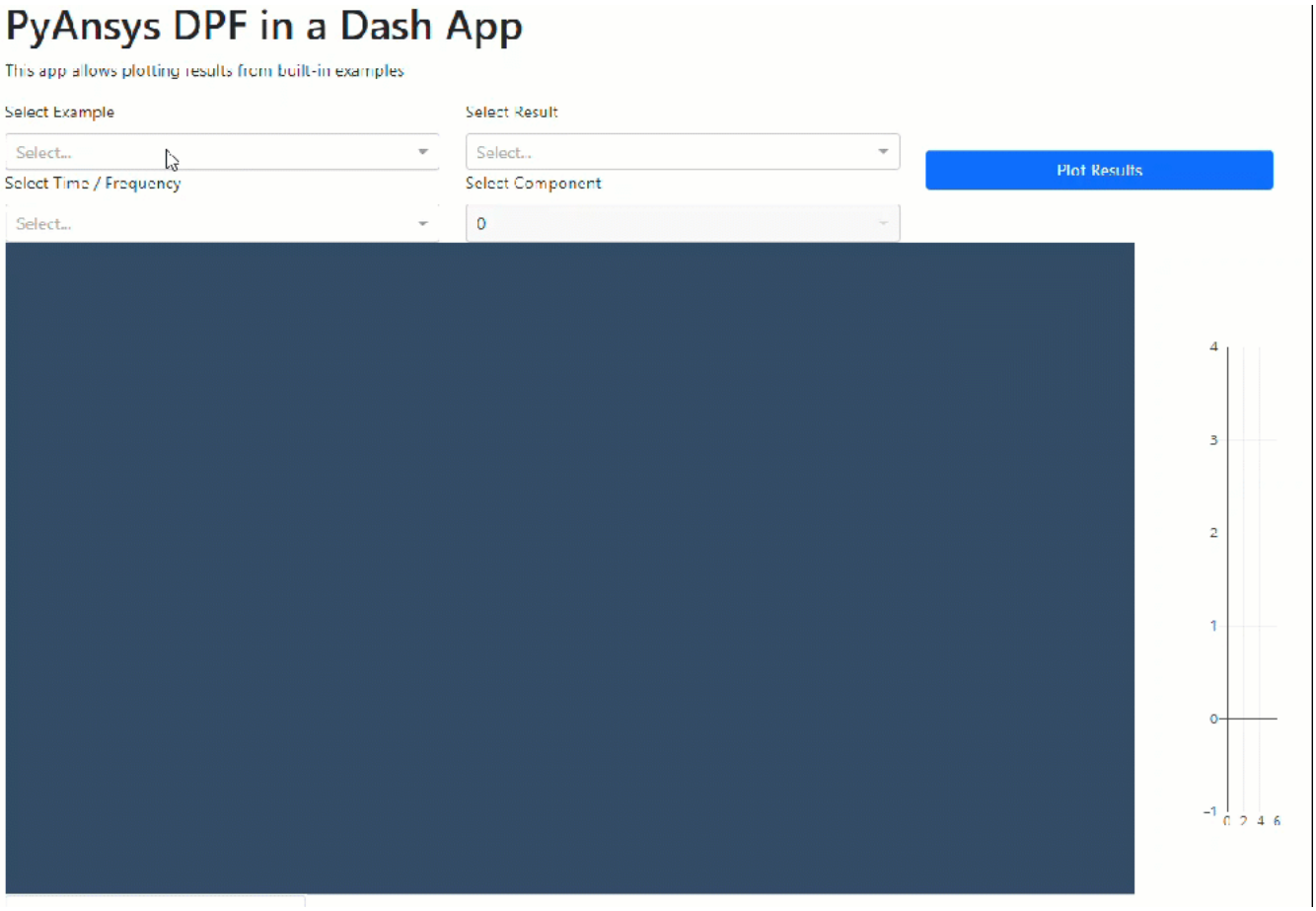
ANSYS in a Python Web App, Part 1: Post Processing with PyDPF

Integrating PyAnsys with Plotly's Dash and the Dash-VTK component to build an Ansys structural analysis post-processing web application

Steve Kiefer Jan 5 · 9 min read ★



The final post-processing app. Image by author

multiphysics simulation and equation solver and PyDPF-Core (& its simplified sibling PyDPF-Post) for post-processing Ansys results files. This is really intriguing to me as a structural analyst that uses ANSYS and Python pretty much daily. I have also used Plotly's Dash framework to build fairly simple web-apps for post processing data and solving various equations or systems.

