ANSYS in a Python Web App, Part 2: Pre Processing & Solving with PyMAPDL

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

Steve Kiefer Jan 20 · 9 min read ★

ANSYS recently announced it is supporting an open source project: PyAnsys. PyAnsys is split into multiple packages: PyMAPDL for interacting with an instance of an Ansys multiphysics simulation and equation solver and PyDPF-Core (& its simplified sibling PyDPF-Post) for post-processing Ansys results files. This is a companion article to Ansys in a Python Web App, Part 1: Post Processing with PyDPF.

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…

towardsdatascience.com

In this article I'll walk through using PyMAPDL, Dash & Dash-VTK to build a web app that builds a model, runs a modal analysis, and presents some results. You can see the full files (notebook and Dash app) in this GitHub repository. Here it is in action:

The PyMAPDL Web App in action! Image by author.

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

🌟 **Introducing Dash** 🌟 Create Reactive Web Apps in pure Python medium.com	

You can also see more about Dash_VTK in this article:

PyMAPDL

MAPDL stands for Mechanical Ansys Parametric Design Language and has been the scripting interface to the Ansys mechanical solver for decades. Prior to Ansys Mechanical (the Workbench application / UI) APDL was the way to parameterize simulation easily. Even with the 'new' Workbench Mechanical UI, MAPDL can still be very useful to add functionality the UI doesn't have (via 'Command Snippets'). PyMAPDL is the PyAnsys package for communicating with an instance of Ansys via Python. According to the PyAnsys Documentation:

> *With PyMAPDL it is easier than ever to integrate the simulation capabilities of the Ansys MAPDL multi-physics solver directly into novel applications thanks to an API that will look familiar to APDL and Python users alike. The package presents a Python-friendly interface to drive the software that manages the submission of low-level APDL commands, while exchanging data through high-performance gRPC interfaces.*

When you launch an MAPDL instance it does require an appropriate license to start the server and will pull an hold the license until the server instance is exited (or killed). Lets looks at what this looks like in a Jupyter notebook (I am using the VS Code 'Interactive window' & 'Python code files')

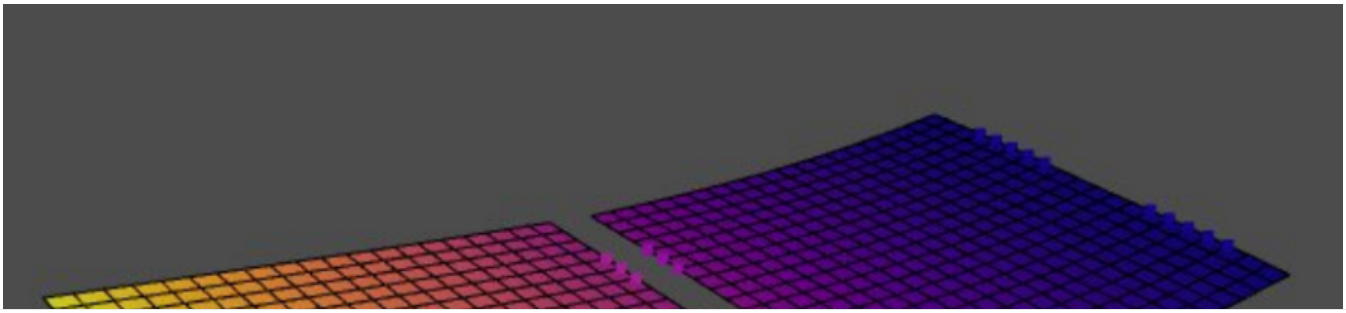
In this example we are using PyMAPDL much like a normal MAPDL script. First we create the `mapdl` instance (optionally specify license type) and then start executing MAPDL commands using the pythonic format. In this example we will create a very simple model of a deployed spacecraft solar array wing (albeit, extremely simplified). At first we define some materials and sections, joint properties as representations for hinges), then geometry (by creating an area then using the `agen` command to copy it with an offset). Then we mesh the shells. Starting at line 87 I am using a helper function `make_sbc` to create some 'remote points' and then the `combin250` 6 DOF springs

