

3D data processing and visualization with PyVista

ALEX RAZOUMOV
alex.razoumov@westdri.ca



SIMON FRASER
UNIVERSITY



**Digital Research
Alliance** of Canada

Current visualization tools

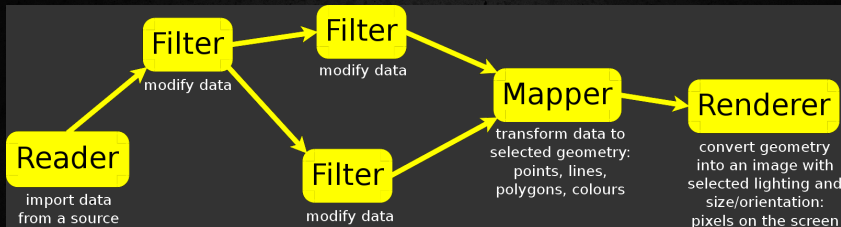
- ▶ Using and teaching ParaView for almost 20 years
 - scalable (MPI), open-source, multi-platform, actively developed, 100+ filters and plugins, GUI and Python scripting, client-server, off-screen batch rendering
 - large ecosystem of tools: VTK, VTK.js (Glance, write your own app), trame, Catalyst2, Cinema, ...
- ⇒ our default recommendation for sci-vis
 - full-day workshop, ~20 recorded webinars
- ▶ Distant second choice: VisIt (full-day workshop, few recorded webinars)
- ▶ Plethora of Python tools for lower-dimensional, publication-quality visualization: matplotlib, plotly, ...

VTK

- ▶ ParaView is built on top of VTK = Visualization Toolkit library (+ same authors)
- ▶ Open-source software system for 3D computer graphics, data processing and visualization
 - de facto industry standard for sci-vis
 - over 5 million lines of code
- ▶ Goes back to the early 1990s, originally C++, now also available in JS (partial since 2016) and Python (more or less full port)
- ▶ Hundreds of data-processing functions + its own file formats and I/O functions
- ▶ Insanely powerful features, but quite complex to use at the API level
 - ParaView makes it easy and accessible in the GUI and in ParaView Python
 - how about using VTK itself from Python?

Data flow in VTK

<https://book.vtk.org/en/latest/VTKBook/13Glossary.html>



- ▶ 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
- ▶ **RendererWindow** (on the screen) can hold multiple renderers
- ▶ Set the **Interactor** for your **RendererWindow** for keyboard+mouse control

How to create a cone in VTK: shortest version

- ▶ Can be made slightly shorter at the expense of performance
- ▶ C++ and JS versions are equally verbose; ParaView Python version is much shorter

long.py

```
import vtkmodules.vtkInteractionStyle # initialize mouse/keyboard controls
import vtkmodules.vtkRenderingOpenGL2 # initialize OpenGL graphics
from vtkmodules.vtkFiltersSources import vtkConeSource
from vtkmodules.vtkRenderingCore import vtkActor, vtkPolyDataMapper, vtkRenderWindow,
    vtkRenderWindowInteractor, vtkRenderer

cone = vtkConeSource(height=2.0, radius=1.0, resolution=30)

mapper = vtkPolyDataMapper()
mapper.SetInputConnection(cone.GetOutputPort())

actor = vtkActor()
actor.SetMapper(mapper)
actor.GetProperty().SetColor(0.9, 0.4, 0.2) # RGB for orange

renderer = vtkRenderer()
renderer.AddActor(actor)
renderer.SetBackground(0.1, 0.1, 0.1) # dark grey background

renderWindow = vtkRenderWindow()
renderWindow.AddRenderer(renderer)
renderWindow.SetWindowName("VTK Cone"); renderWindow.SetSize(900, 900)

interactor = vtkRenderWindowInteractor()
interactor.SetRenderWindow(renderWindow)

renderer.ResetCamera()
renderWindow.Render()
interactor.Start() # put the app into a listening state
```

