C1_E2_CIL Cilindro (1)

November 8, 2018

We are going to emulate the behavior of a magnetic field in microwires. Using the boundary element method and using Laplace as Green function, we're going to solve:

$$\begin{bmatrix} -K_{ext} - K_{int} & \alpha V_{int} + V_{ext} \\ -K'_{ext} - K'_{int} & \frac{\alpha-1}{2} + \alpha V'_{int} + V'_{ext} \end{bmatrix} \begin{bmatrix} \varphi_{ext} \\ \frac{\partial}{\partial n}\varphi_{ext} \end{bmatrix} = \begin{bmatrix} \varphi_{inc} \\ \frac{\partial \varphi_{inc}}{\partial n} \end{bmatrix} \tag{1}$$

Being K the double layer operator, V the single layer operator and I the identity. First of all, we call all the libraries that we'll use.

```
In [2]: #Import libraries
        import bempp.api
        import numpy as np
        import timeit
        bempp.api.set_ipython_notebook_viewer()

        #Set quadrature options
        bempp.api.global_parameters.quadrature.near.double_order = 2
        bempp.api.global_parameters.quadrature.medium.double_order = 2
        bempp.api.global_parameters.quadrature.far.double_order = 2
```

Now we import the mesh that we'll use, ploting them and printing the number of elements presents in the mesh.

```
In [3]: #Define grid
        grid_0 = bempp.api.import_grid("BH21_a5_l10_E5550.msh")

        # Print out the number of elements
        number_of_elements = grid_0.leaf_view.entity_count(0)
        print("The grid has {0} elements.".format(number_of_elements))

        #Plot the grid
        #grid_0.plot()
```

```
The grid has 5550 elements.
```

First of all we have to define some constants:

```
In [4]: #Problem data
        omega = 2.*np.pi*10.e9
        e0 = 8.854*1e-12*1e-18
        mu0 = 4.*np.pi*1e-7*1e6
        mue = (1.)*mu0
        ee = (16.)*e0
        mui = (-2.9214+0.5895j)*mu0
        ei = (82629.2677-200138.2211j)*e0
        k = omega*np.sqrt(e0*mu0)
        lam = 2*np.pi/k
        nm = np.sqrt((ee*mue)/(e0*mu0))
        nc = np.sqrt((ei*mui)/(e0*mu0))
        alfa_m = mue/mu0
        alfa_c = mui/mue
        antena = np.array([[1e4],[0.],[0.]])
        Amp = 1
        print("Numero de onda exterior:", k)
        print("Indice de refraccion matriz:", nm)
        print("Indice de refraccion conductor:", nc)
        print("Numero de onda interior matriz:", nm*k)
        print("Numero de onda interior conductor:", nm*nc*k)
        print("Indice de transmision matriz:", alfa_m)
        print("Indice de transmision conductor:", alfa_c)
        print("Longitud de onda:", lam, "micras")
```

```
Numero de onda exterior: 0.0002095822793
Indice de refraccion matriz: 4.0
Indice de refraccion conductor: (510.829219424+619.966251289j)
Numero de onda interior matriz: 0.000838329117198
Numero de onda interior conductor: (0.428243008559+0.519735760136j)
Indice de transmision matriz: 1.0
Indice de transmision conductor: (-2.9214+0.5895j)
Longitud de onda: 29979.5637693 micras
```

Also, we have to define the boundary functions that we will use to apply the boundary conditions. In this case an armonic wave for Dirichlet and his derivate for Neumann

```
In [5]: #Dirichlet and Neumann functions
        def dirichlet_fun(x, n, domain_index, result):
                result[0] = Amp * np.exp(1j * k * x[1])
        def neumann_fun(x, n, domain_index, result):
                result[0] = Amp * 1j * k * n[1] * np.exp(1j * k * x[1])
```

