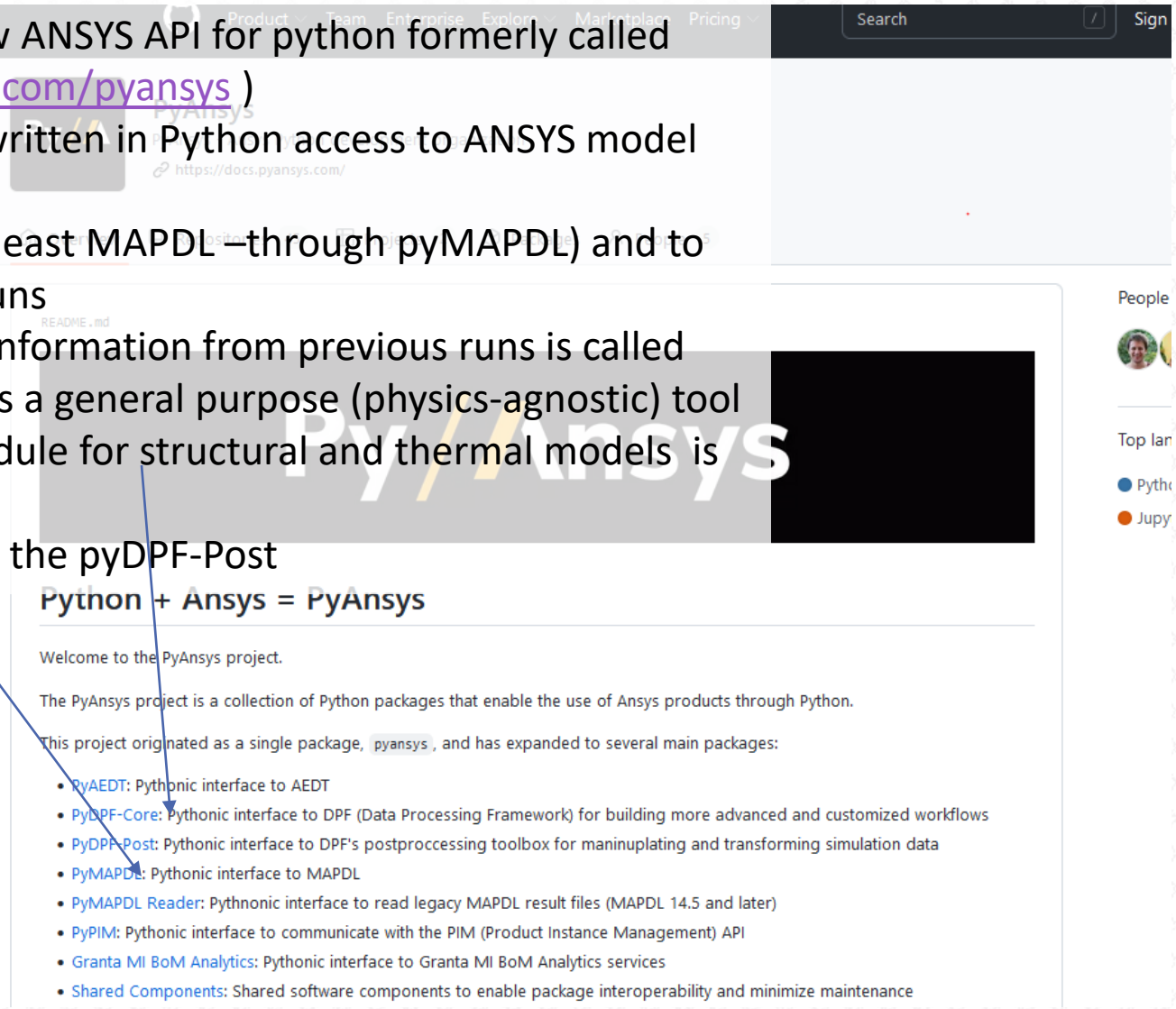
Using Ansys pyMAPDL, pyDPF-Post and more

Part 1: Running and post-processing an ANSYS model outside of ANSYS

Alex Grishin, PhD
PADT, Inc

Preliminaries: What is pyANSYS?

- In this article, we'll explore a relatively new ANSYS API for python formerly called pyANSYS (see the link here: https://github.com/pyansys)
- It's main use is to give external programs written in Python access to ANSYS model information
- There are hooks to both launch ANSYS (at least MAPDL –through pyMAPDL) and to access model information from previous runs
- The main framework for accessing model information from previous runs is called pyDPF (Data Processing Framework). This is a general purpose (physics-agnostic) tool for accessing the data. A more specific module for structural and thermal models is called pyDPF-Post.
- In this article, we'll focus on pyMAPDL and the pyDPF-Post



Preliminaries: What are the benefits of such an interface?

- Most serious ANSYS users would benefit from learning at least the basics of how the DPF-Post and pyMAPDL work for a couple reasons:

    1. To extend their current simulation capabilities. Although we won't dive deeply into it, in this article we'll explore how users may export a model's system equations and post-process a model outside of Ansys These two tasks may seem unrelated, but a future article will tie them together (and show users how to solve the system equations outside of ANSYS!)
    2. To extend their current post-processing capabilities. This is actually the main 'killer-app' of the DPF-Core product. Although ANSYS users have access to the ACT customization process automation tools, that framework is limited to what's currently available in the .NET version that ships with ANSYS. To take a simple example: Suppose users want to perform a custom spectrum analysis (perhaps an industry-specific mode-combination technique) using the eigenvectors from an ANSYS modal solution. In such a case, the ACT framework is not ideal as it has no native capability for manipulating large matrix systems*

- Read more about the DPF-Core and view a quick example here:
  https://dpfdocs.pyansys.com/

*at least none that are exposed to users

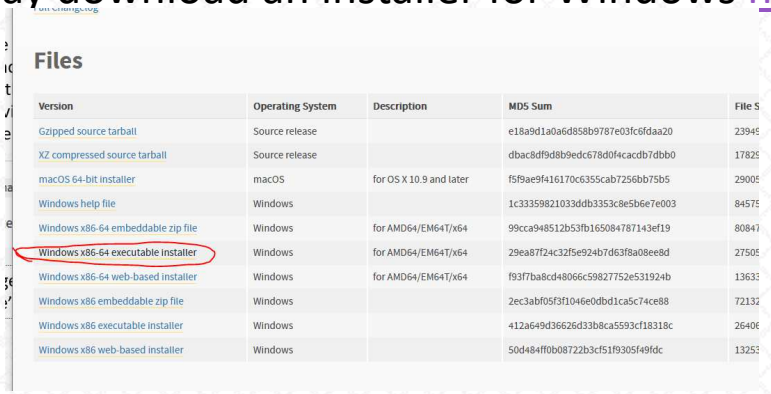
Preliminaries: Installing. From Scratch

- pyDPF-Core, pyDPF-Post and pyMADPL have a complex web of dependencies.
- The best and easiest way to ensure that everything goes smoothly is to allow these installations to install everything they need 'from scratch' –that is to say, outside and independently of any package manager
- This is true even for users who already have a powerful package manager like Anaconda (if you have a standard, default installation). There should be no reason you can't have both an external python installation AND Anaconda. And we're suggesting this is going to be the cleanest solution!*
- When you visit the two github download links (here for DPF-Core, and here for pyMADPL), you'll see the following version support:

The `ansys-mapdl-core` package currently supports Python 3.6 through Python 3.8 on Windows, Mac OS, and Linux.

Install the latest release from PyPi with:

- So, we suggest installing Python 3.8 if you dont' already have it. And again: We mean a 'standalone' installation. Users may download an installer for Windows here



Files

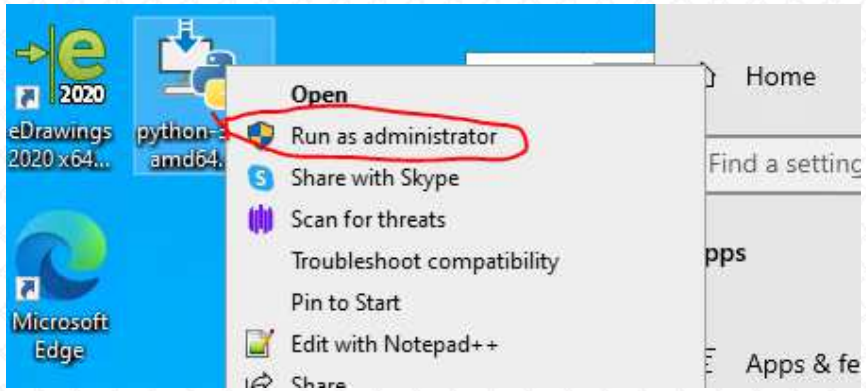
Version	Operating System	Description	MD5 Sum	File S
Gzipped source tarball	Source release		e18a9d1a0a6d858b9787e03fc6fdaa20	2394
XZ compressed source tarball	Source release		dbac8df9d8b9edc678d0f4cacdb7dbb0	1782
macOS 64-bit installer	macOS	for OS X 10.9 and later	f5f9ae9f416170c6355cab7256bb75b5	2900
Windows help file	Windows		1c333598210 33ddb3353c8e5b6e7e003	8457
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	99cca948512b53fb165084787143ef19	8084
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	29ea87f24c32f5e924b7d63f8a08ee8d	2750
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	f93f7ba8cd48066c59827752e531924b	1363
Windows x86 embeddable zip file	Windows		2ec3abf05f3f1046e0dbd1ca5c74ce88	7213
Windows x86 executable installer	Windows		412a649d36626d33b8ca5593cf18318c	2640
Windows x86 web-based installer	Windows		50d484ff0b08722b3cf51f9305f49fdc	1325

*veteran python users should also know that the following useful modules ship WITH the pyANSYS products:

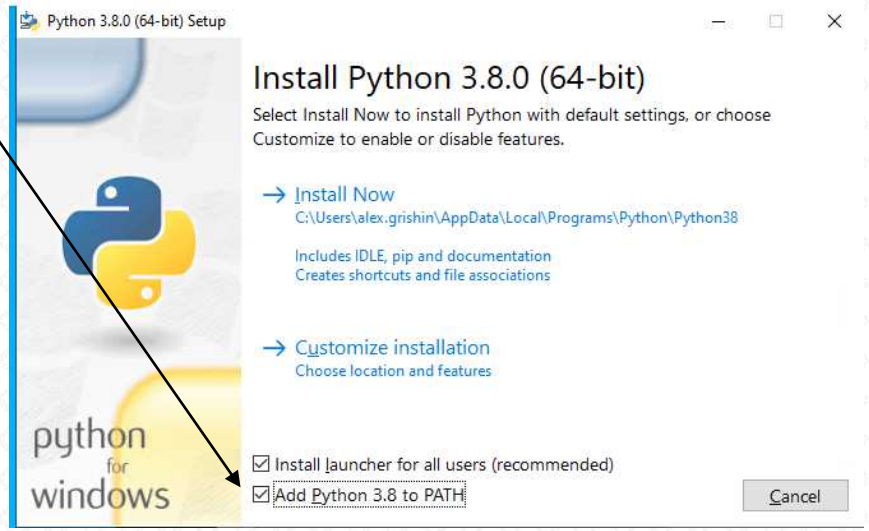
- numpy
- scipy
- pyVista
- matplotlib
- and more...

Preliminaries: Installing. From Scratch

- After downloading the installer, right-click on it and “Run as administrator” (you really should have administrative privileges to install software)

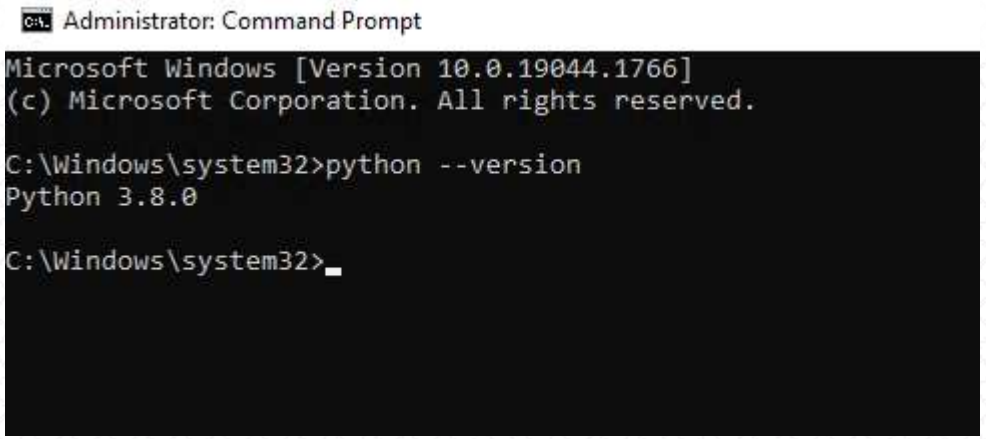


- Check “Add Python 3.8 to PATH” –but only if you don’t already have a path variable set for another installation (or you’re ok with overriding it. Or you simply specify the Scripts folder each time you want to launch an editor).
- An alternative (if you insist on having multiple versions of python installed) is to set custom environment variables –one for each installation. Then execute the environment variable corresponding to the installation you want to access at the command prompt
- Click “Install Now”



Preliminaries: Installing. From Scratch

- Now, launch a command prompt as adminstrator (right-click on Command Prompt from the main menu and select Run as administrator)
- If everything went smoothly so far, you should be able to type “python –version” at the prompt and see the following:



- Install pymadl by typing the following at the prompt (followed by <enter>)

```
pip install ansys-mapdl-core
```