Higher-level abstractions over VTK

- ▶ PyEVTK <https://github.com/paulo-herrera/PyEVTK> is a standalone package for writing to binary VTK files, requires only NumPy (no need for full VTK)
- ▶ PyVista <https://pyvista.org> library for reading, writing, analyzing, and plotting 3D meshes
 - “Swiss-army knife for data encapsulation, all powered by VTK”
 - wraps complex VTK functionality into concise, readable syntax
 - comes with its own, quite advanced visualization capabilities
 - recently I started using it more and more for mesh/data manipulation and I/O
 - does not replace ParaView, but can supplement it in several ways

PyVista history

- ▶ 2016: started by Alex Kaszynski as a module for the finite element library FEMORPH, providing a cleaner, higher-level abstraction layer over the C++ Visualization Toolkit (VTK) library
- ▶ 2017: first public release on GitHub as *vtki*
- ▶ 2019: rebranded and released as *PyVista*
- ▶ present: open-source (MIT license), NumFOCUS-affiliated project, 180+ contributors

Installation

```
uv pip install pyvista           # base PyVista
uv pip install "pyvista[jupyter]" # base PyVista and Jupyter extension
uv pip install "pyvista[all]"    # base PyVista and all extensions

pip install pyvista

conda install -c conda-forge pyvista
```

Also available as a Python wheel on our HPC clusters:

```
module load vtk/9.3.0 python/3.11.5
python -m venv env-pyvista
source env-pyvista/bin/activate
python -m pip install --upgrade pip --no-index
python -m pip install --no-index pyvista
...
deactivate
```

Dependence on / interoperability with other packages

```
uv pip show pyvista # show dependencies and dependents
```

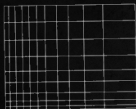
- ▶ **Heavily dependent** on NumPy to represent points, cells, fields and other mesh data
 - ▶ VTK for 3D visualization and data processing
 - ▶ Matplotlib for colormaps and some 2D plotting
-
- ▶ **Works** with Trame for interactive client or server-side rendering in a browser
 - ▶ Jupyter for interactive notebook computing
 - ▶ Xarray, SciPy and many other Python libraries

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



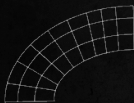
(a) Image Data

Image Data/Structured Points: *.vti, points on a regular rectangular lattice, scalars or vectors at each point



(b) Rectilinear Grid

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



(c) Structured Grid

Structured Grid: *.vts, regular topology and irregular geometry, need to indicate coordinates of each point



(d) Unstructured Points

Particles/Unstructured Points: *.particles



(e) Polygonal Data

Polygonal Data: *.vtp, unstructured topology and geometry, point coordinates, 2D cells only (i.e. no polyhedra), suited for maps



(f) Unstructured Grid

Unstructured Grid: *.vtu, irregular in both topology and geometry, point coordinates, 2D/3D cells, suited for finite element analysis, structural design

Plot some of these with PyVista:

```
from pyvista import demos
demos.plot_datasets() # plot major VTK grid types
```

left mouse	rotate
scroll wheel	zoom in/out in steps
middle mouse	pan
right mouse	continuous zoom in/out
q	close the render window
r	fit all objects without rotation
v	reset the camera (fit all objects with rotation)
f	fly to the point under the mouse cursor

```
demos.plot_logo(window_size=(1920,1080))
```

```
import pyvista as pv
pv.Cone().plot()
```

Other plot destinations

► Plot into a file:

file.py

```
import pyvista as pv

mesh = pv.examples.load_random_hills()

pl = pv.Plotter(off_screen=True, window_size=[1920,1080])
pl.add_mesh(mesh, show_edges=True)
pl.screenshot("111.png")
pl.close()
```

► To plot inline in a Jupyter notebook, you could set

```
pl = pv.Plotter(notebook=True)    # plot inline a Jupyter notebook
```

but it's not really necessary: `pv.Plotter()` works out of the box

Load a simple mesh example

`pv.examples.load` and `pv.examples.download` have 201 functions

createMesh.py

```
import pyvista as pv

mesh = pv.examples.load_random_hills()
print(mesh)    # print info - PolyData
mesh.plot()    # just the surface; opens in a separate blocking window

contours = mesh.contour()    # apply a Contour filter
print(contours)    # print info
contours.plot()    # just the contours

pl = pv.Plotter()    # new Plotting object to display multiple VTK meshes
pl.add_mesh(mesh, opacity=0.85)
pl.add_mesh(contours, color='white', line_width=5)
pl.show()    # surface with contours
```