Now it's time to define the multitrace operators that represent the diagonal of the matrix. This operators have the information of the transmision between the geometries. The definition of the multitrace (A) is posible to see below:

$$A = \begin{bmatrix} -K & S \\ D & K' \end{bmatrix}$$

where K represent the double layer boundary operator, S the single layer, D the hypersingular and K′ the adjoint double layer bounday operator

```
In [6]: #Multitrace Operators
        Ai_0 = bempp.api.operators.boundary.helmholtz.multitrace_operator(grid_0, nm * nc * k)
        Ae_0 = bempp.api.operators.boundary.helmholtz.multitrace_operator(grid_0, nm * k)

        #Multitrace Transmission
        Ai_0[0,1] = Ai_0[0,1]*alfa_c
        Ai_0[1,1] = Ai_0[1,1]*alfa_c

        #Interior and exterior link up
        op_0 = (Ai_0 + Ae_0)
```

In order to obtain the spaces created with the multitrace opertaor it's posible to do the following:

```
In [7]: #Spaces
        dirichlet_space_0 = Ai_0[0,0].domain
        neumann_space_0 = Ai_0[0,1].domain
```

To make the complete diagonal of the main matrix showed at beggining is necessary to define the identity operators:

```
In [8]: #Identity
        ident_0 = bempp.api.operators.boundary.sparse.identity(neumann_space_0, neumann_space_
```

The first subinedx corresponds to the domain space, the second one to the range space. Now is time to create the big block that will have all the operators together, in this case the size is 2X2. And we create the right hand side and the left hand side of the equation

```
In [9]: #Left hand side
        blocked = bempp.api.BlockedOperator(2,2)
        blocked[0,0] = op_0[0,0]
        blocked[0,1] = op_0[0,1]
        blocked[1,0] = op_0[1,0]
        blocked[1,1] = op_0[1,1]
        blocked[1,1] = blocked[1,1] + 0.5 * ident_0 * (alfa_c - 1)

        #Boundary conditions
        dirichlet_grid_fun_0 = bempp.api.GridFunction(dirichlet_space_0, fun=dirichlet_fun)
        neumann_grid_fun_0 = bempp.api.GridFunction(neumann_space_0, fun=neumann_fun)

        #Discretizacion lado derecho
        #rhs = np.concatenate([dirichlet_grid_fun_0.coefficients, neumann_grid_fun_0.coefficie
        #Discretizacion lado izquierdo
        #blocked_discretizado = blocked.strong_form()
```

Now we solve the equation system, we use *gmres*. We print the number of iterations. Notice that the argument $use\_strong\_form = True$ is the responsable for the discretization of both sides of the equation.

```
In [10]: start = timeit.default_timer()
         #Solving the matrix system
         sol, info, it_count = bempp.api.linalg.gmres(blocked, [dirichlet_grid_fun_0, neumann_g
                                                      use_strong_form=True, return_iteration_cou
         stop = timeit.default_timer()
         #Print iterations
         print("The linear system was solved in {0} iterations".format(it_count))
         print('Time: ', stop - start)
```

```
/usr/lib/python3/dist-packages/scipy/sparse/linalg/dsolve/linsolve.py:243: SparseEfficiencyWarn
  warn('splu requires CSC matrix format', SparseEfficiencyWarning)


The linear system was solved in 199 iterations
Time:  352.68147672800114
```

```
In [11]: #Divide the solution
         solution_dirichl, solution_neumann = sol
         print(sol)
         #Plot the solution
         solution_neumann.plot()
```

```
[<bempp.api.assembly.grid_function.GridFunction object at 0x7ff04af96b00>, <bempp.api.assembly
```

```
In [26]: Nx = 150
         Ny = 150
         di=10
         xmin, xmax, ymin, ymax = [-di,di,-di,di]
         plot_grid = np.mgrid[xmin:xmax:Nx * 1j, ymin:ymax:Ny * 1j]
         points = np.vstack((plot_grid[0].ravel(),
                             plot_grid[1].ravel(),
                             0*np.ones(plot_grid[0].size)))
         u_evaluated = np.zeros(points.shape[1], dtype=np.complex128)

         x, y = points[:2]
         idx_ext = np.sqrt(x**2 + y**2) > di
         idx_int = np.sqrt(x**2 + y**2) <= di

         points_exterior = points[:, idx_ext]
         points_interior = points[:, idx_int]

         print(points_interior)
```

```
[[-9.86577181 -9.86577181 -9.86577181 ...,  9.86577181  9.86577181
   9.86577181]
 [-1.54362416 -1.40939597 -1.27516779 ...,  1.27516779  1.40939597
   1.54362416]
```

```
 [ 0.          0.          0.          ...,  0.          0.          0.        ]]
```

