Karin Metzgar

ASTE 546

HW 11

Part 1: Electric Field Correction

Im confused on the first part about the “correction” and definition of Ea, It looks like

dE\_dt = c2 \* (curlB) – J/e0 in the provided code and that is accurate with Ampere’s law..

So continuing with the assignment assuming that is true. I’ve added the function to calculate the divergence of E:

def computeDivE(Ex, Ey):

divE = numpy.zeros((ni,nj))

for i in range(1, ni-1):

for j in range(1, nj-1):

dEx\_dx = (Ex[i+1][j] - Ex[i-1][j]) / (2\*dx)

dEy\_dy = (Ey[i][j+1] - Ey[i][j-1]) / (2\*dy)

divE[i][j] = dEx\_dx + dEy\_dy

# Linear interpolation on the red??? internal nodes

for i in range(1, ni-1, 2): #red nodes are every other node

for j in range(1, nj-1, 2):

divE[i][j] = 0.25 \* (divE[i-1][j-1] + divE[i+1][j-1] + divE[i-1][j+1] + divE[i+1][j+1])

return divE

Now use linear interpolation to set divE on the “red” internal nodes of the mesh (averaging the four centroids). I don’t know what is meant by “red” nodes, so I’m assuming this portion is asking to get the divE values, by linear interpolation, on the node points, which we would get by averaging the surrounding centroids..

def computeInterpolation(divE):

divE\_inter = numpy.zeros((ni+1, nj+1))

#more nodes than centroids

for i in range(1, ni):

for j in range(1, nj):

#average the surrounding centroid values

divE\_inter[i, j] = 0.25 \* ( divE[i-1, j-1] + divE[i-1, j] + divE[i, j-1] + divE[i, j])

return divE\_inter

Compute b = rho/e0 – divE (using the interpolated divE?)

def computeB(rho, divE\_inter, eps0):

b = numpy.zeros((ni, nj))

for i in range(1, ni-1):

for j in range(1, nj-1):

b[i][j] = ( rho[i, j] / eps0 ) - divE\_inter[i, j]

return b

Use a Poisson solver to solve ..

def solvePoisson(b, tol=1e-5, max\_iter=10000):

phi = numpy.zeros((ni, nj))

phi[0, :] = 0

phi[ni-1, :] = -1

for iteration in range(max\_iter):

max\_diff = 0

for i in range(1, ni-1):

for j in range(1, nj-1):

if j == 0 or j == nj-1: # boundary conditions

phi\_new = 0.5 \* (

(phi[i+1, j] + phi[i-1, j]) / (dx\*\*2) -

b[i, j]) / (1 / (dx\*\*2))

else:

phi\_new = 0.5 \* (

(phi[i+1, j] + phi[i-1, j]) / (dx\*\*2) +

(phi[i, j+1] + phi[i, j-1]) / (dy\*\*2) -

b[i, j]) / (1 / (dx\*\*2) + 1 / (dy\*\*2))

diff = abs(phi\_new - phi[i, j])

max\_diff = max(max\_diff, diff)

phi[i, j] = phi\_new

if max\_diff < tol:

print(f'Converged after {iteration} iterations')

break

return phi

Compute E\_star = -div\_phi

def computeEStar(phi):

Ex\_star = numpy.zeros((ni, nj))

Ey\_star = numpy.zeros((ni, nj))

for i in range(1, ni-1):

for j in range(1, nj-1):

Ex\_star[i, j] = -(phi[i+1, j] - phi[i-1, j]) / (2 \* dx)

Ey\_star[i, j] = -(phi[i, j+1] - phi[i, j-1]) / (2 \* dy)

return Ex\_star, Ey\_star

Update Ex = Exa + Ex\_star and Ey..

With the code like this:

# -\*- coding: utf-8 -\*-

"""

ASTE-546 EM\_PIC Example, Part B

"""

import numpy

import pylab as pl

#computes curl of E

def computeCurlE(Ex, Ey):

curl = numpy.zeros((ni-1,nj-1))

for i in range(ni-1):

for j in range(nj-1):

S = 0.5\*(Ex[i][j]+Ex[i+1][j])

E = 0.5\*(Ey[i+1][j]+Ey[i+1][j+1])

N = -0.5\*(Ex[i+1][j+1]+Ex[i][j+1])

W = -0.5\*(Ey[i][j+1]+Ey[i][j])

curl[i][j] = (S\*dx + E\*dy + N\*dx + W\*dy)/(dx\*dy)

return curl

#computes curl B on [1:nx-1],[1-ny-1]

def computeCurlB(Bz):

curlX = numpy.zeros((ni,nj))

curlY = numpy.zeros((ni,nj))

for i in range(ni-2):

for j in range (nj-2):

N = 0.5\*(Bz[i+1][j+1] + Bz[i][j+1])

S = 0.5\*(Bz[i][j] + Bz[i+1][j])

E = 0.5\*(Bz[i+1][j] + Bz[i+1][j+1])

