2-D FEM Simulation for Induction Positioner (Rev 2)

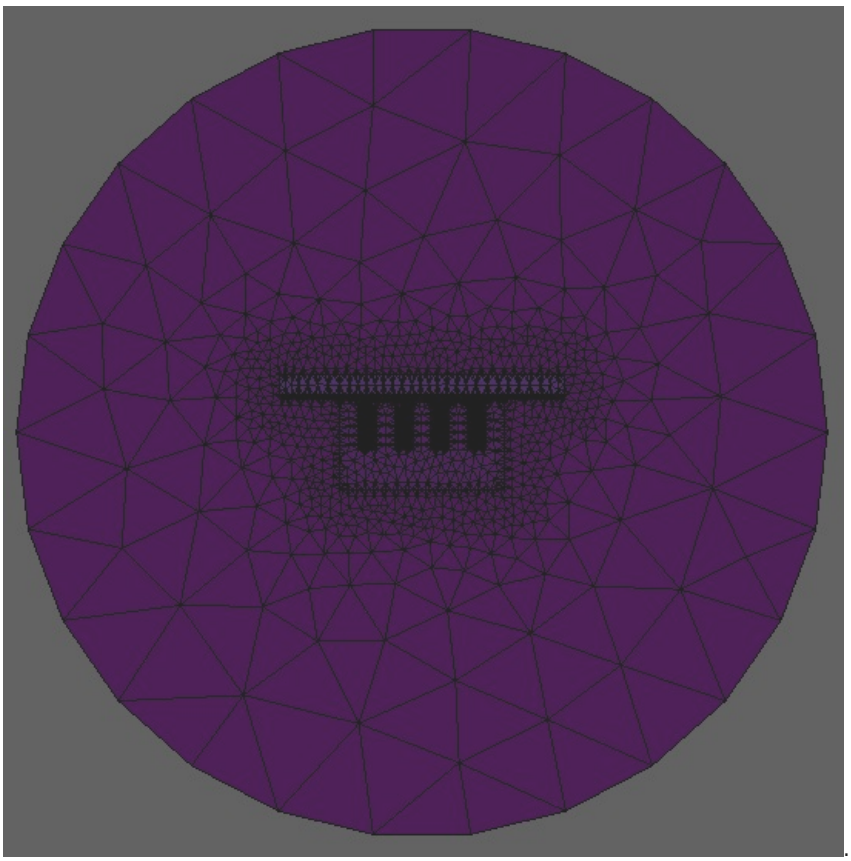
(The addition of time-stepping to the algorithm presented in Rev 1 for the purpose of viewing the induced current in the copper layer of the inducion plate.)

Michael E. Aiello 12/31//22

This file is based on code from application em.ipynb written by Jørgen S. Dokken https://jorgensd.github.io/dolfinx-tutorial/chapter3/em.html

This program runs in an open souce simulation program known as FENiCSx https://jorgensd.github.io/dolfinx-tutorial/

Simulation results documented at the end of this file.



Comments as to modifications to the original code provided by Jørgen S. Dokken are provided in boldface below.