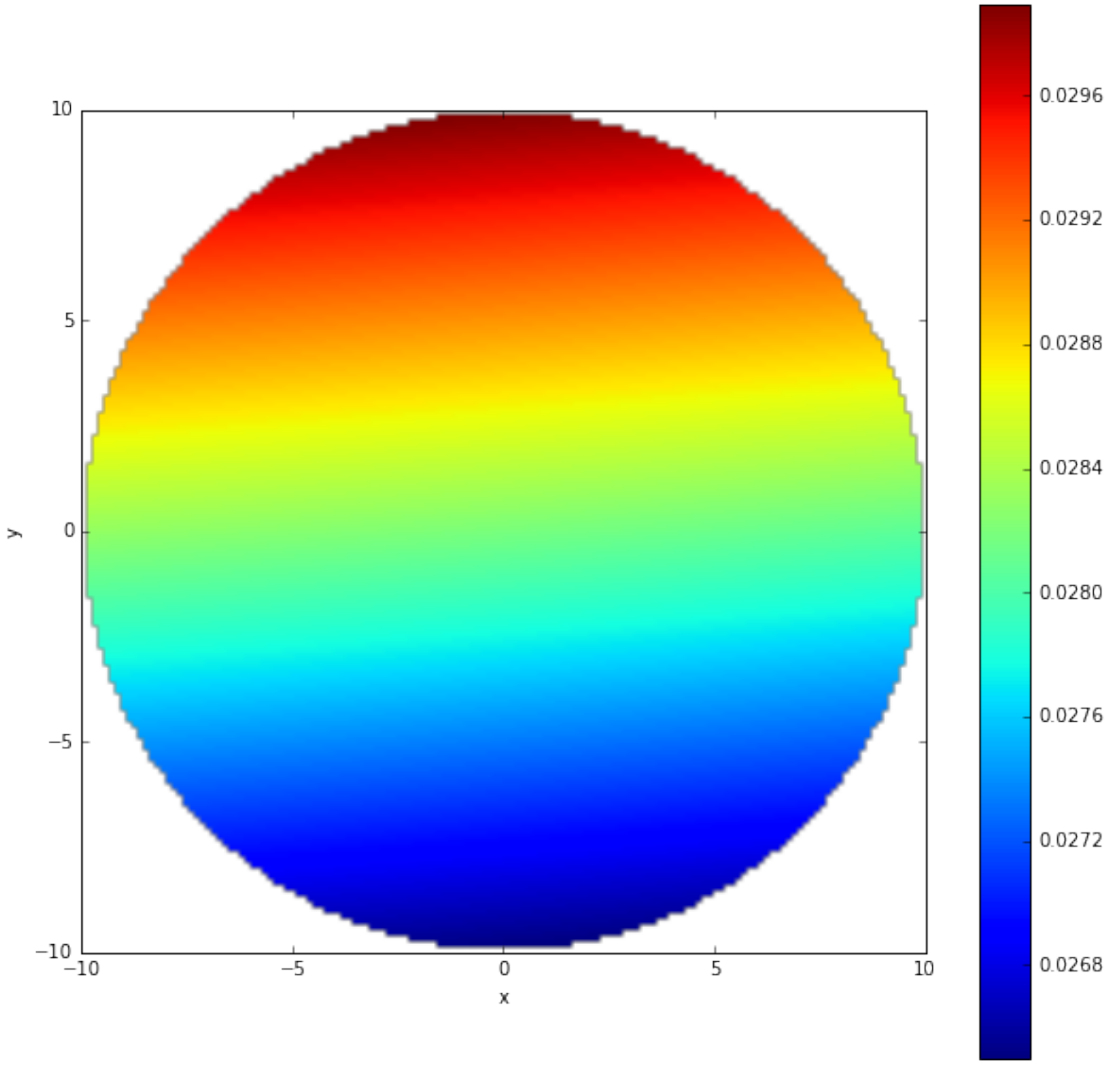
In [17]:
```
slp_pot_int = bempp.api.operators.potential.helmholtz.single_layer(
    dirichlet_space_0, points_interior, nm * k)
slp_pot_ext = bempp.api.operators.potential.helmholtz.single_layer(
    dirichlet_space_0, points_exterior, k)
dlp_pot_int = bempp.api.operators.potential.helmholtz.double_layer(
    dirichlet_space_0, points_interior, nm * k)
dlp_pot_ext = bempp.api.operators.potential.helmholtz.double_layer(
    dirichlet_space_0, points_exterior, k)

total_field_int = (slp_pot_int * solution_neumann
                   - dlp_pot_int * solution_dirichl).ravel()
total_field_ext = (dlp_pot_ext * solution_dirichl
                   - slp_pot_ext * solution_neumann).ravel() \
    + np.exp(1j * k * points_exterior[0])

total_field = np.zeros(points.shape[1], dtype='complex128')
total_field[idx_ext] = total_field_ext
total_field[idx_int] = total_field_int
total_field = total_field.reshape([Nx, Ny])
```

