Supplementary Materials

Program code in Python written and debugged in PyCharm IDE for solving the integro-differential equation (7) by the iteration method.

```python
#
# The program calculates the current distribution and the dipole moment in a wire
# inclusion (l - length, a - radius)
# with the impedance boundary conditions on its surface and surrounded with a
# medium with epsilon and mu.
#
# Yujie Zhao @ March 2021
#
# University of St. Andrews, Fife, Scotland
#

import pandas as pd
import numpy as np
from scipy import integrate
from scipy.interpolate import CubicSpline

# Wire and material parameters
epsilon = complex(4.0, 0.0)  # effective dielectric constant of the matrix, (e1,e2)
mu = complex(1.0, 0.0)  # effective magnetic constant of the matrix, (mu1,mu2)
l = 2.5  # wire length in cm
a = 9.3e-4  # wire radius in cm

# Some constants
c = 2.9979250e+10  # speed of light in cm/s (cgs system)
i = complex(0.0, 1.0)  # imaginary unit
unit = complex(1.0, 0.0)  # real unit in the complex form
e0 = 1.0 / c  # electric field amplitude

# Calculation parameters
NCUR = 50  # number of points in the current distribution along the wire
accuracy1 = 1.0e-3  # relative error when integrating in the first approximation
# for the current
accuracy2 = 1.0e-3  # relative error when integrating in the dipole moments
accuracy3 = 1.0e-3  # relative error when integrating in the coefficients in the
# main loop by frequency
filename = 'Field5.50e_ZeroStrain.csv'  # csv file for the experimental impedance
# dispersion (freq, real, imag)

df = pd.read_csv(filename, sep = ',', header = None)  # reading the impedance file
freq = df[0]  # frequency array
NFREQ = len(df[0])  # number of frequency points
# Recalculating the experimental impedance (Ohms) into the surface impedance (cgs)
Impedance = (df[1] + df[2] * 1j) * 1.0e+9 * (a / l) / (2.0 * c)

x1 = -l / 2.0  # wire left end
x2 = l / 2.0  # wire right end
XDATA = []  # points on the wire used for calculating the current distribution
for n in range(NCUR):
    XDATA.append(x1 + l * n / (NCUR - 1))

# General Green's functions G(r) and Gf(r)
def G(r):
    return np.exp(-i * k * r) / (4.0 * np.pi * r)

def Gf(r):
    return (a ** 2) * (unit + i * k * r) * np.exp(-i * k * r) / (2.0 * (r ** 3))

# Function for calculating Q and Qf factors
def FQ(x):
    r = (x ** 2 + a ** 2)**0.5
    return (G(r)).real

def FQf(x):
    r = (x ** 2 + a ** 2)**0.5
```

```
    return (Gf(r)).real

# Functions for coef1 and coef2
def S1(x, p):
    r = ((x - p) ** 2 + a ** 2)**0.5
    return -i * (G(r)).imag / Q

def S2(x, s, p):
    r = ((s - p) ** 2 + a ** 2)**0.5
    return (i * (kn ** 2 - k ** 2) / (Q * kn)) * np.sin(kn * (x - s)) * (G(r)).imag

def S3(x, s, p):
    r = ((s - p) ** 2 + a ** 2)**0.5
    return (-f * epsilon * Z / (a * c * Q * kn)) * np.sin(kn * (x - s)) * (Gf(r)).imag

def S23(x, s, p):
    return S2(x, s, p) + S3(x, s, p)

def Re_F1a11(p, s):
    return (S23(l / 2.0, s, p) * np.sin(kn * p)).real

def Im_F1a11(p, s):
    return (S23(l / 2.0, s, p) * np.sin(kn * p)).imag

def Re_F2a11(p):
    return (S1(l / 2.0, p) * np.sin(kn * p)).real

def Im_F2a11(p):
    return (S1(l / 2.0, p) * np.sin(kn * p)).imag

def Re_F1a12(p, s):
