Skills 04 - ParaView for FEM Output

Visual inspection, pipeline thinking, and reproducible figures

1 Introduction

Finite-element simulations do not end when the linear system has been solved. Before we interpret numbers, export figures, or compare designs, we need to inspect the computed field on the actual mesh. ParaView is a standard tool for this step: it is free, open-source, scriptable, and can handle data sets that are much larger than what we want to load into a notebook.

In this course, ParaView has three jobs:

1. Check whether the mesh and boundary tags look like the problem we meant to solve.
2. Inspect scalar and vector fields without hiding discretization artifacts.
3. Produce reproducible screenshots, line plots, animations, and exported data for reports (with `pvpython`).

> **Note**
>
> ParaView is not a replacement for quantitative verification. It is a fast way to see mistakes that would otherwise survive for a long time: swapped boundaries, wrong units, clipped ranges, discontinuities, bad refinement zones, or a solution plotted on the wrong data association.

2 Topics

After this notebook you should be able to:

1. Open `.vtu`, `.pvd`, and `.xdmf` output from a FEM code.
2. Explain the difference between the **Pipeline Browser**, **Properties**, and **Render View**.
3. Use `Apply`, visibility, coloring, rescaling, and representation modes intentionally.
4. Build common filters: `Calculator`, `Contour`, `Warp By Scalar`, `Extract Surface`, `Feature Edges`, `Plot Over Line`, and `Probe Location`.
5. Export a figure, a CSV line sample, or a saved `.pvsm` state in a reproducible way.

3 Create a Small FEM-like Data Set

The next cell creates a tiny triangular mesh and writes a time series that ParaView can open. This is not a PDE solve; it is a controlled visualization data set with:

- `temperature` as point data,

- `heat_flux` as vector point data,
- `material_id` as cell data,
- one `.pvd` file that collects several `.vtu` time steps.

Run the cell, then open `paraview_output/heat_demo.pvd` in ParaView.

> ℹ Note
>
> This is a minimal example. In practice, you create the file as a result of a PDE solve. This might only be used later for time series data.

```python
from pathlib import Path
import numpy as np

out_dir = Path("paraview_output")
out_dir.mkdir(exist_ok=True)

def vtk_values(values):
    values = np.asarray(values)
    if values.ndim == 1:
        return " ".join(map(str, values.tolist()))
    return " ".join(map(str, values.reshape(-1).tolist()))

def data_array(name, values, vtk_type="Float64", components=None):
    component_attr = f' NumberOfComponents="{components}"' if components else ""
    return (
        f'        <DataArray type="{vtk_type}" Name="{name}"{component_attr} format="ascii">\n
        f'          {vtk_values(values)}\n'
        f'        </DataArray>'
    )

def write_vtu(filename, points, triangles, temperature, heat_flux, material_id):
    n_cells = len(triangles)
    connectivity = triangles.reshape(-1)
    offsets = 3 * np.arange(1, n_cells + 1, dtype=np.int64)
    cell_types = np.full(n_cells, 5, dtype=np.uint8)  # VTK_TRIANGLE
    lines = [
        '<?xml version="1.0"?>',
        '<VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">',
        '  <UnstructuredGrid>',
        f'    <Piece NumberOfPoints="{len(points)}" NumberOfCells="{n_cells}">',
        '      <PointData Scalars="temperature" Vectors="heat_flux">',
        data_array("temperature", temperature),
        data_array("heat_flux", heat_flux, components=3),
        '      </PointData>',
        '      <CellData Scalars="material_id">',
        data_array("material_id", material_id, vtk_type="Int32"),
        '      </CellData>',
        '      <Points>',
        data_array("Points", points, components=3),
        '      </Points>',
        '      <Cells>',
```

```
        data_array("connectivity", connectivity, vtk_type="Int64"),
        data_array("offsets", offsets, vtk_type="Int64"),
        data_array("types", cell_types, vtk_type="UInt8"),
        '      </Cells>',
        '    </Piece>',
        '  </UnstructuredGrid>',
        '</VTKFile>',
    ]
    filename.write_text("\n".join(lines) + "\n")

nx, ny = 42, 24
xs = np.linspace(0.0, 2.5, nx + 1)
ys = np.linspace(0.0, 1.0, ny + 1)
points_2d = np.array([(x, y) for y in ys for x in xs])
points = np.column_stack([points_2d, np.zeros(len(points_2d))])

def pid(i, j):
    return j * (nx + 1) + i

triangles = []
cell_centers = []
for j in range(ny):
    for i in range(nx):
        quads = [pid(i, j), pid(i + 1, j), pid(i + 1, j + 1), pid(i, j + 1)]
        xmid = xs[i : i + 2].mean()
        ymid = ys[j : j + 2].mean()
        # Leave a small circular void so that surface extraction and edges are visible.
        if (xmid - 1.2) ** 2 + (ymid - 0.5) ** 2 < 0.16 ** 2:
            continue
        triangles.extend([[quads[0], quads[1], quads[2]], [quads[0], quads[2], quads[3]]])
        cell_centers.extend([(xmid, ymid), (xmid, ymid)])

triangles = np.array(triangles, dtype=np.int64)

x = points[:, 0]
y = points[:, 1]
base_temperature = 1.0 - x / x.max() + 0.18 * np.exp(-45.0 * ((x - 1.2) ** 2 + (y - 0.5) ** 2)
material_id = np.array([1 if cx < 1.25 else 2 for cx, _ in cell_centers], dtype=np.int32)

written = []
for step, time in enumerate(np.linspace(0.0, 1.0, 6)):
    oscillation = 0.08 * np.sin(2.0 * np.pi * time) * np.sin(np.pi * x / x.max()) * np.sin(np.
    temperature = base_temperature + oscillation
    dtdx = -1.0 / x.max() + oscillation * (np.pi / x.max()) * np.cos(np.pi * x / x.max())
    dtdy = oscillation * np.pi * np.cos(np.pi * y)
    heat_flux = np.column_stack([-dtdx, -dtdy, np.zeros_like(dtdx)])
    filename = out_dir / f"heat_demo_{step:03d}.vtu"
    write_vtu(filename, points, triangles, temperature, heat_flux, material_id)
    written.append((time, filename.name))

pvd_lines = [
```

```
    '<?xml version="1.0"?>',
    '<VTKFile type="Collection" version="0.1" byte_order="LittleEndian">',
    '  <Collection>',
]
for time, filename in written:
    pvd_lines.append(f'    <DataSet timestep="{time:.6f}" group="" part="0" file="{filename}"/
pvd_lines.extend(["  </Collection>", "</VTKFile>"])
(out_dir / "heat_demo.pvd").write_text("\n".join(pvd_lines) + "\n")

print(f"Open this file in ParaView: {out_dir / 'heat_demo.pvd'}")
print(f"Wrote {len(written)} time steps with {len(points)} points and {len(triangles)} cells."
```

```
Open this file in ParaView: paraview_output/heat_demo.pvd
Wrote 6 time steps with 1075 points and 1948 cells.
```

```
from xml.etree import ElementTree as ET
import matplotlib.pyplot as plt
import matplotlib.tri as mtri

root = ET.parse("paraview_output/heat_demo_000.vtu").getroot()

def read_array(name, dtype=float):
    node = root.find(f".//DataArray[@Name='{name}']")
    return np.fromstring(node.text, sep=" ", dtype=dtype)

preview_points = read_array("Points").reshape(-1, 3)[:, :2]
preview_temperature = read_array("temperature")
preview_triangles = read_array("connectivity", dtype=np.int64).reshape(-1, 3)

triangulation = mtri.Triangulation(
    preview_points[:, 0], preview_points[:, 1], preview_triangles
)

fig, ax = plt.subplots(figsize=(9, 3.8), constrained_layout=True)
field = ax.tripcolor(triangulation, preview_temperature, shading="gouraud", cmap="coolwarm")
ax.triplot(triangulation, color="black", linewidth=0.18, alpha=0.35)
ax.set_aspect("equal")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Generated ParaView demo data: temperature on triangular mesh")
fig.colorbar(field, ax=ax, label="temperature")
plt.show()
```

