. load `data/sineEnvelope.nc` (discretized on a 100^3 grid)
2. apply `Threshold` keeping only data from 1.2 to 2
3. apply `Clip`: origin $O = (49.5, 15, 49.5)$, normal $N = (0, -1, 0)$
4. colour by the right quantity

Two possible solutions:


1. bring up **Time Manager** to animate `Clip's O2` from 0 to 99, for best results save animation as a sequence of PNG files
2. covered in `Scripting`: `Start/Stop Trace` to record the workflow, save the corresponding **Python script**, enclose **parts of it** into a loop changing `O2` from 0 to 99 and writing a series of PNG screenshots, run it inside `ParaView` to produce 100 frames

in either case, merge PNGs using a 3rd-party tool, e.g.

```
ffmpeg -r 30 -i frame%04d.png -c:v libx264 -pix_fmt yuv420p \
-vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" movie.mp4
```

Camera animation in the GUI

Good introductory resource <https://docs.paraview.org/en/latest/UsersGuide/animation.html>

1. Start with any static visualization and adjust the view to your liking
2. Click 'Adjust Camera' icon  (one of the left-side icons on top of the vis window)
 - make sure that Center of Rotation = Camera Focal Point
3. Bring up Time Manager (or erase all previous timelines) and then do the following:

3a. Circular orbit

- select Camera - Follow Path
- click "+" to create a new timeline
- double-click on the white (or black) timeline
- click Create Orbit, accept the defaults
- adjust the number of frames

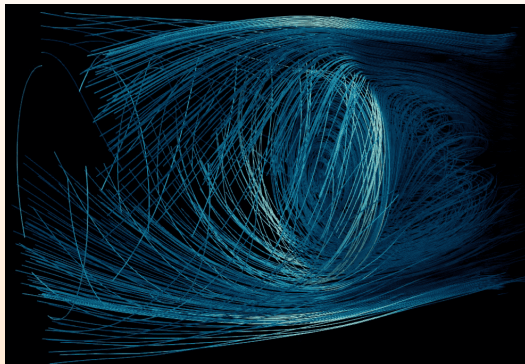
3b. Custom path

- select Camera - Follow Path
- click "+" to create a new timeline
- double-click on the white (or black) timeline
- double-click on Path... in the right column
- click on Camera Position
 - a yellow path with spheres will appear
 - drag the spheres around
- also can change Camera Focus and Up Direction
- click Apply in the Animation Keyframes window

4. Click "▶"

Animating stationary flow: streamlines through a slice

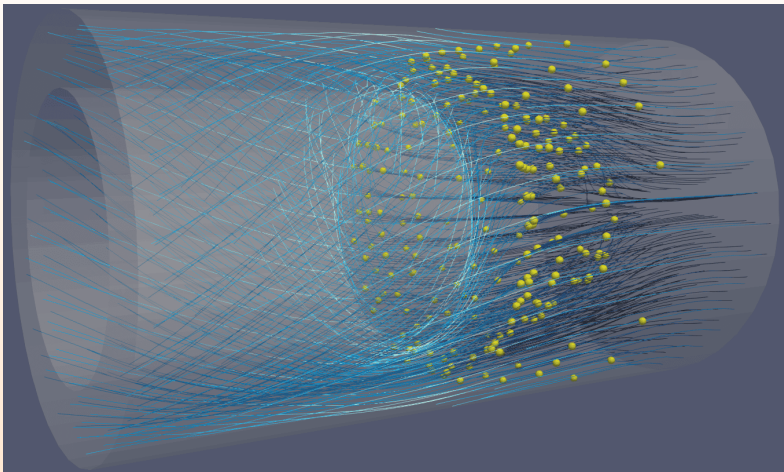
- There is no temporal dimension in this dataset, yet we want to produce an animation with streamlines
- This is one possible direction – what other creative ideas can we explore?
- <https://youtu.be/67qTsW8-kKk> or [hidden/radialSlice.mp4](#) on presenter's laptop
- <https://youtu.be/2uSRrZ6oW0o> or [hidden/xySlice.mp4](#) on presenter's laptop



Animating stationary flow: streamlines through a slice (cont.)

1. Load `disk_out_ref.ex2` making sure to load velocity
2. Draw a radius-z plane slice through the center
 - origin $O = (0, 0, 0)$
 - normal $N = (1, 0, 0)$
3. Stream Tracer With Custom Source
 - `input = disk_out_ref.ex2`
 - `seedSource = Slice1`
4. Tube filter with $r = 0.015$
5. Time Manager: animate Slice's O_0 from -1 to 1 (full range [-5.75,5.75])
6. Use 100 frames, black background, blue2cyan colourmap, colour with vorticity
7. Unselect "Show Plane"
8. Save animation as PNGs, encode at 10 fps

Animating a stationary flow: time contours



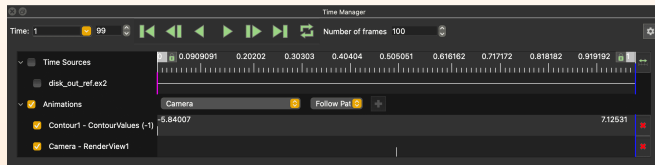
<https://youtu.be/cTBKeybfFyk> or [hidden/timeContours.mp4](#) on presenter's laptop

Animating a stationary flow: time contours (cont.)