Relative to Rev 1 document there is no changes made to this section

In [ ]:
```
import gmsh
import numpy as np
from mpi4py import MPI   # NOTE: Not running with MPI effects precision of computations!

from GeneratePCTraces_Rev_3 import *
from GenerateCore_2_Gap_Rev_3 import *
from GenerateCore_4_Gap_Rev_3 import *
from GenerateCore_Enhanced import *
from GenerateConductionPlate_copper import *
from GenerateConductionPlate_iron import *
gmsh.initialize()

r = 0.05
R = 1     # Radius of domain
gdim = 2  # Geometric dimension of the mesh

air_gap = .0025

center_x_pos_off = - .5 * (3 * (.025 + .01 + .01) + 4 * (.06 - .015))
center_y_pos_off = - .5 * (.05 + .01 + air_gap + (.1 + (.0014 + .0087) * (10 + 2)))

layer_start_pos = .1 + .0087 + center_y_pos_off
row_start_pos = .06 - .015 + .01 + center_x_pos_off - .5 * (.06 + .03)
```

```
core_y_start_pos = 0 + center_y_pos_off
core_x_start_pos = 0 + center_x_pos_off


plate_y_offset = air_gap + (.1 + (.0014 + .0087) * (10 + 2))
plate_x_offset = - .5 * (.7 - 3 * (.025 + .01 + .01) - 4 * (.06 - .015))
y_axis_cond_plate_start = plate_y_offset + 0 + center_y_pos_off
x_axis_cond_plate_start = plate_x_offset + 0 + center_x_pos_off

rank = MPI.COMM_WORLD.rank


if rank == 0:
    gmsh.model.occ.synchronize()

    # Define geometry for background
    background = gmsh.model.occ.addDisk(0, 0, 0, R, R)
    gmsh.model.occ.synchronize()

    #Select to use two gap core...
    Use_2_Gap_Core = 0

    if Use_2_Gap_Core == 1:
        core_x_start_pos = core_x_start_pos + .5 * (.06 + .03)
        GenCoreRtnVals = GenerateCore_2_Gap_Rev_3(core_y_start_pos, core_x_start_pos)
    else:
        core_x_start_pos = core_x_start_pos - .5 * (.06 + .03)
        GenCoreRtnVals = GenerateCore_4_Gap_Rev_3(core_y_start_pos, core_x_start_pos)

    core = GenCoreRtnVals[1]
    core_mass = GenCoreRtnVals[0]

    core_tuple = [(2, core)]

    GenPCRtnVals =  GeneratePCTraces_Rev_3(layer_start_pos, row_start_pos)
    traces = GenPCRtnVals[1]
    traces_mass = GenPCRtnVals[0]

    GenCondPltRtnVals_copper = GenerateConductionPlate_copper(y_axis_cond_plate_start, x_axis_cond_plate_start)
    copper = GenCondPltRtnVals_copper[1]
    copper_mass = GenCondPltRtnVals_copper[0]

    copper_tuple = [(2, copper)]

    GenCondPltRtnVals_iron = GenerateConductionPlate_iron(y_axis_cond_plate_start, x_axis_cond_plate_start)
    iron = GenCondPltRtnVals_iron[1]
    iron_mass = GenCondPltRtnVals_iron[0]

    iron_tuple = [(2, iron)]


    trace_tuples = []
    for i in range(len(traces)):
        trace_tuples.append((2,traces[i]))

    # Resolve all boundaries of the different wires in the background domain
    all_surfaces = []
    all_surfaces.extend(core_tuple)
    all_surfaces.extend(copper_tuple)
    all_surfaces.extend(iron_tuple)
    all_surfaces.extend(trace_tuples)
    whole_domain = gmsh.model.occ.fragment([(2, background)], all_surfaces)
    gmsh.model.occ.synchronize()

    # Create physical markers for the different components in the backround.
    # We use the following markers:
    # - Vacuum: 0 (backround disk)
    # - Traces 1 to len(traces)
    # -
    # -
    core_tag = 1
    copper_tag = 2
    iron_tag = 3
    trace_tag = 4
    background_surfaces = []
    other_surfaces = []
    for domain in whole_domain[0]:
        com = gmsh.model.occ.getCenterOfMass(domain[0], domain[1])
        mass = gmsh.model.occ.getMass(domain[0], domain[1])
        # Identify the core...
        if np.isclose(mass, core_mass, atol = .002):
            gmsh.model.addPhysicalGroup(domain[0], [domain[1]], core_tag)
            core_tag +=1
            other_surfaces.append(domain)
        # Identify the copper plate...
        elif np.isclose(mass, copper_mass):
            gmsh.model.addPhysicalGroup(domain[0], [domain[1]], copper_tag)
            copper_tag +=1
            other_surfaces.append(domain)
```

```
        # Identify the iron plate...
        elif np.isclose(mass, iron_mass):
            gmsh.model.addPhysicalGroup(domain[0], [domain[1]], iron_tag)
            iron_tag +=1
            other_surfaces.append(domain)
        # Identify the traces in the PC board.
        elif np.isclose(mass, traces_mass):
            gmsh.model.addPhysicalGroup(domain[0], [domain[1]], trace_tag)
            trace_tag +=1
            other_surfaces.append(domain)
    #    elif np.allclose(com, [0, 0, 0]):       (Something wrong here. com center way off!)
        else:
            background_surfaces.append(domain[1])

    # Add marker for the vacuum
    gmsh.model.addPhysicalGroup(2, background_surfaces, tag=0)
    # Create mesh resolution that is fine around the wires and
    # iron cylinder, coarser the further away you get
    gmsh.model.mesh.field.add("Distance", 1)
    edges = gmsh.model.getBoundary(other_surfaces, oriented=False)

    gmsh.model.mesh.field.setNumbers(1, "EdgesList", [e[1] for e in edges])
    gmsh.model.mesh.field.add("Threshold", 2)
    gmsh.model.mesh.field.setNumber(2, "IField", 1)
    gmsh.model.mesh.field.setNumber(2, "LcMin", r / 2)
    gmsh.model.mesh.field.setNumber(2, "LcMax", 5 * r)
    gmsh.model.mesh.field.setNumber(2, "DistMin", 2 * r)
    gmsh.model.mesh.field.setNumber(2, "DistMax", 4 * r)
    gmsh.model.mesh.field.setAsBackgroundMesh(2)
    # Generate mesh
    gmsh.option.setNumber("Mesh.Algorithm", 7)
    gmsh.model.mesh.generate(gdim)
```