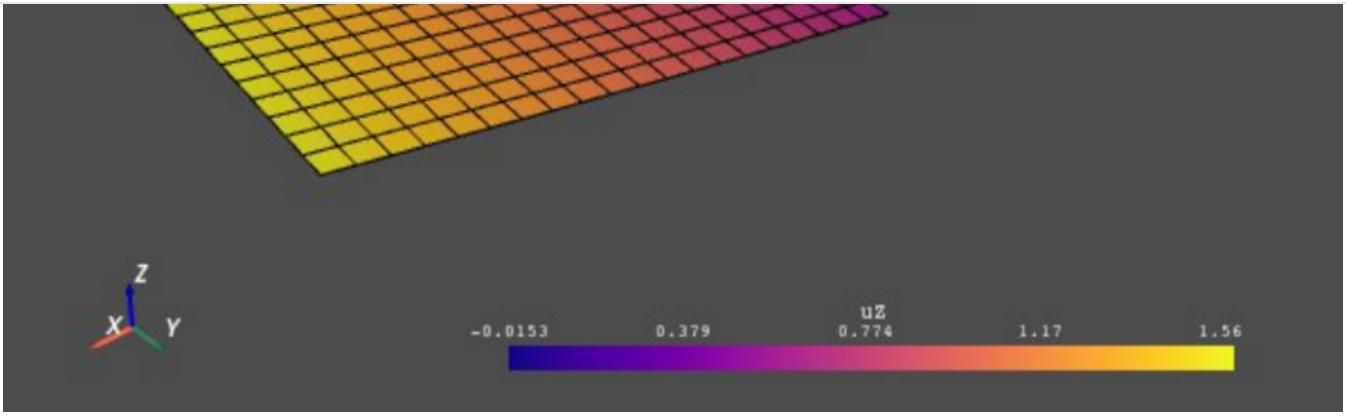
full file.

```python
# %%
import pandas as pd
from ansys.mapdl.core import launch_mapdl

mapdl = launch_mapdl(override=True, license_type="ansys", cleanup_on_exit=True)
mapdl.clear()
mapdl.prep7()
mapdl.units("BIN")
mapdl.csys(kcn=0)

# PV NSM (lb/in^2)
nsm = .002

# Faceseet material
tfs = 0.03
mid = 1
Ex = 10.0e6
nu_xy = 0.3
dens = 0.1 / 386.089
mapdl.mp('EX', mid, Ex)
mapdl.mp('PRXY', mid, nu_xy)
mapdl.mp('DENS', mid, dens)

# Core material
tc = 0.25
mid = 2
Ez = 75e3
Gxz = 45e3
Gyz = 22e3
dens = (3.1 / 12**3) / 386.089

mapdl.mp('EX', mid, 10)
mapdl.mp('EY', mid, 10)
mapdl.mp('EZ', mid, Ez)
mapdl.mp('GXY', mid, 10.)
mapdl.mp('GXZ', mid, Gxz)
mapdl.mp('GYZ', mid, Gyz)
mapdl.mp('PRXY', mid, 0.)
mapdl.mp('PRXZ', mid, 0.)
mapdl.mp('PRYZ', mid, 0.)
mapdl.mp('DENS', mid, dens)

# create shell property
```

```
mapdl.secdata(tfs, 1, 0, 3)
mapdl.secdata(tc, 2, 0, 3)
mapdl.secdata(tfs, 1, 0, 3)
mapdl.secoffset('MID')
mapdl.seccontrol(val4=nsm)

# create joint property
mid = 10
K_1 = 1e6
K_2 = 1e6
K_3 = 1e6
K_4 = 1e3
K_5 = 1e5
K_6 = 1e5
mapdl.et(mid,250,0)
mapdl.r(mid)
mapdl.rmore(K_1, K_2, K_3, K_4, K_5, K_6)

# create geometry (areas)
w = 20.
h = 20.
xc = 0.
yc = 0.
dx = 2.0

a1 = mapdl.blc4(xcorner=xc, ycorner=yc, width=w, height=h)
mapdl.agen(itime=2, na1='ALL', dx=w+dx)

# mesh
mapdl.allsel()
mid=1