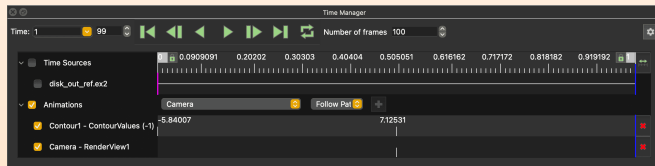
1. Start with the streamtracer lines, however drawn
2. Apply a Countour filter to the output of Streamtracer
 - contour by Integration Time
 - probe the range of values that works best
3. Apply Glyph filter to the output of Countour
4. Time Manager: animate Contour | Isosurfaces
5. This video was recorded with 2000 frames at 60 fps
 - such high resolution only for the final production video
 - debugging animation with 100 frames is perfectly Ok

Exercise: several timelines in one animation

1. Start with the previous integration-time-contour animation from $t = 0$ to $t = 1$
2. Add the second timeline to the animation: Camera - Follow Path from $t = 0.5$ to $t = 1$ (while the first animation is still playing for its second half)

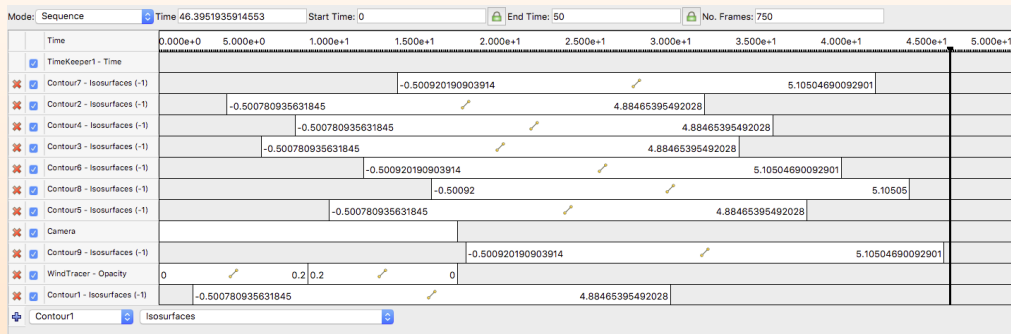


3. Now complete integration-time-contour animation before rotation



Combining many timelines in one animation (cont.)

- In principle, you can add as many timelines to the animation as you like, each with its own time interval and variables
- Here is an example from WestGrid's 2017 *Visualize This* competition submission by Nadya Moisseeva (UBC)



<https://youtu.be/eT8tj8BoYfg> or [hidden/complexAnimation.mp4](#) on presenter's laptop

PYTHON SCRIPTING IN PARAVIEW

Standard Python scripting in ParaView

- Why use scripting?

- automate mundane or repetitive tasks, e.g., making frames for a movie
- document and store your workflow, perhaps share it with colleagues
- use ParaView on clusters from the command line and/or via batch jobs

1. In the GUI: View | Python Shell opens a Python interpreter

- write or paste your script there
- use the button to run an external script from a file

2. `/usr/local/bin/ /Applications/Paraview*.app/Contents/bin/ C:\Program Files\ParaView*\bin\pvpython` will give you a Python shell connected to a ParaView server (local or remote) without the GUI

3. `/usr/local/bin/ /Applications/Paraview*.app/Contents/bin/ C:\Program Files\ParaView*\bin\pvbatch --force-offscreen-rendering script.py` is a serial (on some machines parallel) application using a local ParaView server  **make sure to save your visualization**

4. `/usr/local/bin/ /Applications/Paraview*.app/Contents/bin/ C:\Program Files\ParaView*\bin\paraview --script=codes/script.py` to start ParaView GUI and auto-run the script

Less obvious ways to use Python scripting in ParaView

5. Python Calculator Filter (vs. regular Calculator Filter) to create new data arrays

- treats data as NumPy arrays
- can use short syntax `0.5*rho*mag(V)**2` or long syntax `0.5*inputs[0].PointData['rho']*inputs[0].PointData['V']**2`
- can access data from multiple time steps, multiple inputs, point vs. cell data
- can control array precision / type, but not the underlying discretization
- I don't use it myself; I typically use the regular Calculator or the Programmable Filter

6. Programmable Source / Filter to create new geometry

- to create new discretizations / VTK objects
- useful for creating custom objects from scratch, e.g. projecting a volume onto a plane
- an alternative way to build your own custom file reader (if format not supported out of the box)

7. Python state files: more robust than `*.pvsm` files for complex workflows

First script

1. Bring up View | Python Shell
2. “Run Script” codes/displaySphere.py

displaySphere.py

```
from paraview.simple import *

sphere = Sphere()           # create a sphere pipeline object
print(sphere.ThetaResolution) # print one of the attributes of the sphere
sphere.ThetaResolution = 16

Show()                      # make it visible in the view
Render()
```

3. Can always get help from the command line

```
help(paraview.simple)  # will display a help page on paraview.simple module
help(Sphere)
help>Show)
help(sphere)           # to see this object's attributes
dir(paraview.simple)   # to see everything inside this module's namespace
```

Using filters

1. Reset ParaView
2. “Run Script” codes/displayWireframe.py

displayWireframe.py

```
from paraview.simple import *

sphere = Sphere(ThetaResolution=36, PhiResolution=18)

wireframe = ExtractEdges(Input=sphere)    # apply Extract Edges to sphere

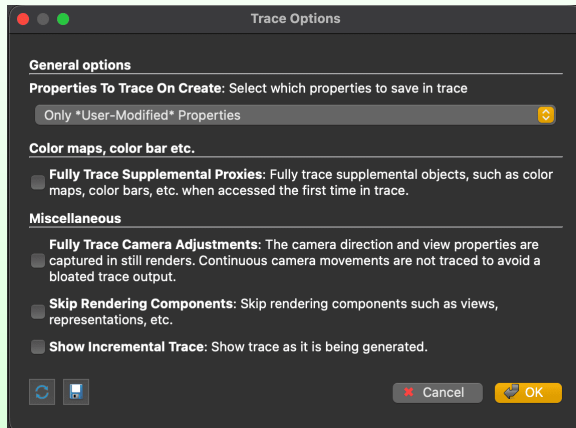
Show()                                     # make the last object visible in the view
Render()
```

3. Try replacing Show() with Show(sphere)
4. Also try replacing Render() with
SaveScreenshot('/path/to/wireframe.png') and running via pvbatch