`MeshTags` for the physical cell data are created below. **There is no change in the code from the original example cited above**.

In [ ]:
```
from dolfinx.io import (cell_perm_gmsh, distribute_entity_data, extract_gmsh_geometry,
                        extract_gmsh_topology_and_markers, ufl_mesh_from_gmsh)
from dolfinx.cpp.mesh import to_type
from dolfinx.graph import create_adjacencylist
from dolfinx.mesh import create_mesh, meshtags_from_entities
if rank == 0:
    # Get mesh geometry
    x = extract_gmsh_geometry(gmsh.model)

    # Get mesh topology for each element
    topologies = extract_gmsh_topology_and_markers(gmsh.model)
    # Get information about each cell type from the msh files
    num_cell_types = len(topologies.keys())
    cell_information = {}
    cell_dimensions = np.zeros(num_cell_types, dtype=np.int32)
    for i, element in enumerate(topologies.keys()):
        properties = gmsh.model.mesh.getElementProperties(element)
        name, dim, order, num_nodes, local_coords, _ = properties
        cell_information[i] = {"id": element, "dim": dim, "num_nodes": num_nodes}
        cell_dimensions[i] = dim

    # Sort elements by ascending dimension
    perm_sort = np.argsort(cell_dimensions)

    # Broadcast cell type data and geometric dimension
    cell_id = cell_information[perm_sort[-1]]["id"]
    tdim = cell_information[perm_sort[-1]]["dim"]
    num_nodes = cell_information[perm_sort[-1]]["num_nodes"]
    cell_id, num_nodes = MPI.COMM_WORLD.bcast([cell_id, num_nodes], root=0)

    cells = np.asarray(topologies[cell_id]["topology"], dtype=np.int64)
    cell_values = np.asarray(topologies[cell_id]["cell_data"], dtype=np.int32)
else:
    cell_id, num_nodes = MPI.COMM_WORLD.bcast([None, None], root=0)
    cells, x = np.empty([0, num_nodes], dtype=np.int64), np.empty([0, gdim])
    cell_values = np.empty((0,), dtype=np.int32)
gmsh.finalize()
```

Distribute the mesh over multiple processors. **There is no change in the code from the original example cited above**.

In [ ]:
```
# Create distributed mesh
ufl_domain = ufl_mesh_from_gmsh(cell_id, gdim)
gmsh_cell_perm = cell_perm_gmsh(to_type(str(ufl_domain.ufl_cell())), num_nodes)
cells = cells[:, gmsh_cell_perm]
mesh = create_mesh(MPI.COMM_WORLD, cells, x[:, :gdim], ufl_domain)
tdim = mesh.topology.dim

local_entities, local_values = distribute_entity_data(mesh, tdim, cells, cell_values)
mesh.topology.create_connectivity(tdim, 0)
```

```
adj = create_adjacencylist(local_entities)
ct = meshtags_from_entities(mesh, tdim, adj, np.int32(local_values))
```

Create data files to optionally inspect the mesh using Paraview. (No changes to the original example).