In [14]:
```
%matplotlib inline
from matplotlib import pylab as plt
radius = np.sqrt(plot_grid[0]**2 + plot_grid[1]**2)
total_field[radius>di] = np.nan
fig = plt.figure(figsize=(10, 10))
plt.imshow(np.real(total_field.T), extent=[-di,di,-di,di])
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar()
```

Out[14]:
```
<matplotlib.colorbar.Colorbar at 0x7ff0151bce10>
```




```
In [20]: Nx = 300
         Ny = 300
         xmin, xmax, ymin, ymax = [-13, 13, -13, 13]
         plot_grid = np.mgrid[xmin:xmax:Nx * 1j, ymin:ymax:Ny * 1j]
         points = np.vstack((plot_grid[0].ravel(),
                             plot_grid[1].ravel(),
                             np.zeros(plot_grid[0].size)))
         u_evaluated = np.zeros(points.shape[1], dtype=np.complex128)

         x, y = points[:2]
         idx_ext = np.sqrt(x**2 + y**2) > 10
         idx_int = np.sqrt(x**2 + y**2) <= 10

         points_exterior = points[:, idx_ext]
         points_interior = points[:, idx_int]
```

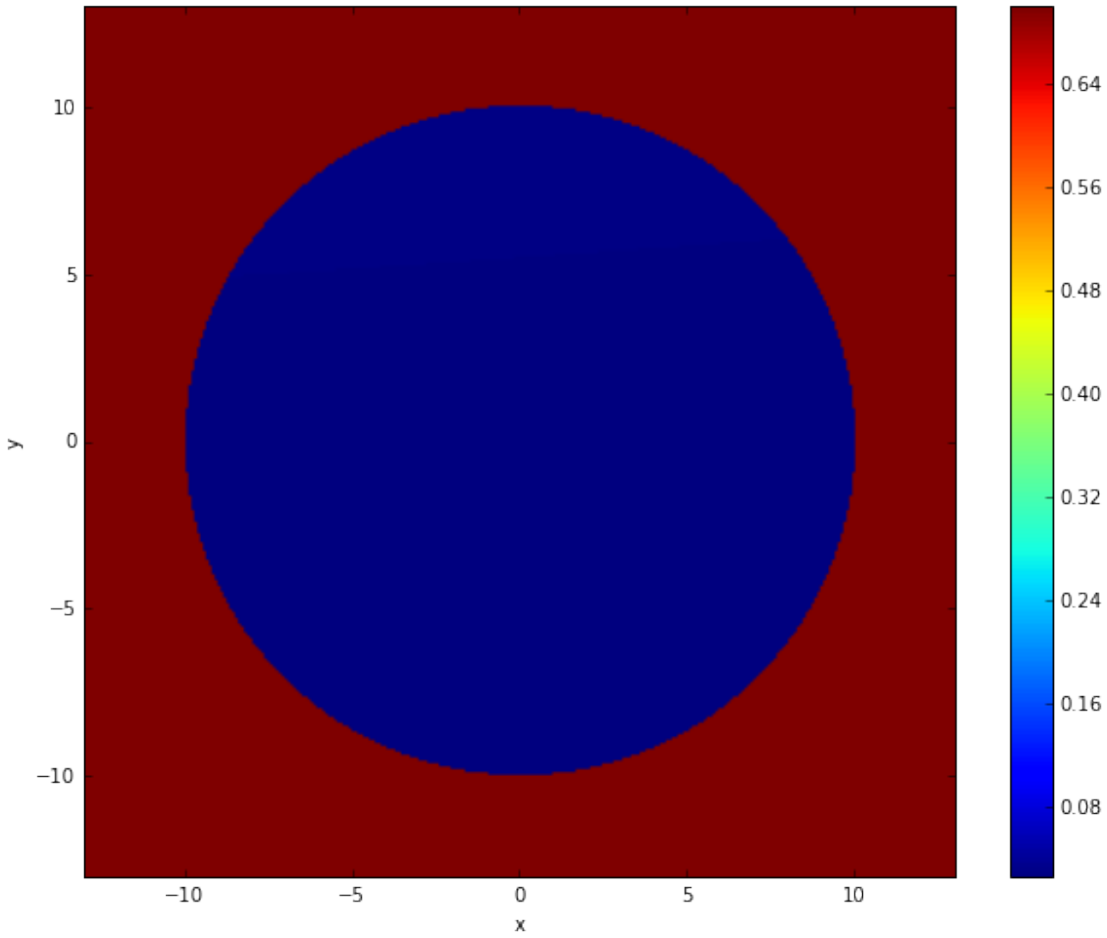
In [23]:
```
slp_pot_int = bempp.api.operators.potential.helmholtz.single_layer(
    dirichlet_space_0, points_interior, nm * k)