Trace tool

Generate Python code from GUI operations

- Tools | Start / Stop Trace



Passing information down the pipeline

... and other useful high-level workflow functions

- `GetSources()` gets a list of pipeline objects
- `GetActiveSource()` gets the active object
- `SetActiveSource()` sets the active object
- `GetRepresentation()` returns the *view representation* for the active pipeline object and the active view
- `GetActiveCamera()` returns the active camera for the active view
- `GetActiveView()` returns the active view
- `CreateRenderView()` creates standard 3D render view
- `ResetCamera()` resets the camera to include the entire scene but preserve orientation (or does nothing ☺)

There is quite a bit of overlap between these two:

```
help(GetActiveCamera())  
help(GetActiveView())
```

Camera animation with scripting

1. Let's load `data/sineEnvelope.nc` and draw an isosurface at $\rho = 0.15$
2. Ensure the focal point is at the dataset center to keep the object in view
 - in a 3D scene, the focal point is the specific coordinate in space that the camera is looking at and rotating around
 - if not \Rightarrow `ResetCamera()`

```
v1 = GetActiveView()  
print(v1.GetFocalPoint())
```

3. Look up azimuthal rotation

```
camera = GetActiveCamera()  
dir(camera)  
help(camera.Azimuth)
```

4. Rotate by 10° around the view-up vector

```
camera.Azimuth(10)  
Render()
```

Camera animation: full rotation

✎ Can paste longer commands from `clipboard.txt`

5. Do full rotation and save to disk

```
nframes = 360
v1 = GetActiveView()
camera = GetActiveCamera()
for i in range(nframes):
    print(v1.CameraPosition)
    camera.Azimuth(360./nframes)    # rotate by 1 degree
    SaveScreenshot('/path/to/frame%04d'%(i)+'.png')
```

6. Merge all frames into a movie at 30 fps

```
ffmpeg -r 30 -i frame%04d.png -c:v libx264 -pix_fmt yuv420p \
-vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" spin.mp4
```

Camera animation: flying towards the focal point

1. Optionally reset the view manually or with `ResetCamera()`
2. Now let's fly 2/3 of the way towards the focal point

```
initialCameraPosition = v1.CameraPosition[:]    # force a real copy
nframes = 100
for i in range(nframes):
    coef = float(i+0.5)/float(1.5*nframes)    # runs from 0 to 2/3
    print(coef, v1.CameraPosition)
    v1.CameraPosition = [(1.-coef)*a + coef*b) \
        for a, b in zip(initialCameraPosition, v1.CameraFocalPoint)]
    SaveScreenshot('/path/to/out%04d'%(i)+'.png')
```

3. Create a movie

```
ffmpeg -r 30 -i out%04d.png -c:v libx264 -pix_fmt yuv420p \
    -vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" approach.mp4
```

Exercise: write and run a complete off-screen script

1. Mac/Linux/Windows: create a script with standalone ParaView GUI

- use Start/Stop Trace
- load `data/sineEnvelope.nc` and draw an isosurface at $\rho = 0.15$
- save the image as PNG

2. Test-run your script with `pvbatch` on your laptop

```
$ pvbatch --force-offscreen-rendering script.py
```

- **Linux:** `pvbatch` should be in one of your system's `bin` directories
- **Mac:** `pvbatch` should be in `/Applications/ParaView*.app/Contents/bin`
- **Windows:** `pvbatch` does not exist (or so I am told), but you can use `pvpython`
 - you will need to locate it yourself
- those of you with an Alliance account can run this script on one of our HPC clusters with

```
$ module load paraview
```

```
$ pvbatch --force-offscreen-rendering script.py
```

3. Modify the script to create some animation

- try a camera animation from the previous slides, all from the terminal
- try animating the contour's isovalue

Extracting data from VTK objects

Do this from *View* | *Python Shell* or from *pvython* (either shell will work)

```
# codes/extractValues.py
from paraview.simple import *

dir = '/Users/razoumov/training/paraviewWorkshop/data/' # edit the path
data = NetCDFReader(FileName=[dir+'sineEnvelope.nc'])
print(type(data)) # class 'paraview.servermanager.NetCDFReader'

local = servermanager.Fetch(data) # get this data from the server
print(local.GetNumberOfPoints())
print(local.GetDimensions())

# the following are populated with data only after servermanager.Fetch(data) is called
pointInfo = data.GetDataInformation().GetPointDataInformation()
narrays = pointInfo.GetNumberOfArrays()
print(narrays)

for i in range(narrays):
    print(pointInfo.GetArrayInformation(i).GetName())

for i in range(10):
    print(local.GetPoint(i)) # coordinates of the first 10 points
```

Feeding data into arrays for further post-processing

```
vtkArray = local.GetPointData().GetArray('density')
print(vtkArray.GetDataSize(), vtkArray.GetRange())

for i in range(10):
    print(vtkArray.GetValue(i))          # values at the first 10 points -- awkward to use

from vtk.numpy_interface import dataset_adapter as dsa

wrapped = dsa.WrapDataObject(local)    # wrap it in a NumPy-friendly object
flatArray = wrapped.PointData['density']
print(flatArray.shape)
print(flatArray)

import numpy as np

dims = local.GetDimensions()
array3d = np.reshape(flatArray, (dims[2], dims[1], dims[0]))
print(array3d.shape)
print(array3d)
```

Using 3rd-party libraries from ParaView's Python

- `pvpython` includes few common 3rd-party libraries such as `numpy`, `scipy`, `pandas`
- What if you want to use other libraries that were not bundled with ParaView?

Using 3rd-party libraries from ParaView's Python

- `pvpython` includes few common 3rd-party libraries such as `numpy`, `scipy`, `pandas`
- What if you want to use other libraries that were not bundled with ParaView?