W = 0.5\*(Bz[i][j+1] + Bz[i][j])

curlX[i+1][j+1] = (N-S)/dy

curlY[i+1][j+1] = -(E-W)/dx

return curlX,curlY

def computeDivE(Ex, Ey):

divE = numpy.zeros((ni, nj))

for i in range(1, ni-1):

for j in range(1, nj-1):

divEx = (Ex[i+1, j] - Ex[i-1, j]) / (2 \* dx)

divEy = (Ey[i, j+1] - Ey[i, j-1]) / (2 \* dy)

divE[i, j] = divEx + divEy

return divE

def computeInterpolation(divE):

divE\_inter = numpy.zeros((ni+1, nj+1))

#more nodes than centroids

for i in range(1, ni):

for j in range(1, nj):

#average the surrounding centroid values

divE\_inter[i, j] = 0.25 \* ( divE[i-1, j-1] + divE[i-1, j] + divE[i, j-1] + divE[i, j])

return divE\_inter

def computeB(rho, divE\_inter, eps0):

b = numpy.zeros((ni, nj))

for i in range(1, ni-1):

for j in range(1, nj-1):

b[i][j] = ( rho[i, j] / eps0 ) - divE\_inter[i, j]

return b

# def solvePoisson(b, tol=1e-5, max\_iter=10000):

# phi = numpy.zeros((ni, nj))

# phi[0, :] = 0

# phi[ni-1, :] = -1

# for iteration in range(max\_iter):

# max\_diff = 0

# for i in range(1, ni-1):

# for j in range(1, nj-1):

# if j == 0 or j == nj-1: # boundary conditions

# phi\_new = 0.5 \* (

# (phi[i+1, j] + phi[i-1, j]) / (dx\*\*2) -

# b[i, j]) / (1 / (dx\*\*2))

# else:

# phi\_new = 0.5 \* (

# (phi[i+1, j] + phi[i-1, j]) / (dx\*\*2) +

# (phi[i, j+1] + phi[i, j-1]) / (dy\*\*2) -

# b[i, j]) / (1 / (dx\*\*2) + 1 / (dy\*\*2))

# diff = abs(phi\_new - phi[i, j])

# max\_diff = max(max\_diff, diff)

# phi[i, j] = phi\_new

# if max\_diff < tol:

# print(f'Converged after {iteration} iterations')

# break

# return phi

def solvePoisson(b, tol=1e-4, max\_iter=10000):

phi = numpy.zeros((ni, nj))

# Boundary conditions

phi[0, :] = 0 # phi = 0 at i = 0

phi[-1, :] = -1 # phi = -1 at i = ni-1

# Relaxation factor

omega = 1.5

for iteration in range(max\_iter):

max\_diff = 0

for i in range(1, ni-1):

for j in range(1, nj-1):

if j == 0 or j == nj-1: # Neumann boundary conditions

phi[i, j] = (1 - omega) \* phi[i, j] + omega \* (

(phi[i+1, j] + phi[i-1, j]) / (dx\*\*2) -

b[i, j]) / (2 / (dx\*\*2))

else:

phi\_new = 0.5 \* (

(phi[i+1, j] + phi[i-1, j]) / (dx\*\*2) +

(phi[i, j+1] + phi[i, j-1]) / (dy\*\*2) -

b[i, j]) / (1 / (dx\*\*2) + 1 / (dy\*\*2))

diff = abs(phi\_new - phi[i, j])

max\_diff = max(max\_diff, diff)

phi[i, j] = phi\_new

if max\_diff < tol:

print(f'Converged after {iteration} iterations')

break

return phi

def computeEStar(phi):

Ex\_star = numpy.zeros((ni, nj))

Ey\_star = numpy.zeros((ni, nj))

for i in range(1, ni-1):

for j in range(1, nj-1):

Ex\_star[i, j] = -(phi[i+1, j] - phi[i-1, j]) / (2 \* dx)

Ey\_star[i, j] = -(phi[i, j+1] - phi[i, j-1]) / (2 \* dy)

return Ex\_star, Ey\_star

#plots data with a given title

def plot(ax,data,title=""):

pl.sca(ax)

pl.cla()

cf = pl.contourf(pos\_x, pos\_y, numpy.transpose(data),8,alpha=.75,cmap='jet')

ax.set\_yticks(pos\_y)

ax.set\_xticks(pos\_x)

ax.xaxis.set\_ticklabels([])

ax.yaxis.set\_ticklabels([])

pl.xlim(min(pos\_x),max(pos\_x))

pl.ylim(min(pos\_y),max(pos\_y))

#ax.grid(b=True,which='both',color='#444',linestyle='-')

ax.set\_aspect('equal', adjustable='box')

pl.title(title)

pl.colorbar(cf,ax=pl.gca(),orientation='vertical',shrink=0.75, pad=0.01)

#sets field to zero along domain boundaries to prevent reflection

def clearBoundaries(F):

ni,nj = numpy.shape(F)

F[0,:] = 0

F[ni-1,:] = 0

F[:,0] = 0

F[:,nj-1] = 0

#constants

eps0 = 8.854187e-12 #permitivity of free space

mu0 = 1.2566370e-6 #permeability of free space

c2 = 1/(eps0\*mu0) #speed of light squared

c = numpy.sqrt(c2) #speed of light

E = 1.602e-19 #elementary charge

#define electric mesh

ni = 81

nj = 41

dx = 0.025

dy = 0.025

#time step

dt = 0.5\*(dx+dy)/c

Ex = numpy.zeros((ni,nj))