mapdl.aatt(mat=mid, type_=mid, secn=mid)
mapdl.aesize('all', dx/2.)
mapdl.mshape(0, '2D')
mapdl.mopt('SPLIT', 'OFF')
mapdl.smrtsize(sizlvl=4)
mapdl.csys(kcn=0)
mapdl.amesh("all")

# %%
# base remote points
mapdl.lsel(type_='s', item='LOC', comp='X', vmin=0.)
mapdl.nsll(type_='s', nkey=1)
n_base_1, r_base = make_sbc(mapdl, 0., h/4., 0., pinb=dx)
mapdl.lsel(type_='s', item='LOC', comp='X', vmin=0.)
```

```
# ref remote points
mapdl.lsel(type_='s', item='LOC', comp='X', vmin=w)
mapdl.nsll(type_='s', nkey=1)
nr1, rr1 = make_sbc(mapdl, w+dx/2., h/4., 0., pinb=dx)
mapdl.lsel(type_='s', item='LOC', comp='X', vmin=w)
mapdl.nsll(type_='s', nkey=1)
nr2, rr2 = make_sbc(mapdl, w+dx/2., 3*h/4., 0., pinb=dx)

# mob remote points
mapdl.lsel(type_='s', item='LOC', comp='X', vmin=w+dx)
mapdl.nsll(type_='s', nkey=1)
nm1, rm1 = make_sbc(mapdl, w+dx/2., h/4., 0., pinb=dx)
mapdl.lsel(type_='s', item='LOC', comp='X', vmin=w+dx)
mapdl.nsll(type_='s', nkey=1)
nm2, rm2 = make_sbc(mapdl, w+dx/2., 3*h/4., 0., pinb=dx)

# Add joints
mapdl.allsel()
mid = 10
mapdl.mat(mid)
mapdl.type(mid)
mapdl.real(mid)
mapdl.e(nr1,nm1)
mapdl.e(nr2,nm2)

# solving
mapdl.allsel()
mapdl.slashsolu()
mapdl.outres(item='ALL', freq='ALL')
mapdl.d(n_base_1, "ALL")
mapdl.d(n_base_2, "ALL")
mapdl.antype('MODAL')
mapdl.modopt('LANB', 10)
mapdl.mxpand(elcalc="YES")
o = mapdl.solve()
print(o)
```