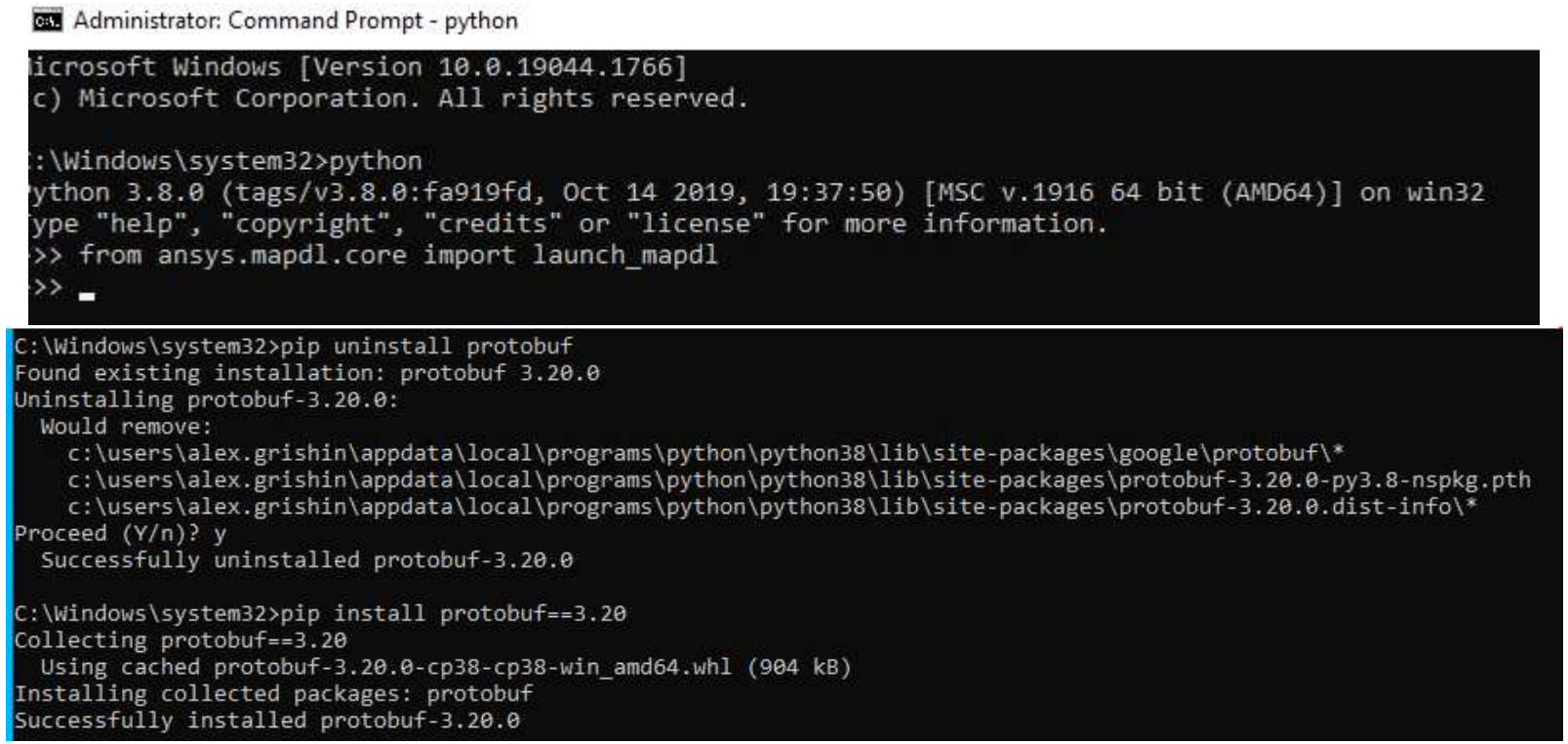
- Install pypdf-post by typing the following at the prompt

```
pip install ansys-dpf-post
```

Preliminaries: Installing. From Scratch

- Try a simple test of the installation at the command prompt. Open a python shell by typing “python”
- Then, at the prompt, type “from ansys-mapdl-core import launch_mapdl”
- If successful, you wont’ see any errors or warnings (like below)
- However, some users may get an error related to the version of Google’s protobuf installed
- If you fall into this category, uninstall protobuf, and then install protobuf version 3.20* (first type “pip uninstall protobuf <enter>” followed by “pip install protobuf == 3.20 <enter>”

- From here, you can install an editor of your choice (or use the command shell)
- We’ll use the standard python shell



*WARNING: This is true as of this writing on 7/13/2022, but will probably not be true a year from now)

The Example Problem: 2D NAFEMS benchmark problem

- We chose the following problem for demonstration
- This is a NAFEMS “Challenge Problem” from 2015 (too late to submit our solution!). The link may be found here: https://www.nafems.org/downloads/nbc1.pdf
- We thought it might help users understand the problem more intimately by looking at the solution in different frameworks (that post-process element results slightly differently) and understanding how the post-processing is done
- In a future article, even more insight will be gained by actually looking at the underlying system equations

Stress at the Centre of a Square Plate with Linear Boundary Tractions

A unit square homogeneous and isotropic steel plate is centred at the origin of the XY-plane with edges parallel with the coordinate axes and is loaded with linearly distributed normal and tangential boundary tractions as shown in figure 1. The plate can be assumed to be thin so that a plane-stress constitutive relationship is appropriate and for convenience a unit thickness may be used.

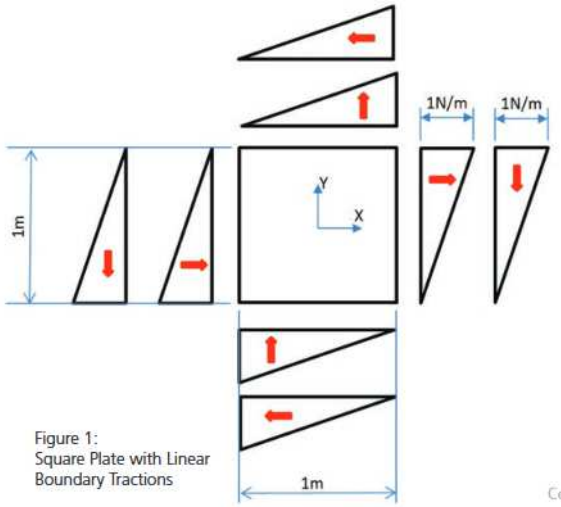


Figure 1:
Square Plate with Linear Boundary Tractions

The Challenge

The challenge is to produce **two models** of this problem in your finite element software and then answer some questions. The first model should use a **single four-noded element** and the second a **single eight-noded element**. As engineers interested in the integrity of the plate we might wish to see the distribution of von Mises stress over the plate. We would like you to provide:

a) Numerical values for the von Mises stress at the centre of the plate for both models,

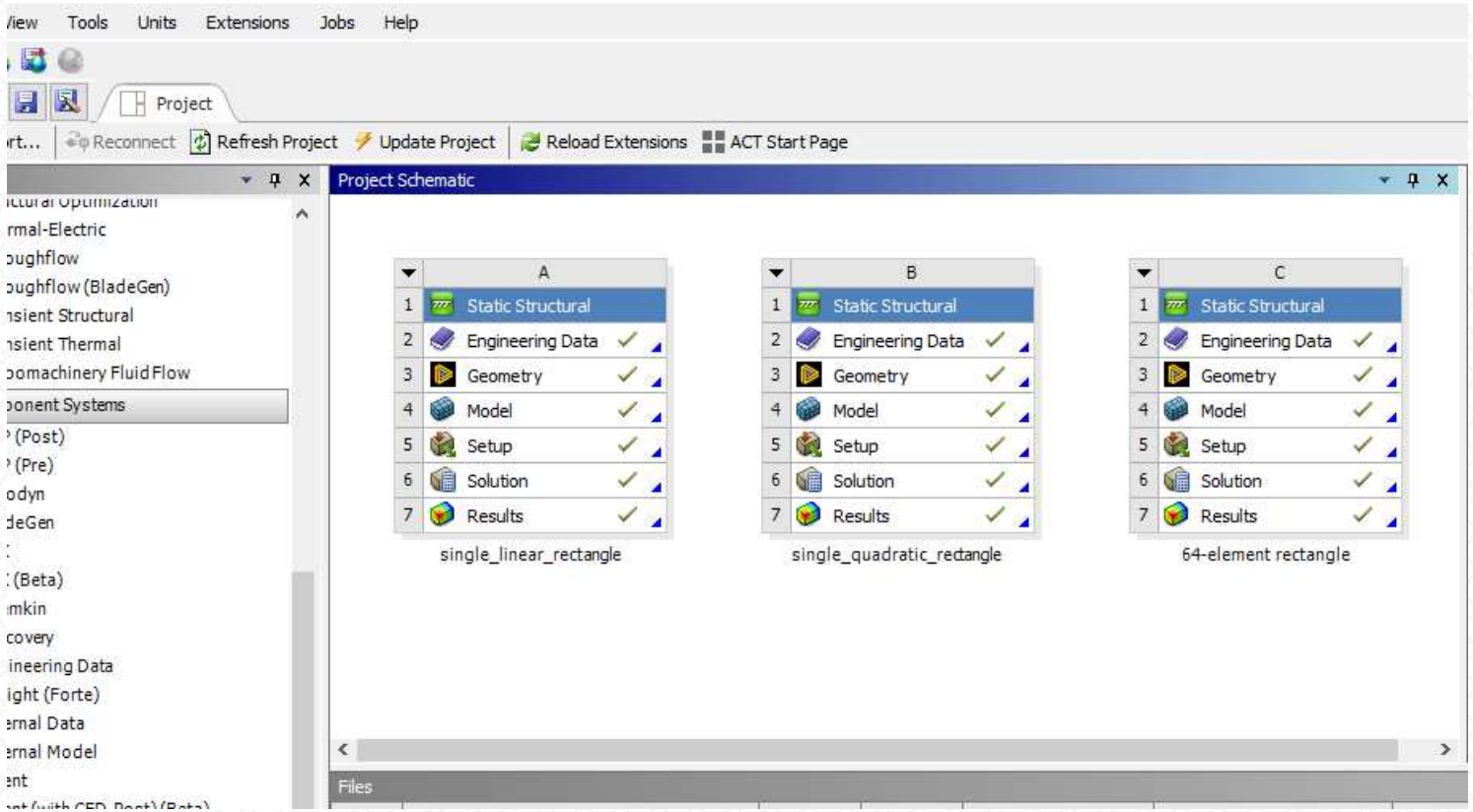
b) A statement as to which of these values is correct,

c) Contour plots of von Mises Stress for both models,

d) A brief commentary on how you modelled the problem and what, if anything, of interest you note about this problem – please include details of the software that you used.

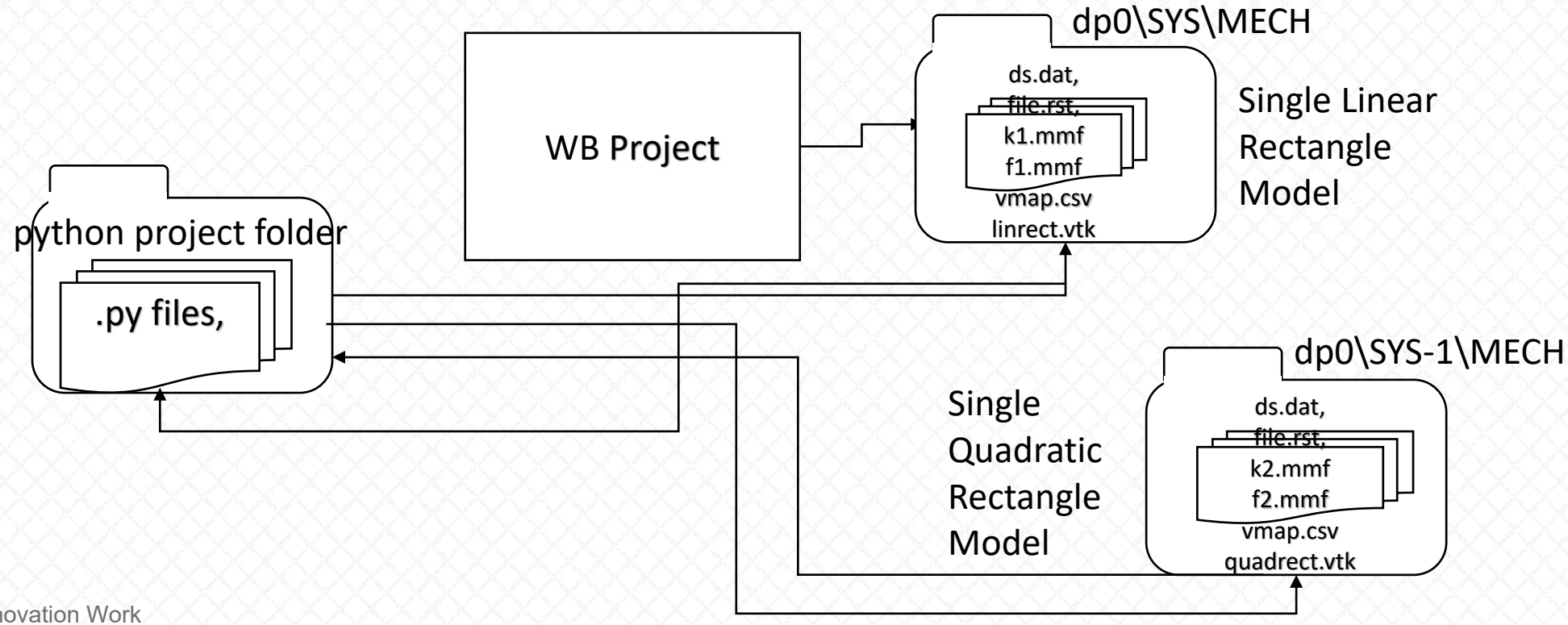
The Example Problem: In Workbench...

- The challenge problem asks us to build both a single linear element model, as well as a single quadratic element model. The “challenge” is to explain the differences and make statements about which is “correct”. These models are represented in our Workbench project in systems A and B respectively
- In addtion, we’ve added another model for completeness in system C: A 64-element quadratic model. We should be able to spot a trend as to what’s going on as we go from left to right..



The Example Problem: What we'll be doing...