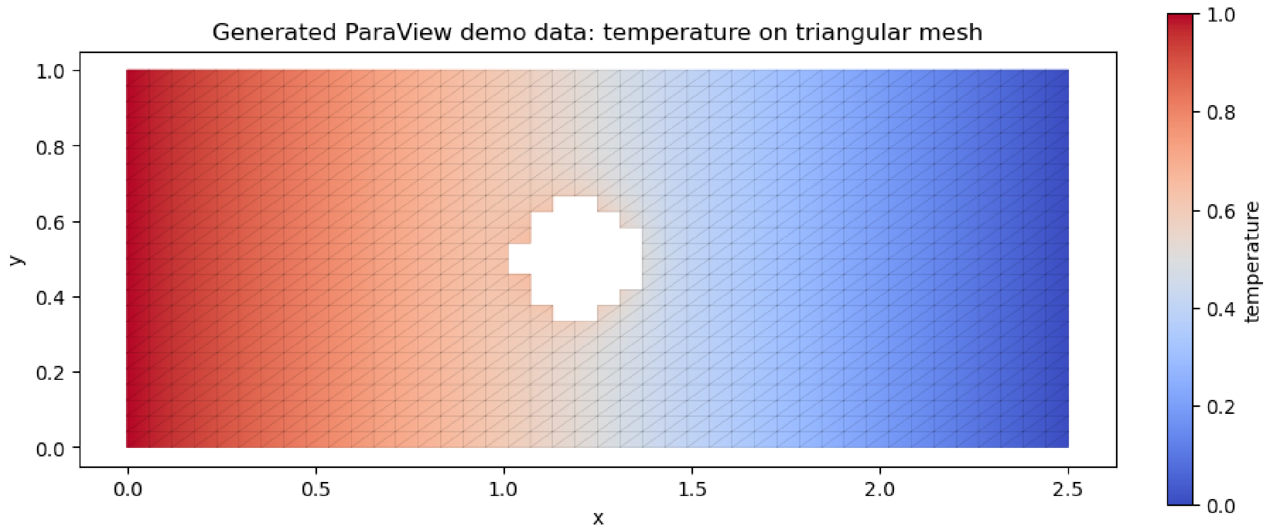


Figure 1: Generated demo data for the ParaView workflow: temperature on a triangular mesh written to VTK.

4 ParaView's Mental Model

A ParaView session is a data-flow graph. Every object in the **Pipeline Browser** is either a data source or a filter derived from another object.

Three interface habits matter immediately:

1. Select the object you want to edit in the **Pipeline Browser**.
2. Change parameters in **Properties**.
3. Press **Apply**.

ParaView deliberately does not recompute every time you touch a property. This is useful for large data, but it also means that many beginners think nothing happened. If the view does not update, first ask: did I select the right pipeline object, and did I press **Apply**?

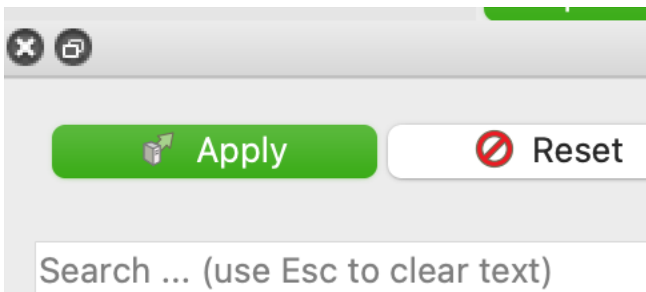
The eye icon controls visibility. It does not delete data. This is important when you compare a raw mesh, a contour result, and a warped result in the same view.

The gear icon exposes advanced properties. Use it when a filter has more options than the default panel shows.