1st mode out of plan displacement plot. Image by author.

So far this feels just like writing an MAPDL script. Lets functionalize this a bit before integrating into an application. Here we setup the function to allow the user to define the panel width (`panel_w`) and height (`panel_h`) as well as the number of panels (`n_panels`) and a few other properties. We create the mapdl object (which launches the instance and pulls the license) within the function. The `agen` command and hinge-spring representation creation have been updated to account for the user-defined number of panels. We mesh and solve as usual, but then we extract the `grid` and `result` object from the `mapdl` solved `mapdl` object. We also pass the mapdl object to another helper function `get_sene_comps` which extracts the relative portion of strain energy from the hinge and panel components for each mode and returns the data in a pandas dataframe. We then exit the `mapdl` object to release the license and kill the instance before returning the `grid` , `result` , and `df_sene` objects.

```
def make_pv_array(panel_w, panel_h, n_panels=2, nsm_pv = 0.002, tc=0.25, tfs=0.03, k_hinge=10e3
    mapdl = launch_mapdl(override=True, license_type="ansys", cleanup_on_exit=True)
    mapdl.clear()
    mapdl.prep7()
    mapdl.units("BIN")
    mapdl.csys(kcn=0)

    # material property definitions ommitted,  See github repo for full file.

    # create geometry (areas)
    a1 = mapdl.blc4(xcorner=0., ycorner=-panel_w/2, width=panel_h, height=panel_w)
    mapdl.agen(itime=n_panels, na1='ALL', dx=panel_h + dx_panel)

    # mesh
    mapdl.allsel()
```

```
    mapdl.mshape(0, '2D')
    mapdl.mopt('SPLIT', 'OFF')
    mapdl.smrtsize(sizlvl=4)
    mapdl.csys(kcn=0)
    mapdl.amesh("all")

    # base remote points
    mapdl.lsel(type_='s', item='LOC', comp='X', vmin=0.)
    mapdl.nsll(type_='s', nkey=1)
    n_base_1, r_base = make_sbc(mapdl, 0., -panel_w/4., 0., pinb=dx_panel)
    mapdl.lsel(type_='s', item='LOC', comp='X', vmin=0.)
    mapdl.nsll(type_='s', nkey=1)
    n_base_2, r_base = make_sbc(mapdl, 0., panel_w/4., 0., pinb=dx_panel)

    if n_panels > 1:
        for i in range(n_panels):
            x_ref = panel_h * (i + 1) + dx_panel * i
            x_mob = x_ref + dx_panel
            # ref remote points
            mapdl.lsel(type_='s', item='LOC', comp='X', vmin=x_ref)
            mapdl.nsll(type_='s', nkey=1)
            nr1, rr1 = make_sbc(mapdl, x_ref + dx_panel / 2., -panel_w / 4., 0., pinb=dx_panel)
            mapdl.lsel(type_='s', item='LOC', comp='X', vmin=x_ref)
            mapdl.nsll(type_='s', nkey=1)
            nr2, rr2 = make_sbc(mapdl, x_ref + dx_panel / 2., panel_w / 4., 0., pinb=dx_panel)

            # mob remote points
            mapdl.lsel(type_='s', item='LOC', comp='X', vmin=x_mob)
            mapdl.nsll(type_='s', nkey=1)
            nm1, rm1 = make_sbc(mapdl, x_ref + dx_panel / 2., -panel_w / 4., 0., pinb=dx_panel)
            mapdl.lsel(type_='s', item='LOC', comp='X', vmin=x_mob)
            mapdl.nsll(type_='s', nkey=1)
            nm2, rm2 = make_sbc(mapdl, x_ref + dx_panel / 2., panel_w / 4., 0., pinb=dx_panel)

            # Add joints
            mapdl.allsel()
            mid = 10
            mapdl.mat(mid)
            mapdl.type(mid)
            mapdl.real(mid)
            mapdl.e(nr1, nm1)
            mapdl.e(nr2, nm2)

    # make components
    mapdl.esel(type_='s', item='TYPE', vmin=1)
```

```
    mapdl.allsel()

    # solving
    mapdl.allsel()
    mapdl.slashsolu()
    mapdl.outres(item='ALL', freq='NONE')
    mapdl.outres(item='NSOL', freq='ALL')
    mapdl.outres(item='RSOL', freq='ALL')
    mapdl.outres(item='ESOL', freq='ALL')
    mapdl.outres(item='VENG', freq='ALL')
    mapdl.outres(item='MISC', freq='ALL')
    mapdl.d(n_base_1, "ALL")
    mapdl.d(n_base_2, "ALL")
    mapdl.antype('MODAL')
    mapdl.modopt('LANB', 3)
    mapdl.mxpand(elcalc="YES")
    o = mapdl.solve()

    grid = mapdl.mesh.grid
    result = mapdl.result

    df_sene = get_sene_comps(mapdl, ['shell', 'joint'])
    mapdl.exit()
    return result, grid, df_sene
```

Running an analysis is as simple as defining the parameters and calling the function:

```
panel_w = 20.
panel_h = 20.
tc = 0.25
r, g, df_sene = make_pv_array(panel_w=panel_w, panel_h=panel_h,
tc=tc)
```

With this functionalized analysis its time to build the web app.