1. Let's assume you work on an Alliance cluster; check your ParaView's Python version

```
module load paraview
pvpython    # let's assume it says Python 3.13.2
```

2. Load the closest Python module, create a virtual env. and install your library there

```
module avail python          # python/3.13.2 is one of them
module load python/3.13.2
python -m venv ~/env-astro    # this will install a new virtual environment into ~/astro
source ~/env-astro/bin/activate
python -m pip install --upgrade pip --no-index
python -m pip install --no-index xarray # install an external package into this new environment
```

3. Next time you log in to the cluster, start `pvpython`:

```
module load paraview
pvpython
```

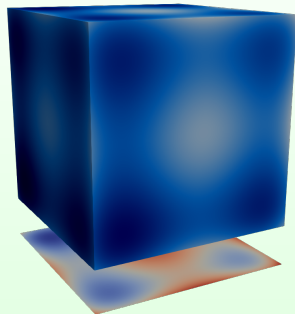
4. Load your new virtual environment directly from Python:

```
venv_path = '/home/user01/env-astro/lib/python3.13/site-packages'
import sys
sys.path.insert(0, venv_path) # activate your new env from the shell
from paraview.simple import *
import xarray                 # this xarray comes from your new virtual environment
```

5. For batch workflows, replace `pvpython` with `pvbatch`

Creating/modifying VTK objects

Let's say we want to plot a projection of a cubic dataset along one of its principal axes, or do some other transformation for which there is no filter



- Calculator / Python Calculator filter cannot modify the geometry ...

Programmable filter

Watch our webinar <https://bit.ly/programmablefilter>

1. Apply Programmable Filter with `OutputDataSetType = vtkUnstructuredGrid`
2. Paste the following code codes/`projectionUnstructured.py` into the filter
(this code was last tested in ParaView 6.0.1)

```
numPoints = inputs[0].GetNumberOfPoints()
side = int(round(numPoints**(1./3.))) # round() in this Python returns float type
layer = side*side
rho = inputs[0].PointData['density'] # 1D flat array
points = vtk.vtkPoints() # create vtkPoints instance, to contain 100^2 points in the projection
proj = vtk.vtkDoubleArray(); proj.SetName('projection') # create the projection array
for i in range(layer): # loop through 100x100 points
    x, y, z, column = inputs[0].GetPoint(i)[0:2], -20., 0.
    for j in range(side):
        column += rho.GetValue(i+layer*j)
    points.InsertNextPoint(x,y,z) # also points.InsertPoint(i,x,y,z)
    proj.InsertNextValue(column) # add value to this point

output.SetPoints(points) # add points to vtkUnstructuredGrid
output.GetPointData().SetScalars(proj) # add projection array to these points

quad = vtk.vtkQuad() # create a cell
output.Allocate(side, side) # allocate space for side^2 'cells'
for i in range(side-1):
    for j in range(side-1):
        quad.GetPointIds().SetId(0,i+j*side)
        quad.GetPointIds().SetId(1,(i+1)+j*side)
        quad.GetPointIds().SetId(2,(i+1)+(j+1)*side)
        quad.GetPointIds().SetId(3,i+(j+1)*side)
    output.InsertNextCell(vtk.VTK_QUAD, quad.GetPointIds())
```

REMOTE AND DISTRIBUTED VISUALIZATION

Visualizing remote data

Details at <https://docs.alliancecan.ca/wiki/ParaView>

If your dataset is on a remote cluster, there are several options:



download data to your desktop and visualize it locally

- limited by the dataset size and your desktop's CPU/GPU + memory



run ParaView remotely on the cluster via X11 forwarding

- your desktop $\xrightarrow{\text{ssh } -X/Y}$ cluster running ParaView



run ParaView remotely via remote desktop (via Open OnDemand or JupyterHub)

- your desktop $\xrightarrow{\text{VNC}}$ cluster running ParaView
- easiest option: use <https://jupyterhub.fir.alliancecan.ca> or similar front-end (described 2 slides from now)
- manual option: you can start a VNC server on an interactive cluster compute node by hand as described in our documentation <https://docs.alliancecan.ca/wiki/VNC>
- serial (default) or parallel (start a remote server + connect from a remote client)



run ParaView in client-server mode

- ParaView client on your desktop \rightleftharpoons serial or parallel ParaView server on larger machine



run ParaView via a GUI-less batch script (interactively or scheduled)

- render server can run with GPU rendering or purely in software
- data/render servers can run on single-core, or across several cores/nodes with MPI
- for interactive GUI work on clusters you should schedule interactive jobs, as opposed to running on the login nodes

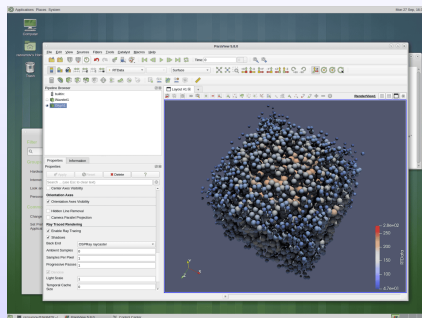
Special remote vis cases

1. **In-situ visualization** = instrumenting a simulation code on the cluster to
 - 1.1 output graphics and/or
 - 1.2 act as on-the-fly server for a visualization frontend (ParaView/VisIt client on your laptop)
 - need to use a special library (ParaView's Catalyst or VisIt's libsim)
 - very advanced topic for another time
2. **Web-based visualization** with data served from another location

ParaView via remote desktop

<https://docs.alliancecan.ca/wiki/ParaView> | “Small-scale interactive” tab

- 1a. On **Fir**, **Rorqual**, or **Narval**: launch a **JupyterLab** instance
- 1b. On **Nibi** or **Trillium**: launch an **Open OnDemand** instance
2. Start a remote desktop
3. In the terminal inside the remote desktop, load the ParaView module
- 4a. For **serial rendering**: start ParaView inside the remote desktop
- 4b. For **multi-core rendering**: start `pvserver` with `mpirun`, start ParaView inside the remote desktop, connect to the local server
5. Load your data and render