ImageData from scratch

imageDataCreate.py

```
import pyvista as pv
import numpy as np

grid = pv.ImageData(dimensions=(100, 200, 300))
print('points:', grid.points)      # 0..99, 0..199, 0..299
print('center:', grid.center)      # (49.5, 99.5, 149.5)
print('dims:', grid.dimensions)    # (100, 200, 300)
print('origin:', grid.origin)       # default (0.0, 0.0, 0.0)
print('spacing:', grid.spacing)     # default (1.0, 1.0, 1.0)
print(grid)                        # no arrays

dxdydz = grid.points - grid.center # dx,dy,dz from the centre for each
point; shape (6000000,3)
distance = np.linalg.norm(dxdydz, axis=1) # 1D flattened array of
distances from the centre
grid['density'] = distance
print(grid)                        # now 1 array
```

Inspect and plot ImageData

```
print('names:', grid.array_names)           # list of all array names
print('association:', grid.get_array_association('density')) # point data
print('data:', grid.point_data['density'])   # 1D flattened array
```

```
dims = grid.dimensions           # (100, 200, 300)
rho = grid.point_data['density'].reshape(dims)
print('shape:', rho.shape)       # (100, 200, 300)
```

```
pl = pv.Plotter(window_size=(1200, 1200)) # new Plotting object
opacity = np.ones(100)                    # fully opaque
pl.add_volume_clip_plane(grid, normal='-x', opacity=opacity,
                          cmap='magma')    # interactive clip filter as volume
pl.show()
```

```
pl = pv.Plotter(window_size=(1200, 1200)) # new Plotting object
opacity = np.zeros(100)
opacity[::10] = np.geomspace(0.01, 0.75, 10) # numbers on a log scale go
into each 10th value of opacity => ring pattern
pl.add_volume_clip_plane(grid, normal='-x', opacity=opacity[::-1],
                          cmap='magma')        # clip filter
pl.show()
```

ImageData with cell values

imageDataCell.py

```
import numpy as np
import pyvista as pv

values = np.linspace(0, 10, 1000).reshape((20, 5, 10)) # 3D array of shape
                (20, 5, 10) -- now want to assign it to cells
grid = pv.ImageData()
grid.dimensions = np.array(values.shape) + 1 # we'll inject our values
        into cells => need one more for points in each dim
grid.origin = (100, 33, 55.6) # bottom left corner
grid.spacing = (1, 5, 2) # cell sizes in each dim
grid.cell_data['density'] = values.flatten() # assign the flattened array
grid.cell_data['density'].shape # (1000,)

grid.plot(show_edges=True)

# back into a 3D array
dim = np.array(grid.dimensions) - 1 # cell-based array has one fewer point
    in each dimension
rho = grid.cell_data['density'].reshape(dim)
print(rho.shape) # (20, 5, 10)
print(values[5,4,3] == rho[5,4,3]) # check that we get the right order
```

ImageData sources

This also creates 3D ImageData:

wavelet.py

```
import pyvista as pv

h = 15                                     # half-extent in each dim
wavelet = pv.Wavelet(extent=(-h,h,-h,h,-h,h)) # 30^3 cells, 31^3 points
print('center:', wavelet.center)           # default (0.0, 0.0, 0.0)
print('spacing:', wavelet.spacing)         # default (1.0, 1.0, 1.0)
print('bounds:', wavelet.bounds)           # -15.0 .. 15.0 in each dimension

wavelet.plot()
```

PolyData from the lists of vertices and faces

polyDataCreate.py

```
import numpy as np
import pyvista as pv

vertices = np.array([[0,0,0], [1,0,0], [1,1,0],
                    [0,1,0], [0.5,0.5,-1]]) # 5 points

square     = [4, 0,1,2,3] # lists of vertices
triangle1  = [3, 0,1,4]
triangle2  = [3, 1,2,4]

faces = np.concat([square,triangle1,triangle2]) # [4012330143124]
surface = pv.PolyData(vertices, faces)

surface.plot(scalars=np.arange(3), cpos=[-1, 1, 0.5], show_edges=True,
            line_width=5) # each face with a diff colour, cpos = camera position
```