    return (S23(l / 2.0, s, p) * np.cos(kn * p)).real

def Im_F1a12(p, s):
    return (S23(l / 2.0, s, p) * np.cos(kn * p)).imag

def Re_F2a12(p):
    return (S1(l / 2.0, p) * np.cos(kn * p)).real

def Im_F2a12(p):
    return (S1(l / 2.0, p) * np.cos(kn * p)).imag

def Re_Fa21(p):
    return (S1(-l / 2.0, p) * np.sin(kn * p)).real

def Im_Fa21(p):
    return (S1(-l / 2.0, p) * np.sin(kn * p)).imag

def Re_Fa22(p):
    return (S1(-l / 2.0, p) * np.cos(kn * p)).real

def Im_Fa22(p):
    return (S1(-l / 2.0, p) * np.cos(kn * p)).imag

def Re_F1B1(p, s):
    return (S23(l / 2.0, s, p)).real

def Im_F1B1(p, s):
    return (S23(l / 2.0, s, p)).imag

def Re_F2B1(p):
    return (S1(l / 2.0, p)).real

def Im_F2B1(p):
    return (S1(l / 2.0, p)).imag

def Re_FB2(p):
    return (S1(-l / 2.0, p)).real

def Im_FB2(p):
    return (S1(-l / 2.0, p)).imag

# Low, G(x), and upper, H(x), integration y-curves in a 2D integral (horizontal
```

```
# lines in our case)
def G1(x):
    return -l / 2.0 + x - x

def H1(x):
    return l / 2.0 + x - x

# Current distribution - zero approximation
def j0(x):
    current = (i * f * epsilon * e0 / (2.0 * Q * (kn**2))) * (np.cos(kn * x) - np.cos(kn * l / 2.0)) / np.cos(kn * l / 2.0)
    return current

def Re_j0(x):
    return (j0(x)).real

def Im_j0(x):
    return (j0(x)).imag

# Current distribution - first iteration
def j1(x):
    x0 = x
    def Re_FA1j(p, s):
        return (S23(x0, s, p) * np.sin(kn * p)).real

    def Im_FA1j(p, s):
        return (S23(x0, s, p) * np.sin(kn * p)).imag

    def Re_FA2j(p):
        return (S1(x0, p) * np.sin(kn * p)).real

    def Im_FA2j(p):
        return (S1(x0, p) * np.sin(kn * p)).imag

    def Re_FB1j(p, s):
        return (S23(x0, s, p) * np.cos(kn * p)).real

    def Im_FB1j(p, s):
        return (S23(x0, s, p) * np.cos(kn * p)).imag

    def Re_FB2j(p):
        return (S1(x0, p) * np.cos(kn * p)).real

    def Im_FB2j(p):
        return (S1(x0, p) * np.cos(kn * p)).imag

    def Re_FC1j(p, s):
        return (S23(x0, s, p)).real

    def Im_FC1j(p, s):
        return (S23(x0, s, p)).imag

    def Re_FC2j(p):
        return (S1(x0, p)).real

    def Im_FC2j(p):
        return (S1(x0, p)).imag

    def G2(x):
        return -l / 2.0 + x - x

    def H2(x):
        return x0 + x - x

    coef3 = -i * f * epsilon * e0 / (2.0 * Q * (kn ** 2))

    Integral1 = list(integrate.dblquad(Re_FA1j, x1, x2, G2, H2, epsabs = accuracy1, epsrel = accuracy1))
    Result1 = Integral1[0]

    Integral2 = list(integrate.dblquad(Im_FA1j, x1, x2, G2, H2, epsabs = accuracy1, epsrel = accuracy1))
    Result2 = Integral2[0]
```

```
    Integral3 = list(integrate.quad(Re_FA2j, x1, x2, epsabs = accuracy1, epsrel = accuracy1))
    Result3 = Integral[0]

    Integral4 = list(integrate.quad(Im_FA2j, x1, x2, epsabs = accuracy1, epsrel = accuracy1))
    Result4 = Integral[0]

    current = coef1 * (np.sin(kn * x) + (Result1 + Result3) * unit + (Result2 + Result4) * i)

    Integral1 = list(integrate.dblquad(Re_FB1j, x1, x2, G2, H2, epsabs = accuracy1, epsrel = accuracy1))
    Result1 = Integral1[0]

    Integral2 = list(integrate.dblquad(Im_FB1j, x1, x2, G2, H2, epsabs = accuracy1, epsrel = accuracy1))
    Result2 = Integral2[0]

