PHY407 Lab 4

Monday, November 18, 2024

Jace Alloway - 1006940802 - alloway1

This assignment was compiled in VS Code using the Python and LaTeX extensions. Matplotlib does not produce inline plots in terminal, so the `%matplotlib inline` command was commented out. **Collaborators for all questions: none.**

Problem 1

(a) To create the contour plot, one must solve the elliptical Laplace equation for a potential φ:

$$\nabla^2 \varphi = 0. \tag{1.1}$$

This was done by implementing the Gauss-Seidel method:

$$\varphi(x,y) \longleftarrow \frac{1+\omega}{4}\left[\varphi(x+a,y)+\varphi(x-a,y)+\varphi(x,y+a)+\varphi(x,y-a)\right]-\omega\varphi(x,y) \tag{1.2}$$

In part (a), we will investigate the absence of relaxation ($\omega = 0$), while over-relaxation will be examined in part (b). The initial conditions of the problem were set up as follows:

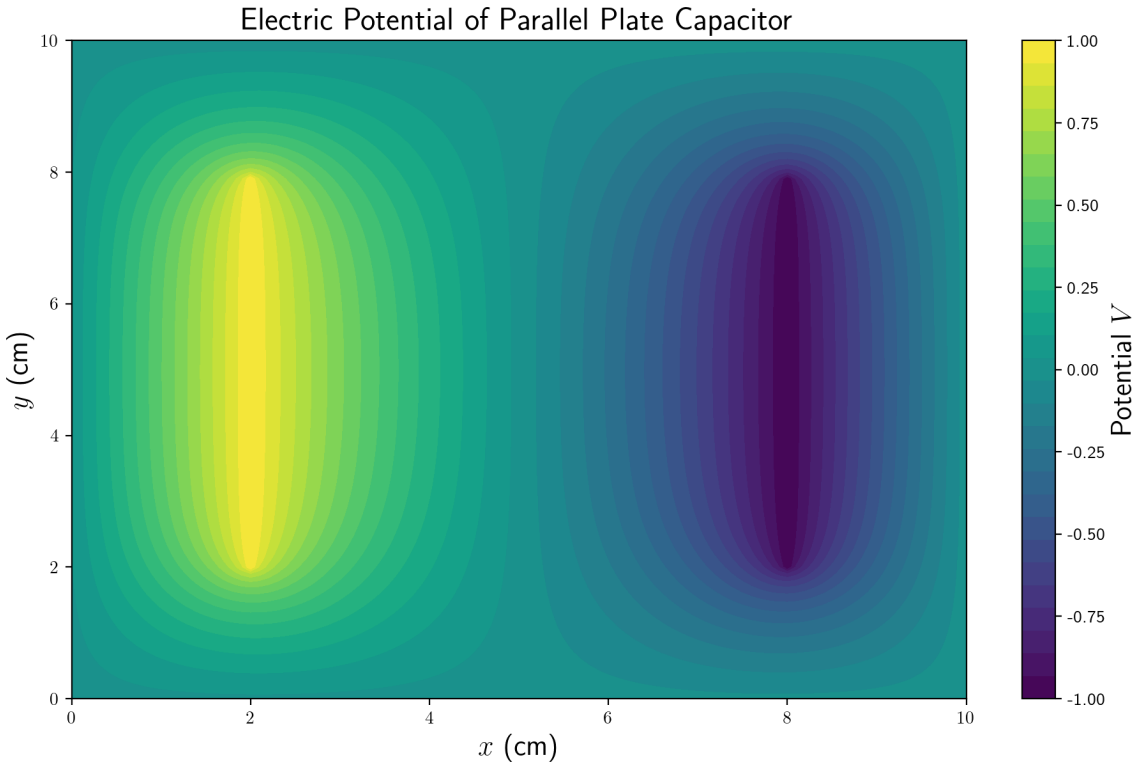
```
M = 100 #number of gridpoints in x and y
width = 10 #in cm
x = np.linspace(0, width, M+1)
X, Y = np.meshgrid(x, x)    #create the grid
a= x[1] - x[0] #should be 0.1
phi = np.zeros((M+1, M+1), dtype = float) #empty phi array

        #when x = 2cm and 8cm, find these indices
v1idx = int(2/a)
v2idx = int(8/a)

    #by symmetry, set the corresponding values to 1 / -1
phi[v1idx:v2idx, v1idx] = 1
phi[v1idx:v2idx, v2idx] = -1
```