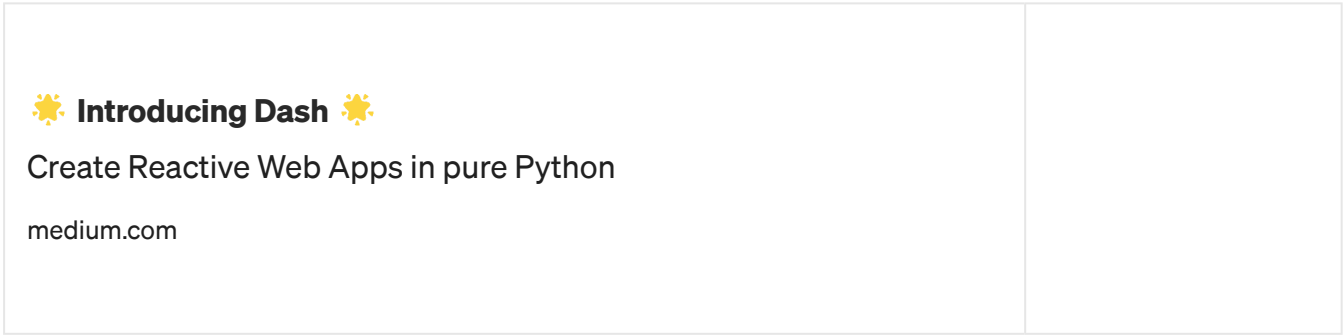
After playing around with PyANSYS in a Jupyter notebook for a while I decided to see if I could mash together a web-app that uses PyANSYS as a backend for a purpose-built analysis tool that a non-analyst could use and get insight (or at least build something that heads in that direction). PyANSYS is split into several packages. PyMAPDL is focused on communicating with a running MAPDL instance (local or remote) which pulls a license while running. PyDPF-Core and PyDPF-Post are focused on post-processing already-solved files (eg *.rst). These libraries do require Ansys to be installed.

In this article I'll walk through using PyDPF, Dash & Dash-VTK to build a web app that loads and plots results from a few PyDPF examples. You can see the full files (notebook and Dash app) in this GitHub repository.

Plotly's Dash & Dash-VTK

Plotly's Dash open source framework is described as a framework that can be used to build a fully functioning web-application with only python. I really like it for building applications I expect other users (besides myself) will use.

While its grown a lot since its debut, you can read the 2017 announcement essay to get more context to what Dash is all about:



web-app. I put together an example with a writeup you can check out here:

With Dash we are able to build an application with a simple and clean user interface and thanks to Dash-VTK view 3D mesh and associated results. If you are using Ansys to solve your analysis problems you would need to export the results from Ansys into some fixed (likely text) format so they could be read in by the web-app. You would have to export the model (mesh) as well as the results from Ansys and handle all the mapping. This could be fragile and would be limited to whatever results you (as the developer) would code for reading. This is where PyDPF comes in to help eliminate that export of model and mesh step and go right from a full results file to data processing and plotting.

PyDPF

The DPF in PyDPF stands for Data Processing Framework and (according to the PyAnsys Documentation) is dedicated to post-processing:

> *DPF is a workflow-based framework which allows simple and/or complex evaluations by chaining operators. The data in DPF is defined based on physics agnostic mathematical quantities described in a self-sufficient entity called field. This allows DPF to be a modular and easy to use tool with a large range of capabilities. It's a product designed to handle large amount of data.*

The PyAnsys documentation and examples highlight its use in Jupyter notebooks (including plotting). I explored the libraries using VS Code 'Interactive window' & 'Python code files' which are effectively a Jupyter notebook. Let's see what it would look like in a notebook:

```
from ansys.dpf import core as dpf
from ansys.dpf.core import examples

rst = examples.simple_bar
model = dpf.Model(rst)
tfs = model.metadata.time_freq_support.time_frequencies.data
print(model)


```

pyAnsys_DPF_example_1.py hosted with ❤ by GitHub view raw

….and voila! Lots of info about the results file such as results sets, units, and mesh statistics.