PolyData sources

- Some polygonal objects are built in $\Rightarrow$ no need to define them manually

polyDataSources.py

```
import pyvista as pv

cyl = pv.Cylinder(radius=1.5, height=8.0, resolution=30, direction=(1,0,0))
cyl.plot()

pl = pv.Plotter()
pl.add_mesh(cyl, opacity=0.85)
cyl2 = pv.Cylinder(radius=1.5, height=8.0, resolution=30, direction=(0,1,0))
pl.add_mesh(cyl2, opacity=0.85)
pl.show()

sphere = pv.Sphere()
box = pv.Box()

pl = pv.Plotter(shape=(1,3))
pl.subplot(0, 0); pl.add_mesh(cyl, color='lightblue', show_edges=True)
pl.subplot(0, 1); pl.add_mesh(sphere, color='lightblue', show_edges=True)
pl.subplot(0, 2); pl.add_mesh(box, color='lightblue', show_edges=True)
pl.show()

pl.close()
```

3D graphs as PolyData

createGraph.py

```
import networkx as nx, numpy as np, pyvista as pv

H = nx.dorogovtsev_goltsev_mendes_graph(5)    # 5th generation
pos = nx.spring_layout(H, dim=3)
xyz = np.array([pos[i] for i in pos])          # (npoints,3) array
graph = pv.PolyData(xyz)                       # add nodes

lines = []
for e in H.edges():
    lines.extend([2, e[0], e[1]])              # each edge adds a line with 2 points

graph.lines = np.array(lines)                  # add edges

plotter = pv.Plotter(window_size=[1920, 1088])
spheres = graph.glyph(scale=False, orient=False,
                       geom=pv.Sphere(radius=0.03))    # glyph filter
plotter.add_mesh(spheres, color="crimson", render_points_as_spheres=True)

tubes = graph.tube(radius=0.005)              # tube filter
plotter.add_mesh(tubes, color="slategray")

plotter.show()
```

3D graphs to files

It's already PolyData ⇒ can save to a VTP file:

```
graph.save("dgm5.vtp") # possible since graph is PolyData
```

How about saving it as UnstructuredGrid?

```
cellTypes = np.full(H.number_of_edges(), pv.CellType.LINE, dtype=np.uint8)
               # fill with pv.CellType.LINE=3
unstructuredGraph = pv.UnstructuredGrid(cells, cellTypes, xyz) # of type
                       UnstructuredGrid
unstructuredGraph.save("dgm5.vtu")
```

UnstructuredGrid from the lists of points and cells

createUnstructured.py

```
import numpy as np
import pyvista as pv

points = [[1,1,0], [-1,1,0], [-1,-1,0], [1,-1,0], [0,0,1.6]] # 5 points
cell = [len(points), *list(range(len(points)))] # cell = [5, 0,1,2,3,4]
pyramid = pv.UnstructuredGrid(cell, [pv.CellType.PYRAMID], points)

pyramid.plot(scalars=np.arange(5), show_edges=True,
             line_width=3, full_screen=True) # put values into points

pyramid.plot(scalars=10, show_edges=True,
             line_width=3, full_screen=True) # put a value into the cell

from pyvista.examples import cells
cells.plot_cell(pyramid, full_screen=True) # no values, nice plot with dots
```

Combine multiple UnstructuredGrid cells into one

Fairly common to see this in some datasets

combineCells.py

```
from pyvista.examples import cells

g1 = cells.Hexahedron()    # in this case a cube
g1.points += [0, 1, 0]    # shift the cube along the y-axis
g2 = cells.Tetrahedron()
combined = g1 + g2

cells.plot_cell(combined, cpos='iso')
```

- ▶ Common cell types: VERTEX, LINE, TRIANGLE, POLYGON, PIXEL, TETRA, VOXEL, HEXAHEDRON, WEDGE, TETRAHEDRON, PYRAMID
- ▶ Inside `pv.CellType` counted 64 predefined cell types