PyMAPDL Dash Web App

in an easily digestible format. For this application I am expecting the user to primarily change the `panel_h`, `panel_w`, `n_panels`, and `tc` inputs so we will make these obvious with 4always visible numeric inputs. However, we still want the user to be able to change other parameters if needed so we will hide these inside a `collapse` component. Lets check out the user input portion of the layout definition.

```
app.layout = dbc.Container([
    html.H1('PyAnsys MAPDL in a Dash App', className="mt-3"),
    html.P('Design your solar array! (not really, dont use these assumed properties)'),
    dbc.Row([
        dbc.Col([
            dbc.Form([
                dbc.Label('Number of Panels', html_for=f'{APP_ID}_n_panels_input'),
                dbc.Input(id=f'{APP_ID}_n_panels_input', type="number", min=0, max=10, step=1,
            ])
        ]),
        dbc.Col([
            dbc.Form([
                dbc.Label('Panel Width (in)', html_for=f'{APP_ID}_w_panel_input'),
                dbc.Input(id=f'{APP_ID}_w_panel_input', type="number", min=1., value=20, step='
                dbc.FormText("Y direction"),
            ])
        ]),
        dbc.Col([
            dbc.Form([
                dbc.Label('Panel Height (in)', html_for=f'{APP_ID}_h_panel_input'),
                dbc.Input(id=f'{APP_ID}_h_panel_input', type="number", min=1., value=20, step='
                dbc.FormText("X / deployment direction")
            ])
        ]),
        dbc.Col([
            dbc.Form([
                dbc.Label('Panel Core Thickness (in)', html_for=f'{APP_ID}_tc_input'),
                dbc.Input(id=f'{APP_ID}_tc_input', type="number", min=.125, value=.25, step='an
            ])
        ]),
    ]),
    dbc.ButtonGroup([
        dbc.Button('Hide / Show Additional Options', id=f'{APP_ID}_options_collapse_button', co
        dbc.Button('Solve', id=f'{APP_ID}_solve_button', color='primary'),
    ]),
   dbc.Collapse(
      id=f'{APP_ID}_options_collapse',
```

```
                dbc.Col([
                    dbc.Form([
                        html.Div([
                            dbc.Label('Non Structural Mass (lbs/in^2)', html_for=f'{APP_ID}_nsm_i
                            dbc.Input(id=f'{APP_ID}_nsm_input', type="number", min=0., value=0.00
                            dbc.FormText("Photo-voltaics, harness, etc"),
                            ], className="mb-3",
                        )
                    ]),
                    dbc.Form([
                        html.Div([
                            dbc.Label('Facesheet Thickness (in)', html_for=f'{APP_ID}_tfs_input')
                            dbc.Input(id=f'{APP_ID}_tfs_input', type="number", min=0.01, value=0.
                            dbc.FormText("per skin, 2 skins per panel")
                            ], className="mb-3",
                        )
                    ])
                ]),
                dbc.Col([
                    dbc.Form([
                        html.Div([
                            dbc.Label('Hinge Effective Rotational Stiffness (in-lbs/rad)', html_f
                            dbc.Input(id=f'{APP_ID}_k_hinge_input', type="number", min=1000, valu
                            dbc.FormText("Stiffness about pin axis"),
                            ], className="mb-3",
                            )
                    ]),
                    dbc.Form([
                        html.Div([
                            dbc.Label('Panel Separation (in)', html_for=f'{APP_ID}_dx_panel_input
                            dbc.Input(id=f'{APP_ID}_dx_panel_input', type="number", min=0., value
                        ], className="mb-3",
                        )
                    ])
                ]),
                dbc.Col([
                    dbc.Form([
                        html.Div([
                            dbc.Label('Displacement Scale Factor', html_for=f'{APP_ID}_scale_inpu
                            dbc.Input(id=f'{APP_ID}_scale_input', type="number", value=5.),
                        ], className="mb-3",
                        )
                    ]),
                    dbc.Form([
                        html.Div([
```

```
                                label='Show Edges',
                                value=True,
                            ),
                        ], className="mb-3",
                        )
                    ])
                ]),
            ], className="mt-3",
            )
        ]
    )
```

We have our main inputs in lines 8, 14, 21, & 28 (of the gist). We then have a couple of buttons. The first is for opening and closing the collapse to expose additional options, and the other is to execute the solve with all of the set parameters. The solve button will be the input to our main callback. Starting on line 36 we have the Collapse object which holds a grid layout of the additional options: non structural mass, facesheet thickness, hinge effective stiffness, the gap between the panels, the deformation scale factor in the plot, and a toggle to turn on or off element edges. Immediately below the collapse we have an initially hidden progress bar that we will use to update the status of the solve followed by the `dash_vtk.View` and `dcc.Graph` components used to show the actual 3D model and color bar (as explained in the previou pyAnsys article). Below this we have (effectively) a `html.Div` element that we will pass in a summary table.

```
    dbc.Row([
        dbc.Col([
            html.Progress(id=f'{APP_ID}_progress_bar', value='0', max='10', style={'visibility'
        ])
    ]),
    dbc.Row([
        dbc.Col([
            html.Div(
                style={"width": "100%", "height": "60vh"},
                children=[
                    dash_vtk.View(
                        id=f'{APP_ID}_vtk_view',
                        children=dash_vtk.GeometryRepresentation(
                            id=f'{APP_ID}_geom_rep_mesh',
                            children=[],
```

```
                    ),
                ]
            )
        ],
            width=10
        ),
        dbc.Col([
            dcc.Graph(
                id=f'{APP_ID}_results_colorbar_graph',
                style={"width": "100%", "height": "60vh"},
            ),

        ],
            width=2
        )
    ],
    className="g-0"
    ),
    dbc.Row([
        dbc.Col(
            id=f'{APP_ID}_mem_sene_dt_div',
        ),
    ],
        className="mt-3"
    )
])
```