    Integral3 = list(integrate.quad(Re_FB2j, x1, x2, epsabs = accuracy1, epsrel = accuracy1))
    Result3 = Integral3[0]

    Integral4 = list(integrate.quad(Im_FB2j, x1, x2, epsabs = accuracy1, epsrel = accuracy1))
    Result4 = Integral4[0]

    current = current + coef2 * (np.cos(kn * x) + (Result1 + Result3) * unit + (Result2 + Result4) * i)

    Integral1 = list(integrate.dblquad(Re_FC1j, x1, x2, G2, H2, epsabs = accuracy1, epsrel = accuracy1))
    Result1 = Integral1[0]

    Integral2 = list(integrate.dblquad(Im_FC1j, x1, x2, G2, H2, epsabs = accuracy1, epsrel = accuracy1))
    Result2 = Integral2[0]

    Integral3 = list(integrate.quad(Re_FC2j, x1, x2, epsabs = accuracy1, epsrel = accuracy1))
    Result3 = Integral3[0]

    Integral4 = list(integrate.quad(Im_FC2j, x1, x2, epsabs = accuracy1, epsrel = accuracy1))
    Result4 = Integral4[0]



    current = current + coef3 * (unit + (Result1 + Result3) * unit + (Result2 + Result4) * i)
    return current

# Sampling j1(x), its real and imaginary parts
def Re_j1():
    FDATA = []
    for n in range(NCUR):
        FDATA.append((j1(XDATA[n])).real)
    return FDATA

def Im_j1():
    FDATA = []
    for n in range(NCUR):
        FDATA.append((j1(XDATA[n])).imag)
    return FDATA

def moment():
    Integral = list(integrate.quad(Re_j0, x1, x2, epsabs = accuracy2, epsrel = accuracy2))
    Result = Integral[0]
    dip0 = unit * Result
    Integral = list(integrate.quad(Im_j0, x1, x2, epsabs = accuracy2, epsrel = accuracy2))
    Result = Integral[0]
    dip0 = - (dip0 + i * Result) * i /(2.0 * np.pi * f)

    FDATA = Re_j1()
    cs = CubicSpline(XDATA, FDATA)  # cubic spline interpolation of Re_j1(x)
# calculated in discrete points
    cubfun = lambda x: cs(x)
    Integral = list(integrate.quad(cubfun, x1, x2, epsabs = accuracy2, epsrel = accuracy2))
    Result = Integral[0]
    dip1 = unit * Result
```

```

    FDATA = Im_j1()
    cs = CubicSpline(XDATA, FDATA)
    cubfun = lambda x: cs(x)
    Integral = list(integrate.quad(cubfun, x1, x2, epsabs = accuracy2, epsrel = accuracy2))
    Result = Integral[0]
    dip1 = - (dip1 + i * Result) * i / (2.0 * np.pi * f)

    return [dip0, dip1]

dip0_real = []  # real part of the dipole moment - zero approximation
dip0_imag = []  # imaginary part of the dipole moment - zero approximation
dip1_real = []  # real part of the dipole moment - first approximation
dip1_imag = []  # imaginary part of the dipole moment - first approximation

for n in range(NFREQ):
    f = freq[n]  # frequency array
    Z = Impedance[n]  # surface impedance array calculated from the experimental
# impedance (Ohms)
    k = 2.0 * np.pi * f * (epsilon * mu)**0.5 / c  # wavenumber in a medium with
# epsilon and mu

    Integral = list(integrate.quad(FQ, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Q = Integral[0]
    Integral = list(integrate.quad(FQf, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Qf = Integral[0]

    kn = k * (unit - i * c * Z * Qf/(4.0 * (np.pi**2) * a * f * mu *Q))**0.5
# normalised wavenumber

    Integral = list(integrate.dblquad(Re_F1a11, x1, x2, G1, H1, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a11 = unit * Result
    Integral = list(integrate.dblquad(Im_F1a11, x1, x2, G1, H1, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a11 = a11 + i * Result