- As we mentioned at the outset, we chose a simple model as a benchmark to demonstrate pyMAPDL and pydpf-post's capabilities
- The setup is this: We built the model in Workbench, and so we'll be accessing the project's solution folder to open the model in pyMAPDL and export the matrices.
- We'll also run the model in pyMAPDL and view the results using pydpf-post
- The python files will reside in a python project folder



The Example Problem: Single Linear Rectangle Model

- In a challenges like this, the authors intentionally omit details. For example, how should we constrain the model? Well, that's part of the challenge
- Without exploiting the symmetry (or anti-symmetry in this case), the mininal constraint condition suggests that we can only apply constraints where the applied loads are zero (locations 1 and 2 below)
- And since we don't know how to apply this sort of combined non-uniform traction in Workbench, we script it in*

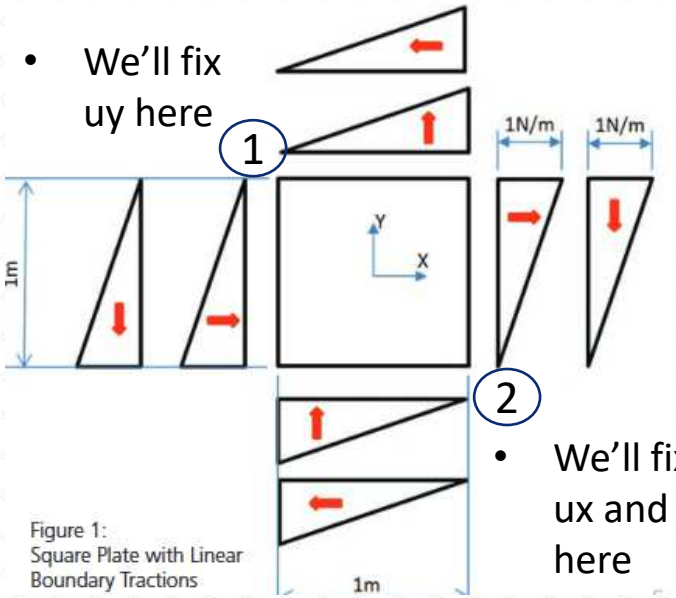
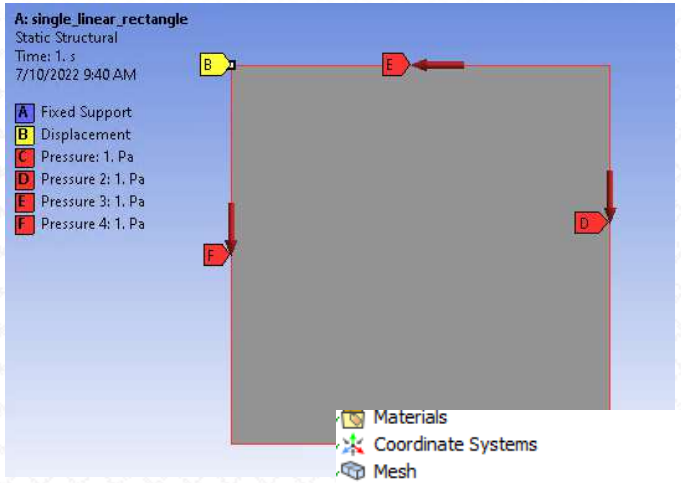
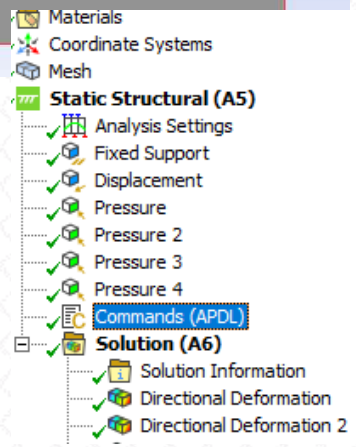


Figure 1:
Square Plate with Linear Boundary Tractions






```
sfedele,all,all,all
esel,s,type,,2
sfe,all,1,pres,1,-1.0,0.0 !x-dir
sfe,all,2,pres,1,1.0,0.0  !y-dir

esel,s,type,,3
sfe,all,1,pres,1,0.0,1.0 !x-dir
sfe,all,2,pres,1,0.0,-1.0  !y-dir

esel,s,type,,4
sfe,all,1,pres,1,-1.0,0.0 !x-dir
sfe,all,2,pres,1,1.0,0.0  !y-dir

esel,s,type,,5
sfe,all,1,pres,1,0.0,1.0 !x-dir
sfe,all,2,pres,1,0.0,-1.0  !y-dir
allsel,
```

*The details of this are beyond the scope of this article. Contact the author if you have questions...

The Example Problem: Single Linear Rectangle Model

- We've already run this model in Workbench, but before we show the result, we'll do the following:
  - Re-read the model in the solution folder and run it using pyMAPDL
  - Export the matrices using MADPL's using APDL Math functions specifically designed for this purpose (see Eric Miller's introductory article on APDL Math here: https://www.padtinc.com/2012/02/23/apdl-math-access-to-the-ansys-solver-matrices-with-apdl/ (again, we won't be using the exported matrices in this article –that will have to wait for a future article)
  - View the results in pypdf-post
  - to make things a little more compact, we've bundled most of the necessary pyMADPL code into a python file called kmat.py shown at right

file: **kmat.py**

```
from ansys.mapdl.core import launch_mapdl
from ansys.dpf import post
import numpy as np
import sys,os

def runandsolve(mapdl,path):
    #mapdl = launch_mapdl(run_location=rundir,override=True)
    mapdl.finish()
    mapdl.clear()
    mapdl.input(path)
    mapdl.run('/solu')
    mapdl.ematwrite('yes')
    mapdl.solve()

def matrix_export(mapdl,kname='kmat',fname='fmat'):
    mapdl.finish()
    mapdl.run('*smat,vks,d,import,full,file.full,stiff')
    mapdl.run('*dmat,f,d,import,full,file.full,rhs')
    mapdl.run('*smat,usr2solv,d,import,full,file.full,usr2solv')
    mapdl.run('*dim,snums,,usr2solv_rowdim')
    mapdl.run('*vfill,snums(1),ramp,1,1')

    mapdl.run('*vec,vsnums,d,import,apdl,snums')
    mapdl.run('*mult,usr2solv,tran,vsnums,,vmap')
    runstr = '*export,vks,mmf,' + kname + '.mmf'
    mapdl.run(runstr)
    runstr = '*export,f,mmf,' + fname + '.mmf'
    mapdl.run(runstr)
    mapdl.run('*export,vmap,csv,vmap.csv')
```

The Example Problem: Single Linear Rectangle Model

- Ok. Now, all we have to do is open a python shell and execute the following lines of python (see next slide)

file**: linrect.py**

- Change this path to your location before execution!

```
import os
import sys
wpath = r"C:\Users\alex.grishin\exporting_matrices"
sys.path.append(wpath)
os.chdir(wpath)
from kmat import *

solverfilesloc = wpath + "\\plane_stress_files\\dp0\\SYS\\MECH"
dspath = solverfilesloc + "\\ds.dat"
resultpath = solverfilesloc + "\\file.rst"
mapdl = launch_mapdl(run_location=solverfilesloc,override=True)
runandsolve(mapdl,dspath)
matrix_export(mapdl,kname='k1',fname='f1')

solution = post.load_solution(resultpath)
disp = solution.displacement()
stress = solution.stress()
disp.x.plot_contour(cpos='xy',cmap='rainbow',n_colors=9)
stress.xx.plot_contour(cpos='xy',cmap='rainbow',n_colors=9)
mapdl.exit()
```

- Specify the python working directory
- import kmat.py

- Export a stiffness matrix called k1 and a load vector called f1

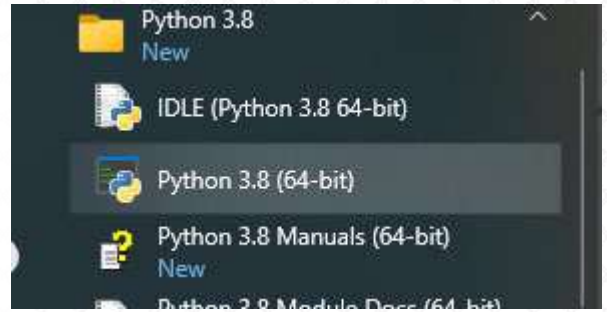
- This function is defined in kmat.py
- This function is defined in kmat.py

- Plot ux
- Plot sx

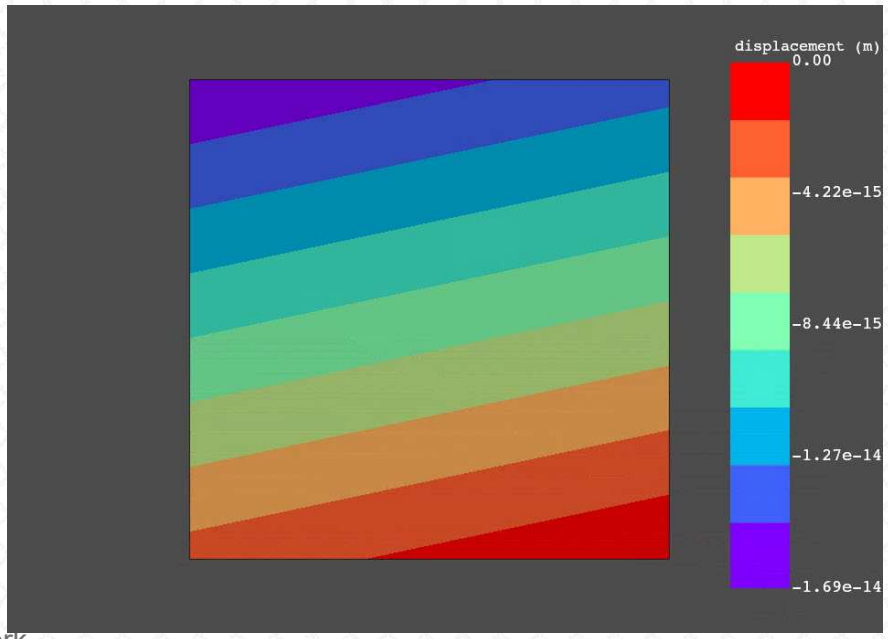
- close the MAPDL connection

The Example Problem: Single Linear Rectangle Model

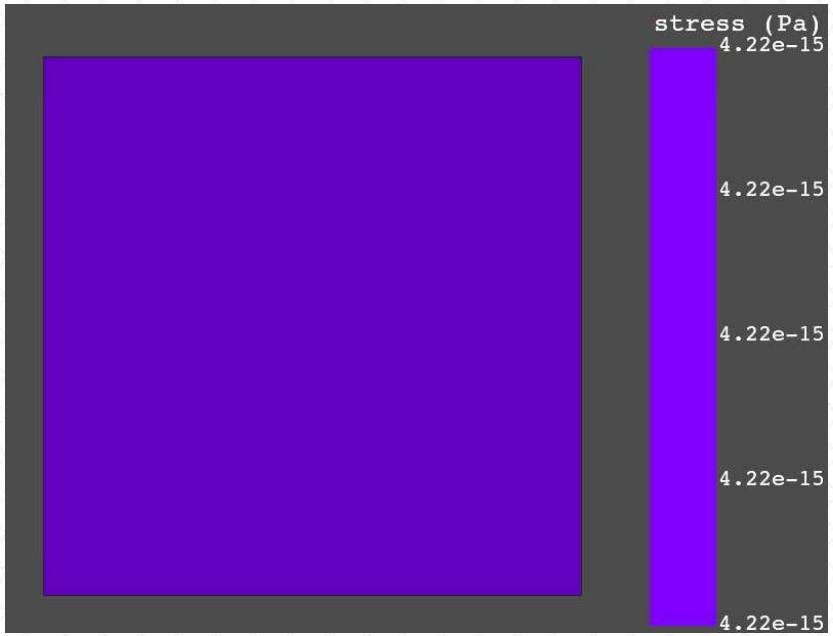
- From the Start Menu, open a python shell and simply cut-and-paste the lines from the previous slide into the shell
- You'll see the displacement plot first pop up in a 'blocking' window
- You'll have to explicitly close this window before you can see the stress result, which will pop up the same way
- Just make sure to close both windows to execute all the code
- Note that this behavior is a feature of the python shell we're using to execute this code. Different editors will have behave differently when confronted with an interactive blocking process



u_x



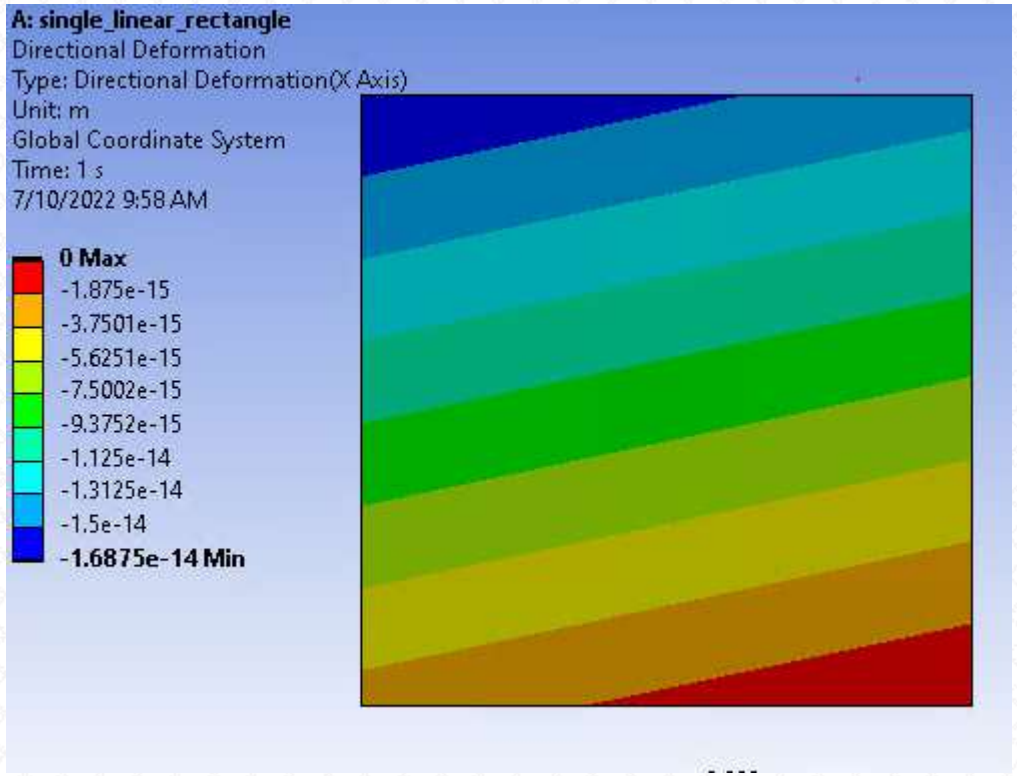
σ_x



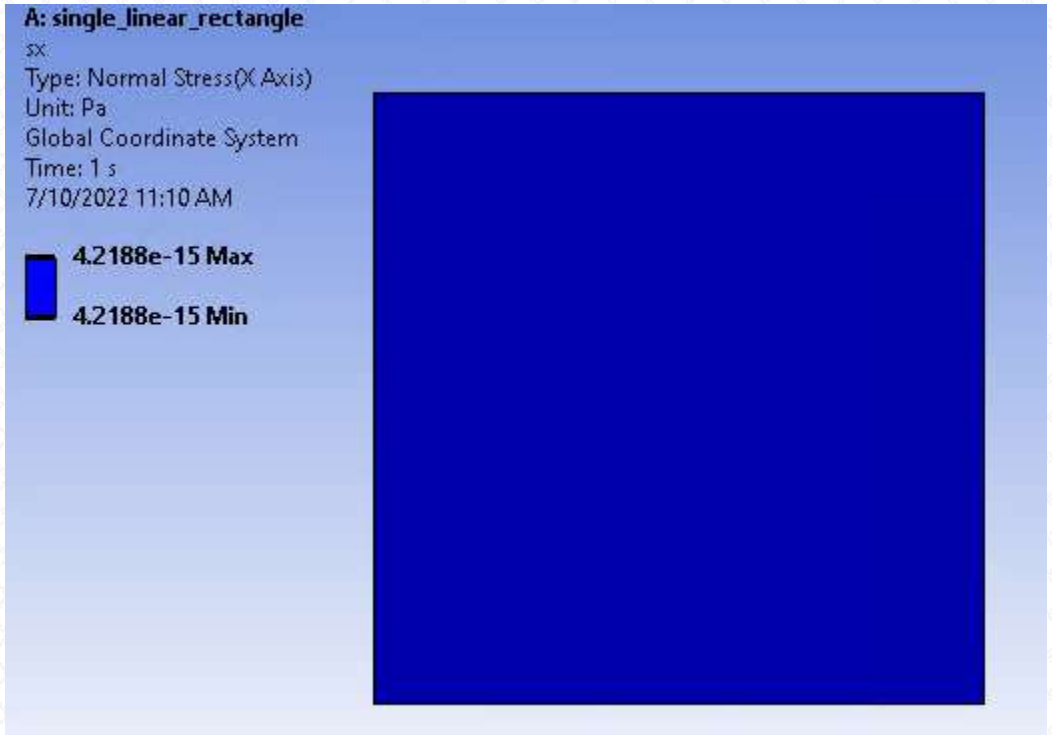
The Example Problem: Single Linear Rectangle Model