In [ ]:
```
from dolfinx.io import XDMFFile
with XDMFFile(MPI.COMM_WORLD, "gmsh_test_data.xdmf", "w") as xdmf:
    xdmf.write_mesh(mesh)
    xdmf.write_meshtags(ct)
```

Visualize the subdommains interactively using PyVista. **There is no change in the code from the original example cited above**.

In [ ]:
```
import pyvista
pyvista.set_jupyter_backend("pythreejs")
from dolfinx.plot import create_vtk_mesh

plotter = pyvista.Plotter()
grid = pyvista.UnstructuredGrid(*create_vtk_mesh(mesh, mesh.topology.dim))
num_local_cells = mesh.topology.index_map(mesh.topology.dim).size_local
grid.cell_data["Marker"] = ct.values[ct.indices<num_local_cells]
grid.set_active_scalars("Marker")
actor = plotter.add_mesh(grid, show_edges=True)
plotter.view_xy()
if not pyvista.OFF_SCREEN:
    plotter.show()
else:
    pyvista.start_xvfb()
    cell_tag_fig = plotter.screenshot("cell_tags.png")
```

Next, we define the discontinous functions for the permability μ
μ
and current J_z
J_z
using the `MeshTags` as in Defining material parameters through subdomains

For Rev 2, time-stepping code is added to this section

The code section below has been modified relative to the original example (em.ipymb) to execute 20 steps between 0 and 360 degrees of an alternating current (J) applied to the three phase copper traces contained in the PC board inserted onto the motor core. The Vector potential is displayed for each ineration.

In addition, time stepping has been added to the code below. Within each of the 20 steps, 50 inerations of the code is executed in order to compute the first and second derivative of the FEM computed vector potential at each ineration step.

The first derivative of the vector potential represents the generated "Electric Field". Here we are interested only in the electric field within the copper portion of the induction plate (which can conduct current). The second derivative of the vector potential represents the generated current or "J". Again, we are only interested in current generated in the copper portion of the induction plate. For purposes of demonstration, it is assumed that all other elements within the FEM grid have no conductivity. This includes the copper traces in the PC board which under the assumption in this simulation, are pure current sources.

It should be noted that in this experiment, the generated "J" in the copper plate IS NOT fed back into the algorithm (although it would be a simple process to do so). The point here is to demonstrate that the second derivative of the vector potential when induced in a conducting material should be proportional to the "minus" value of the sourced vector vector potential produced by the current flowing in the copper traces within the core. This in turn is what produces force on the conducting plate.

It may also be of interest to see the affect of adding a DC current to all three phases during the time-stepping process.

Question: Is it possible to make use of damping in dynamic operation (motion). One idea (unrelated to the demonstration at hand), is the damping of a induction motor spindle running at high speed? The interesting thing here is that the effect of damping is limited only to the performance of the current loop and not any of the higher order control loops within the system.

A slice of the vector potential, the generated first directive of the vector potential, and the second derivative of the vector potential in the center of the copper sheet (coating) is viewed by setting view_slice to 1.

DC current is set by assigning a value to the variable damping_current (in this case a value of .2 which is would be an extreme use case and only used here for demonstration purposes.) The DC current is added to all three phases. This provides a minor problem with the power amplifier. To apply an equal amount of bias current to each of the three phases requires that the WYE connection of the motor be referenced to power return or a seperate amplifier be used to control each phase of the motor stator. (The intial thought was adding a balanced offset to Phases A, B and C which would not require the WYE connection to be reference to power return. However, this may an optimium approach because there is always one point in comutation where this offset would be canceled?)

Question: Is adding an equal amount of DC current to each phase the correct approach for adding a damping effect to the system. Or since Phase B winding is reversed, should a positive current offset be applied to Phase A and C and an negative offset applied to Phase B?