Outline view

outline.py

```
import pyvista as pv

wavelet = pv.Wavelet(extent=(-15,15,-15,15,-15,15))

pl = pv.Plotter()
pl.add_mesh(wavelet.outline(), color="black", line_width=2)
pl.show()
```

► Or you can replace the last three lines with:

```
wavelet.outline().plot(color="black", line_width=2)
```

Surface view

- ▶ When plotting a 3D mesh, you get a surface view **by default**, scaled to the 3D data range:

surface.py

```
wavelet.plot(window_size=[1600, 1600])
```

- ▶ Same result:

```
pl = pv.Plotter(window_size=[1600, 1600])
pl.add_mesh(wavelet, cmap='viridis', scalar_bar_args={'title': 'density'})
# default style='surface'
pl.show()
```

- ▶ Also surface view, but now scaled to the surface data range:

```
pl = pv.Plotter(window_size=[1600, 1600])
outer_surface = wavelet.extract_surface(algorithm=None) # PolyData
pl.add_mesh(outer_surface, cmap='viridis', scalar_bar_args={'title': 'density'})
pl.show()
```

- ▶ Custom data range:

```
wavelet.plot(clim=[50, 160], window_size=[1600, 1600])
```

Volume view

[volume.py](#)

```
wavelet.plot(volume=True, window_size=[1600, 1600])
```

► Same result:

```
pl = pv.Plotter(window_size=[1600, 1600])
pl.add_volume(wavelet, cmap='viridis')
pl.show()
```

► Let's add some opacity mapping (custom transfer function) for the scalar array:

```
pl = pv.Plotter(window_size=[1600, 1600])
pl.add_volume(wavelet, opacity=[0,0,0,0.1,0.3,0.6,1])
pl.show()
```

```
opacity="linear"
opacity="linear_r"
opacity="geom"
opacity="sigmoid"
```

```
same as opacity=[0,1]
same as opacity=[1,0]
evenly spaced on the log scale
similar to linear, bigger jump in the middle
```

► Alternatively:

```
wavelet.plot(volume=True, opacity=[0,0,0,0.1,0.3,0.6,1],
              window_size=[1600, 1600])
```

Point view

point.py

```
pl = pv.Plotter(window_size=[1600, 1600])
pl.add_mesh(wavelet, style='points', point_size=8,
            render_points_as_spheres=True, cmap='viridis')
pl.show()
```

- Point size stays constant with zoom

Wireframe view

wireframe.py

```
pl = pv.Plotter(window_size=[1600, 1600])
pl.add_mesh(wavelet, style='wireframe', line_width=2, cmap='viridis')
pl.show()
```

Both points and wireframe

pointWireframe.py

```
pl = pv.Plotter(window_size=[1600, 1600])
pl.add_mesh(wavelet, style='points', point_size=10,
            render_points_as_spheres=True, cmap='viridis')
pl.add_mesh(wavelet, style='wireframe', cmap='viridis')
pl.show()
```

- Point size stays constant with zoom

Point Gaussian view

gaussian.py

```
pl = pv.Plotter(window_size=[1600, 1600])
pl.set_background('grey')
points = wavelet.cast_to_poly_points() # PolyData
pl.add_mesh(points, style='points_gaussian', point_size=3, cmap='Spectral',
            render_points_as_spheres=True)
pl.show()
```

- Use the GPU to render all $(30 + 1)^3 = 29791$ points as 2D textured shader-calculated splats facing the camera, as opposed to rendering each sphere as a separate multi-polygon 3D mesh

Filters

- ▶ The real power comes from the ability to run filters on data
- ▶ We already saw `contour()` , `extract_surface()` , `outline()` and few others

