Mario L. Gutierrez Abed
364009832
mlg3843@rit.edu

Problem Set 5
Computational Astrophysics

11-22-2021

The Maxwell Equations in vacuum (in the geometric form of Gaussian units) take the form:

$$\boxed{\begin{aligned} \frac{\partial \vec{B}}{\partial t} &= -\vec{\nabla} \times \vec{E} && \text{(Faraday)} \quad (1a)\\ \frac{\partial \vec{E}}{\partial t} &= \vec{\nabla} \times \vec{B} && \text{(Ampère)} \quad (1b)\\ \vec{\nabla} \cdot \vec{E} &= 0 && \text{(Gauss)} \quad (1c)\\ \vec{\nabla} \cdot \vec{B} &= 0. && \quad (1d) \end{aligned}}$$

The latter two are constraint equations that must be satisfied at each time step, while the former two are evolution equations. If we specialize to plane waves moving in the $\pm z$ direction, then we have

$$E_z = B_z = 0 \tag{2a}$$
$$\partial_x \vec{E} = \partial_x \vec{B} = 0 \tag{2b}$$
$$\partial_y \vec{E} = \partial_y \vec{B} = 0, \tag{2c}$$

so that the relevant Maxwell equations become

$$\partial_t B_x = \partial_z E_y \tag{3a}$$
$$\partial_t B_y = -\partial_z E_x \tag{3b}$$
$$\partial_t E_x = -\partial_z B_y \tag{3c}$$
$$\partial_t E_y = \partial_z B_x. \tag{3d}$$

Problem 1. *Write Eqs. (3) in the form*

$$\partial_t \vec{U} = A\, \partial_z \vec{U}$$

and find the eigenvectors and eigenvalues of A *(note there is degeneracy here).*

Solution. Let $\vec{U} = [B_x \ B_y \ E_x \ E_y]^\top$; then we have

$$\partial_t \begin{bmatrix} B_x \\ B_y \\ E_x \\ E_y \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}}_{A} \partial_z \begin{bmatrix} B_x \\ B_y \\ E_x \\ E_y \end{bmatrix}. \tag{4}$$

Now, from $A\vec{v} = \lambda\vec{v}$, we get

$$0 = \det \begin{bmatrix} -\lambda & 0 & 0 & 1 \\ 0 & -\lambda & -1 & 0 \\ 0 & -1 & -\lambda & 0 \\ 1 & 0 & 0 & -\lambda \end{bmatrix} = \lambda^4 - 1,$$

so the eigenvalues are ± 1, each with multiplicity 2. The corresponding eigenvectors are then

$$\left\{ \begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}.$$

♠

Problem 2. *Perform a Von-Neumann stability analysis of the discrete version of Eqs.* (3)*, where*

$$\partial_z \vec{U} = \frac{U_{i+1} - U_{i-1}}{2h}$$

and the resulting system of ODEs is solved using standard RK4. *Determine (roughly) the maximum value of* $c = \Delta t/h$ *for which the system is stable.*

Solution. To perform a von Neumann analysis on one component of $\vec{U}$ (say, the x-component) we put [1]

$$U_i^n := S^n e^{\iota \omega x_i},$$

where S^n is some scalar factor (when working with the full vector $\vec{U}$ we will have a matrix instead of a scalar). Then we have

$$\begin{aligned}
\frac{U_{i+1} - U_{i-1}}{2h} &= S^n \frac{e^{\iota\omega(x+h)} - e^{\iota\omega(x-h)}}{2h} \\
&= S^n\, e^{\iota\omega x} \frac{e^{\iota\omega h} - e^{-\iota\omega h}}{2h} \\
&= S^n\, e^{\iota\omega x}\, \iota \frac{\sin(\omega h)}{h}.
\end{aligned}$$

We now apply RK4 to the full vector and extract the eigenvalues from the amplification matrix:

```
from sympy import *
import numpy as np
from matplotlib import pyplot as plt

var(('h', 'dt', 'U', 'B_x', 'B_y', 'E_x', 'E_y', 'omega', 'x', 'c'))

A = Matrix([ [0, 0, 0, 1],
             [0, 0, -1, 0],
             [0, -1, 0, 0],
             [1, 0, 0, 0],
           ])
U = Matrix([
            [B_x],
            [B_y],
            [E_x],
            [E_y]
            ])

def rhs(u):
    return A * u * I * sin(omega*h) / h

# apply RK4 routine
K1 = rhs(U)
K2 = rhs(U + dt/2 * K1)
K3 = rhs(U + dt/2 * K2)
K4 = rhs(U + dt * K3)
U1 = U + dt/6 * (K1 + 2 * K2 + 2 * K3 + K4)


# Simplify U1
U1.simplify()

# get the coefficients
coef_U0 = diff(U1,U[0])
coef_U1 = diff(U1,U[1])
coef_U2 = diff(U1,U[2])
coef_U3 = diff(U1,U[3])
```