In [ ]:

In [ ]:
```
from dolfinx.fem import (dirichletbc, Expression, Function, FunctionSpace,
                         VectorFunctionSpace, locate_dofs_topological)
from dolfinx.fem.petsc import LinearProblem
from dolfinx.mesh import locate_entities_boundary
from ufl import TestFunction, TrialFunction, as_vector, dot, dx, grad, inner
from petsc4py.PETSc import ScalarType

import time
from IPython.display import clear_output

from dolfinx.mesh import compute_midpoints

# Set direction of MMF wave here (0 or 1)...
mmf_negative = 0


# Select to view vector potential....
view_vector_pontential = 1

# Select to show magnetic field....
view_magnetic_field = 0

# Select full 3-D view or "slice" along the y axis (center of the copper part of the conduction plate)
view_slice = 0

Q = FunctionSpace(mesh, ("DG", 0))
material_tags = np.unique(ct.values)
mu = Function(Q)
J = Function(Q)

V = FunctionSpace(mesh, ("CG", 1))
facets = locate_entities_boundary(mesh, tdim-1, lambda x: np.full(x.shape[1], True))
dofs = locate_dofs_topological(V, tdim-1, facets)
bc = dirichletbc(ScalarType(0), dofs, V)

u = TrialFunction(V)
v = TestFunction(V)
a = (1 / mu) * dot(grad(u), grad(v)) * dx
L = J * v * dx

A_z = Function(V)
problem = LinearProblem(a, L, u=A_z, bcs=[bc])

W = VectorFunctionSpace(mesh, ("DG", 0))
B = Function(W)

A_z_1 = Function(V)
A_z_2 = Function(V)

d2A_z_dt2 = Function(V)
dA_z_dt = Function(V)


idx_range = 1000   # Do not exceed this value. There seems to be float operation mixed in with double!


idx_display = int(idx_range) / 20   # Always display 20 snapshots no matter the idx_range

    #Use this for viewing results ("setting to 0" is the vector potential itself) ...
        # 1- first derivative of vector potential
        # 2- second derivative of vector potential
display_derivative = 2

    #Add desired DC offset to the current command for all three phases to represent
    #the "damping" factor.  (Use 0 or .2)
damping_current = 0

for idx in range(0, idx_range):
    if mmf_negative == 1:
```