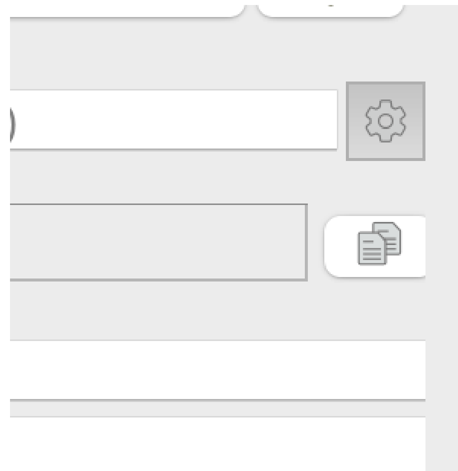
5 First Inspection Workflow

Open `paraview_output/heat_demo.pvd` and use this checklist:

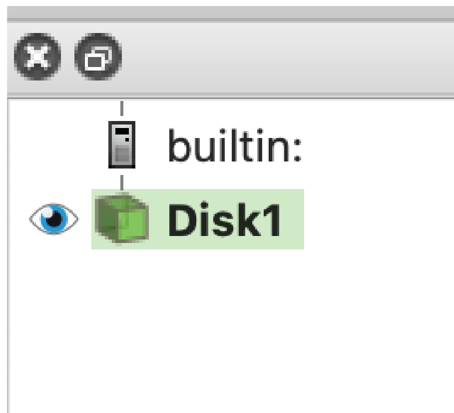
> **Lecture 03 output files**
>
> The coupled bracket example from Lecture 03 can also be opened in ParaView. Download the XDMF file together with its paired HDF5 file and keep both files in the same directory:
>
> - Temperature: `bracket_advective_heat_T.xdmf`, `bracket_advective_heat_T.h5`
> - Pressure: `bracket_advective_heat_p.xdmf`, `bracket_advective_heat_p.h5`



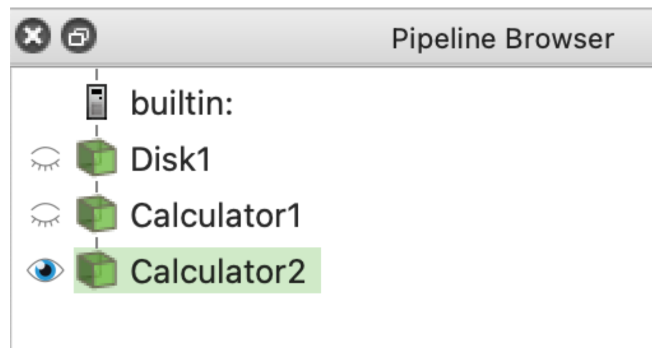
(a) Apply button in the Properties panel.



(b) Gear icon for advanced properties.



(c) Visibility eye in the Pipeline Browser.



(d) Pipeline Browser with a filter chain.

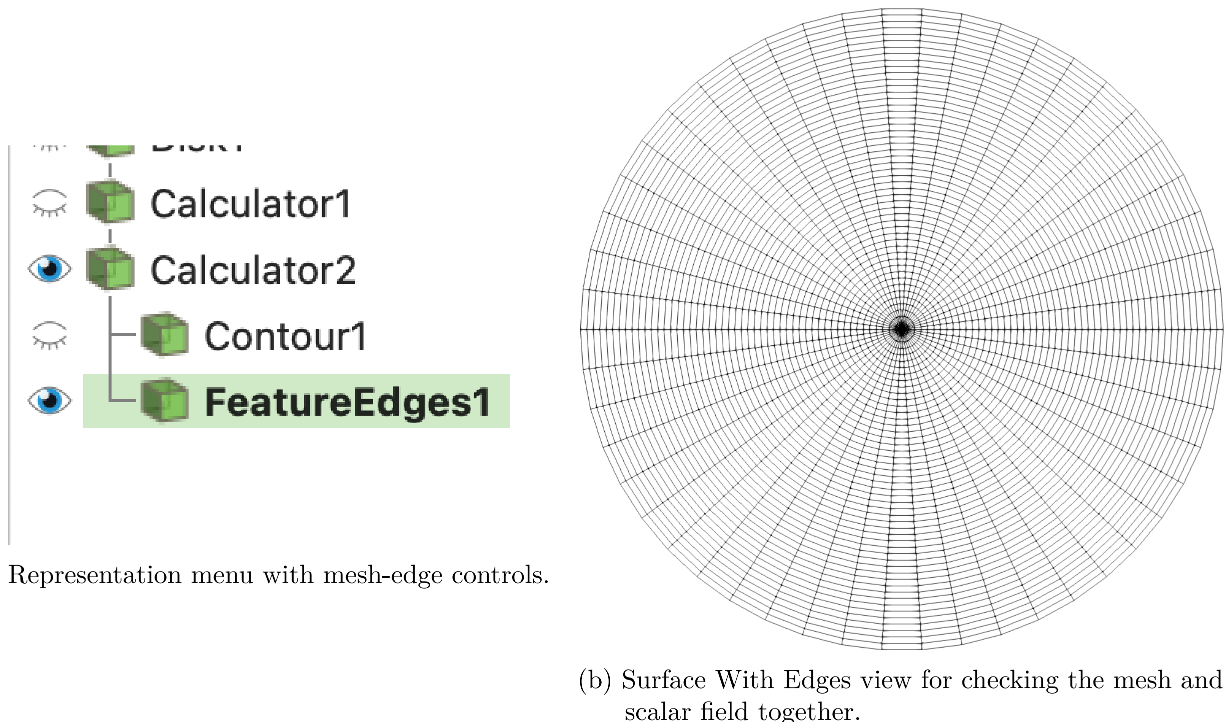
Figure 2: Core ParaView interface controls used throughout the tutorial: Apply, advanced properties, object visibility, and the pipeline browser.

- Velocity: bracket_advective_heat_u.xdmf, bracket_advective_heat_u.h5

Open the .xdmf file in ParaView; it will read the corresponding .h5 data file automatically.

1. Press **Apply** in the Properties panel.
2. Set representation to `Surface With Edges`.
3. Color by `temperature`.
4. Rescale the color range to the current data range.
5. Move through time with the time controls.
6. Switch coloring to `material_id` and notice that this is cell data, not point data.

A first plot should answer boring but essential questions: is the domain correct, is the void visible, are the elements plausible, does the field live on the expected part of the mesh, and does the time slider actually change the data?