Hands-on: ParaView via remote desktop

Let's try this on the training cluster:

`https://jupyter.cass.vastcloud.org`

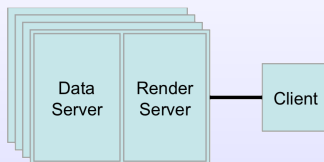
1. choose a username and get a password from the instructor
2. copy data into your remote directory, e.g.

```
$ unzip /project/def-sponsor00/shared/paraview.zip data/sineEnvelope.nc
```

3. try serial rendering
4. try parallel rendering on 4 cores + colour by ProcessID

Taking the next step: distributed client-server ParaView

<https://docs.alliancecan.ca/wiki/ParaView> | “Large-scale interactive” tab



- Local client **on your computer**
- Run `pvserver` on a multi-core server or a distributed cluster
- Alliance’s general-purpose (variety of workloads) clusters: Fir, Nibi, Rorqual, Narval
 - located at SFU, UofWaterloo, École de technologie supérieure (Montreal)
 - specs in our documentation wiki
 - specs at <https://docs.alliancecan.ca/wiki/Fir>
(replace Fir with Nibi or Rorqual or Narval)
 - batch-oriented environment for parallel and serial jobs ⇒ use Slurm scheduler
 - identical software setup https://docs.alliancecan.ca/wiki/Available_software

Question 1: should I use CPUs or GPUs for rendering?

- We can render on GPUs (*hardware acceleration*) or CPUs (*software rendering*) with both interactive and batch visualization
 - On desktops, graphics-focused RTX GPUs have traditionally been faster for rendering graphics
 - in a standard graphics card, a large portion of the chip is dedicated to "Graphics Processing Clusters" to run things like rasterization and texture mapping
- (1) H100-class data-centre GPUs were designed for AI / HPC, not graphics
 - to lower cost, the H100 allows OpenGL/Vulkan pipelines to run on only 2 out of its ~66 thread controllers
⇒ rendering will utilize the card only at ~ 3% efficiency ... not the best use of this very expensive device
⇒ don't use it for visualization
 - (2) On clusters, far more CPU cores than GPUs ⇒ much easier to allocate many CPU cores
 - with enough CPU cores, open-source rendering libraries such as Intel's OSPRay (ray tracing) and OpenSWR / OSMesa (rasterization) have largely closed the performance gap for many visualization tasks
 - (3) Much faster to load *partitioned* data in parallel on 16/32/64/128 CPU cores ⇒ reuse the same cores for parallel rendering, with no data reshuffling!
 - more on I/O and data partitioning later
- Some older NVIDIA GPUs and AMD GPUs can still be used for rendering on clusters, but these are rather corner cases, so we won't focus on them here
 - In this workshop, all exercises assume **CPU rendering**; slides on **GPU rendering** are also included

Question 2: how many CPUs/GPUs do I need?

- How many processors do we need? From *ParaView documentation*:
 - structured (Structured Points, Rectilinear Grid, Structured Grid): one CPU core per ~ 20 million cells
 - unstructured (Unstructured Points, Polygonal Data, Unstructured Grid): one CPU core per ~ 1 million cells
- Your initial bottlenecks will be **physical memory** and **disk read speed**, followed eventually by **CPU/GPU rendering time** \Rightarrow to simplify things, to decide on the number of CPU cores for initial dataset exploration, use the dataset size
 - consider 80 GB dataset (single timestep)
 - base nodes have 128 GB memory with 32 cores \Rightarrow 3.5 GB/core (accounting for the OS, system tools, etc.) \Rightarrow 23 cores for this dataset
 - need to account for filters (and other processing), MPI buffers \Rightarrow minimum 32 cores
 - for comfortable processing with complex filters use 48 – 64 cores
- On large HPC clusters ParaView has been shown to scale to $\sim 10^{12}$ cells (Structured Points) on $\sim 10,000$ cores and beyond
- Always do a scaling study before attempting to visualize large datasets
- It is important to understand the **memory requirements of filters**
 - a typical (structured \rightarrow unstructured) filter increases memory footprint by $\sim 3X$

Important setting: Remote Render Threshold

In ParaView's preferences can set (Render View | Remote/Parallel Rendering Options | Remote Render Threshold) beyond which rendering will be remote

- **default 20MB** ⇒ small rendering will be done on your laptop's GPU, interactive rotation with a mouse will be fast, but anything modestly intensive (under 20MB) will be shipped to your laptop and might be slow
- **0MB** ⇒ all rendering (including rotation) will be remote, so you will be really using the cluster's CPU(s)/GPU(s) for everything
 - good for large data processing
 - not so good for interactivity, especially on a slower connection
- experiment with the threshold to find a suitable value

Hands-on: local client, remote software rendering

Goal: start with interactive client-server, proceed to remote offscreen batch rendering

1. On the cluster start remote parallel ParaView server:

```
$ module load paraview
$ salloc --ntasks=4 --time=0:60:0 --mem-per-cpu=3600      # --account=def-someuser
$ mpirun -np 4 pvserver --force-offscreen-rendering --opengl-window-backend OSMesa
```

- ✚ usually no need to specify the OpenGL window backend, but sometimes the default backend is not determined properly
- ✚ how would you adapt these commands to render on **one CPU**?