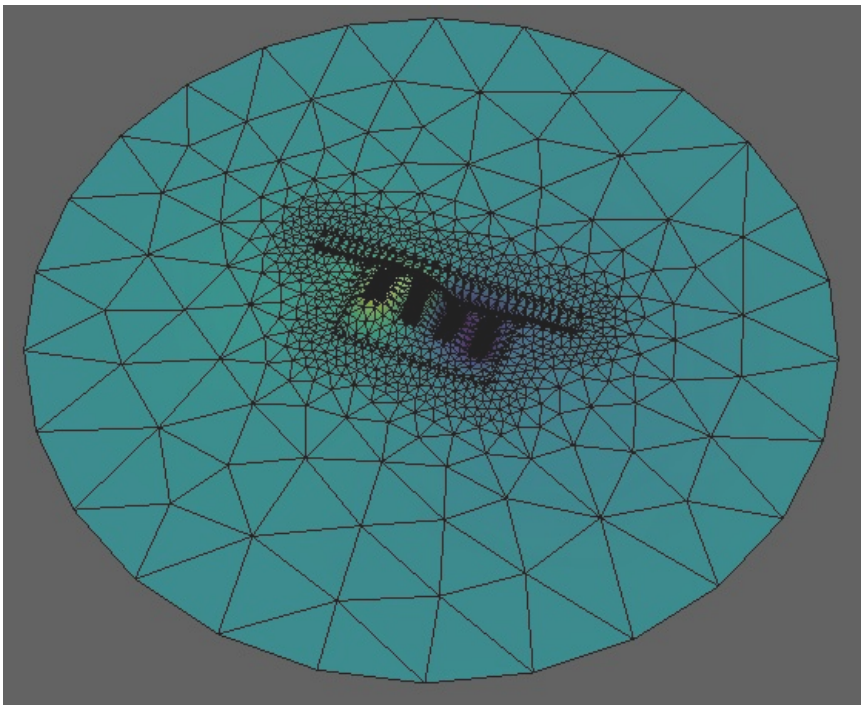
```
        Theta = ((idx_range - 1) - idx) * 2.0 * np.pi / idx_range
    else:
        Theta = idx * 2.0 * np.pi / idx_range
    K = 30.0
    Phase_A_cur = K * (np.sin(Theta) + damping_current)
    Phase_B_cur = K * (np.sin(Theta - 2.0 * np.pi / 3.0) + damping_current)
    Phase_C_cur = K * (np.sin(Theta - 4.0 * np.pi / 3.0) + damping_current)

#    Q = FunctionSpace(mesh, ("DG", 0))
#    material_tags = np.unique(ct.values)
#    mu = Function(Q)
#    J = Function(Q)
    # As we only set some values in J, initialize all as 0
    J.x.array[:] = 0
    for tag in material_tags:
        cells = ct.indices[ct.values==tag]
        num_cells = len(cells)
        # Set values for mu
        if tag == 0:
            mu_ = 4 * np.pi*1e-7 # Vacuum
        elif tag == 1:
            mu_ = 1e-5 # Core (This should really be 6.3e-3)
        elif tag == 3:
            mu_ = 1e-5 # Conduction Plate (iron) (This should really be 6.3e-3)
        else:
            mu_ = 1.26e-6 # Else, Copper traces and Conduction Plate (copper)
        mu.x.array[cells] = np.full(num_cells, mu_)
        # Now, assign the currents to traces representing Phase A, B and C
        #      Conductors left side (bottom to top)       [4, 8,  12, 16, 20, 24, 28, 32, 36, 40]
        #      Conductors in first slot (bottom to top)   [5, 9,  13, 17, 21, 25, 29, 33, 37, 41]
        #      Conductors in second slot (bottom to top)  [6, 10, 14, 18, 22, 26, 30, 34, 38, 42]
        #      Conductors right side (bottom to top)      [7, 11, 15, 19, 23, 27, 31, 35, 39, 43]


#               # !!!! Verified OK above !!!!!!~!
#
#       if tag in [28]:
#           J.x.array[cells] = np.full(num_cells, 300.0)



        # Phase A, right side...
        if tag in [4, 8,  12, 16, 20]:
            J.x.array[cells] = np.full(num_cells, Phase_A_cur)
        # Phase A, first slot (return)...
        elif tag in [5, 9,  13, 17, 21]:
            J.x.array[cells] = np.full(num_cells, - Phase_A_cur)

        # Phase B, first slot (current direction flipped)...
        elif tag in [25, 29, 33, 37, 41]:
            J.x.array[cells] = np.full(num_cells,  - Phase_B_cur)
        # Phase B, second slot (current direction flipped, return)
        elif tag in [26, 30, 34, 38, 42]:
            J.x.array[cells] = np.full(num_cells,  Phase_B_cur)

        # Phase C, second slot...
        elif tag in [6, 10, 14, 18, 22]:
            J.x.array[cells] = np.full(num_cells, Phase_C_cur)
        # Phase C, right side (return)...
        elif tag in [7, 11, 15, 19, 23]:
            J.x.array[cells] = np.full(num_cells, - Phase_C_cur)



#    V = FunctionSpace(mesh, ("CG", 1))
#    facets = locate_entities_boundary(mesh, tdim-1, lambda x: np.full(x.shape[1], True))
#    dofs = locate_dofs_topological(V, tdim-1, facets)
#    bc = dirichletbc(ScalarType(0), dofs, V)
#
#    u = TrialFunction(V)
#    v = TestFunction(V)
#    a = (1 / mu) * dot(grad(u), grad(v)) * dx
#    L = J * v * dx
#
#    A_z = Function(V)
#    problem = LinearProblem(a, L, u=A_z, bcs=[bc])
    problem.solve()

#    W = VectorFunctionSpace(mesh, ("DG", 0))
#    B = Function(W)
    B_expr = Expression(as_vector((A_z.dx(1), -A_z.dx(0))), W.element.interpolation_points)
    B.interpolate(B_expr)




        #https://fenicsproject.discourse.group/t/dolfinx-fem-function-function-add-with-dolfinx-fem-function-f
```

```
    d2A_z_dt2.x.array[:] = (A_z.x.array[:] - 2*A_z_1.x.array[:] + A_z_2.x.array[:]) * idx_display * idx_display
