3D visualization in Jupyter Notebooks

Martin Sandve Alnæs (martinal@simula.no),
Vidar T. Fauske, Min Ragan-Kelley

[simula . research laboratory]
- by thinking constantly about it

FEniCS'17, Luxembourg, 2017-06-13

Overview

Basic technologies

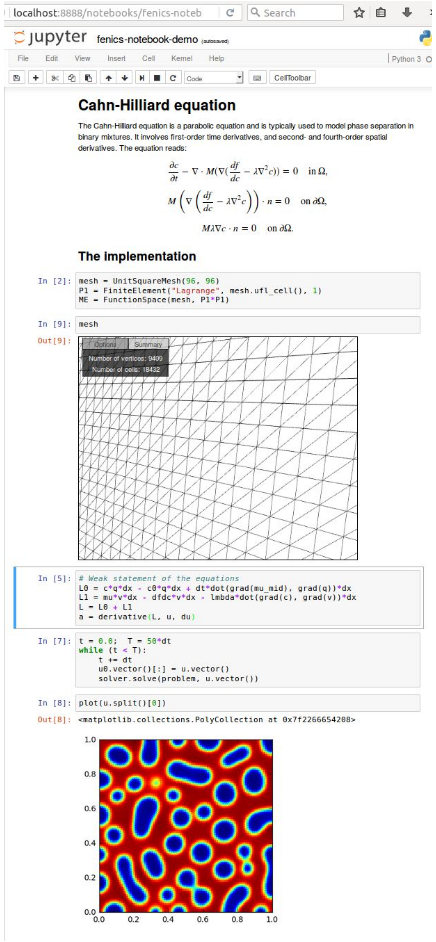
A quick overview

Some new visualization packages

Different goals, different API flavours

A suggestion for FEniCS

Supporting lots of visualization packages



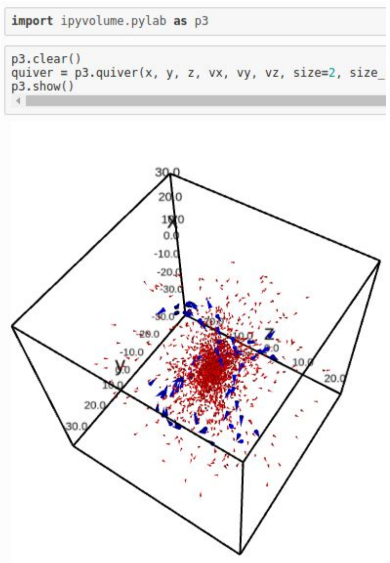
A quick overview of core tech for 3D visualization in notebooks

Jupyter Notebook ecosystem

- Notebook cell outputs can contain arbitrary HTML and Javascript
- Ipywidgets provides generic GUI widgets for notebooks

3D web technologies

- At the core is WebGL, a somewhat limited and slightly high level OpenGL
- Three.js library handles some tedious parts, adds abstractions, scenegraph



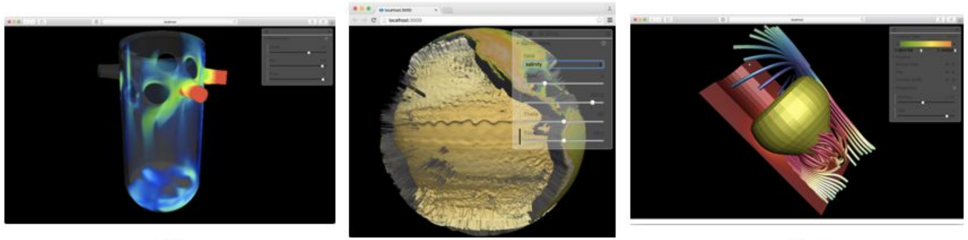
Quite a few visualization tools have added web versions lately

Paraview web

- Visualizer, ArcticViewer
- vtk-js to replace Three.js

Other

- MayaVi
- VisPy
- ipyvolume



(a) (b) (c)