Ey = numpy.zeros((ni,nj))

den\_i = numpy.zeros((ni,nj))

#define magnetic mesh

Bz = numpy.zeros((ni-1,nj-1))

#placeholder for current density

den = numpy.zeros((ni,nj))

den[30:50,18:22] = 1e4

rho = den\*E

ux = 1e3

Jx = rho\*ux

Jy = numpy.zeros((ni,nj))

# #main loop

# for it in range(1):

# dB\_dt = -computeCurlE(Ex,Ey)

# Bz += dB\_dt\*dt

# curlBx, curlBy = computeCurlB(Bz)

# dEx\_dt = c2\*curlBx - Jx/eps0

# dEy\_dt = c2\*curlBy - Jy/eps0

# Ex += dEx\_dt\*dt

# Ey += dEy\_dt\*dt

# div\_E = computeDivE(Ex,Ey)

# div\_E\_interpolated = computeInterpolation(div\_E)

# b = computeB(rho,div\_E\_interpolated,eps0)

# phi = solvePoisson(b)

# Ex\_star, Ey\_star = computeEStar(phi)

# Ex\_total = Ex + Ex\_star

# Ey\_total = Ey + Ey\_star

# #clear fields on boundaries to prevent reflection

# clearBoundaries(Ex)

# clearBoundaries(Ey)

# clearBoundaries(Bz)

#main loop

for it in range(100):

dB\_dt = -computeCurlE(Ex,Ey)

Bz += dB\_dt\*dt

curlBx, curlBy = computeCurlB(Bz)

dEx\_dt = c2\*curlBx - Jx/eps0

dEy\_dt = c2\*curlBy - Jy/eps0

Ex += dEx\_dt\*dt

Ey += dEy\_dt\*dt

div\_E = computeDivE(Ex,Ey)

div\_E\_interpolated = computeInterpolation(div\_E)

b = computeB(rho,div\_E\_interpolated,eps0)

# phi = solvePoisson(b)

# Ex\_star, Ey\_star = computeEStar(phi)

# Ex\_total = Ex + Ex\_star

# Ey\_total = Ey + Ey\_star

#clear fields on boundaries to prevent reflection

clearBoundaries(Ex)

clearBoundaries(Ey)

clearBoundaries(Bz)

#plotting

fig = pl.figure(num=None, figsize=(12, 10), dpi=80, facecolor='w', edgecolor='k')

sub = (pl.subplot(5,2,1),pl.subplot(5,2,2),pl.subplot(5,2,3),pl.subplot(5,2,4),

pl.subplot(5,2,5),pl.subplot(5,2,6),pl.subplot(5,2,7),pl.subplot(5,2,8),pl.subplot(5,2,9),pl.subplot(5,2,10))

pos\_x = numpy.linspace(0,(ni-1)\*dx,ni)

pos\_y = numpy.linspace(0,(nj-1)\*dy,nj)

plot(sub[0],Ex,title="Exa")

plot(sub[1],Ey,title="Eya")

plot(sub[2],Ex\_star,title="Ex\_star")

plot(sub[3],Ey\_star,title="Ey\_star")

plot(sub[4],Ex\_total,title="Ex")

plot(sub[5],Ey\_total,title="Ey")

plot(sub[6],b,title="b")

plot(sub[7],phi,title="phi")

plot(sub[9],div\_E,title="div\_E")

#recompute pos\_x/pos\_y for the magnetic mesh

pos\_x = numpy.linspace(0.5\*dx,(ni-2)\*dx,ni-1)

pos\_y = numpy.linspace(0.5\*dy,(nj-2)\*dy,nj-1)

plot(sub[8],Bz,title="Bz")

I’m able to get this ..

A screenshot of a computer screen

Description automatically generated

But once I add in the commented out portions and change the number of iterations to 1 I get this:

A screenshot of a computer

Description automatically generated

Alright, I uncommented out all the lines and let it run 100 times, and got this:

#main loop

for it in range(100):

dB\_dt = -computeCurlE(Ex,Ey)

Bz += dB\_dt\*dt

curlBx, curlBy = computeCurlB(Bz)

dEx\_dt = c2\*curlBx - Jx/eps0

dEy\_dt = c2\*curlBy - Jy/eps0

Ex += dEx\_dt\*dt

Ey += dEy\_dt\*dt

div\_E = computeDivE(Ex,Ey)

div\_E\_interpolated = computeInterpolation(div\_E)

b = computeB(rho,div\_E,eps0)

phi = solvePoisson(b)

Ex\_star, Ey\_star = computeEStar(phi)

Ex\_total = Ex + Ex\_star

Ey\_total = Ey + Ey\_star

#clear fields on boundaries to prevent reflection

clearBoundaries(Ex)

clearBoundaries(Ey)

clearBoundaries(Bz)

A screenshot of a computer

Description automatically generated