(a) Representation menu with mesh-edge controls.

(b) Surface With Edges view for checking the mesh and scalar field together.

Figure 3: First inspection view: enable mesh edges, color by the target field, and verify the geometry before interpreting the solution.

> **Tip**
>
> For debugging, `Surface With Edges` is often better than a beautiful smooth surface. It reveals element size, holes, boundaries, and interpolation artifacts.

6 Point Data vs Cell Data

FEM output can attach arrays to different geometric objects:

- **Point data** lives on mesh vertices and is interpolated inside cells.
- **Cell data** is one value per element.

- **Field data** is global metadata and is not directly drawn on the mesh.

This distinction changes what a plot means. Continuous P_1 temperature fields are naturally point data. Material labels, subdomain ids, and many DG quantities are naturally cell data. If ParaView offers both point and cell versions of an array, read the label carefully before interpreting discontinuities or smoothness.

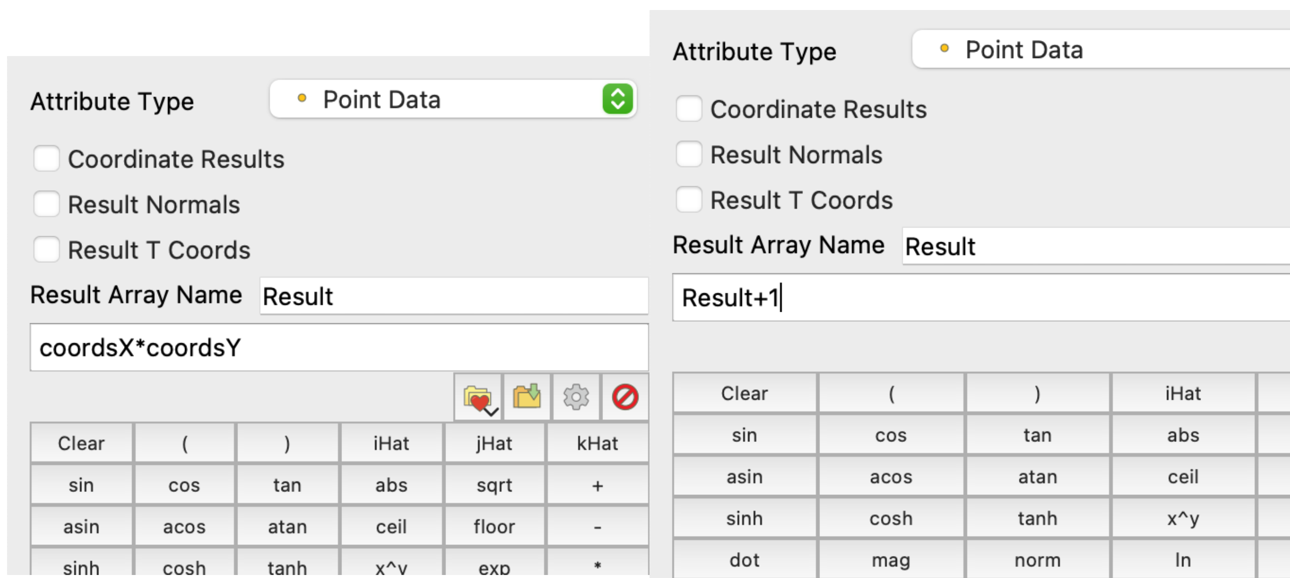
7 Calculator Filter

`Filters > Alphabetical > Calculator` creates a new array from existing arrays and coordinates. Typical uses are unit conversion, derived magnitudes, nondimensional quantities, and quick sanity checks.

Examples for this data set:

- `temperature - 273.15` if a field was exported in Kelvin and should be inspected in Celsius.
- `mag(heat_flux)` to inspect the flux magnitude.
- `coordsX*coordsY` as a simple coordinate-dependent test field.

Use a clear **Result Array Name** such as `heat_flux_magnitude`. Then press **Apply**, color by the new array, and rescale the color range.



(a) Calculator expression and result-array name.

(b) New calculator output in the pipeline.

Figure 4: Calculator workflow: define a derived array, apply the filter, and inspect the generated pipeline object.

> ⚠ Warning
>
> Calculator expressions act on the selected data association. If the input array is point data, the result is point data. If you need cell-wise results, convert or select the correct association first.

8 Color Ranges and Legends

Color is part of the numerical argument. The same field can look stable, unstable, smooth, or broken depending on the chosen range.

Useful buttons and settings:

1. **Rescale to Data Range** for one static snapshot.
2. **Rescale to Temporal Range** for time series comparisons.
3. **Use Discrete Colors** when levels should correspond to contour lines or material ids.
4. **Edit Color Legend Properties** to set titles, number formatting, font size, and labels.