- Just verify that these plots are the same as what's shown in Workbench
- This verifies the NAFEMS solution for the linear rectangle case: A null result. Dont' worry, there's nothing wrong here. The authors of the challenge problem specifically designed it to produce a null result over a linear rectangle!

u_x



σ_x



The Example Problem: Single Quadratic Rectangle Model

- Following the NAFEMS challenge problem directions, next verify the single quadratic element by cutting-and-pasting the code at right into a new python shell
- Notice, this code is nearly identical to the code we previously executed for the linear rectangle, but this time we're pointing to a different folder for the ds.dat and results file (see slide 10)

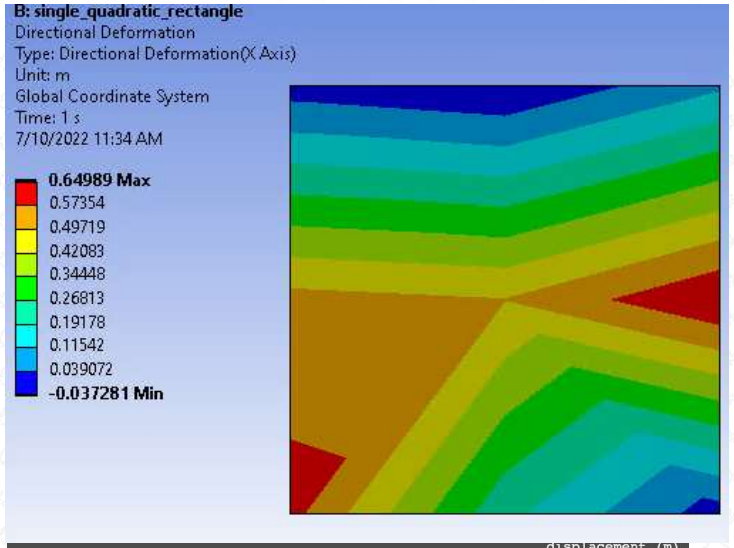
file**: quadrect.py**

```
import os
import sys
wpath = r"C:\Users\alex.grishin\exporting_matrices"
sys.path.append(wpath)
os.chdir(wpath)
from kmat import *

solverfilesloc = wpath + "\\plane_stress_files\\dp0\\SYS-1\\MECH"
dspath = solverfilesloc + "\\ds.dat"
resultpath = solverfilesloc + "\\file.rst"
mapdl = launch_mapdl(run_location=solverfilesloc,override=True)
runandsolve(mapdl,dspath)
matrix_export(mapdl,kname='k2',fname='f2')

solution = post.load_solution(resultpath)
disp = solution.displacement()
stress = solution.stress()
disp.x.plot_contour(cpos='xy',cmap='rainbow',n_colors=9)
stress.xx.plot_contour(cpos='xy',cmap='rainbow',n_colors=9)
mapdl.exit()
```

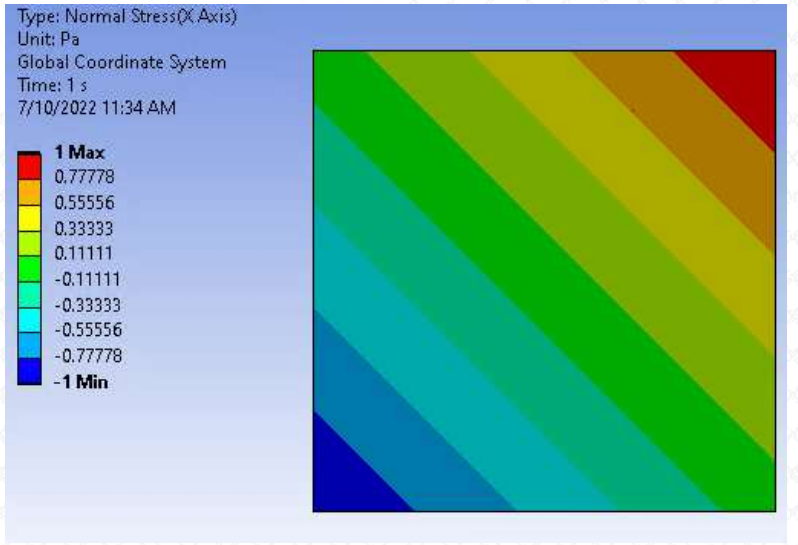
The Example Problem: Single Quadratic Rectangle Model

- And again: compare

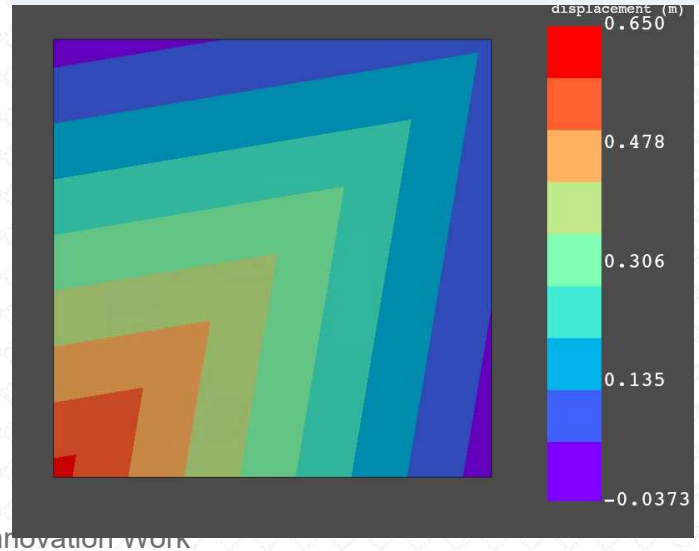
u_x (Workbench)



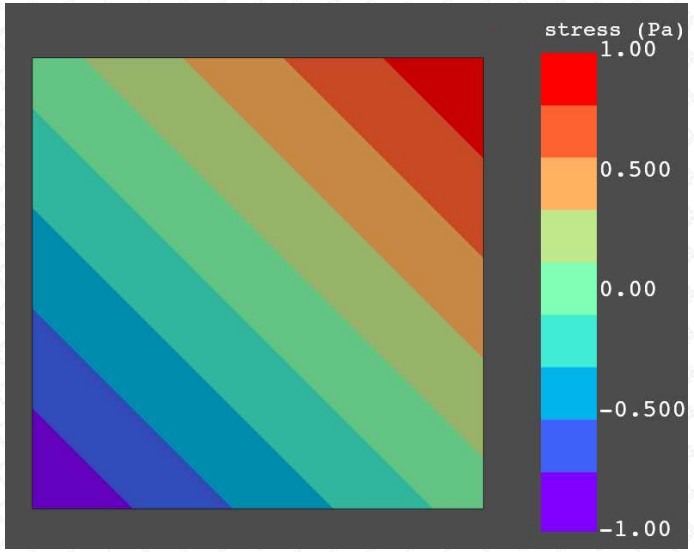
σ_x (Workbench)



u_x (dpf-post)



σ_x (dpf-post)



The Example Problem: Single Quadratic Rectangle Model

- Everything looks identical except for the displacement plot. Why?
- Well, first verify that the nodal values ARE the same

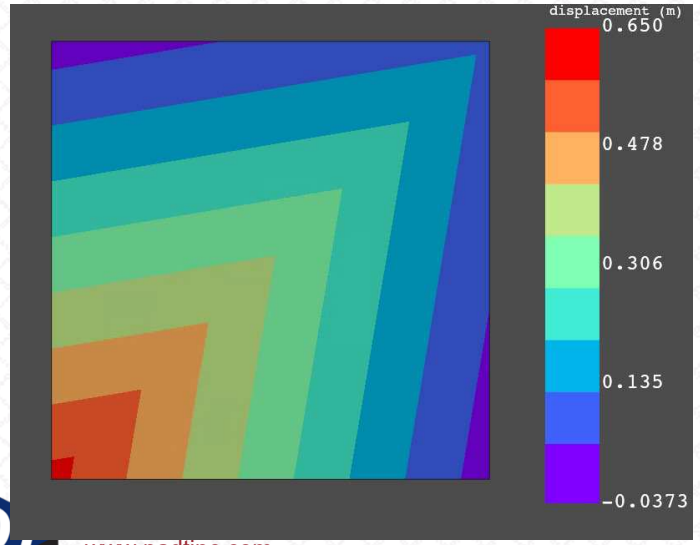
- This line in DPF-Post returns a numpy array of ux values...

```
disp.x.get_data_at_field()
✓ 0.5s
array([ 5.99561404e-01,  4.34974731e-13,  5.25109649e-01,  0.00000000e+00,
        2.12280702e-01,  1.00438596e-01,  6.49890351e-01, -3.72807018e-02])
```

- And this line tells us what nodes those values correspond to...

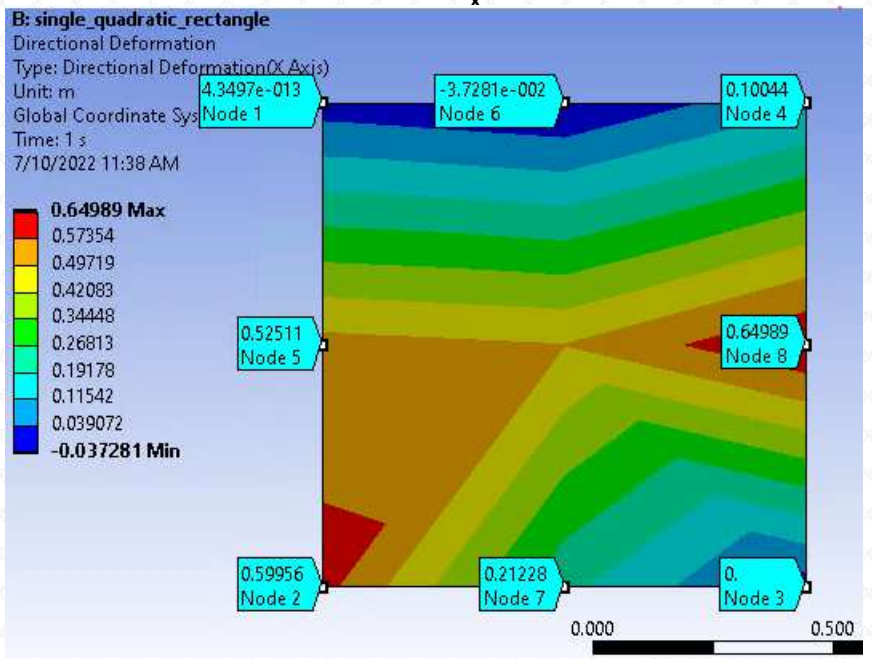
```
disp.x.get_scoping_at_field()
✓ 0.4s
[2, 1, 5, 3, 7, 4, 8, 6]
```

u_x



- They ARE the same values
- It's just that the DPF-post plot seems to just be interpolating the 4 corner node values (or something. We'll get to the bottom of this!)

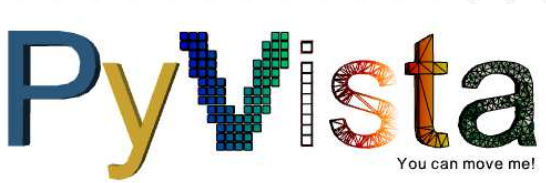
u_x



The Example Problem: Single Quadratic Rectangle Model

- It's worth spending a little time contemplating what's going on here
- To help with this, we'll first introduce a VERY useful tool which came to us silently, without charge and without any mention when we downloaded the pyANSYS modules: pyVista
- All the mesh-based contour plots we see in pyMAPDL, pyDPF-Post, and DPF-Core are provided using this tool 'under the hood'
- To prove this, type the following in the same open python session (we want the 'mapdl' and 'solution' objects to be instantiated before doing this)

```
import pyvista as pv <enter>
mapdl.mesh.grid <enter>
solution.mesh.grid <enter>
```



- You should see something like this:

```
In [21]: type(mapdl.mesh.grid)
Out[21]: pyvista.core.pointset.UnstructuredGrid

In [22]: type(solution.mesh.grid)
Out[22]: pyvista.core.pointset.UnstructuredGrid
```

- So, both the mapdl.mesh and the solution.mesh use a pyvista core pointset.UnstructuredGrid)

The Example Problem: Single Quadratic Rectangle Model

- Now, copy the grid to a new unstructured grid called 'copygrid' by typing:

  `copygrid = solution.mesh.grid.copy() <enter>`
- Then type `copygrid <enter>`
- This is what you should see:

```
In [67]: copygrid
Out[67]:
UnstructuredGrid (0x2a350ff2580)
  N Cells:  1
  N Points: 8
  X Bounds: 0.000e+00, 1.000e+00
  Y Bounds: 0.000e+00, 1.000e+00
  Z Bounds: 0.000e+00, 0.000e+00
  N Arrays: 1
```

- This makes sense, right? We should be looking at a single quadrilateral rectangle, right?
- Note that if you copied the mapdl.mesh.grid instead, you would see 5 elements. This is because that mesh differs in that it also contains the surface effect elements (and these are just line elements coincident with the quadrilateral edges) which are used to apply the surface tractions...