[1](Here we are using $\iota = \sqrt{-1}$ to avoid confusion with the index i.

```python

# form the matrix
m = Matrix( [ [coef_U0, coef_U1, coef_U2, coef_U3] ] )
m = simplify(m.subs({dt : c * h, omega * h : x}))

# get the eigenvalues
evals=[]
for key in m.eigenvals().keys():
    print(key)
    key.simplify()
    evals.append(key)

# evaluate the eigenvalues for a given courant factor c
f = evals[0].subs({c : 2.8}).simplify()
g = lambdify(x, f)

# output plot
z_grid = np.linspace(0, 2*np.pi, 500, dtype=np.complex128)
y_grid = np.abs(g(z_grid))
x_grid = z_grid.real
plt.plot(x_grid, y_grid)
plt.show()
```

As the following plot shows, the maximum Courant value for which we are guaranteed stability is roughly $c = 2.827$:

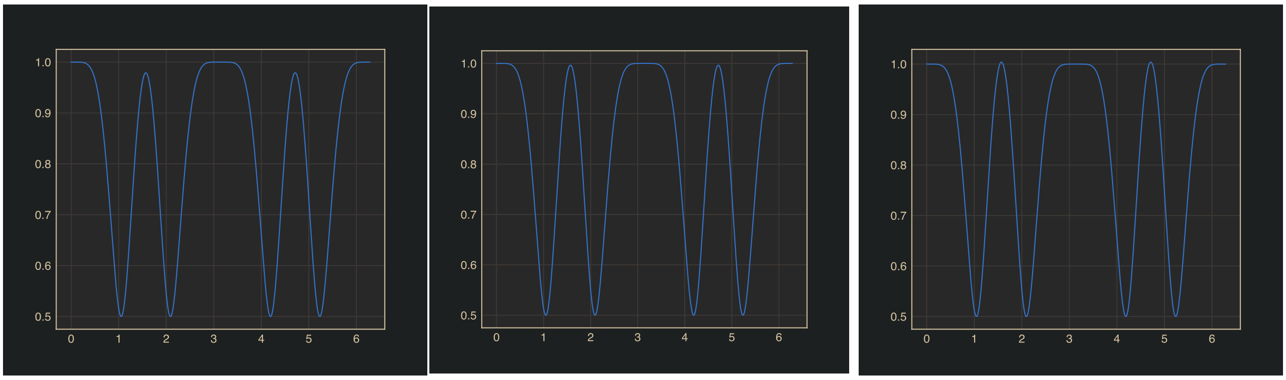


Figure 1: Absolute value of eigenvalues of the amplification matrix. From left to right we have Courant values $c = 2.82, 2.827, 2.83$. At $c = 2.82$ the system is quite stable; at $c = 2.827$ we still have a stable system, but we are just about approaching the limit of stability ... for $c = 2.83$ we see that we are getting eigenvalues with magnitude just bigger than 1, so we have already crossed the stability threshold at this point.

♠

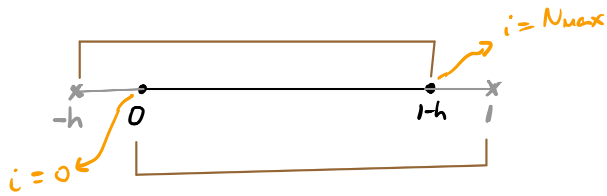
Problem 3. *Implement the above finite-difference algorithm on a grid $0 \leqslant z < 1$ with periodic boundary conditions (of period 1). Choose initial conditions for which you can also obtain an exact solution. Using those conditions, evolve the Maxwell system to at least $t = 10$ and determine the L^∞-norm of the error on the grid at $t \sim 10$ as a function of the number of gridpoints. (Hint: This step is much easier to do if you can guarantee that $t = 10$ is an integer number of timesteps from $t = 0$.) Make a simple animation of one of the fields as it changes in time.*