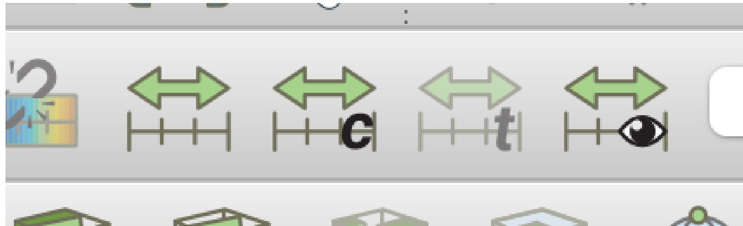
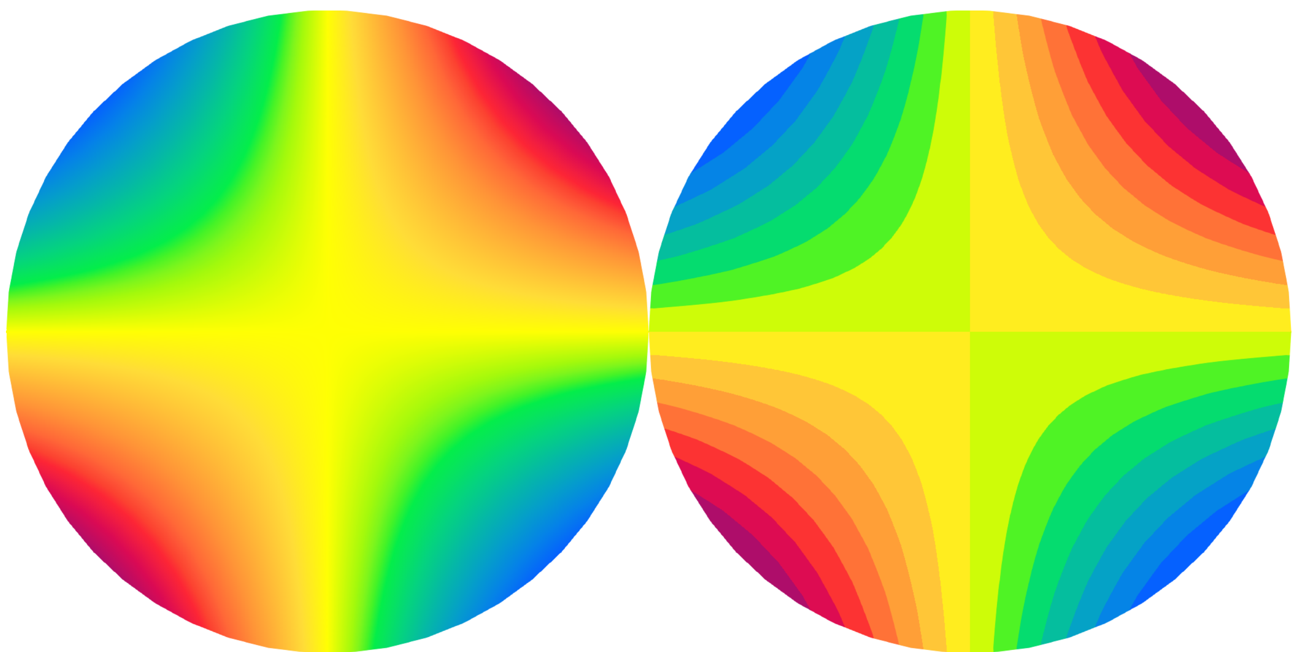


Figure 5: Color-range rescaling controls for static and temporal data ranges.



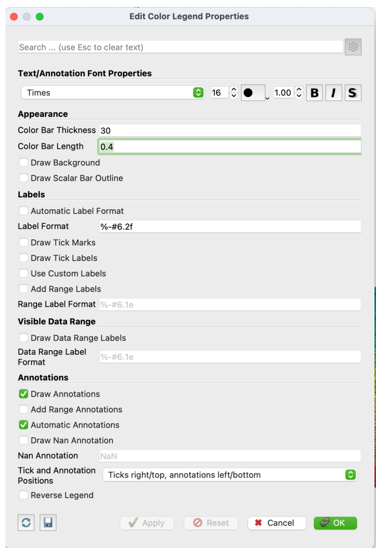
(a) Continuous color map for smooth scalar fields. (b) Discrete color map for levels or labeled regions.

Figure 6: Continuous and discrete color maps communicate different numerical assumptions about the same data.

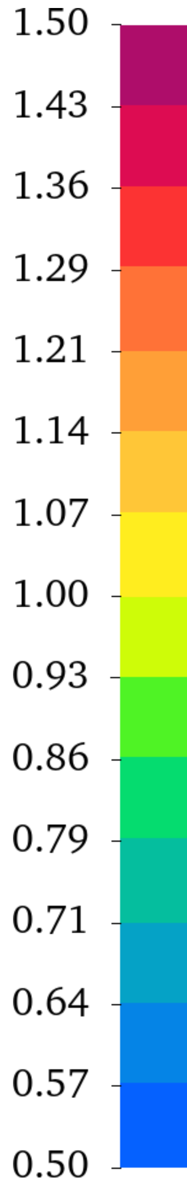
For reports, do not rely on ParaView defaults blindly. Use a readable legend title with units, set a range that is comparable across cases, and document if the range was clipped.

9 Contour and Isolines

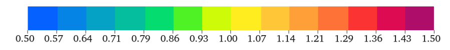
`Contour` extracts curves or surfaces where a scalar field has prescribed values. In 2D heat conduction this gives isotherms.



(a) Legend settings for titles, labels, and fonts.



(b) Color legend in the render view.



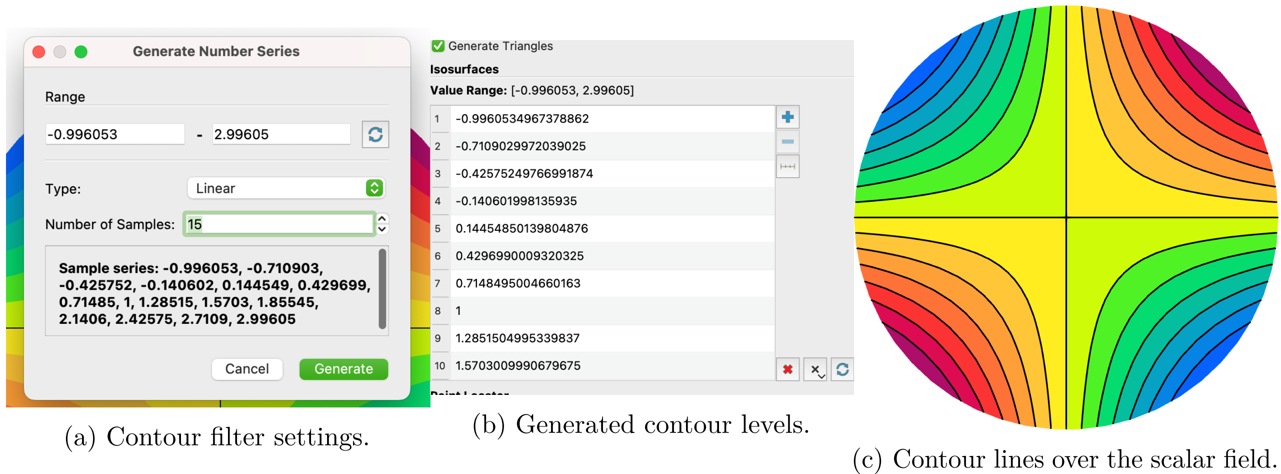
(c) Compact custom legend for report figures.