pyAnsys_MAPDL_dash_layout_2.py hosted with ❤ by GitHub view raw

The callbacks for the edge toggle and collapse are pretty simple so I will focus on the main callback. Here we are going to use a relatively new Dash feature: 'long callback'. While this toy example may return results within ~30s (before the browser times out) a more complicated / detailed analysis may not. We have to setup a cache manager and will use the simpler `diskcache` for this example. This is fairly well explained in the dash documentation.

```
@app.long_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}_mem_sene_dt_div', 'children'),
    Input(f'{APP_ID}_solve_button', 'n_clicks'),
    State(f'{APP_ID}_n_panels_input', 'value')
```

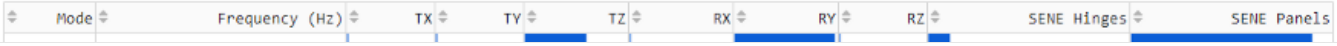
```
        State(f'{APP_ID}_nsm_input', 'value'),
        State(f'{APP_ID}_tfs_input', 'value'),
        State(f'{APP_ID}_k_hinge_input', 'value'),
        State(f'{APP_ID}_dx_panel_input', 'value'),
        State(f'{APP_ID}_tc_input', 'value'),
        State(f'{APP_ID}_scale_input', 'value'),
        prevent_initial_call=True,
        running=[
            (
                Output(f'{APP_ID}_solve_button', "disabled"),
                True,
                False
            ),
            (
                Output(f'{APP_ID}_progress_bar', "style"),
                {"visibility": "visible"},
                {"visibility": "hidden"},
            ),
        ],
        progress=[
            Output(f'{APP_ID}_progress_bar', "value"),
            Output(f'{APP_ID}_progress_bar', "max")
        ],
    )
    def dash_vtk_update_grid(set_progress, n_clicks, n_panels, panel_w, panel_h, nsm_pv, tfs, k_hin

        if any([v is None for v in [n_clicks, n_panels, panel_w, panel_h, nsm_pv, tfs, k_hinge, dx_
            raise PreventUpdate



        set_progress((str(1), str(10)))
        res, ugrid, df_sene_mem = make_pv_array(panel_w, panel_h, n_panels=n_panels, nsm_pv=nsm_pv,
        set_progress((str(8), str(10)))
        f0 = df_sene_mem['freq'].iloc[0]
        dof ='uZ'
        ugrid = plot_nodal_disp(ugrid, res, scale=scale, dof=dof)

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

        fig_cb = make_colorbar(f'Z Displacement (in)<Br>Freq: {f0:0.3f}', rng)

        mesh_state = to_mesh_state(ugrid, field_to_keep=dof)
        set_progress((str(9), str(10)))
```

```
        sort_action='native',
        columns=[
            {'name': 'Mode', 'id': 'mode', 'type': 'numeric', 'format': Format(precision=2, sch
            {'name': 'Frequency (Hz)', 'id': 'freq', 'type': 'numeric', 'format': Format(precis
            {'name': 'TX', 'id': 'TX', 'type': 'numeric', 'format': Format(precision=2, scheme=
            {'name': 'TY', 'id': 'TY', 'type': 'numeric', 'format': Format(precision=2, scheme=
            {'name': 'TZ', 'id': 'TZ', 'type': 'numeric', 'format': Format(precision=2, scheme=
            {'name': 'RX', 'id': 'RX', 'type': 'numeric', 'format': Format(precision=2, scheme=
            {'name': 'RY', 'id': 'RY', 'type': 'numeric', 'format': Format(precision=2, scheme=
            {'name': 'RZ', 'id': 'RZ', 'type': 'numeric', 'format': Format(precision=2, scheme=
            {'name': 'SENE Hinges', 'id': 'joint', 'type': 'numeric', 'format': Format(precisio
            {'name': 'SENE Panels', 'id': 'shell', 'type': 'numeric', 'format': Format(precisio
        ],
        style_data_conditional=(
            data_bars(df_sene_mem, 'TX', clr='rgb(11, 94, 215)') +
            data_bars(df_sene_mem, 'TY', clr='rgb(11, 94, 215)') +
            data_bars(df_sene_mem, 'TZ', clr='rgb(11, 94, 215)') +
            data_bars(df_sene_mem, 'RX', clr='rgb(11, 94, 215)') +
            data_bars(df_sene_mem, 'RY', clr='rgb(11, 94, 215)') +
            data_bars(df_sene_mem, 'RZ', clr='rgb(11, 94, 215)') +
            data_bars(df_sene_mem, 'joint', clr='rgb(11, 94, 215)') +
            data_bars(df_sene_mem, 'shell', clr='rgb(11, 94, 215)')
        ),
    )

    set_progress((str(10), str(10)))
    return dash_vtk.Mesh(state=mesh_state), rng, fig_cb, dt
```