Solution. First we write the differential matrix corresponding to the spatial derivatives

$$\frac{U_{i+1} - U_{i-1}}{2h}. \tag{5a}$$

Given the stated periodic boundary conditions, we have

$$
\begin{aligned}
U(0) &= U(1) \\
U(-h) &= U(1-h).
\end{aligned}
$$



Thus, for $i = 0$, (5a) becomes

$$\frac{U_1 - U_{-1}}{2h} = \frac{U_1 - U_{N_{max}}}{2h}. \tag{5b}$$

Similarly, for $i = N_{max}$,

$$\frac{U_{N_{max}+1} - U_{N_{max}-1}}{2h} = \frac{U_0 - U_{N_{max}-1}}{2h}. \tag{5c}$$

Hence, writing

$$\vec{U} = \begin{bmatrix} {}^{(0)}U \\ {}^{(1)}U \\ {}^{(2)}U \\ {}^{(3)}U \end{bmatrix} = \begin{bmatrix} B_x \\ B_y \\ E_x \\ E_y \end{bmatrix},$$

the fully discrete form of the RHS of Eq. (4) becomes

$${}^{(k)}\dot{U} = \frac{1}{2h} \begin{bmatrix} 0 & 1 & & & & -1 \\ -1 & 0 & 1 & & & \\ & -1 & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 0 & 1 \\ 1 & & & & -1 & 0 \end{bmatrix} \begin{bmatrix} A^{(k)}U_0 \\ \vdots \\ A^{(k)}U_i \\ \vdots \\ A^{(k)}U_{N_{max}} \end{bmatrix}, \tag{6}$$

where the (k) left superscript indicates the k^{th}-component of $\vec{U}$.

```python
import numpy as np

A = np.array([
             [0., 0., 0.,  1.],
             [0., 0., -1., 0.],
             [0., -1., 0., 0.],
             [1., 0.,  0., 0.]
            ])

'''
    RHS of Maxwell's Equations
'''
def maxwell_rhs(u, h, N):

    rhs      = np.zeros_like(u)
    M        = np.zeros((N+1,N+1))
    num_cols = len(u[0])

    for i in range (N):
        M[i,i+1] = 1.
        M[i+1,i] = -1.

    M[0,-1] = -1.
    M[-1,0] = 1.
    M      = .5/h * M

```

```python
    u_vec = np.dot(A,u.T)
    u_vec = u_vec.T

    for col in range(num_cols):
        rhs[:,col] = np.dot(M,u_vec[:,col])

    return rhs

'''
    RK4 Routine
'''
def rk4_step(u, rhs_func, dt, h, **kwargs):
    k1 = rhs_func(u, h, **kwargs)
    k2 = rhs_func(u + .5 * dt * k1, h, **kwargs)
    k3 = rhs_func(u + .5 * dt * k2, h, **kwargs)
    k4 = rhs_func(u + dt * k3, h, **kwargs)

    return u + dt/6. * (k1 + 2.*k2 + 2.*k3 + k4)
```

The system is then evolved using the RK4 scheme, as described above. In order to obtain an exact solution, we choose to set

$$B_x(t,z) = \sin\left[2\pi(t-z)\right] \tag{7a}$$

$$E_x(t,z) = \cos\left[2\pi(t-z)\right]. \tag{7b}$$

Then, from Eqs. (3), we get

$$B_y(t,z) = \cos\left[2\pi(t-z)\right] \tag{7c}$$

$$E_y(t,z) = -\sin\left[2\pi(t-z)\right]. \tag{7d}$$

Hence the initial conditions are given by

$$\vec{U}^0 = \begin{bmatrix} B_x(0,z) \\ B_y(0,z) \\ E_x(0,z) \\ E_y(0,z) \end{bmatrix} = \begin{bmatrix} -\sin(2\pi z) \\ \cos(2\pi z) \\ \cos(2\pi z) \\ \sin(2\pi z) \end{bmatrix}. \tag{8}$$

```python
from matplotlib import pyplot as plt

'''
    Evolve the system using RK4
'''

T_FINAL = 10.
COURANT = .5
z0      = 0.
zf      = 1.

def solve_system(N_MAX):

    h       = (zf - z0) / N_MAX
    z_grid  = np.linspace(z0, zf-h, N_MAX + 1)
    dt      = h * COURANT

    U  = np.zeros((N_MAX+1,4))
    t  = 0.

    while t <= T_FINAL:

        w = 2. * np.pi * (t - z_grid)
        U[:,0] = np.sin(w)      # B_x
        U[:,1] = np.cos(w)      # B_y
        U[:,2] = np.cos(w)      # E_x
        U[:,3] = -np.sin(w)     # E_y

        U = rk4_step(U, maxwell_rhs, dt, h, N = N_MAX)
        t += dt

    return z_grid, U