                #NOTE: idx_display * idx_display in numerator satisfies denomenator dt**2 since delta(idx) is a


    dA_z_dt.x.array[:] = (A_z.x.array[:] - A_z_1.x.array[:]) * idx_display

         #https://fenicsproject.discourse.group/t/copying-a-function-in-dolfinx/7425
         #https://fenicsproject.discourse.group/t/function-assign-in-dolfinx/7992/4
    A_z_2.x.array[:] = A_z_1.x.array[:]
    A_z_1.x.array[:] = A_z.x.array[:]



    if idx % idx_display == 0:

        plotter = pyvista.Plotter()

        Az_grid = pyvista.UnstructuredGrid(*create_vtk_mesh(V))

        if display_derivative == 2:

            Az_grid.point_data["d2A_z_dt2"] = d2A_z_dt2.x.array
            Az_grid.set_active_scalars("d2A_z_dt2")
            warp = Az_grid.warp_by_scalar("d2A_z_dt2", factor=1e7 * 13)   # Need to adjust scale here.

        elif display_derivative == 1:

            Az_grid.point_data["dA_z_dt"] = dA_z_dt.x.array
            Az_grid.set_active_scalars("dA_z_dt")
            warp = Az_grid.warp_by_scalar("dA_z_dt", factor=1e7 * 5)   # Need to adjust scale here.

        else:

            Az_grid.point_data["A_z"] = A_z.x.array
            Az_grid.set_active_scalars("A_z")
            warp = Az_grid.warp_by_scalar("A_z", factor=1e7)


        actor = plotter.add_mesh(warp, show_edges=True)
        if not pyvista.OFF_SCREEN:



            if view_vector_pontential == 1:

                if view_slice == 1:

                    # This code constructed from info derived from https://docs.pyvista.org/examples/01-filter/
                    #  "Single slice - origin defaults to the center of the mesh"
                    # and https://docs.pyvista.org/api/core/_autosummary/pyvista.UnstructuredGrid.slice.html
                    # The slicing offset is .01/2, .01 being the thickness of the copper part of the conduction
                    single_slice = warp.slice(normal='y', origin=([x_axis_cond_plate_start, y_axis_cond_plate_s
                    p = pyvista.Plotter()
                    p.add_mesh(warp.outline(), color="k")
                    p.add_mesh(single_slice, show_edges=True)
                    p.view_xz()
                    p.show()

                else:

                    plotter.view_xz()
                    plotter.show()


            if view_magnetic_field == 1:

                plotter = pyvista.Plotter()
                plotter.set_position([0,0,5])

                # We include ghosts cells as we access all degrees of freedom (including ghosts) on each proces
                top_imap = mesh.topology.index_map(mesh.topology.dim)
                num_cells = top_imap.size_local + top_imap.num_ghosts
                midpoints = compute_midpoints(mesh, mesh.topology.dim, range(num_cells))

                num_dofs = W.dofmap.index_map.size_local +  W.dofmap.index_map.num_ghosts
                assert(num_cells == num_dofs)
                values = np.zeros((num_dofs, 3), dtype=np.float64)
                values[:, :mesh.geometry.dim] = B.x.array.real.reshape(num_dofs, W.dofmap.index_map_bs)
                cloud = pyvista.PolyData(midpoints)
                cloud["B"] = values
                glyphs = cloud.glyph("B", factor=1e5)   # (original was factor=2e6)
                actor = plotter.add_mesh(grid, style="wireframe", color="k")
                actor2 = plotter.add_mesh(glyphs)

                if not pyvista.OFF_SCREEN:
                    plotter.show()
                else:
                    pyvista.start_xvfb()
                    B_fig = plotter.screenshot("B.png")
```