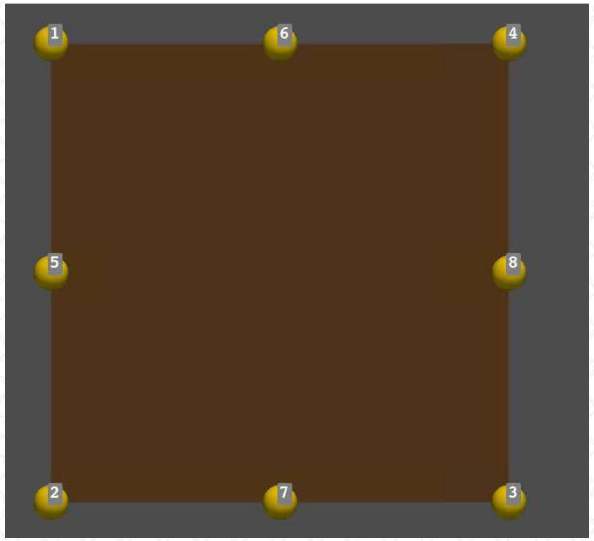
The Example Problem: Single Quadratic Rectangle Model

- Ok. Now, we're going to use pyVista to create a useful and pretty plot of this mesh and it's nodes. Just cut-and-paste the following into your open python session:

```
nodes = pv.PolyData(copygrid.points)
plabels = [str(i+1) for i in range(myquad.n_points)]
p = pv.Plotter()
p.add_mesh(copygrid,color='brown')
p.add_point_labels(nodes,plabels,font_size=20,
        point_color='gold',
        point_size=40,render_points_as_spheres=True,
        always_visible=True)
p.camera_position='xy'
p.show()
```

- create a pv PolyData object to render the nodes
- create node labels
- create a Plotter instance
- add the mesh to the plotter
- add the node (PolyData) object
- set the camera for 2D viewing
- show the mesh

- Here's what you should see!

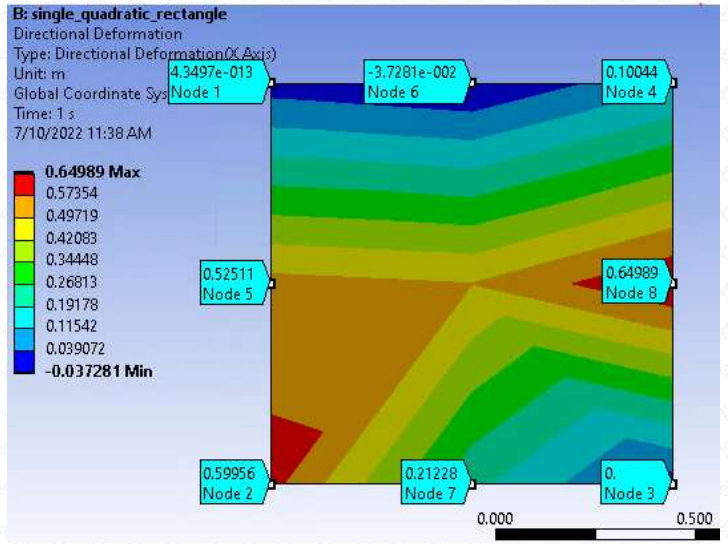


The Example Problem: Single Quadratic Rectangle Model

- Returning to our problem. From slide 18, we learned the following about the nodal displacement x-component values:

- According to Workbench, the solution should be as shown as shown at the right:

node	ux
1	4.34974731E-13
2	5.99561404E-01
3	0.00000000E+00
4	1.00438595E-01
5	5.25109649E-01
6	-3.72807018E-02
7	2.12280702E-01
8	6.49890351E-01



- This solution can be verified in pyDPF-Post by creating a numpy array to store the data like so:

```
ux = np.zeros(8)
ids = np.array(disp.x.get_scoping_at_field())
ux[ids-1] = disp.x.get_data_at_field()
```

- pyvista arrays are zero-based, while ANSYS numbering is always ones-based (a FORTRAN legacy) so we have to subtract 1

- Initialize a numpy array to store the ux values
- store the nodal ids (like in slide 18)
- get the solution but store it ine correct order

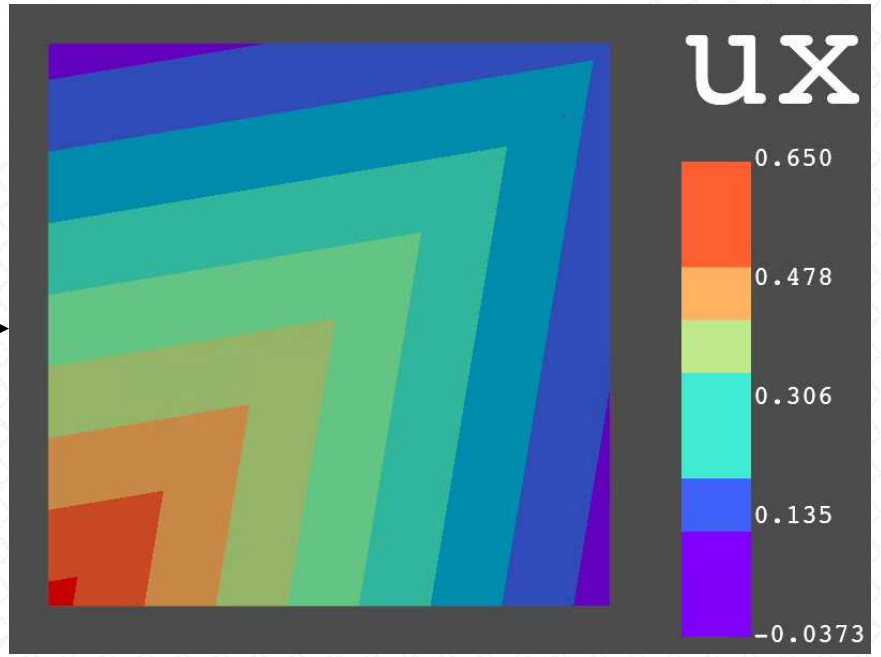
The Example Problem: Single Quadratic Rectangle Model

- We'll use our copied pyVista mesh to plot these values!
- This is remarkably easy to do with pyVista. All we have to do is create a 'point_data' array with these values. And pyVista supplies us with a shortcut for doing so. Such arrays are managed by pyVista within python dictionaries.

```
copygrid['ux'] = ux
```

- This line adds the result data to the grid for plotting. Note the dictionary-like syntax
- Plot the displacement like so:

```
p = pv.Plotter()
p.add_mesh(copygrid,scalars='ux',
        cmap='rainbow',n_colors=9)
p.camera_position='xy'
p.show()
```



- This is what you should see!
- This is identical to the pyDPF-Post plot

The Example Problem: Single Quadratic Rectangle Model

- Ok. We've verified the pyDPF solution. But we still neeed to explain why it differs from the WB solution
- Here's where things get really interesting (for readers interested in the guts of FEA)
- We menionted (on slide 18) that pyDPF-Post seems to be only interpolating the corner nodes, and so it does –even for the plot we just created OUTSIDE of both ANSYS and pyDPF-Post where we were quite careful to map all the nodal results to their correct place.
- We can verify that pyDPF-Post is doing that by querying the mesh's connectivity:

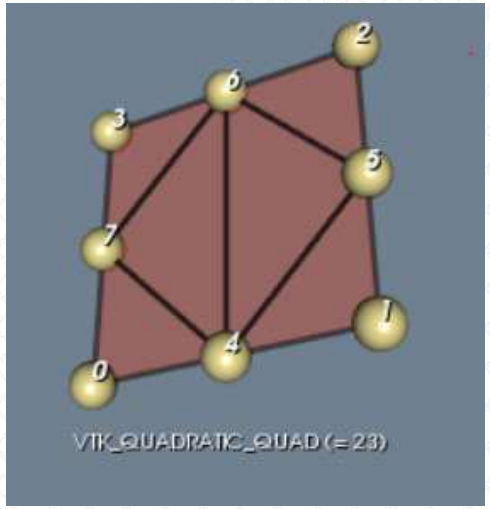
```
In [139]: mygrid.cell_connectivity
Out[139]: array([1, 2, 3, 0, 1, 2, 3, 0], dtype=int64)
```

- Remember that in pyVista (and python generally), arrays are indexed by zero. So when we query this mesh's node connectivity, we expect to see (according to the ANSYS convention and mesh we see in Workbench):
  - [0,1,2,3,4,6,7,5]
- What's happening is that *pyDPF-Post is copying the mid-side nodes to lie on top of the corner nodes*!

The Example Problem: Single Quadratic Rectangle Model

- We can use pyVista to generate a contour plot like the one shown in Workbench
- Hopefully, doing so will reveal much about how modern FEA software works AND the NAFEMS challenge problem
- First, let's create a better displacement plot. We'll build on what we've learned so far and create a quadratic quadrilateral mesh 'from scratch'. This time, we'll properly define the midside nodes
- pyVista is a high-level python wrapper around a larger and older framework called VTK
- When we define a mesh ('grid' in VTK parlance) in pyVista, we're using the names and conventions of VTK. So, for example, to define a quadratic quadrilateral, we first have to look that up in VTK (see the link here).
- The type we're looking for is called VTK_QUADRATIC_QUAD (id = 23)
- To create this grid type in pyVista with the nodes we already have, simply cut-and-paste the following into the python session:

```
import vtk
cells = np.array([8,0,1,2,3,4,6,7,5])
cell_type = np.array([vtk.VTK_QUADRATIC_QUAD])
myquad = pv.UnstructuredGrid(cells,cell_type,copygrid.points)
```



The Example Problem: Single Quadratic Rectangle Model

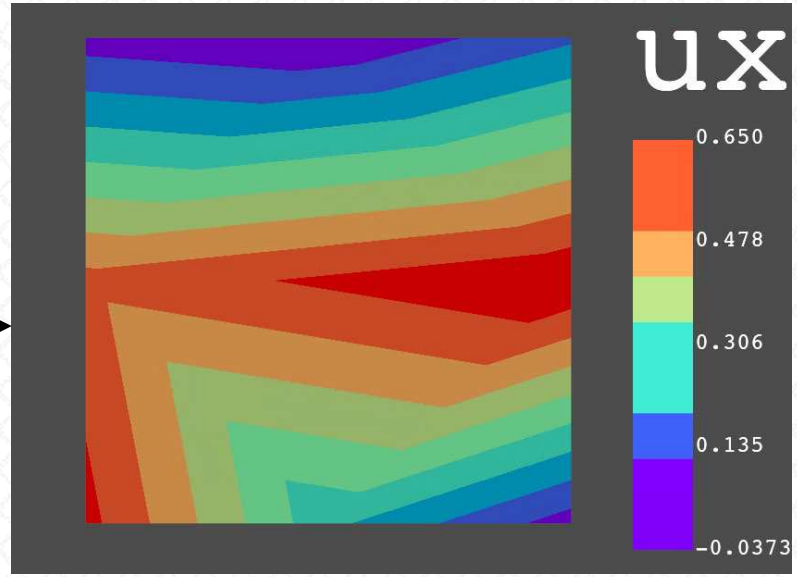
- Now add the nodal displacement data as before:

```
myquad['ux'] = ux
```

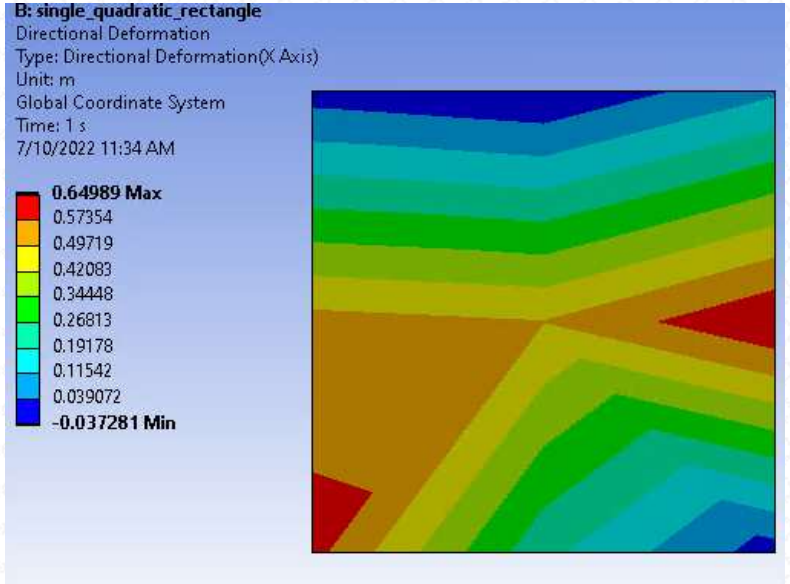
- And plot the displacement:

```
p = pv.Plotter()
p.add_mesh(myquad,scalars='ux',cmap='
rainbow',n_colors=9)
p.camera_position='xy'
p.show()
```

- This is what we get!



- Compare!