filters.py

```
import pyvista as pv

wavelet = pv.Wavelet(extent=(-15,15,-15,15,-15,15))
# wavelet.<TAB> # shows 501 possible extensions, many are filters

outline = wavelet.outline()
threshold = wavelet.threshold(value=200)
contours = wavelet.contour()
slices = wavelet.slice_orthogonal()
glyphs = wavelet.glyph(factor=5e-3, geom=pv.Sphere(), orient=False, progress_bar=True)
clip = wavelet.clip(normal=(1,1,1))
wavelet_with_vectors = wavelet.compute_derivative(scalars='RTData', gradient='gradient')
streamlines = wavelet_with_vectors.streamlines(vectors='gradient', n_points=100,
                                                source_radius=10.0, progress_bar=True)

pl = pv.Plotter(shape=(2,3), window_size=[1920, 1088])

pl.subplot(0, 0); pl.add_mesh(outline, color='black'); pl.add_mesh(threshold)
pl.subplot(0, 1); pl.add_mesh(outline, color='black'); pl.add_mesh(contours, cmap='coolwarm')
pl.subplot(0, 2); pl.add_mesh(outline, color='black'); pl.add_mesh(slices)
pl.subplot(1, 0); pl.add_mesh(outline, color='black'); pl.add_mesh(glyphs, cmap='RdBu')
pl.subplot(1, 1); pl.add_mesh(outline, color='black'); pl.add_mesh(clip)
pl.subplot(1, 2); pl.add_mesh(outline, color='black'); pl.add_mesh(streamlines, line_width=5)

pl.remove_scalar_bar('RTData')
pl.remove_scalar_bar('gradient')
pl.show()
```

Two linked views: outer surface mesh vs. full 3D mesh

delaunay.py

```
import numpy as np, pyvista as pv

np.random.seed(42) # for reproducible results
coords = np.random.uniform(low=0.0, high=1.0, size=(1000, 3)) # 1,000 random coordinates
pointCloud = pv.PolyData(coords)
delaunayMesh = pointCloud.delaunay_3d(alpha=0, progress_bar=True)

pl = pv.Plotter(shape=(1,2), window_size=[1920, 1088])

pl.subplot(0, 0)
pl.add_text('\nOuter surface mesh', font_size=24, position='upper_edge')
pl.add_mesh(pointCloud, color="cyan", point_size=10, render_points_as_spheres=True)
pl.add_mesh(delaunayMesh, style='wireframe', opacity=0.3, line_width=8) # UnstructuredGrid's outer surface

pl.subplot(0, 1)
pl.add_text('\nInner 3D mesh', font_size=24, position='upper_edge')
pl.add_mesh(pointCloud, color="cyan", point_size=10, render_points_as_spheres=True)
edges = delaunayMesh.extract_all_edges()
pl.add_mesh(edges, color="black", line_width=1) # 2D PolyData throughout the entire volume

pl.link_views()
pl.show()
```

- ▶ The flag `render_points_as_spheres=True` uses GPU rendering for smooth 3D spherical appearance
⇒ not entirely sure if it'll work with Zoom running at the same time

Animation

```
from pyvista import demos
demos.plot_wave(wavetime=10)
```

```
uv pip install imageio[ffmpeg]
```

explode.py

```
import pyvista as pv

n = 6
grid = pv.ImageData(dimensions=(n, n, n)).cast_to_unstructured_grid()

pl = pv.Plotter(window_size=[1920, 1088])
pl.open_movie('expand.mp4', framerate=30)
pl.add_mesh(grid, show_edges=True)
pl.write_frame() # the initial frame

for i in range(100):
    exploded_mesh = grid.explode(0.005)
    grid.copy_from(exploded_mesh) # copy into the existing mesh (which is
    shown in each frame)
    pl.add_text(f'Frame: {i}', name='time-label')
    pl.write_frame()

pl.close()
```

Physics-based rendering (PBR) for PolyData

PyVista itself does not bundle OSPRay. Instead, use its PBR engine alongside an environment texture, to deliver high-quality optical effects such as **glass transparency**, **refraction**, **roughness**, and **ambient reflection**, all using default OpenGL hardware acceleration without OSPRay ... For reflections, this can be illustrated with a skybox background reflecting off a mesh sculpture:

pbr.py

```
import pyvista as pv

mesh = pv.examples.download_louis_louvre()
mesh.rotate_z(180, inplace=True)
mesh.rotate_x(270, inplace=True)

landscape = pv.read_texture("kloppenheim_lowres.jpg")
landscape.InterpolateOn() # linear interpolation 2D images -> 3D surface
landscape.MipmapOn()      # for better texture rendering

pl = pv.Plotter()
pl.add_actor(landscape.to_skybox()) # set the scene background image
pl.set_environment_texture(landscape) # reflect background off the mesh
pl.add_mesh(mesh, color='gray', pbr=True, metallic=0.3, roughness=0.1,
             diffuse=1)
pl.view_xy(negative=True)
pl.show()
```