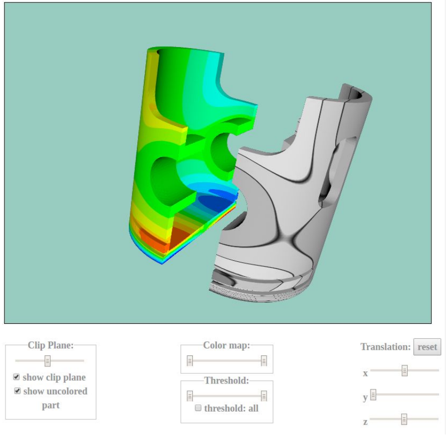
Packages developed by or contributed to by OpenDreamKit

Pythreejs is a wrapper for Three.js based on ipywidgets

Compose scenegraph in Python, render with Three.js

- Exposes many of the Three.js classes as ipywidgets
- Objects include camera, lights, basic shapes such as spheres and boxes, text, and also custom triangle meshes
- Great for semi-interactive 3D illustrations and animations
- Not really a scientific visualization library
- Not created by ODK but currently being updated on ODK time

Scivijs is a lightweight Paraview like visualization pipeline written in Javascript



Could be suitable for FEniCS:

- Interactive inspection of functions inline in a notebook (cutplanes, isosurfaces, and more)
- Jupyter widget under development
- Proof of concept FEniCS -> Scivijs exists
- No demo to show right now

K3D aims for a simple 3D plotting interface

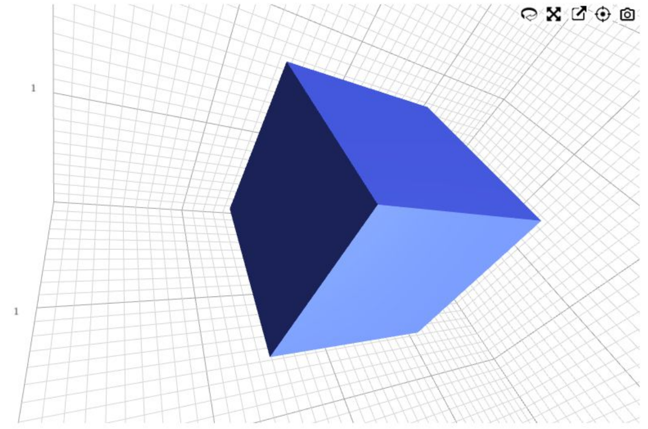
```
positions = boundary.coordinates()
triangles = boundary.cells()
vertexmap = boundary.entity_map(0).array()
scalars = u.compute_vertex_values()
scalars = scalars[vertexmap]

# Scale and compute colors
scalar_range = (scalars.min(), scalars.max())
print(scalar_range)

scalars[:] = 0
scalar_range = (0,1)

# Plot as surface mesh
K3D.mesh(positions, triangles, vertex_scalars=scalars,
         color_range=scalar_range, color_map=K3D.basic_color_maps.CoolWarm)
```