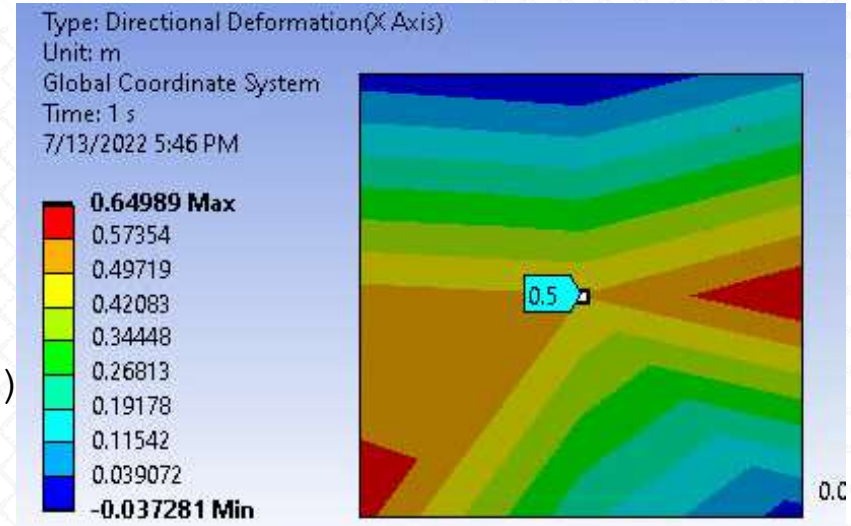
The Example Problem: Single Quadratic Rectangle Model

- Well, that's MUCH better. At least values along edges look right.
- It looks like the Workbench solution still has an advantage, though.
- Notice that we can query the center of the mesh, even though we DON'T HAVE A NODE THERE
- Remember, a finite element solution is smooth throughout it's domain (sometimes called 'the support'). Though we don't have nodes within that domain, we should still be able to query values ANYWHERE (via the shape functions). This fact is leveraged when creating finite element contour plots (even if users can't query any location they want. In a future article, we'll show how DPF-Post and pyVista will allow us to overcome obstacles like that)

- To capture this additional information, we'll use a hgher order grid called VTK_BIQUADRATIC_QUAD (id=28)

- add a point at the center

```
import vtk
eightpts = np.array(myquad.points)
ninepts = np.vstack((eightpts,np.array([0.5,0.5,0.])))
cells = np.array([9,0,1,2,3,4,6,7,5,8])
cell_type = np.array([vtk.VTK_BIQUADRATIC_QUAD])
mybiquad = pv.UnstructuredGrid(cells,cell_type,ninepts)
```



The Example Problem: Single Quadratic Rectangle Model

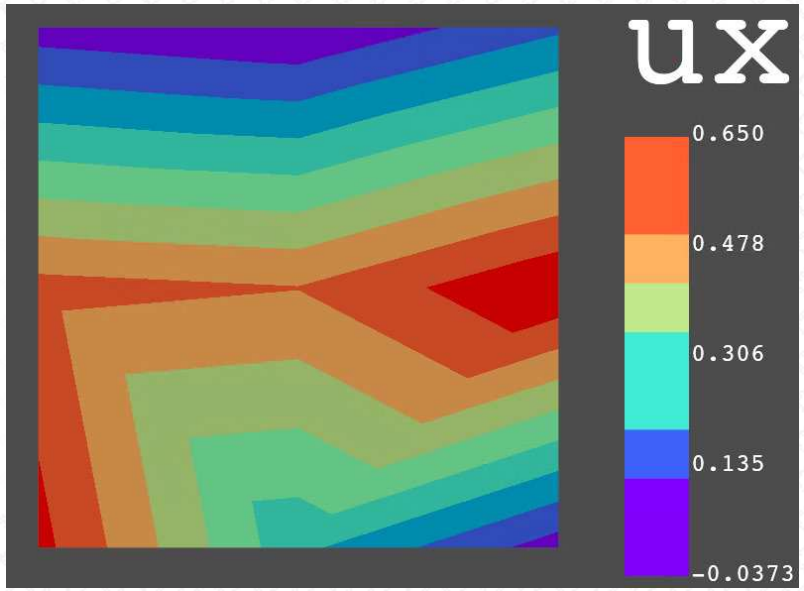
- Now, append the center ‘ux’ value from Workbench to our ux solution and add it the mesh:

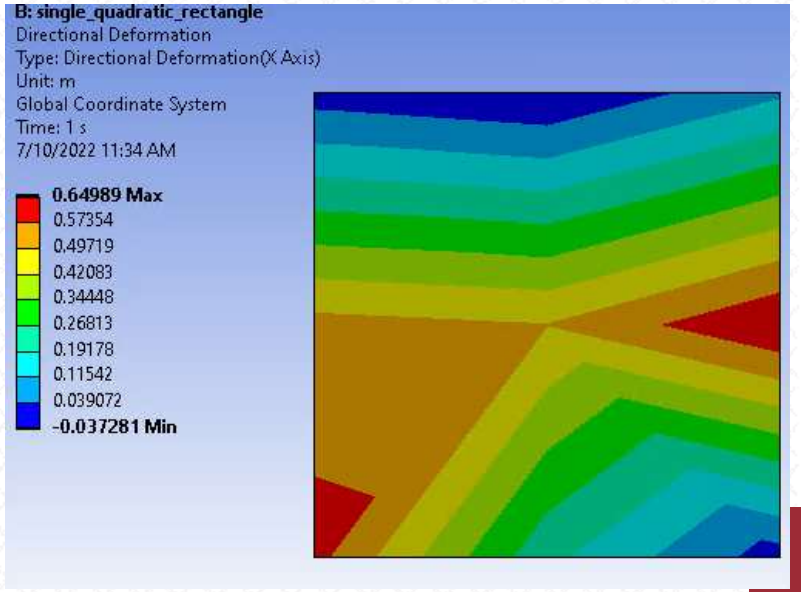
```
ux = np.append(ux,0.5)
mybiquad['ux'] = ux
```

- And plot as before...

```
p = pv.Plotter()
p.add_mesh(mybiquad,scalars='ux',cmap
      ='rainbow',n_colors=9)
p.camera_position='xy'
p.show()
```

- And compare...
- This is as far as we can go with a single cell in pyVista. We can verify that every node has the correct value
- Differences in the two plots are due to different plotting algorithms. We will return to this in a future article





The Example Problem: Single Quadratic Rectangle Model

- Let's return again to the challenge problem. Recall that we get a null solution (both displacements and stresses) with the linear rectangle. We're NOT getting a null solution now
- As we mentioned, this problem was designed to produce a null solution for a linear rectangle, but the exact solution is only zero over a region of the rectangle as reflected in the (exact) component stress solutions shown below (we've already verified σ_x)

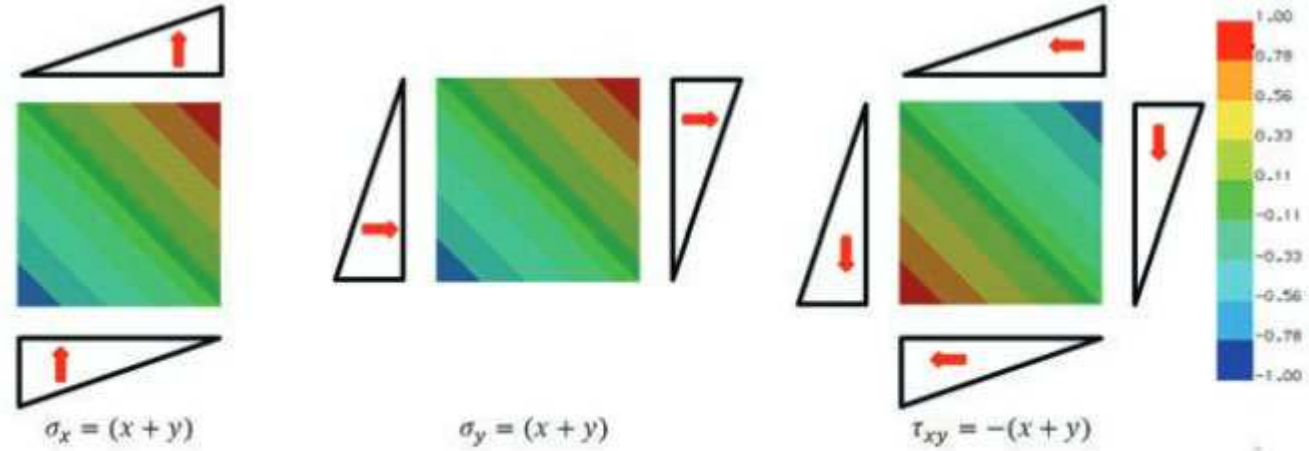


Figure 8: Stress field and corresponding boundary tractions (eight-noded element model)

The Example Problem: Single Quadratic Rectangle Model

- Let's follow the author and reproduce some of the "low fidelity" stress results he reports.
- In particular, we want to show how CS1 and CS2 below can come about with the solution post-processing techniques we've just seen

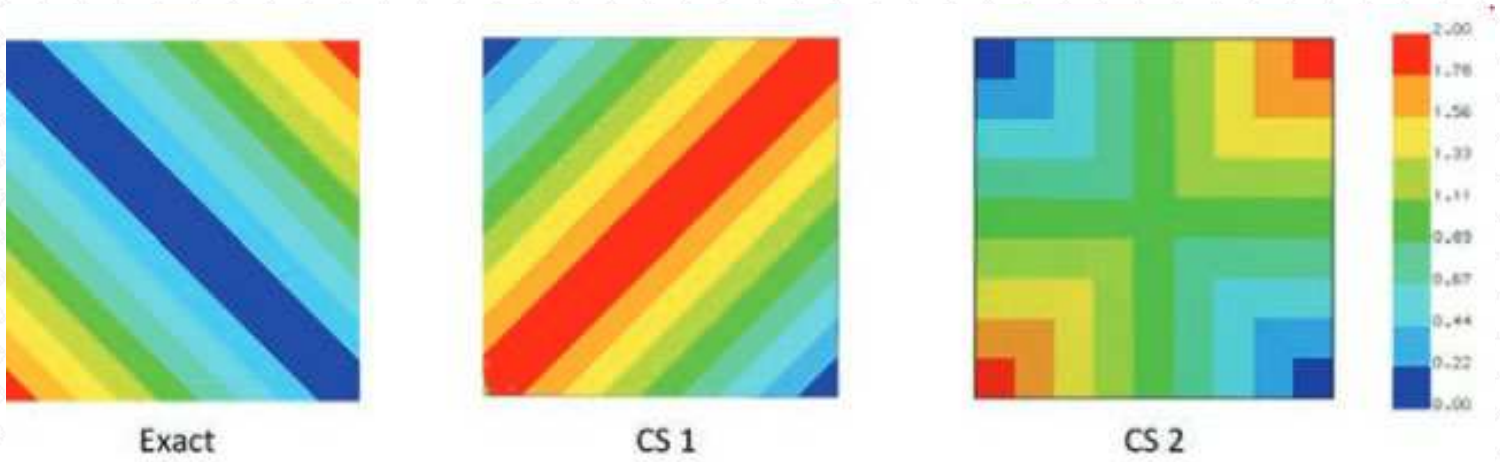


Figure 9: Contour plots of von Mises stress for the challenge problem

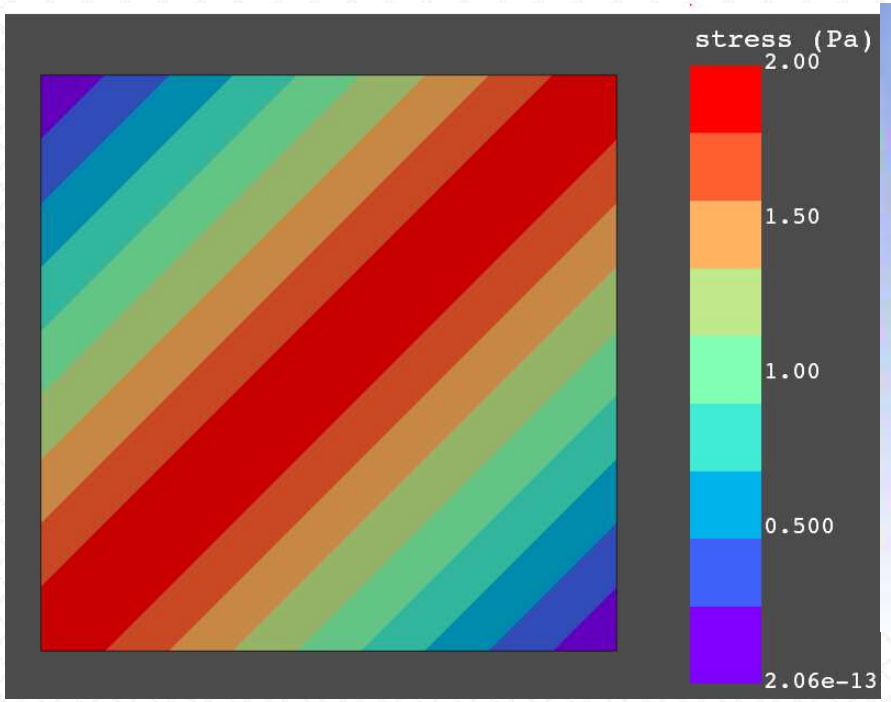
The Example Problem: Single Quadratic Rectangle Model

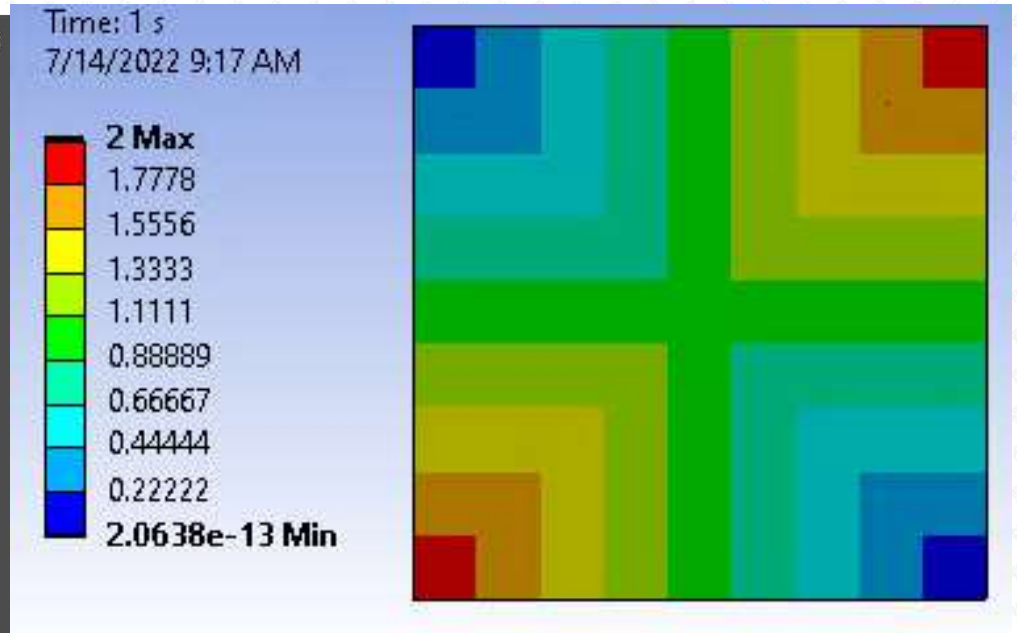
- In the same python session we've been using (we need the 'stress' object active), execute the following:

```
stress.von_mises.plot_contour(cpos='xy',cmap='rainbow',n_colors=9)
```

- This should result in the following:

- This is identical to the plot the NAFEMS author calls CS1 (and it's also what you get in MAPDL)





- Compare to the Workbench von Mises stress plot of the quadratic rectangle (above). This is identical to what the NAFEMS author calls CS2