Figure 7: Legend controls and output: set readable labels and units, then check how the legend appears in the final render.

Exercise:

1. Select the loaded data set.
2. Add `Contour`.
3. Choose `temperature` as the contour variable.
4. Generate around 8 to 12 values.
5. Set contour coloring to `Solid Color`, for example black.
6. Increase line width if you export a high-resolution image.

Contours are easiest to read when the underlying surface uses a continuous color map and the contour lines are a contrasting solid color.



(a) Contour filter settings.

(b) Generated contour levels.

(c) Contour lines over the scalar field.

Figure 8: Contour setup and result: choose the scalar, define readable levels, and draw the isolines over the field.

10 Mesh Edges, Surfaces, and Boundaries

For FEM output, it is often useful to separate geometry from field values:

- `Surface With Edges` shows elements directly in the active representation.
- `Extract Surface` extracts the outer surface of a 3D data set.
- `Feature Edges` extracts sharp or boundary edges and can make hidden boundary structure visible.
- `Extract Selection` can isolate selected cells, points, or thresholded regions.

For DG methods or multi-material meshes, inspect cell boundaries before smoothing or resampling. A nice continuous-looking surface may hide an important discontinuity.

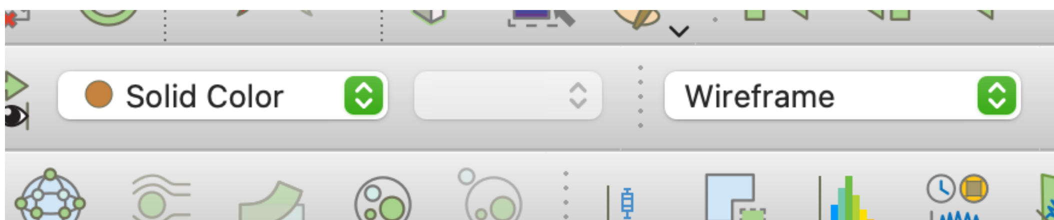


Figure 9: Feature Edges filter controls for extracting boundary and sharp edges.

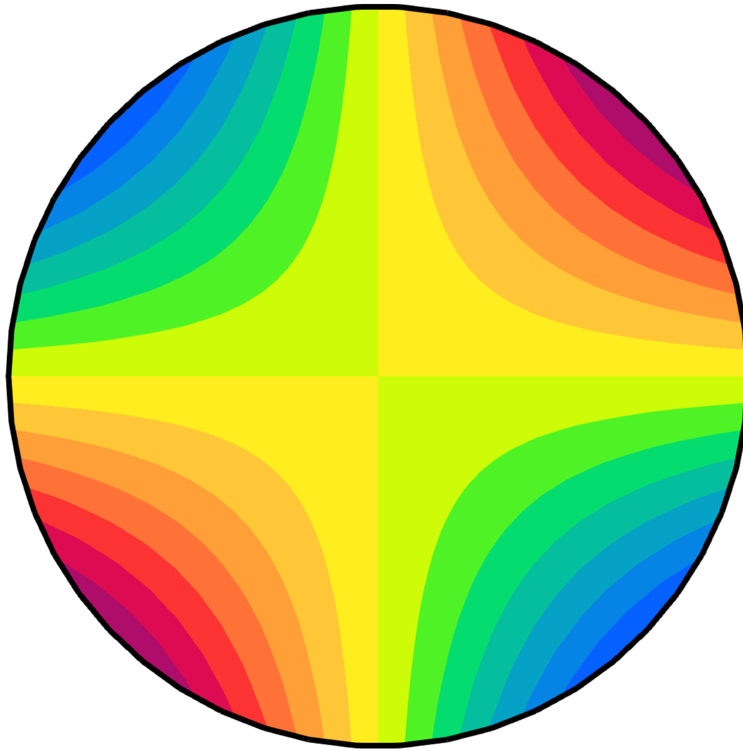


Figure 10: Extracted boundary and feature edges shown without the full scalar surface.

11 Warp By Scalar

`Warp By Scalar` turns a scalar field into vertical displacement. For 2D scalar fields this can be a useful qualitative plot, but it changes the geometry, so it should be used deliberately.

Exercise:

1. Select the loaded data set.
2. Add `Warp By Scalar`.
3. Use `temperature` as the scalar.
4. Start with a small scale factor such as `0.25`.
5. Use the 2D/3D camera buttons to switch between flat and oblique views.

Warping is a visualization transform, not a physical deformation unless the scalar really is a displacement component.

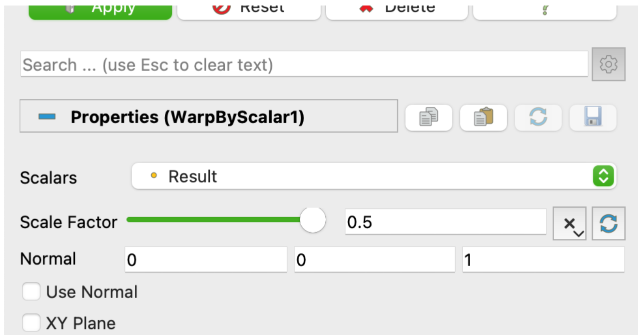
12 Probe and Plot Over Line

A visual plot becomes more useful when we can extract numbers from it.

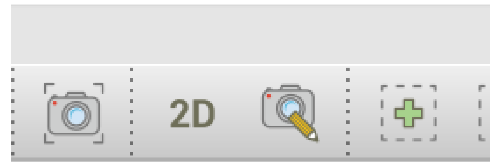
Use these tools:

- Hover over the view to inspect approximate point values.
- `Find Data` to query cells or points satisfying a condition.
- `Probe Location` to sample at one coordinate.
- `Plot Over Line` to sample a field along a segment.