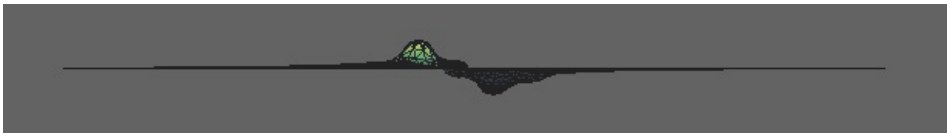
```
            print("Plot number", int(idx / idx_display))

        else:
            pyvista.start_xvfb()
            Az_fig = plotter.screenshot("Az.png")



        time.sleep(.5)
        ch = input("Hit return to continue...")
        clear_output(wait=True)
```

Simulation results (not interactive)

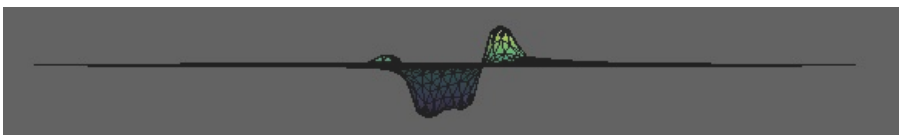
The image below shows the vector potential in 3-D for iteration step 3 (out of 20) for a preset direction of field rotation. Keep in mind that between each iteration, 50 time steps are executed.



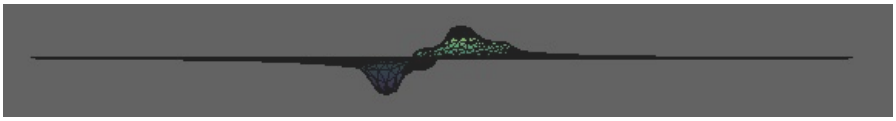
Views on the plane X-Y



The vector potential, no DC offset.



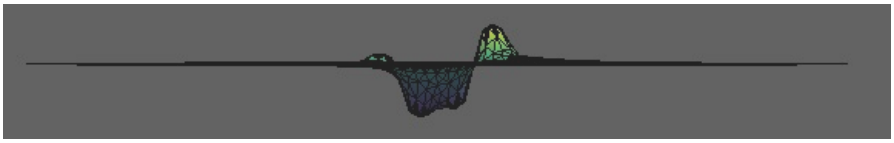
First derivative of the vector potential (electric field), no DC offset.



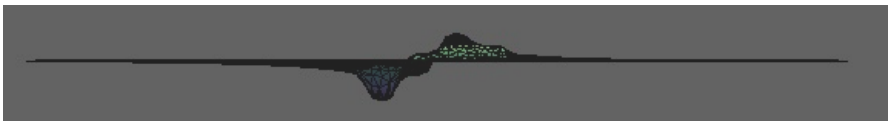
Second derivative of the vector potential, no DC offset (induced electric current), at points where conductivity is not infinity. In this demonstration, this would only be in the cross-section of the copper

coating covering the conduction plate.

The vector potential, with DC offset.



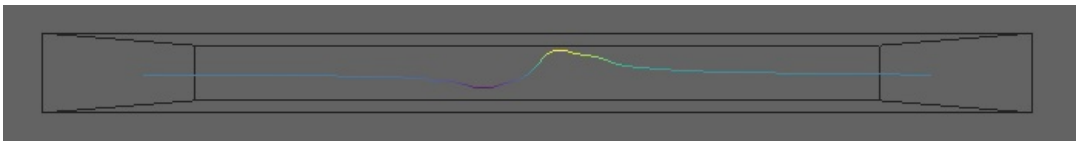
First derivative of the vector potential (electric field), with DC offset (should be the same as no DC offset plot above.



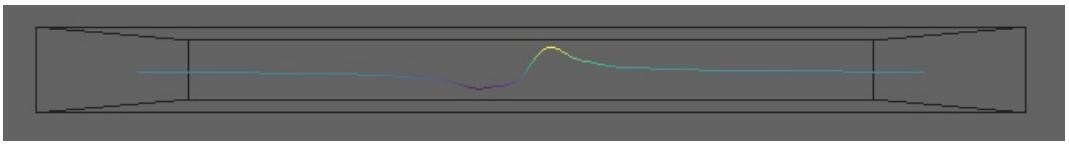
Second derivative of the vector potential with DC offset (induced electric current), at points where conductivity is not infinity (should be the same as no DC offset plot above.

Note that as expected (with scaling), the second derivative of the vector potential reflects the negative of the vector potential (with no DC offset) in the plots above. The shift in position between the two quatities as the field rotates is what produces force.

Views on the plane X-Y sliced at the center of the copper coating of the conduction plate.



Second derivative of the vector potential, no DC offset (induced electric current)



Second derivative of the vector potential with DC offset (induced electric current). Should be same as no DC offset plot above.