The Example Problem: Single Quadratic Rectangle Model

- Based on the work we've already done, we can easily check to see if the quadratic or bi-quadratic pyVista cell can produce solutions similar to the Workbench result.
- We'll re-use the pyVista grids we've already created
- Just as we obtained the ux result, stored it in an array with correct node ordering and plotted it over the pyVista cell vertices, we can do the same with the von Mises stress
- Try the following:

```
seqv = np.zeros(8)
ids = np.array(stress.von_mises.get_scoping_at_field())
seqv[ids-1] = stress.von_mises.get_data_at_field()
myquad['seqv'] = seqv
mybiquad['seqv'] =np.append(seqv,1.0)
```

- add the WB stress at center

- To plot the eight-node quadrilateral von Mises stress:

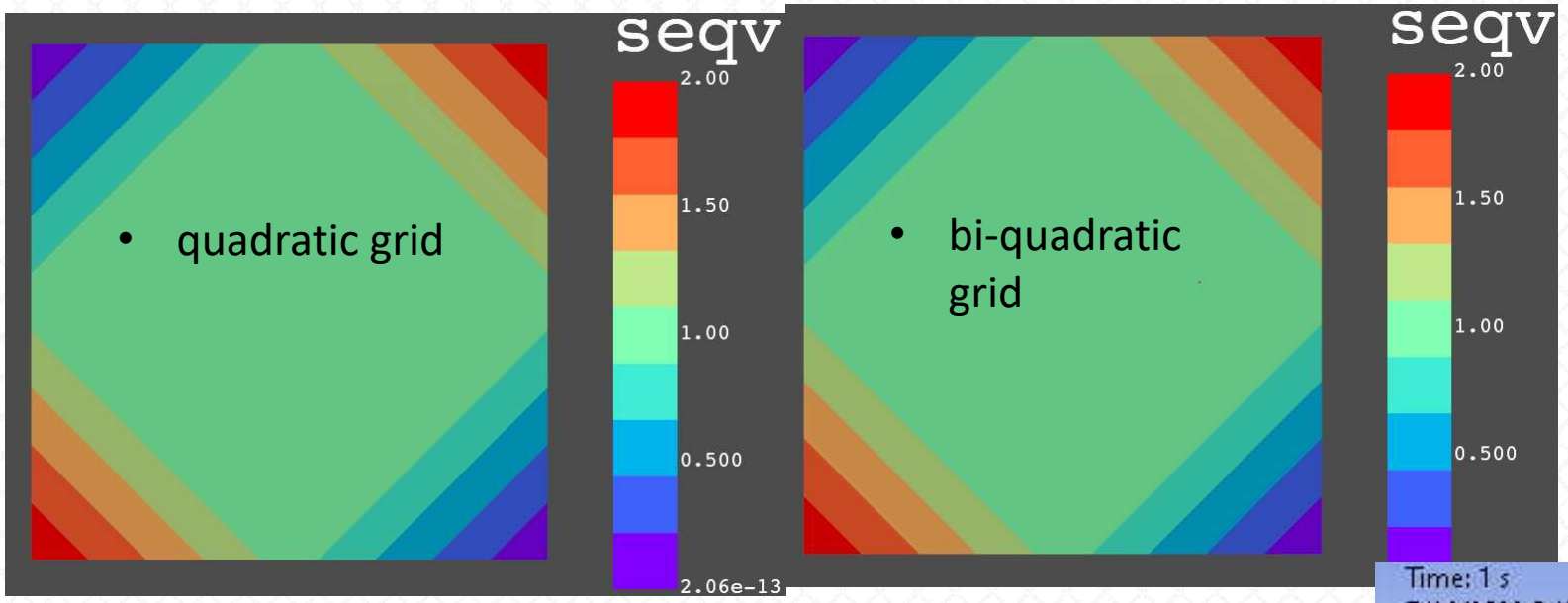
```
p = pv.Plotter()
p.add_mesh(myquad,scalars='seqv',
        cmap='rainbow',n_colors=9)
p.camera_position='xy'
p.show()
```

- To plot the nine-node bi-quadrilateral von Mises stress:

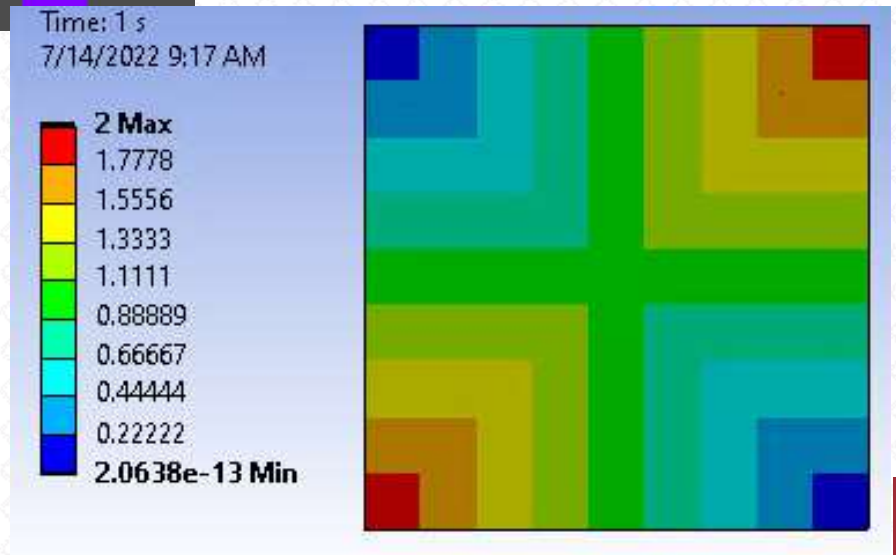
```
p = pv.Plotter()
p.add_mesh(mybiquad,scalars='seqv',
        cmap='rainbow',n_colors=9)
p.camera_position='xy'
p.show()
```

The Example Problem: Single Quadratic Rectangle Model

- And this is what results



- Compare to the Workbench (CS2) stress result
- Again, we can be confident that all node values match
- Only the contour algorithm differs
- A more thorough discussion will have to wait for another article.
- Suffice it so say that what we've been exploring are different ways to post-process a result. pyVista offers one set of assumptions, while WB is using a different graphical tesselation



The Example Problem: Conclusions

- so, what can we say about this model?
- In slide 29, we showed the exact solution, but let's forget we saw that.
- We can't believe the single linear element because loads are only applied at corner nodes in such an element. And the applied tractions all sum to zero at corner nodes (although the stress estimate of the center of that element yields the correct result due to the careful planning of problem's author!)
- But let's look again at what the author calls CS1. We KNOW why that pattern emerges in pyDPF-Post. But it happes in MAPDL too. This is because the MAPDL convention is to extrapolate stress values from integration points to the corner nodes only (this fact may explain why pyDPF-Post simply copies corner nodes when considering midside node values)
- MAPDL stores no stress values for midside nodes, while in pyDPF-Post, midside node values are averaged with corner node values at those corners

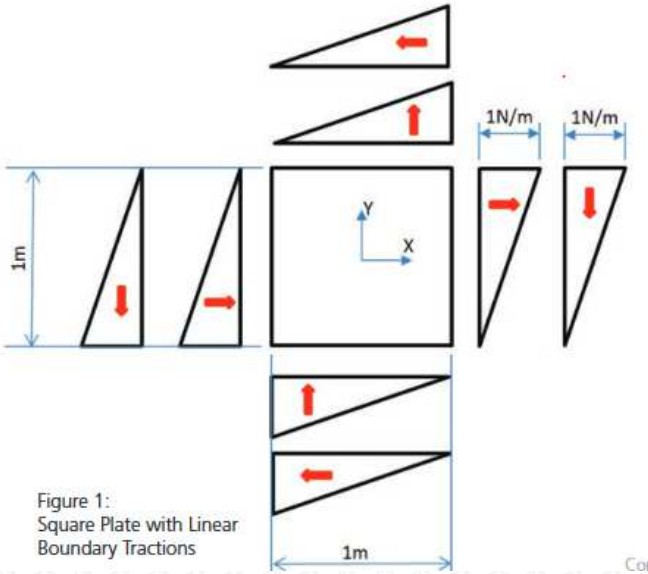
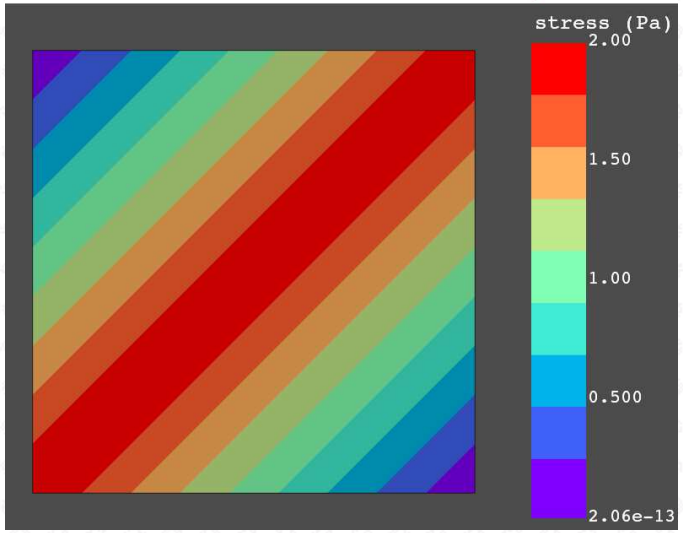


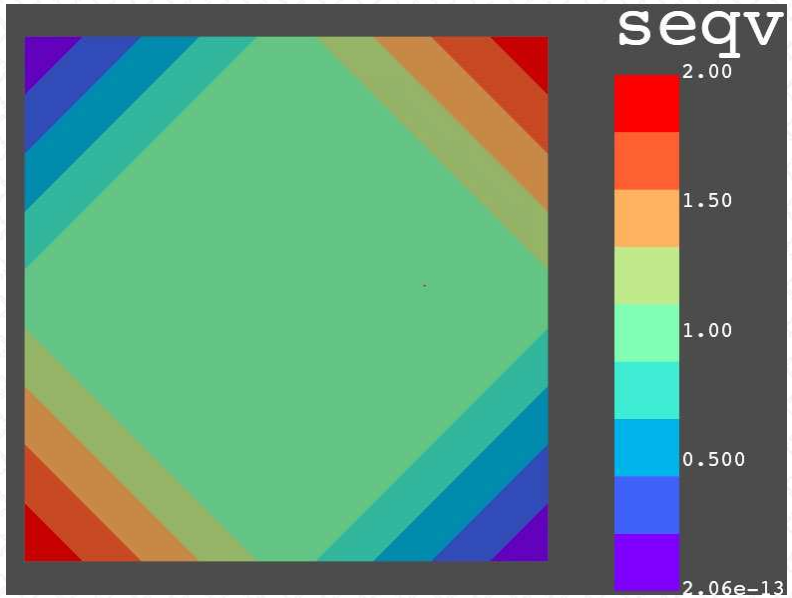
Figure 1:
Square Plate with Linear
Boundary Tractions



The Example Problem: Conclusions

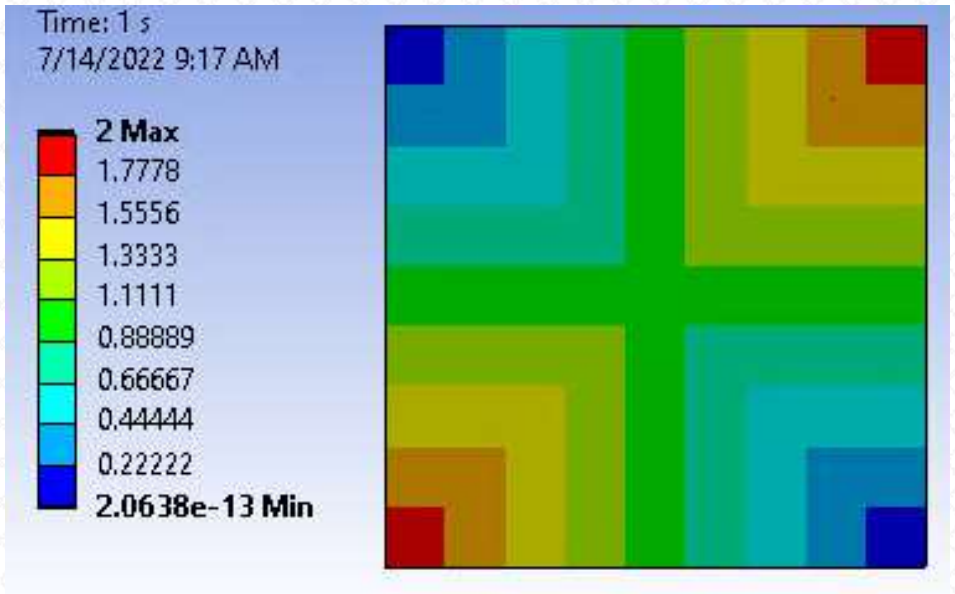
- We defintely believe the CS2 models as a trend (at least at the edges if not the center). It's not really clear if the Workbench plotting algorithm adds any more information than the one we used with pyVista (botom left. In other words: is the Workbench solution 'closer' to the exact result?)

σ_{eqv}



pyVista

σ_{eqv}



CS2: Workbench

The Example Problem: Conclusions

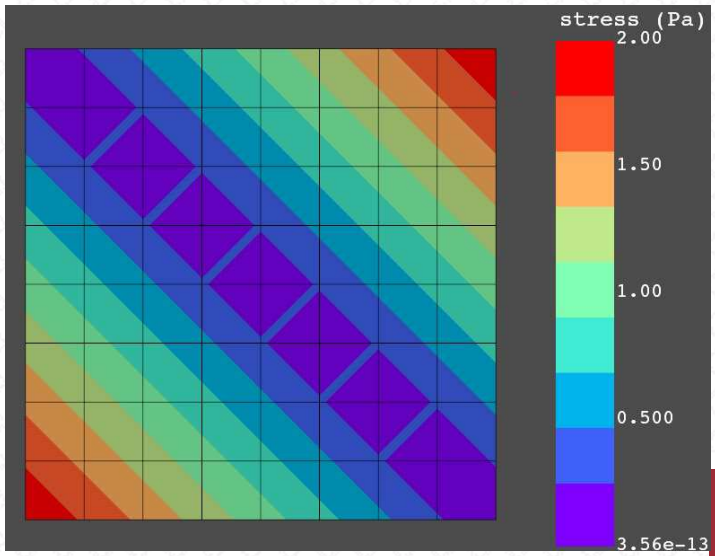
- We conclude by comparing the algorithms on an 8 x 8 mesh of quadratic elements (the third Workbench model in the project)
- First, let's look at the pyDPF-Post plot. Run the following...

file: **8x8rect.py**

```
import os
import sys
wpath = r"C:\Users\alex.grishin\exporting_matrices"
sys.path.append(wpath)
os.chdir(wpath)
from kmat import *

solverfilesloc = wpath + "\\plane_stress_files\\dp0\\SYS-2\\MECH"
dspath = solverfilesloc + "\\ds.dat"
resultpath = solverfilesloc + "\\file.rst"
mapdl = launch_mapdl(run_location=solverfilesloc,override=True)
runandsolve(mapdl,dspath)
matrix_export(mapdl,kname='k3',fname='f3')el = el

solution = post.load_solution(resultpath)
disp = solution.displacement()
stress = solution.stress()
#commenting the displacement plots because we don't need them for demo...
#disp.x.plot_contour(cpos='xy',cmap='rainbow',n_colors=9)
stress.von_mises.plot_contour(cpos='xy',cmap='rainbow',n_colors=9)
mapdl.exit()
```

- This is what you should see...