slp_pot_ext = bempp.api.operators.potential.helmholtz.single_layer(
    dirichlet_space_0, points_exterior, k)
dlp_pot_int = bempp.api.operators.potential.helmholtz.double_layer(
    dirichlet_space_0, points_interior, nm * k)
dlp_pot_ext = bempp.api.operators.potential.helmholtz.double_layer(
    dirichlet_space_0, points_exterior, k)

total_field_int = (slp_pot_int * solution_neumann
                   - dlp_pot_int * solution_dirichl).ravel()
total_field_ext = (dlp_pot_ext * solution_dirichl
                   - slp_pot_ext * solution_neumann).ravel() \
    + np.exp(1j * k * points_exterior[0])

total_field = np.zeros(points.shape[1], dtype='complex128')
total_field[idx_ext] = total_field_ext
total_field[idx_int] = total_field_int
total_field = total_field.reshape([Nx, Ny])
```

In [24]:
```
%matplotlib inline
from matplotlib import pylab as plt
fig = plt.figure(figsize=(10, 8))
plt.imshow(np.real(total_field.T), extent=[-13, 13, -13, 13])
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar()
```

Out[24]:
```
<matplotlib.colorbar.Colorbar at 0x7ff068ab49e8>
```


10
5
0
−5
−10
y
−10
−5
0
5
10
x
0.64
0.56
0.48
0.40
0.32
0.24
0.16
0.08