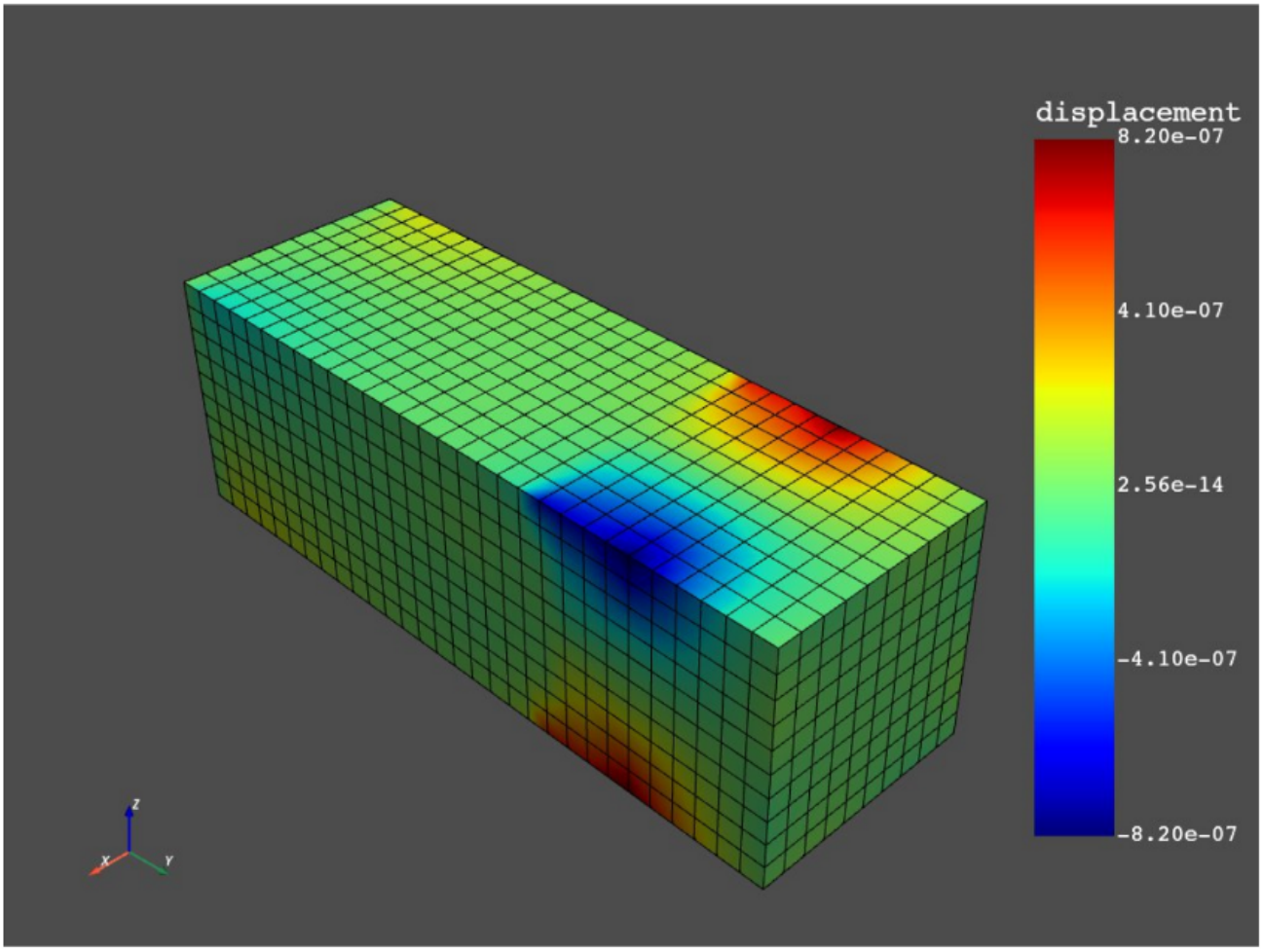
```
DPF Model
------------------------------
DPF Result Info
Analysis: static
Physics Type: mecanic
Unit system: MKS: m, kg, N, s, V, A, degC
Available results:
U Displacement :nodal displacements
ENF Element nodal Forces :element nodal forces
ENG_VOL Volume :element volume
ENG_SE Energy-stiffness matrix :element energy associated with the
stiffness matrix
ENG_AHO Hourglass Energy :artificial hourglass energy
ENG_TH thermal dissipation energy :thermal dissipation energy
ENG_KE Kinetic Energy :kinetic energy
ENG_CO co-energy :co-energy (magnetics)
ENG_INC incremental energy :incremental energy (magnetics)
BFE Temperature :element structural nodal temperatures
------------------------------
DPF  Meshed Region:
3751 nodes
3000 elements
Unit: m
With solid (3D) elements
------------------------------
DPF  Time/Freq Support:
Number of sets: 1
Cumulative     Time (s)       LoadStep       Substep
1              1.000000       1              1
```