    Integral = list(integrate.quad(Re_F2a11, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a11 = a11 + unit * Result
    Integral = list(integrate.quad(Im_F2a11, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a11 = a11 + i * Result

    Integral = list(integrate.dblquad(Re_F1a12, x1, x2, G1, H1, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a12 = unit * Result
    Integral = list(integrate.dblquad(Im_F1a12, x1, x2, G1, H1, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a12 = a12 + i * Result
    Integral = list(integrate.quad(Re_F2a12, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a12 = a12 + unit * Result
    Integral = list(integrate.quad(Im_F2a12, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a12 = a12 + i * Result

    Integral = list(integrate.quad(Re_Fa21, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a21 = unit * Result
    Integral = list(integrate.quad(Im_Fa21, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a21 = a21 + i * Result

    Integral = list(integrate.quad(Re_Fa22, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a22 = unit * Result
    Integral = list(integrate.quad(Im_Fa22, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    a22 = a22+i * Result
```

```
    Integral = list(integrate.dblquad(Re_F1B1, x1, x2, G1, H1, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    B1 = unit * Result
    Integral = list(integrate.dblquad(Im_F1B1, x1, x2, G1, H1, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    B1 = B1 + i * Result
    Integral = list(integrate.quad(Re_F2B1, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    B1 = B1 + unit * Result
    Integral = list(integrate.quad(Im_F2B1, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    B1 = B1 + i * Result
    B1 = (unit + B1) * i * f * epsilon * e0 / (2.0 * Q * (kn**2))

    Integral = list(integrate.quad(Re_FB2, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    B2 = unit * Result
    Integral = list(integrate.quad(Im_FB2, x1, x2, epsabs = accuracy3, epsrel = accuracy3))
    Result = Integral[0]
    B2 = B2 + i * Result
    B2 = (unit + B2) * i * f * epsilon * e0 / (2.0 * Q * (kn**2))

    coef2 = (B1 + B2) / 2.0
    coef2 = coef2 + (B2 - B1) * (a11 + a21) / (4.0 * np.sin(kn * l / 2.0) + 2.0 * (a11 - a21))
    coef2 = coef2 / (np.cos(kn * l / 2.0) + (a22 - a12) * (a11 + a21)
                     /(4.0 * np.sin(kn * l / 2.0) + 2.0 * (a11 - a21)) + (a12 + a22) / 2.0)
    coef1 = ((B1 - B2) + coef2 * (a22 - a12)) / (2.0 * np.sin(kn * l / 2.0) + (a11 - a21))

    dip01 = moment()  # [dip0, dip1] - array of the zero and first approximations
# for the dipole moment
    dip0_real.append(c * (dip01[0]).real)
    dip0_imag.append(-c * (dip01[0]).imag)
    dip1_real.append(c * (dip01[1]).real)
    dip1_imag.append(-c * (dip01[1]).imag)

    print('frequency point = ', n + 1)

disper0 = np.column_stack((freq, dip0_real, dip0_imag))  # dispersion of the dipole
# moment - zero approximation
disper1 = np.column_stack((freq, dip1_real, dip1_imag))  # dispersion of the dipole
# moment - first approximation

np.savetxt('disper0.csv', disper0, delimiter=',')
np.savetxt('disper1.csv', disper1, delimiter=',')
```