2. Wait for it to start waiting for an incoming connection:

```
Waiting for client...
Connection URL: cs://node1.int.cass.vastcloud.org:11111
Accepting connection(s): node1.int.cass.vastcloud.org:11111
```

3. On your computer start SSH port forwarding:

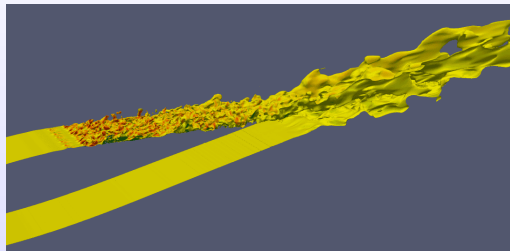
```
$ ssh userXX@cass.vastcloud.org -L 11111:node1:11111  # use the actual compute node
```

- ✚ more details (Linux, MacOS, Windows) at https://docs.alliancecan.ca/wiki/SSH_tunnelling

4. On your computer start ParaView 6.0.x, click  Connect, fill in the connection details, then connect to cs://localhost:11111

Hands-on continued

5. Tools → Start Trace
6. Load and visualize the dataset
 - use one of the four datasets described later in these slides
7. Save the image as a PNG file
8. Tools → Stop Trace
9. Save the generated script as `myscript.py` locally
 - edit it in a text editor, simplify (most generated lines will be setting defaults)
 - provide the correct input / output file paths on the remote system



Hands-on continued

10. Upload the script to the cluster:

```
$ scp myscript.py userXX@cass.vastcloud.org:
```

11. On the cluster try running it via an interactive job:

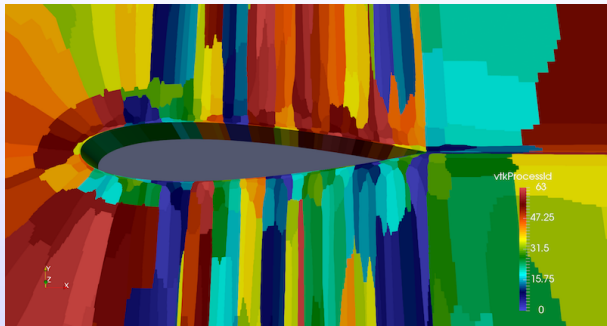
```
$ module load paraview
$ salloc --time=0:60:0 --ntasks=4 --mem-per-cpu=3600      # --account=def-someuser
$ mpirun -np 4 pvbatch --force-offscreen-rendering \
    --opengl-window-backend OSMesa myscript.py
```

and then check the results

12. Once you are happy with the result, write a Slurm job submission script and submit it with sbatch

Check parallel rendering + monitor resource usage

- ParaView processes communicate via MPI
- The variable `vtkProcessID` will show you domain decomposition (more on this later)



See a **more artistic version** of this figure

- Check distributed memory usage: View | Memory Inspector
- Check CPU usage: `srunch --jobid=<jobID> --pty htop`
- Check GPU usage: `srunch --jobid=<jobID> --pty watch -n 1 nvidia-smi`

Next few pages: datasets for remote rendering exercises

Simpler exercise:

1. Create your visualization via interactive client-server using CPU rendering
2. Save your visualization to PNG

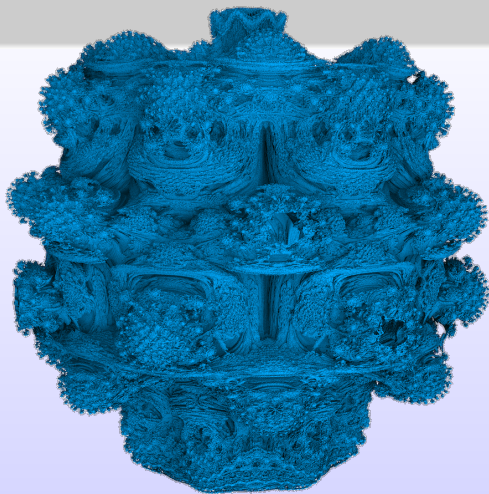
Additional steps (extended exercise):

3. Convert this workflow into a Python script
4. Upload this Python script to the cluster
5. Try running the script inside an interactive (`salloc`) job; debug if needed
6. Once happy with the result, write a Slurm job submission script and submit this rendering as a batch (`sbatch`) job

Dataset 1: Mandelbulb

- Visualize power-8 **Mandelbulb**
- Use the file `mandelbulb800.nc` – sampled at 800^3
- Use 4 - 8 CPU cores on the training cluster via `salloc`
 1. try to recreate the picture on the right: pay attention to the **lights** and **shadows**
 2. use `View` → `Memory Inspector` to keep an eye on memory usage
 3. optionally colour your dataset by `processID`
- Copy the file:

```
$ unzip /project/def-sponsor00/shared/paraview.zip data/mandelbulb800.nc  
$ ls -lh data/mandelbulb800.nc
```

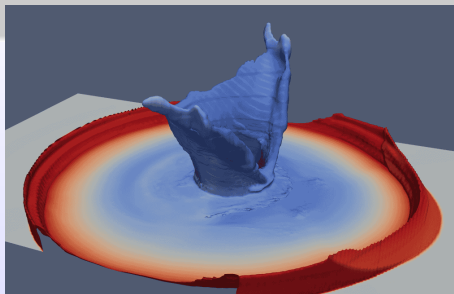


Dataset 2: deep impact dataset

Dataset from IEEE 2018 SciVis Contest

- Dataset from *Deep Water Impact* simulation by John Patchett (LANL) and Galen Gisler (Univ. of Oslo)
 - dataset details [here](#)
 - 269 low-resolution ($460 \times 280 \times 240$) snapshots in time
 - the original simulation is much higher resolution
 - used with permission
- While you could render this dataset in serial, probably best to give it few cores
- Full dataset (115GB in total)
`fir:/project/6003910/razoumov/ieeevis2018-deepWaterImpact/460x280x240`
- Reduced dataset on the training cluster `/project/def-sponsor00/shared/deepImpact`
 - every 10th file in the timeline \Rightarrow 26 files, 12GB
 - use 8 cores on the training cluster
 - load all files, Contour by `v02` (water volume fraction), colour by ρ , Rescale to data range
 - alternatively, visualize `snd` (sound speed)
- To make it easier to navigate to the dataset in ParaView, create a symbolic link:

```
$ mkdir -p ~/data  
$ ln -s /project/def-sponsor00/shared/deepImpact ~/data/deepImpact
```