using the `available_results` attribute of the `model.metadata.result_info` object we can get the result info objects for each result type in the results file (eg displacement, stress, etc). The result info object is not where the result data is actually stored. It is only a metadata object, but has all the information to retrieve the actual results using operators. First we create an operator using the `operator_name` attribute of the metadata object and connect it to the `model.metadata.data_sources` which associates the generic operator with our model object. We then use the `time_scoping` input to select the results set by index (converting from typical 0-based to 1-based in this case). If there is only 1 component in this results set then we get the fields container of the selected data set by calling the `res_op.outputs.field_container()` method. If there are more than 1 component then we create a new operator (`component_selector_fc()`) and chain it to the results operator by connecting the results operator outputs to the `comp_sel` operator on line 21 of the gist. We then add another input to select the component number (line 22) and request the fields container object by calling the `.outputs.fields_container()` method. There should only be 1 `field` object inside our `field_container` and we can access it as the first item in a list. finally we pass our filtered field to the `plot` method on our `mesh` object.

```
1   # %%
2   # 1st result in results set
3   res_idx = 0
4   # last available 'time' index
5   tf_idx = len(tfs) - 1
6   # first component (if more than 1)
7   comp_idx = 0
8
9   result_info = model.metadata.result_info
10  res = result_info.available_results[res_idx]
11  res_op = dpf.Operator(res.operator_name)
12  res_op.inputs.data_sources.connect(model.metadata.data_sources)
13  res_op.inputs.time_scoping([tf_idx + 1])
14  if res.n_components == 1:
15      fields = res_op.outputs.fields_container()
16  elif res.n_components > 1:
17      res_op = dpf.Operator(res.operator_name)
18      res_op.inputs.data_sources.connect(model.metadata.data_sources)
19      res_op.inputs.time_scoping([tf_idx + 1])
20      comp_sel = dpf.operators.logic.component_selector_fc()
21      comp_sel.inputs.connect(res_op.outputs)
```

```
25  mesh.plot(f0)
```

pyAnsys_DPF_example_2.py hosted with ❤ by GitHub view raw



PyAnsys DPF Displacement plot in Jupyter notebook. Image by author.

To change result type, component, or time step you just change the index parameters (`res_idx` , `comp_idx` , or `tf_idx`). If you check out the PyDPF examples you will notice there are shortcuts to plotting displacement, but we will use this more generic method later.

PyDPF is using VTK objects in the background to plot the results. Under the hood PyDPF converts the mesh object to a VTK unstructured grid and adds the arrays to that grid object before plotting. When we get to Dash_VTK we will also want the grid object so lets try and explicitly use it. Lets put it in a function!

(grids/nodes or elements/cells) and then map the field data order to the order of the nodes/cells in the vtk grid object. This is needed because the raw field data may have a different mapping/order than the order of the nodes/elements (so you cant just assign the `field.data` array directly to the `grid` object!). Once we get our `grid` with the results array assigned we can replace the `mesh.plot(f0)` line with `grid.plot(scalars=name)`. The main point here is that we are plotting directly with the VTK object (which we will need for Dash_vtk)….

```python
def get_grid_with_field(meshed_region, field):
    name = '_'.join(field.name.split("_")[:-1])
    location = field.location
    if location == dpf.locations.nodal:
        mesh_location = meshed_region.nodes
    elif location == dpf.locations.elemental:
        mesh_location = meshed_region.elements
    else:
        raise ValueError(
            "Only elemental or nodal location are supported for plotting."
        )
    overall_data = np.full(len(mesh_location), np.nan)
    ind, mask = mesh_location.map_scoping(field.scoping)
    overall_data[ind] = field.data[mask]
    grid = meshed_region.grid
    if location == dpf.locations.nodal:
        grid.point_data[name] = overall_data
    elif location == dpf.locations.elemental:
        grid.cell_data[name] = overall_data

    return grid
```

pyAnsys_DPF_example_3.py hosted with ❤ by GitHub view raw

The pyDPF Dash App

And now for the main course….

Ok we have a script that can filter down a results set to a specific field (by result type, time, and component) and we have a function that can extract a vtk grid object and

time step (by time) and (if appropriate) select the component of the result. That sounds like 4x `dcc.dropdown` components. We will also use dash_bootstrap_components to make it look nice and add some callbacks that crawl through the selected `ansys.dpf.core.examples` example to collect the appropriate options for the dropdowns. we will put the actual plotting behind a "plot" button for stability.