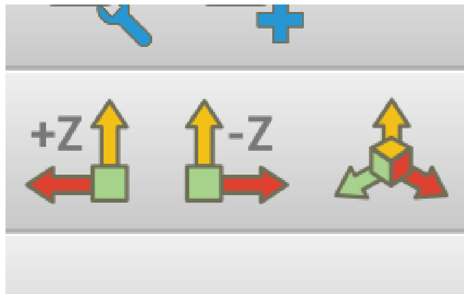
Exercise with `Plot Over Line`:



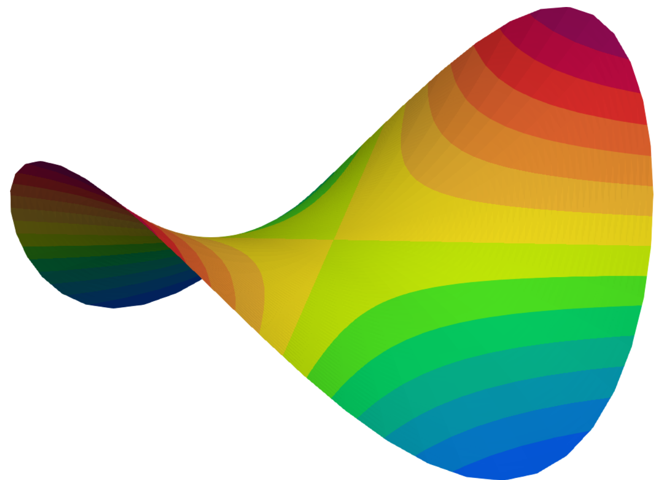
(a) Warp By Scalar filter settings.



(b) 2D and 3D camera buttons.



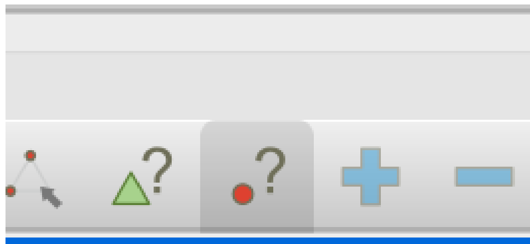
(c) Camera alignment controls.



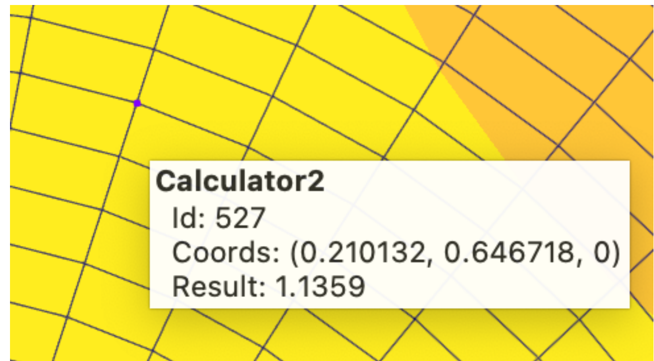
(d) Warped scalar field in an oblique view.

Figure 11: Warp By Scalar workflow: set a conservative scale factor, use camera controls deliberately, and compare the warped view with the flat field.

1. Select the loaded data set.
2. Add `Plot Over Line`.
3. Set point 1 to `(0, 0.5, 0)` and point 2 to `(2.5, 0.5, 0)`.
4. Press **Apply**.
5. Export the spreadsheet view as CSV if you need the values in Python, Julia, or LaTeX.

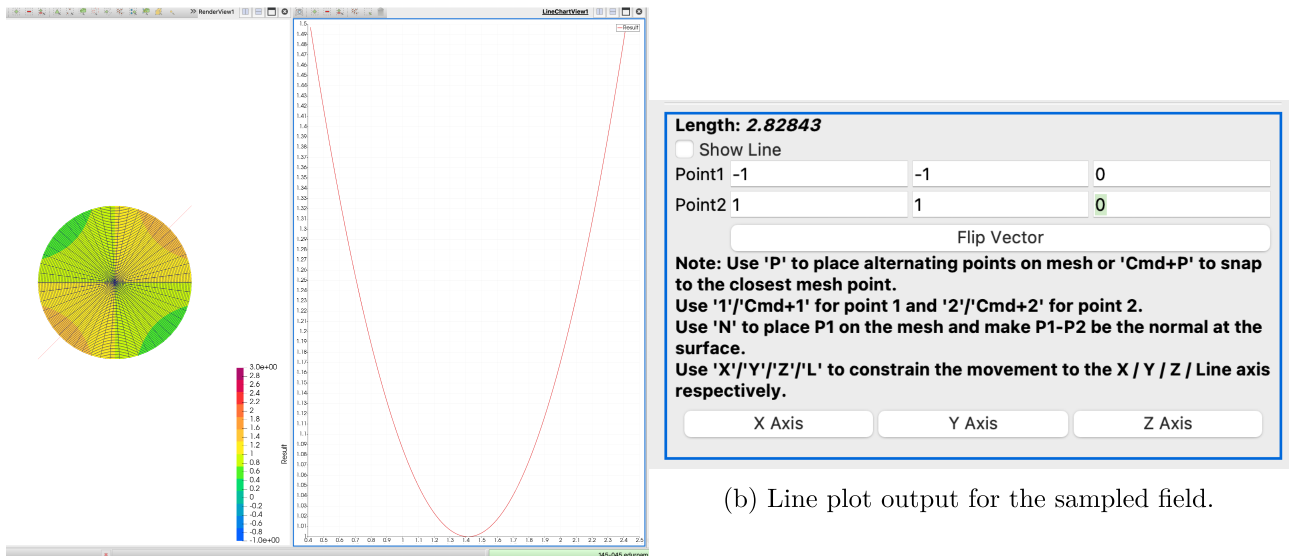


(a) Point-value inspection in the render view.



(b) Find Data dialog for querying cells and points.

Figure 12: Point and cell queries turn a visual inspection into a numerical check.



(a) Plot Over Line sampling segment.

(b) Line plot output for the sampled field.

Figure 13: Plot Over Line samples field values along a chosen segment and exposes the result as a plot or table.