PyVista comes with a couple of skybox images `pv.examples.download_sky_box*`

This code makes heavy use of the GPU and freezes alongside Zoom



`open $(fd louis.mp4 ~/Documents)`

Parallel processing

- ▶ Unlike ParaView, PyVista does not do native parallel processing for mesh computations
 - ⇒ not a scalable tool for large-scale sci-vis
- ▶ You can use process-based parallelism to speed up processing of multiple datasets / timesteps / mesh blocks across multiple CPU cores
 - inside each worker process, process a separate structure
 - use [multi-processing](#), not multi-threading, as VTK might not be always thread-safe with shared memory in Python
 - multi-processing example `parallel.py` in the next slide
- ▶ Automatic (albeit serial) read of a master parallel file and all its partitioned pieces, e.g. PVTU or OpenFOAM or VTKHDF (example `readPartitioned.py` in the next section)
- ▶ PyVista data arrays – which wrap NumPy arrays – can be processed in chunks in parallel using Dask or Ray

Multi-processing example

parallel.py

```
import numpy as np, pyvista as pv, time, warnings
from multiprocessing import Pool

warnings.filterwarnings("ignore", message=".*download_nefertiti.*")
nefertiti = pv.examples.download_nefertiti() # 2,018,232 cells

def simplifyMesh(args):
    mesh, removeFraction = args
    return mesh.decimate(removeFraction)

remove = [0.50, 0.90, 0.99, 0.999]
tasks = [(nefertiti, r) for r in remove]

ncores = 4; pool = Pool(ncores)
start = time.time()
r = pool.map(simplifyMesh, tasks)
pool.close() # turn off your parallel workers
end = time.time()
print("Time in seconds:", round(end-start,3))

pl = pv.Plotter(shape=(2,2), window_size=(1920,1080))
for i in range(4):
    x, y = divmod(i, 2)
    pl.subplot(x, y)
    pl.add_text(f'{r[i].number_of_cells:,}'+' cells', font_size=12, position='upper_edge')
    pl.add_mesh(r[i], style='wireframe', opacity=0.6, line_width=3)

pl.link_views()
pl.show()
```

1 core - 39.376s, 4 cores - 13.271s

GUI clients

- ▶ PyVista does not have a single, standalone official desktop app like ParaView
- ▶ In PyVista itself you can add interactive controls (e.g. a slider) and a callback function that will be triggered on slider movement – demo `slider.py`
- ▶ There are several additional libraries for creating GUIs:
 - **PyVistaQt** module adds interactive VTK widgets (sliders, buttons, probes, ...) directly into your background renderer using Qt – useful for updating your plot in real-time
 - can be used to create standalone apps (Mac, Windows, Linux) with PyInstaller and PyQt
 - build on the same OS you intend to deploy
 - PyVista has a **Trame** backend
 - **PyVista-GUI** (work in progress) is a framework for building PyVista-based desktop apps

► PyVista can load VTK files directly:

loadVTK.py

```
import pyvista as pv

cube = pv.read("sineEnvelope.vti")
cube.plot(window_size=[1600, 1600])

warped = pv.read("warpingSpheres.vtkhdf")
warped.plot(window_size=[1600, 1600])
```

► PyVista cannot load NetCDF files natively, but we can do it via e.g. Xarray:

loadNetCDF1.py

```
import pyvista as pv
import xarray as xr

cube = xr.open_dataset("sineEnvelope.nc") # via xarray => xarray.Dataset
name = list(cube.data_vars)[0] # 'density'
data = cube[name].values # numpy.ndarray

mesh = pv.ImageData()
mesh.dimensions = data.shape
mesh.point_data['density'] = data.flatten()

mesh.plot(window_size=[1600, 1600])
```