Below is an excerpt of the layout with dropdowns and the plot button. I am using a global variable `APP_ID` mostly out of habit to distinguish id names in the case of a multi-page app. Several dropdowns are missing `options` as well as `value` parameters. We will populate these with callbacks.

```
dbc.Row([
    dbc.Col([
        dbc.Label('Select Example', html_for="dropdown"),
        dcc.Dropdown(
            id=f'{APP_ID}_example_dropdown',
            options=[
                {'label': 'simple_bar', 'value': 'simple_bar'},
                {'label': 'msup_transient', 'value': 'msup_transient'},
                {'label': 'static', 'value': 'static'},
            ],
            clearable=False,
        ),
        dbc.Label('Select Time / Frequency', html_for="dropdown"),
        dcc.Dropdown(
            id=f'{APP_ID}_example_tf_dropdown',
            options=[],
            value=None,
            clearable=False
        )
    ]),
    dbc.Col([
        dbc.Label('Select Result', html_for="dropdown"),
        dcc.Dropdown(
            id=f'{APP_ID}_example_result_dropdown',
            options=[],
            value=None,
            clearable=False
        ),
        dbc.Label('Select Component', html_for="dropdown"),
        dcc.Dropdown(
            id=f'{APP_ID}_example_comp_dropdown',
```

```
            disabled=True
        )
    ]),
    dbc.Col(
        dbc.Button(
            'Plot Results',
            id=f'{APP_ID}_plot_button',
            style={'height':'80%', 'width':'80%'},
            class_name='h-50'
        ),
        align="center"
    )
]),
```

pyAnsys_DPF_dash_layout_1.py hosted with ❤ by GitHub view raw

Below are 2 callbacks that populate the result type, time step, and component dropdown options and default values. The first is triggered when the example is selected. This callback crawls through the `available_results` in the `metadata.result_info` object and creates the dropdown options using the name of the result type. We also create the timestep options using the float value as the `label`, but the integer index as the `value` parameter (remember using the index in the notebook example? We also send back default values (last timestep and first result type).

The second callback is fired when a result type is selected. We use the result name to find the appropriate `result_info` item (using `next`). Once we have that we determine the number of components and create the `options`, `values`, and if the dropdown should be `disabled` (for the case of 1 component).

```
@app.callback(
    Output(f'{APP_ID}_example_tf_dropdown', 'options'),
    Output(f'{APP_ID}_example_tf_dropdown', 'value'),
    Output(f'{APP_ID}_example_result_dropdown','options'),
    Output(f'{APP_ID}_example_result_dropdown','value'),
    Input(f'{APP_ID}_example_dropdown', 'value')
)
def dash_vtk_update_result_options(example_name):
    ctx = dash.callback_context
    if not ctx.triggered:
        raise PreventUpdate

```

```
16
17      tf = model.metadata.time_freq_support.time_frequencies.data
18      tf_options = [{'label': tfi, 'value': i} for i, tfi in enumerate(tf)]
19
20      return tf_options, len(tf)-1, res_options, result_info.available_results[0].name
21
22
23  @app.callback(
24      Output(f'{APP_ID}_example_comp_dropdown','options'),
25      Output(f'{APP_ID}_example_comp_dropdown','value'),
26      Output(f'{APP_ID}_example_comp_dropdown','disabled'),
27      Input(f'{APP_ID}_example_result_dropdown','value'),
28      State(f'{APP_ID}_example_dropdown', 'value')
29  )
30  def dash_vtk_update_comp_options(res_name, example_name):
31      ctx = dash.callback_context
32      if not ctx.triggered:
33          raise PreventUpdate
34
35      model = dpf.Model(EXAMPLE_MAP[example_name])
36      result_info = model.metadata.result_info
37      res = next((r for r in result_info.available_results if r.name == res_name), None)
38      if res is not None:
39          if res.n_components == 1:
40              comp_options = [{'label': 0, 'value': 0}]
41              comp_value = 0
42              comp_disabled = True
43          else:
44              comp_options = [{'label': i, 'value': i} for i in range(res.n_components)]
45              comp_value = 0
46              comp_disabled = False
47          return comp_options, comp_value, comp_disabled
48      else:
49          raise PreventUpdate
```