The `Outputs`, `Inputs` and `States` all look pretty normal. we have the 1 'solve' button as our only input and a lot of other `State` objects to collect the user-defined settings when the button was pressed. We are going to add the `prevent_initial_callback=True` to prevent a solution kicking off before we want it to. Now we come to the special features used with long_callback: `running` and `progress`. The running parameter lets us define callbacks to set component properties (1st of 3) while the callback is running (2nd of 3) and when the callback completed (3rd of 3). Here we are disabling the 'solve' button while the callback is running (and enabling it when it completes with `disabled=False`). We are also making the progress bar visible while the callback is running and not when it completes (with `style={visibility: 'hidden'}`). The progress argument lets us update some progress indicator while the callback is running. We can

argument is a function that we pass values to it which correspond to the order of the component properties in the `running` argument list. It only takes a single input, so we put the 2 properties (the progress bar component's `value` and `max` properties) in a `tuple` when we call the `set_progress()` function. So to set the progress bar to a progress of 1 out of 10 we would call `set_progress((1,10))` at the appropriate location in the callback.

Ok now that we got passed that we can move into the callback function. Right up front I am just checking none of the parameters are `None` (which shouldn't happen with `prevent_initial_call=True`). We then set our progress to 1/10 and then call our function to build, mesh, and run our solar array FEM. This function has been tweaked a bit from the notebook version so that we can pass in the `set_progress` function and continue to increment the progress within and a few additional helper functions are used to pull out modal effective mass and merge with the strain energy table. Each of the inputs to the `make_pv_array()` function are parameters pulled from the `State` objects. We get back our result object, the grid, and the dataframe with strain energy breakdown (and now modal effective mass). We need a helper function (`plot_nodal_disp`) to assign the out of plane displacement from the pyMAPDL result object into the vtk grid itself (as a point array). This function just ensures the data set has the same number of elements as the grid and if there are extra nodes in the grid, the result is set to 0. (while making sure the order of data / scoping stays intact). After this we have our grid with the `uz` scalar data and use the min/max from that to help create the colorbar and ensure the min/max of the `dash_vtk` plot is the same as the min/max on the colorbar. We then convert the grid to the `mesh_state` object, and create a Dash `DataTable` to present the modal effective mass and strain energy. Its a bit fancy with another helper function (pulled from the Dash examples) to create bars within each column to easily identify high modal mass DOFs and contributors to strain energy. In the example below we can quickly see the first mode is a cantilever mode (High `TZ` & `RY`) with the panel stiffness contributing most of the strain energy. The 2nd mode is a torsion mode (high `RX`) again dominated by panel stiffness (high `SENE Panels`). To significantly increase these modes we would want to add stiffness to the panels (as opposed to make the hinges stiffer).



Fancy data table highlighting modal mass and strain energy %. Image by author.

Do you see the light?

Echoing what I mentioned in the last article: I believe packaging pyAnsys into a web app is most powerful when the intended user is not a simulation expert. This example was pretty simple, but more advanced MAPDL scripts could be converted to functions. The inputs and outputs here are tailored for a specific use. In a typical Finite Element Analysis you have to weed through a lot of potential results to get the insights you are looking for. This could be trivial for a simulation expert, but not so much for other engineers. A couple of use-cases for integrating pyMAPDL into an app outside of the typical Ansys platforms:

1. Products that are fairly standardized, but may require small well-defined changes or tweaks to meet certain performance parameters (stiffness, strength, etc)
2. Course preliminary models for a business development team, so they can run quick layout / sizing trades to get to a preliminary design without calling in the ‘big guns’

For a final comment I’d like to share why I like web-apps (as opposed to an .exe): everyone uses the same tool. It is always the latest and greatest version and you don’t have to worry about older obsolete (or buggy) older versions floating around.

Thanks for making it this far! (sorry no prize). Check out the GitHub repo for the full files.