σ_{eqv}



The Example Problem: Conclusions

- Now, we'll create our quadratic pyVista solution. This time, we'll use a slightly different syntax (it's easier just to read in an array of cell connectivities than to create the conectivity format we used for a single cell)
- First, we'll create a cell connectivity array like so:

```
cells = np.empty((0,8),int)                          #initialize
for i in range(solution.mesh.grid.n_cells):
    el = solution.mesh.elements.element_by_id(i+1)
    carray = np.array(el.connectivity).astype(int)
    cells = np.append(cells,[carray],axis=0)
```

- And here's the new syntax:

```
newquad = pv.UnstructuredGrid({vtk.VTK_QUADRATIC_QUAD:cells},copygrid.points)
```

- Now, get the ids and seqv data as before…

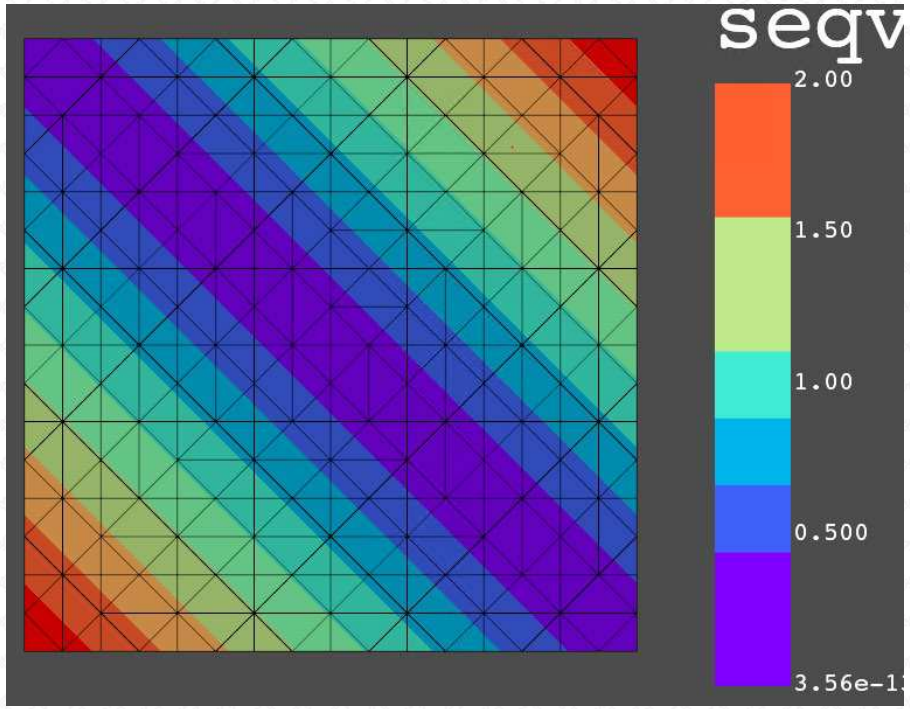
```
seqv = np.zeros(newquad.n_points)
ids = np.array(stress.von_mises.get_scoping_at_field())
seqv[ids-1] = stress.von_mises.get_data_at_field()
newquad['seqv'] = seqv
```

The Example Problem: Conclusions

- Finally, plot as before (but this time, set the 'show_edges' option to 'yes')

```
p = pv.Plotter()
p.add_mesh(newquad,scalars='seqv',
        cmap='rainbow',n_colors=9,
        show_edges='yes')
p.camera_position='xy'
p.show()
```

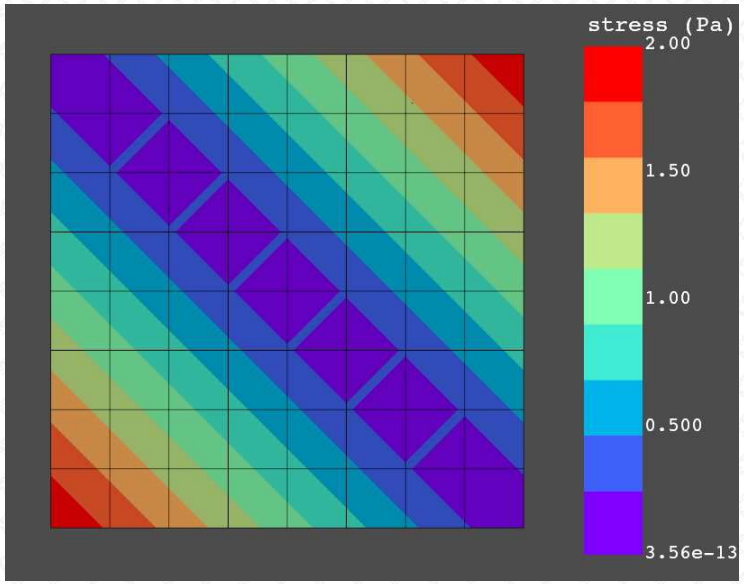
- show edges to revealt the VTK_QUADRILATERAL_QUAD cell edges



The Example Problem: Conclusions

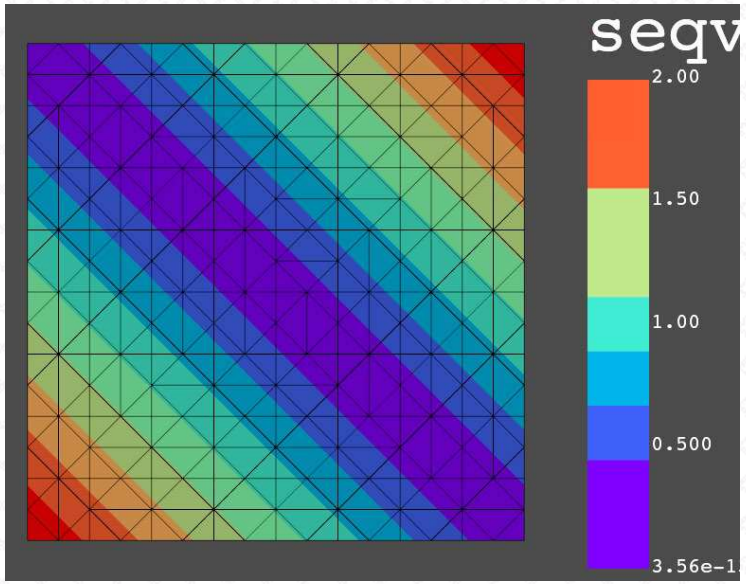
- Finaly, comparing all three algorithms on an 8 x 8 mesh of quadratic elements reveals that the Workbench algorithm (CS2) and our pyVista solution (middle figure) are both smoother (converge faster) than the CS1 solution (DPF-Post and MAPDL)

σ_{eqv}



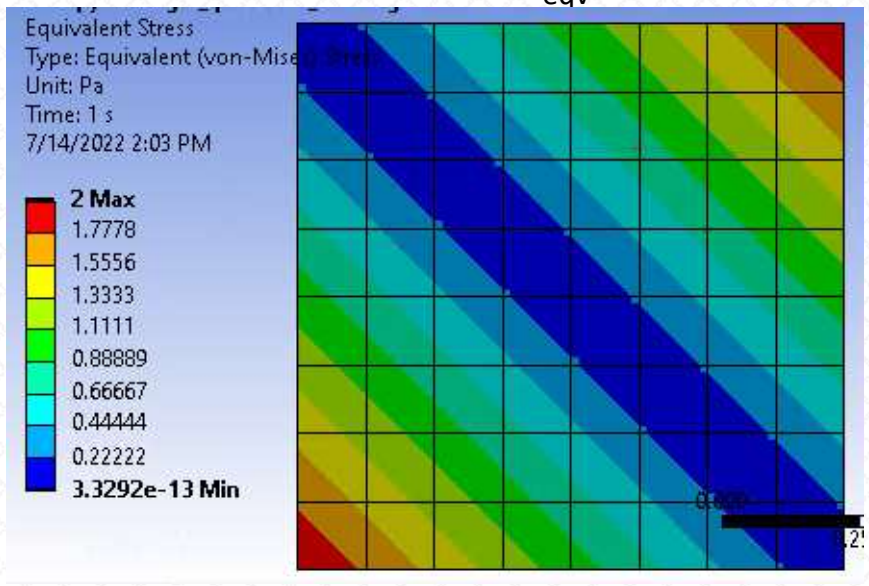
CS1: pyDPF-Post on 8 x 8 quadratic grid

σ_{eqv}



pyVista on 8 x 8 quadratic grid (highlighting the unique edge pattern of this pyvista cell type)

σ_{eqv}

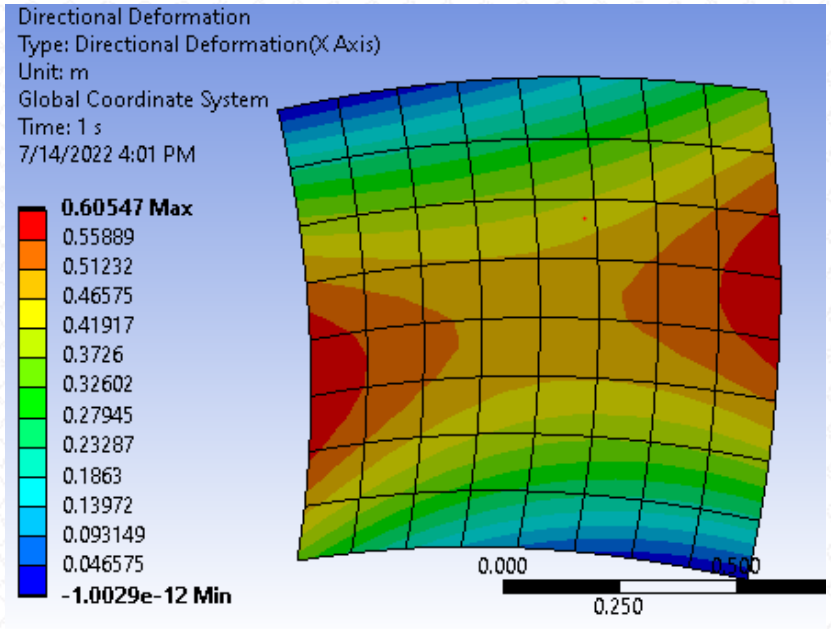


CS2: Workbench with 8 x 8 quadratic mesh

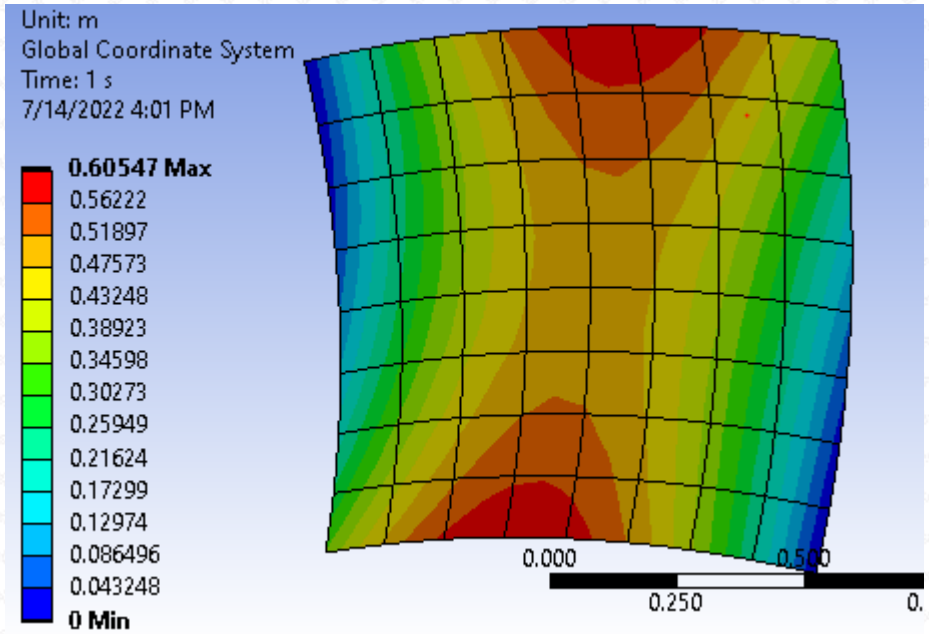
The Example Problem: Conclusions

- One last thing: though the NAFEMS author neglects displacements, we plot the distorted displacement of the refined model just to show readers how this contrived model behaves (compare to single quadratic element on slide 28)

ux



uy



Final Notes

- The author of the NAFEMS challenge problem points out that different commercial FE packages may employ different strategies for graphically rendering nodal solutions
- The differences betweeen them may be especially accute for coarse, high-order element types (quadratic and above)
- The author makes a compelling case that there's nothing to be gained by what he calls 'low fidelity' approaches (see slides 30 and 31 –although that wasn't the intent)
- While good engineering practice would guide most analysts to refine a mesh before making conclusions or identifying trends, a little care in rendering results with more accuracy could potentially reduce the number of such refinements an analyst has to make
- The central challenge posed by the author was to provide an estimate for the von Mises stress at the center fo the plate for both the single linear and quadratic rectangles and choose which one is 'correct'. This was a misdirection from the issue of mesh refinement and the relation of each model to the exact solution. The single linear rectangle 'accidentally' yields the exact solution (from the perspective of analyst refining the mesh –not the problem's author). This is misleading because that solution is only exact in that the loads cancel to within numerical accuracy
- Issues of stress contour rendering (which consumed this article) were only raised by the author in his solution
- We'll return to this problem and explore the issue of contour rendering in a little more detail in a future article