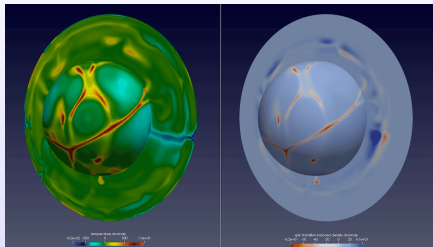


Dataset 3: Earth's mantle convection

Dataset from IEEE 2021 SciVis Contest <https://scivis2021.netlify.app>

- Dataset from *Earth's Mantle Convection* simulation by Hosein Shahnas and Russell Pysklywec (U. of Toronto)

- dataset details at <https://scivis2021.netlify.app/data>
- 251 timesteps on a spherical $180 \times 201 \times 360$ grid
- used with permission



- Full dataset (89GB in total)

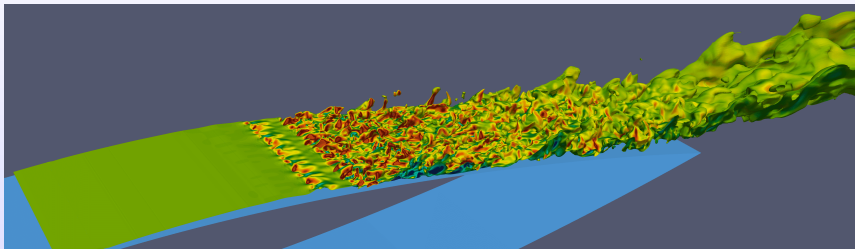
`fir:/project/6003910/razoumov/ieeevis2021-mantleConvection/spherical`

- Reduced dataset on the training cluster `/project/def-sponsor00/shared/mantle`
 - every 10th file in the timeline \Rightarrow 26 files, 11GB
 - you can render this dataset in serial
 - visualize any 3D scalar field, e.g. the *temperature anomaly* or the *spin transition-induced density anomaly*
- To make it easier to navigate to the dataset in ParaView, create a symbolic link:

```
$ mkdir -p ~/data
$ ln -s /project/def-sponsor00/shared/mantle ~/data/mantle
```

Dataset 4 (Fir only): airflow over a turbine blade

Dataset from WestGrid's 2019 <https://computeCanada.github.io/visualizeThis> competition



- OpenFOAM *decomposed* dataset: 512 cores, 86 timesteps, 5 hydro variables, ~1TB in total
 - simulation by Joshua Brinkerhoff (UBC Okanagan), used with permission
 - unstructured mesh \Rightarrow loading a single timestep from the **3D internal mesh** requires 200GB+ physical RAM
 - the **2D airfoil mesh** takes only 13.7 GB virtual memory for 1 timestep + 1 variable
 - data in `fir:/project/6003910/razoumov/visThis2019-airfoil`
- Image at the top shows the air speed isosurface coloured by the Y-component of the vorticity, full animation rendering (86 timesteps) took 17m on 128 CPU cores on Cedar (predecessor to Fir)
- Create a symbolic link to make it easier to navigate to the dataset in ParaView

Let's try animations

1. Time animation

2. Camera animation

- check the “Scripting” section in these slides
- suggestions: rotate 360° around the vertical axis, fly towards/through the object

Off-screen rendering on a GPU

To render on a GPU from an OpenGL application such as ParaView, **traditionally you would require:**

1. OpenGL support in the GPU driver, and
2. an X server that handles windows and surfaces onto which client APIs can draw
 - run X11 server (typically started by root) on the GPU compute node, set `DISPLAY=:0.$gpuindex` (get GPU index from Slurm)

Modern NVIDIA GPU drivers include EGL (*Embedded-System Graphics Library*) support enabling creation of an OpenGL context for off-screen rendering without an X server

- your OpenGL application needs to be **recompiled with EGL support** \Rightarrow we have built this into `paraview/6.0.0` module that provides both **pvserver** for client-server and **pvbatch** for batch rendering
- unlike X11, EGL does not require any special setting to scale to very high resolutions, e.g. 4K (3840×2160) – simply ask it to render a 4K image

Interactive client-server rendering on a cluster's GPU

1. On Narval **submit an interactive job** to the GPU partition, e.g. a serial job:

```
$ salloc --time=0:30:0 --ntasks=1 --gpus-per-node=a100:1 \  
--mem-per-cpu=3600 --account=def-someuser
```

When the job starts, it'll return a prompt on the assigned compute node.

2. On the compute node inside the job **start the ParaView server** using a special version of ParaView with EGL support

```
$ module load paraview  
$ pvserver --force-offscreen-rendering --opengl-window-backend EGL  
# --egl-device-index=0 not needed: first available GPU inside the job is 0
```

For multiple GPUs can use

```
$ nvidia-smi -L # will return 0, 1, ...
```

The pvserver command will return something like

```
Waiting for client...  
Connection URL: cs://ng20201:11111  
Accepting connection(s): ng20201:11111
```


Data partitioning in parallel ParaView

-

Data partitioning in parallel ParaView (cont.)