pyAnsys_DPF_callback_1.py hosted with ❤ by GitHub view raw

Now for the callback that return the 3D object for Dash_VTK. This callback is fired when the 'plot' button is pressed. We get the model object using the `example_dropdown` and a global variable (`dict`) that maps the example names (`simple_bar` , `msup_transient` , and `static`) to their appropriate `ansys.dpf.core.examples` object. We get the `result_info` object as in the previous callback but now we get the

to a `mesh_state` object that the Dash_VTK `GeometryRepresentation` component is expecting. we also set the colorscale range to the min/max of the field results. Not shown here is a function that creates a plotly figure with only a colorbar. While you can make a few edits to the dash_vtk source and rebuild the javascript (using node.js, and npm)to get a scalrabar to work with Dash_VTK (see here) I found it fairly simple to just create a plotly colorbar with the same background as Dash_VTK. See the full project for details.

```python
@app.callback(
    Output(f'{APP_ID}_geom_rep_mesh', 'children'),
    Output(f'{APP_ID}_geom_rep_mesh', 'colorDataRange'),
    Output(f'{APP_ID}_results_colorbar_graph', 'figure'),
    Output(f'{APP_ID}_results_min_max_dt', 'data'),
    Input(f'{APP_ID}_plot_button', 'n_clicks'),
    State(f'{APP_ID}_example_dropdown', 'value'),
    State(f'{APP_ID}_example_result_dropdown','value'),
    State(f'{APP_ID}_example_comp_dropdown', 'value'),
    State(f'{APP_ID}_example_tf_dropdown', 'value')
)
def dash_vtk_update_grid(n_clicks, example_name, result_name, comp_idx, tf_idx):

    if any([v is None for v in [example_name, result_name, tf_idx]]):
        raise PreventUpdate

    model = dpf.Model(EXAMPLE_MAP[example_name])
    result_info = model.metadata.result_info
    res = next((r for r in result_info.available_results if r.name == result_name), None)
    mesh = model.metadata.meshed_region

    if res.n_components == 1:
        res_op = dpf.Operator(res.operator_name)
        res_op.inputs.data_sources.connect(model.metadata.data_sources)
        res_op.inputs.time_scoping([tf_idx+1])
        fields = res_op.outputs.fields_container()
        f0 = fields[0]
        name = '_'.join(f0.name.split("_")[:-1])
        ugrid = get_grid_with_field(mesh, f0)
        mesh_state = to_mesh_state(ugrid.copy(), field_to_keep=name)
    elif res.n_components > 1:
        res_op = dpf.Operator(res.operator_name)
        res_op.inputs.data_sources.connect(model.metadata.data_sources)
        res_op.inputs.time_scoping([tf_idx+1])
        comp_sel = dpf.operators.logic.component_selector_fc()
```

```
        fields = comp_sel.outputs.fields_container()
        f0 = fields[0]
        name = '_'.join(f0.name.split("_")[:-1])
        ugrid = get_grid_with_field(mesh, f0)
        mesh_state = to_mesh_state(ugrid.copy(), field_to_keep=name)
    else:
        raise PreventUpdate

    view_max = ugrid[name].max()
    view_min = ugrid[name].min()
    rng = [view_min, view_max]

    name = '_'.join(f0.name.split("_")[:-1])
    fig = make_colorbar(name, rng)

    dt = [{'index':'Max', 'model':view_max}, {'index':'Min', 'model':view_min}]

    return [dash_vtk.Mesh(state=mesh_state), rng, fig, dt]
```

pyAnsys DPF callback 2.py hosted with ❤ by GitHub view raw

You can see the full files (notebook and Dash app) in this GitHub repository. I also included a yml file for installing all required libraries via conda.

Note: I did notice some instability in the javascript when switching result types that have different quantity of components, eg from `displacement` (3 components)to `stress` (6 components) . I tried a few things, but digging into the javascript (react_VTK) is beyond my expertise. If you see an error you can refresh the page and start over. The first plot always works.

Cool, but….. why?

Good question…. I believe packaging pyAnsys into a web app is most powerful when the intended user is not a simulation expert. Otherwise you could just use the notebook interface, Workbench, or MAPDL. I can think of a couple examples where post-processing could be useful in a web-app:

1. A results file database where results files would be archived with some extra metadata and could be brought up to review very specific results (eg stress)

Well thanks for making it this far. I hope you found something in here useful. In a future article I will cover integrating the PyAnsys pyMAPDL library into a Dash app which may be more useful as the Dash app can act as a very restricted preprocessor. It will include solving and extracting some results as well.

Thanks to Ben Huberman.