```

```python
'''
    Exact Solutions (Eqs (10))
'''

def magnetic_field_exact(t,z):
    w  = 2. * np.pi * (t-z)
    Bx = np.sin(w)
    By = np.cos(w)
    B  = np.array((Bx, By))
    return B

def electric_field_exact(t,z):
    w  = 2. * np.pi * (t-z)
    Ex = np.cos(w)
    Ey = - np.sin(w)
    E  = np.array((Ex, Ey))
    return E

'''
    Numerical Solutions (N_MAX = 100)
'''
z_grid = solve_system(100)[0]
U      = solve_system(100)[1]

B_x = U[:,0]
B_y = U[:,1]
E_x = U[:,2]
E_y = U[:,3]

magnetic_field = np.array((B_x, B_y))
electric_field = np.array((E_x, E_y))


'''
    Compare Solutions (E_x)
'''

plt.plot(z_grid, electric_field_exact(T_FINAL,z_grid)[0], 'ro-', label = r'$E_x$ (Exact)')   # E_x
plt.plot(z_grid, electric_field[0], 'y-', label = r'$E_x$ (Numerical)')      # E_x  (numerical)

plt.xlabel(r'$z$')
plt.ylabel(r'$E_x(T_{\rm{FINAL}}, z)$')
plt.legend(fancybox=True, framealpha=1, borderpad=1, shadow=True)

plt.show()
plt.close()
```

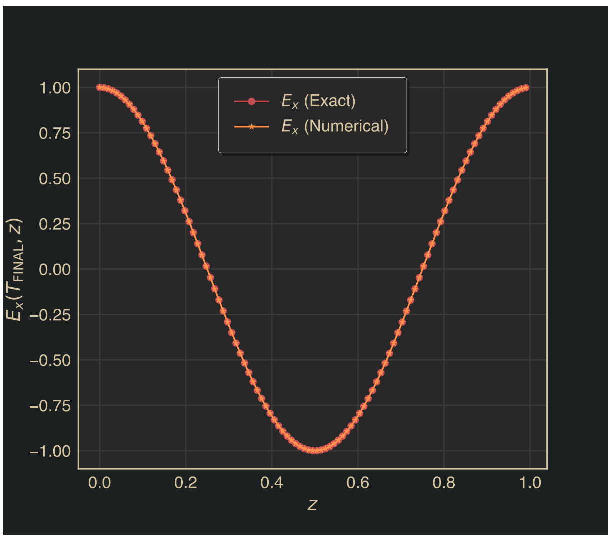


Figure 2: Numerical and closed-form solution of the x-component of the electric field at $t_{final} = 10$.

We get the same excellent match for the remaining fields as well:

```
'''
    Compare Solutions (All Fields)
'''

fig, ((Bx, By), (Ex, Ey)) = plt.subplots(2,2, sharex=True, sharey=True)   #sharey=True

Bx.plot(z_grid, magnetic_field_exact(T_FINAL,z_grid)[0], 'ro-')   # B_x
Bx.plot(z_grid, magnetic_field[0], 'y-')      # B_x (numerical)

By.plot(z_grid, magnetic_field_exact(T_FINAL,z_grid)[1], 'ro-')   # B_y
By.plot(z_grid, magnetic_field[1], 'y-')      # B_y (numerical)

Ex.plot(z_grid, electric_field_exact(T_FINAL,z_grid)[0], 'ro-')   # E_x
Ex.plot(z_grid, electric_field[0], 'y-')      # E_x (numerical)

Ey.plot(z_grid, electric_field_exact(T_FINAL,z_grid)[1], 'ro-')   # E_y
Ey.plot(z_grid, electric_field[1], 'y-')      # E_y (numerical)

Bx.set(ylabel=r'$B_x(T_{\rm{FINAL}}, z)$')
By.set(ylabel=r'$B_y(T_{\rm{FINAL}}, z)$')
Ex.set(ylabel=r'$E_x(T_{\rm{FINAL}}, z)$')
Ey.set(ylabel=r'$E_y(T_{\rm{FINAL}}, z)$')

Ex.set(xlabel=r'$z$')
Ey.set(xlabel=r'$z$')

plt.show()
plt.close()
```