- Can also do the last task via [pyvista-xarray](#) library which provides [.pyvista](#) accessor for [xarray.DataArray](#) and [xarray.Dataset](#) objects:

loadNetCDF2.py

```
import xarray as xr
import pvxarray

cube = xr.open_dataset("sineEnvelope.nc") # xarray.Dataset
rho = cube["density"] # xarray.DataArray

mesh = rho.pyvista.mesh(x="x", y="y", z="z") # generate a PyVista mesh
    by mapping dimensions to coordinates

mesh.plot(window_size=[1600, 1600])
```

Reading partitioned datasets: both VTK and VTKHDF files

readPartitioned.py

```
import pyvista as pv

# --- 1st option (prior to this, need to create master.pvtu for all
#         decomposed_*.pvtu files - can be done easily in Python)
mesh = pv.read('master.pvtu')
clipped = mesh.clip(normal='x', origin=mesh.center)
outer = clipped.extract_surface()

# --- 2nd option
mesh = pv.read('data.vtkhdf') # read as PartitionedDataSet: nCPU partitions
blocks = [p for p in mesh if p is not None] # list all its blocks
print('total number of cells:', sum([b.number_of_cells for b in blocks]))
combined = pv.merge(blocks) # merge into a single UnstructuredGrid
clipped = combined.clip(normal='x', origin=combined.center)
outer = clipped.extract_surface()

pl = pv.Plotter()
pl.set_background('#5c5956')
pl.add_mesh(outer, scalars='Turbulent_Viscosity', cmap='terrain',
            scalar_bar_args={'title': 'Turbulent_Viscosity'})
pl.show()
```

Writing meshes to files

- ▶ PyVista delegates native `.save()` writers directly to VTK's underlying writers for the corresponding file type ... when they exist, e.g.
 - `imageData.save("filename.vti")` works
 - `polygonalMesh.save("filename.vtp")` works
 - `unstructuredMesh.save("dgm5.vtu")` works

How about writing VTKHDF files?

- ▶ In development since 2022; very much a work in progress
- ▶ Ideally could replace other VTK formats
- ▶ Using HDF5 for actual storage ⇒ all the bells and whistles of HDF5, including:
 - better serial and parallel (not in PyVista) I/O performance
 - support for hierarchical data structures
- ▶ At the moment `vtk.vtkHDFWriter` only supports writing `vtkPolyData`, `vtkUnstructuredGrid`, `vtkPartitionedDataset` and `vtkMultiBlockDataSet`, but not e.g. `vtkImageData` – would need to convert it first to either PolyData or UnstructuredGrid:

writeImageDataVTKHDF.py

```
import pyvista as pv

grid = pv.ImageData(dimensions=(10, 10, 10))
grid.point_data["scalars"] = range(1000)

unstructured = grid.cast_to_unstructured_grid()
unstructured.save("111.vtkhdf")
```

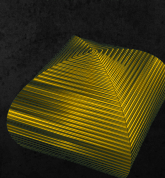
Personal favourite workflows

- ▶ Great for processing data, to be further rendered in an interactive GUI tool (ParaView)
- ▶ PyVista scripts are incredibly compact $\Rightarrow$ should work well for non-interactive workflows with a clear goal in mind, e.g. use your favourite LLM:
 - “in pyvista, visualize an intersection of 2 cylinders”
 - “in pyvista, create a point cloud of $1e6$ points inside a unit cube $[0,1]$ and compute 3D tessellation of this point cloud”
 - “ in pyvista, I have an UnstrucuredGrid; clip it and colour the outer surface by 'density'”
 - “I have the following visualization script in PyVista ... convert it to an animation rotating around the vertical axis and save it as an mp4 file”
 - “convert this script ... to PyVista; do not import any VTK or vtkmodules components directly”

Original intersecting cylinders



Boolean intersection



Summary

- ▶ Source code <https://github.com/pyvista/pyvista>
- ▶ Tutorial <https://tutorial.pyvista.org>
- ▶ User guide <https://docs.pyvista.org/user-guide>
- ▶ Gallery <https://docs.pyvista.org/examples>
- ▶ Documentation <https://docs.pyvista.org>
- ▶ PyVista colourmaps https://docs.pyvista.org/api/utilities/named_colormaps.html
- ▶ Community discussion board <https://github.com/pyvista/pyvista/discussions>

Questions?