13 Time Series and Animation

The `.pvd` file created above is a collection file. ParaView reads it as one time-dependent data source, even though the actual fields live in separate `.vtu` files.

For animations:

1. Open the time controls and step through the solution manually first.
2. Use a fixed color range over the full temporal range.
3. Save the state before exporting.

4. Use `File > Save Animation` only after checking the first and last frame.

A common mistake is to rescale the color map at every time step. That can hide amplitude changes completely.

14 Exporting Figures and States

A reproducible ParaView figure should be recoverable from files, not only from memory.

Recommended export workflow:

1. Set background to white for print figures.
2. Hide orientation axes unless they carry information.
3. Use a fixed camera position and document it by saving state.
4. Set the image size explicitly, for example `2000 x 1400` pixels.
5. Save the state via `File > Save State` as `.pvsm`.
6. Export PNG/JPEG for quick use, and CSV for quantitative line plots.

For final papers, many groups export the main field image and the color legend separately. This gives more control over placement in LaTeX or vector graphics tools.

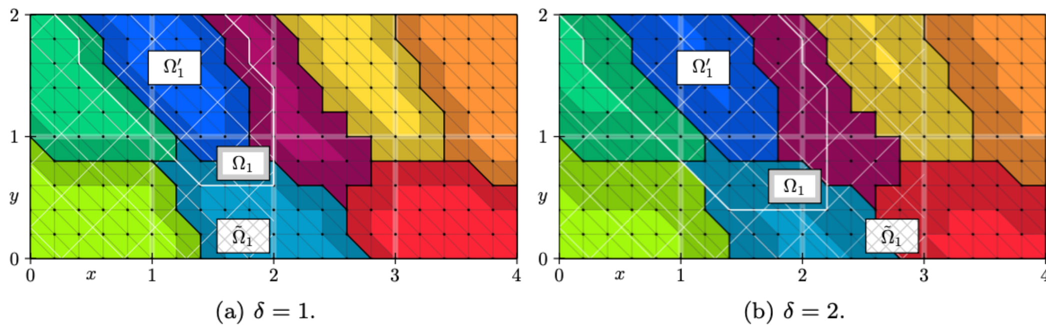


FIG. 3. *Sketch of the non-overlapping* $\{\Omega'_i\}_{i=1}^8$ *(black border), overlapping* $\{\Omega_i\}_{i=1}^8$ *(white border), and periodic neighborhood decomposition* $\{\tilde{\Omega}_i\}_{i=1}^8$ *(cross-hatch) of* $\Omega_4 := (0,4) \times (0,2)$ *for an overlap (dark shades) of* **(a)** $\delta = 1$ *and* **(b)** $\delta = 2$ *layers of elements. An increase of the periodic neighborhood from* $\tilde{\Omega}_1 = (0,2) \times (0,2)$ *for* $\delta = 1$ *to* $\tilde{\Omega}_1 = (0,3) \times (0,2)$ *for* $\delta = 2$ *can be observed.*

Figure 14: Advanced layer composition example with the main field and legend arranged for publication.

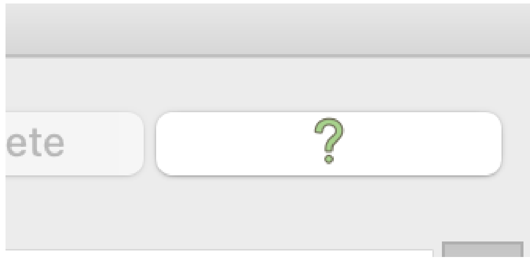
15 Help and Reproducibility

Every filter has built-in help through the `?` button in the Properties panel. Use it when a parameter is unclear; many filters have options that are hard to guess from the name alone.

For reproducibility, remember the following files:

- `.vtu`: one VTK unstructured-grid data set.
- `.pvd`: a collection file, often used for time series.
- `.xdmf` plus `.h5`: common FEniCSx output pair for mesh and field data.
- `.pvsm`: ParaView state file containing the visualization pipeline.
- `.py`: ParaView Python trace or script for automated rendering.

A good submission can include raw simulation output plus a `.pvsm` state. That lets someone reopen the exact pipeline, inspect the filters, and regenerate the figure.



(a) Help button in a filter properties panel.

Contour (Contour)

Generate isolines or isosurfaces using point scalars.

The Contour filter computes isolines or isosurfaces using a selected point-centered scalar array. The Contour filter operates on any type of data set, but the input is required to have at least one point-centered scalar (single-component) array. The output of this filter is polygonal.

Property	Description	Default(s)	Restrictions
Input	This property specifies the input dataset to be used by the contour filter.		Accepts input of following types: • vtkDataSet • vtkHyperTreeGrid. The dataset must contain a field array (point) with 1 component(s). The dataset must contain a field array (cell) with 1 component(s).
Contour By	This property specifies the name of the scalar array from which the contour filter will compute isolines and/or isosurfaces.		An array of scalars is required.
ComputeNormals	If this property is set to 1, a scalar array containing a normal value at each point in the isosurface or isoline will be created by the contour filter; otherwise an array of normals will not be computed. This operation is fairly expensive both in terms of computation time and	1	Accepts boolean values (0 or 1).

(b) Filter documentation opened inside ParaView.

Figure 15: Built-in filter help is the fastest way to check unfamiliar parameters while keeping the current pipeline open.

16 Typical FEniCSx Output Pattern

In FEniCSx, the most common path is to write mesh and functions through XDMF or VTK output. The exact API can vary by version, but the visualization principle is stable: write the mesh, write named fields, and keep time values explicit.

```
# Typical idea in a FEniCSx solver script, not executed here:
#
# from dolfinx import io
#
# with io.XDMFFile(mesh.comm, "solution.xdmf", "w") as xdmf:
#     xdmf.write_mesh(mesh)
#     u.name = "temperature"
#     xdmf.write_function(u, t=0.0)
#
# For a time-dependent solve, call write_function(u, t) after each accepted time step.
# Open solution.xdmf in ParaView and check that the time selector appears.
```

17 Further Reading

- ParaView Reference
- ParaView Ressources