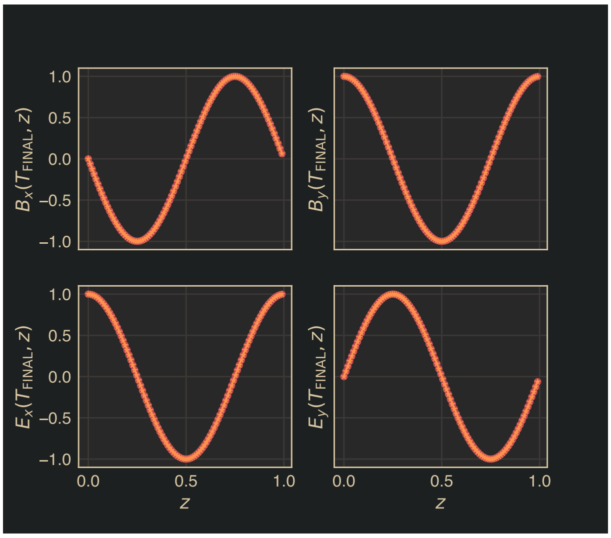


Figure 3: Numerical and closed-form solutions of both electric and magnetic fields at $t_{\rm final} = 10$.

An animation of the evolution of the x-compoent of the magnetic field (up to $t_{\rm final} = 1$) has been uploaded as an .mp4 file. We conclude this exercise by measuring the L^∞-norm of the error between the exact and numerical solutions as a function of the number of cells:

```
'''
    N-test
'''

N_list  = [50, 100, 150, 200, 250, 300]
Bx_test = []
By_test = []
Ex_test = []
Ey_test = []

for N in N_list:

```

```
    sol    = solve_system(N)[1]
    z_grid = solve_system(N)[0]

    B_x = sol[:,0]
    B_y = sol[:,1]
    E_x = sol[:,2]
    E_y = sol[:,3]

    Bx_exact = magnetic_field_exact(T_FINAL,z_grid)[0]
    By_exact = magnetic_field_exact(T_FINAL,z_grid)[1]
    Ex_exact = electric_field_exact(T_FINAL,z_grid)[0]
    Ey_exact = electric_field_exact(T_FINAL,z_grid)[1]

    Bx_error = np.linalg.norm(Bx_exact - B_x, ord = np.inf)
    By_error = np.linalg.norm(By_exact - B_y, ord = np.inf)
    Ex_error = np.linalg.norm(Ex_exact - E_x, ord = np.inf)
    Ey_error = np.linalg.norm(Ey_exact - E_y, ord = np.inf)

    Bx_test.append(Bx_error)
    By_test.append(By_error)
    Ex_test.append(Ex_error)
    Ey_test.append(Ey_error)

'''
    Plot results from N-test
'''

plt.plot(N_list, Bx_test, 'r-', label = r'$B_x$')
plt.plot(N_list, By_test, 'yo-', label = r'$B_y$')
plt.plot(N_list, Ex_test, 'b-', label = r'$E_x$')
plt.plot(N_list, Ey_test, 'go-', label = r'$E_y$')

plt.xlabel(r'$N_{\rm max}$')
plt.ylabel(r'$L^{\infty}$-error')
plt.legend(fancybox=True, framealpha=1, borderpad=1, shadow=True)

plt.show()
plt.close()
```

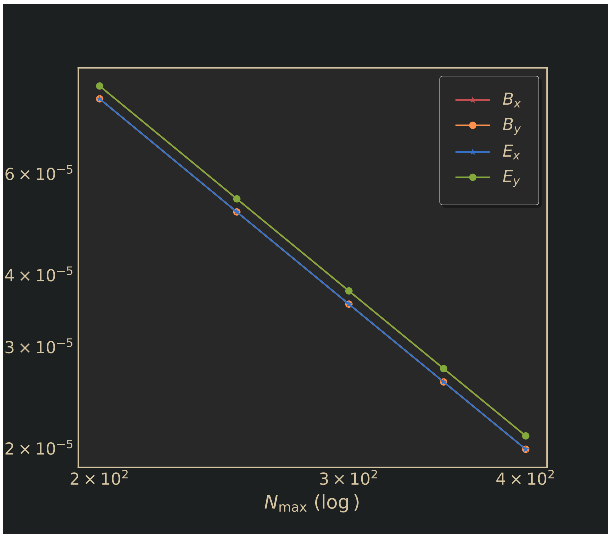


Figure 4: L^{∞}-norm of the error between the exact and numerical solutions for $N_{\max} \in \{200, 250, 300, 350, 400\}$, plotted in a log-log scale.

♠