If you have a large (many GBs) serial `.vtu` file:

1. Read this file in parallel ParaView on 16 cores - **slow**
 - and hope that it does not run out of memory on the reading core!
 - at this point the dataset is sitting in memory on one core
 - example: serial `.vtu` file at 9.1GB \Rightarrow 1'49" reading time
2. Apply D3 filter to distribute the dataset - **slowish** (memory + MPI)
3. **File** \rightarrow **Save data** as `.pvtu` with lz4 level-6 (fast) compression - **fast**
 \Rightarrow 16 files + 1 header file
 - now you have a statically decomposed dataset
4. Restart parallel ParaView on 16 cores, read `.pvtu` from scratch into - **fast!**
 - at this point the dataset is distributed across all 16 cores
 - example: same (but now decomposed) `.pvtu` dataset at 5.1GB (fast compression) \Rightarrow 11" reading time

👉 The same I/O speeds logic applies to `.vti` \rightarrow `.pvti` (but no need for D3)

Exercise: parallel rendering of partitioned data

This is a step-by-step guide for an unpartitioned, multi-block turbine dataset in
/project/def-sponsor00/shared/originalNonDecomposed (~18GB)

```
$ ln -s /project/def-sponsor00/shared/originalNonDecomposed ~/data/originalNonDecomposed
```

1. Partition the data

- 1.1 `salloc --time=0:60:0 --ntasks=16 --mem-per-cpu=3600`
- 1.2 start client-server ParaView session on 16 cores
- 1.3 load all `.vtm` files (all 10 timesteps); each points to a subdirectory with `.vtu` files
- 1.4 apply **Merge Blocks**, output type = Unstructured Grid
- 1.5 apply **Cell Data to Point Data** (so that you could use Contour)
- 1.6 apply **D3**
- 1.7 save data as `decomposed.pvtu`, write all timesteps as a series, use fast compression
⌚ this last step might take a while (10-15 mins) for all 10 timesteps; **what's in the files now?**

2. Visualize it

- 2.1 reset your client-server ParaView session on 16 cores ⌚ to test the reading speed
- 2.2 load all `decomposed.pvtu` files ⌚ do they load faster?
- 2.3 create visualization interactively, save animation as 1000 × 800 PNG files 📁 ~1 min
- 2.4 merge them into a movie with `ffmpeg`

Working with large datasets

Some filters **should not be used with structured data**:
they write unstructured data, can be heavy on memory usage

- Append Datasets
- Append Geometry
- Clean
- Clean to Grid
- Connectivity
- D3
- Delaunay 2D/3D
- Extract Edges
- Linear Extrusion
- Loop Subdivision
- Reflect
- Rotational Extrusion
- Shrink
- Smooth
- Subdivide
- Tessellate
- Tetrahedralize
- Triangle Strips
- Triangulate

Use these with caution: **Clip, Decimate, Extract Cells by Region, Extract Selection, Quadric Clustering, Threshold** (also write unstructured, but not as heavy on memory)

Remote rendering summary: some orthogonal decisions

(1) interactive vs. batch

- Interactive client-server for a quick look, exploration or debugging
 - another option is to download a scaled-down version of your dataset, debug a script locally on your laptop, and then run it as a batch job on the original full-resolution dataset on the cluster
- Batch really preferred for production jobs and producing animations

(2) CPU vs. GPU

- On clusters, CPU rendering is typically the best choice
 - recall: data-centre GPUs perform poorly for graphics workloads
 - you can throw many CPUs at your rendering job
 - modern ray tracing and rasterization libraries can be very fast at scale
 - I/O benefits
- For initial exploration, use the dataset size (in GB) to estimate the appropriate number of CPU cores, then adjust as needed

SUMMARY

Our visualization webinars

● ~2-3 visualization webinars per year

- ✉ subscribe to our mailing list <https://training.westdri.ca/contact>
- keep an eye on our website <https://training.westdri.ca/blog>
- ~50 mins + questions, usually on **fairly specific** or **advanced** topics

● Many past webinars are available with slides and recordings at <https://training.westdri.ca/tools/visualization>

- “Creating interactive online visualizations with Trame”
- “Launching 2023 Visualize This contest” (showing the Programmable Filter in use)
- “Image-based approach to large-scale visualization” (Cinema) ● “In-situ visualization with ParaView Catalyst2”
- “Highlights from the 2021 IEEE SciVis Contest” ● “Text analysis in 3D”
- “Remote visualization on Compute Canada clusters” ● “Data visualization in Julia with the Makie ecosystem”
- “Scientific visualization on NVIDIA GPUs” ● “Workflows with Programmable Filter / Source in ParaView”
- “The Topology ToolKit (TTK)” ● “Command-line image processing with ImageMagick”
- “Web-based 3D scientific visualization” (ParaViewWeb, vtk.js, ParaView Glance)
- “Photorealistic rendering with ParaView and OSPRay” ● “Batch visualization on Compute Canada clusters”
- “Molecular visualization with VMD” ● “Intermediate VMD topics: trajectories, movies, scripting”
- “Using YT for analysis and visualization of volumetric data” (part 1) ● “Working with data objects in YT” (part 2)
- “Scientific visualization with Plotly” ● “Novel visualization techniques from 2017 VISUALIZE THIS competition”
- “Camera animation in ParaView and VisIt” ● “3D visualization on new Compute Canada systems”
- “Using ParaViewWeb for 3D visualization and data analysis in a web browser”
- “Visualization support in WestGrid / Compute Canada”
- “Scripting and other advanced topics in VisIt visualization”
- “CPU-based rendering with OSPRay” ● “3D graphs with NetworkX, VTK, and ParaView” ● “Graph visualization with Gephi”

● We are always looking for new topic suggestions!

● Upcoming Winter Visualization Series in January-March 2026

Documentation and getting help

- Online visualization gallery <https://ccvis.netlify.app>
- Our documentation <https://docs.alliancecan.ca/wiki/Visualization>
- Western Canada research computing visualization resources
<https://training.westdri.ca/tools/visualization> (webinar archive)
- Email support@tech.alliancecan.ca and mention “*visualization*” in the subject line
⇒ goes into our ticketing system
- Email me alex.razoumov@westdri.ca
- Official documentation
 - ParaView documentation <https://docs.paraview.org>
 - VTK documentation <https://docs.vtk.org>
 - ParaView Discourse <https://discourse.paraview.org>
 - self-directed ParaView tutorial
<https://docs.paraview.org/en/latest/Tutorials/SelfDirectedTutorial/index.html>
 - ParaView User’s Guide <https://docs.paraview.org/en/latest/UsersGuide/index.html>