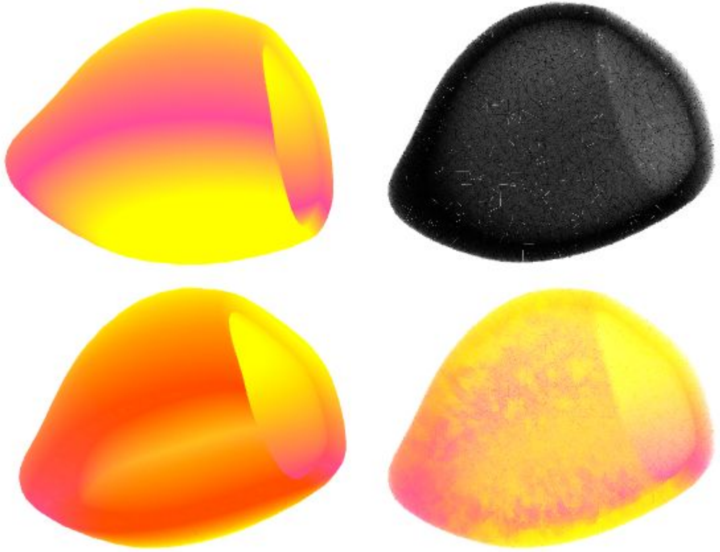
```
(0.54881163609402639, 1.0)
```



Can be suitable for many basic FEniCS plotting needs

- Scatter plots
- Glyphs (quiver)
- Surfaces in 3D
- Under development now at University of Silesia, good time to request features!
- (Missing better figures because of time...)

```
figs = []
for method in ["surface", "min", "xray", "volume"]:
    fig = render(coordinates, cells,
                 density=density, emission=emission,
                 density_lut=density_lut,
                 emission_lut=emission_lut,
                 width=400, height=400,
                 method=method)
    fig.animate = True
    figs.append(fig)
box = ipywidgets.HBox([ipywidgets.VBox(figs[:2]), ipywidgets.VBox(
box
```

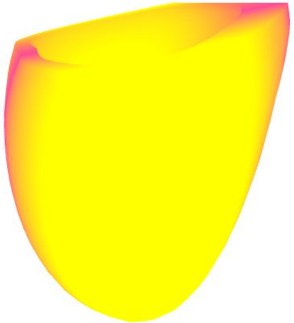


Unray (unreleased) provides volume rendering of tetrahedral meshes

Pipeline overview

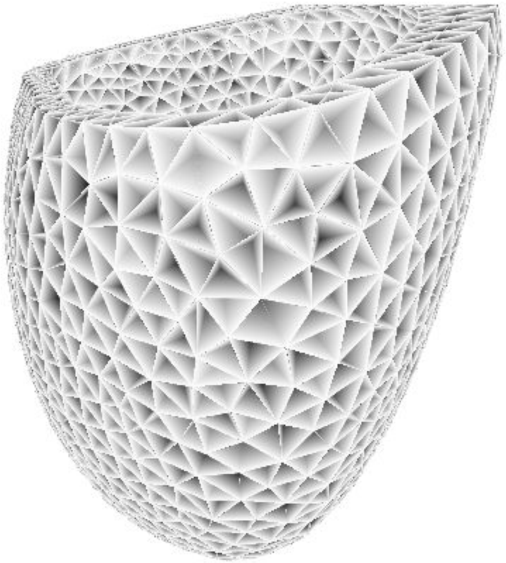
- Upload cells, coordinates, vertex values of functions as numpy arrays
- Data uploaded to GPU textures via Three.js
- Surface of each tetrahedron rasterized as a triangle strip using instanced rendering

Unray can be quite fast for anything that can render in WebGL shaders



Framework in place for

- Passing function data on each tetrahedron to shader
- Computing depth of tetrahedron
- Tested with decent performance with 4 million tetrahedral cells



My suggestion: FEniCS should make it easier to get data that users can feed into plotting libraries, instead of hiding it in plot(...)

Some things are easy to use

- mesh.cells() and mesh.coordinates()
- BoundaryMesh, could be simplified
- MeshFunction.array()
- function.compute_vertex_values()

Other nice-to-haves

- function.compute_dg_vertex_values()
- function.compute_cell_values()
- Functionality such as probes and slices from fenicstools should be in dolfin
- A more consistent interface for all of the above

Another idea is to make a small set of functions to package fenics objects into a generic simple format for visualization

This could be just arrays

- points, vectors = make_glyphs(func)
- points, scalars = make_scatter(func)
- triangles, points, values = make_surface(func)
- (above a very simplified version)

Or some vega-like format

- data = { “f”: f.compute_vertex_values(), ... }
- enc = { “colors”: { “field”: “f”, “range”: [0,1] }, ... }
- plot(“glyphs”, data=data, encoding=enc)

What do you want from visualization tools in notebooks?

Let me know during the breaks, or at martinal@simula.no!

We acknowledge financial support from the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541)