where the last 2 lines were written so that between the y values of 2cm and 8cm at the separate x values of 2cm and 4cm, the potential has conditions $V = +1$ and $V = -1$, respectively, to account for the plate capacitors. A copy of the array was made to account for any previous changes in the loop, using `phinew = np.copy(phi)`.
The PDE was solved by iterating over x and y indices, as long as the maximum difference over cells between `phi` and `phinew` was greater than the target 10^{-6}. If the indices were at the edges, the values of φ were set to 0 for boundary conditions.

```
    #set accuracy
target = 10e-6
    #init grid of error tolerances for each cell
delta = np.ones((M, M), float)
```

```
it = 0
    #set max number of iterations
maxit = 1500

    #while the max tolerance over all cells is greater than target, do the loop
while np.max(delta) > target:
        #if #iterations > max, stop (did not converge )
    if it > maxit:
        print(f"Solution did not converge within maximum number of iterations.")
        break

        #copy new array to old array, repeate
    np.copyto(phi, phinew)

    #iterate over indices
    for k in range(M):
        for l in range(M):
                #boundary conditions (0)
            if k == 0 or k == 100 or l == 0 or l == 100:
                phinew[k,l] = phi[k,l]

                #capacitor conditions (+1, -1)
            if (k in range(v1idx, v2idx)) and (l == v1idx or l == v2idx):
                phinew[k,l] = phi[k,l]

                #determine new values using GS
            else:
                phinew[k, l] = (phi[k+1, l] + phinew[k-1, l] + phi[k, l+1]    \
                                    + phinew[k, l-1])/4

                #if the single cell tolerance is greater than target,
                #compute new tolerance
            if delta[k, l] > target:
                delta[k,l] = np.abs(phinew[k, l] - phi[k, l])

                #else, return to next step of loop
            elif delta[k, l] <= target:
                continue

    it += 1     #up the iterator

    #print num iterations
print('Regular G-S iterations:', it - 1)
```

The resulting plot produced was

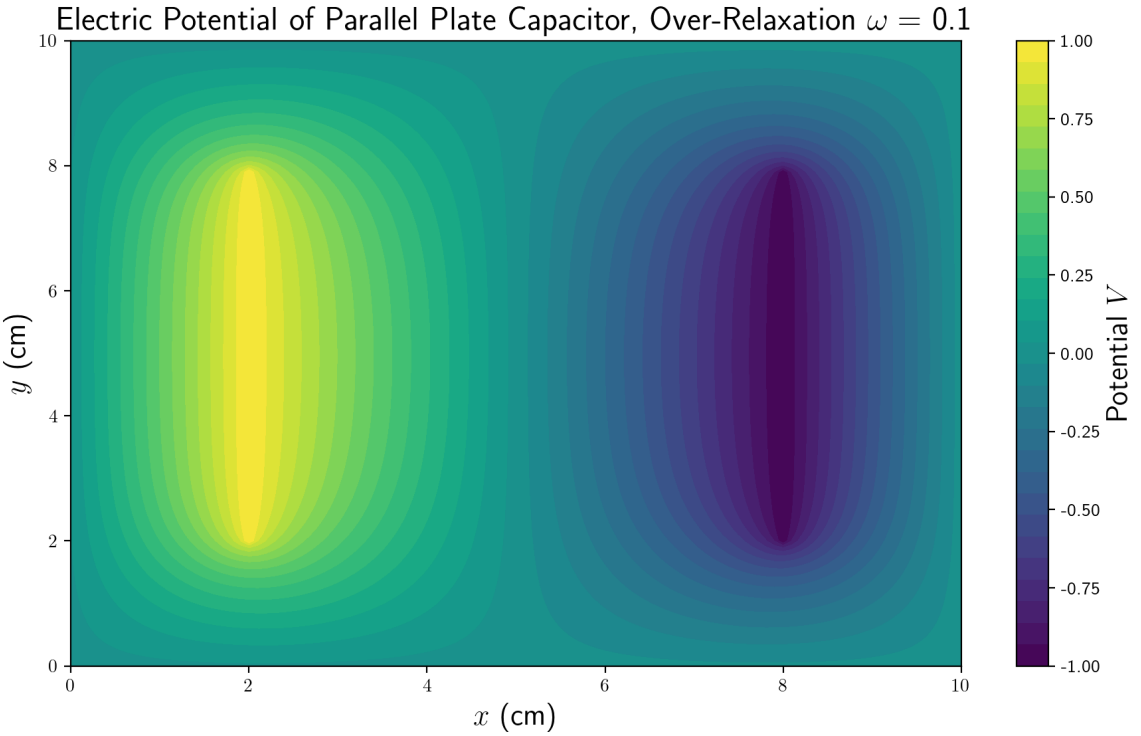


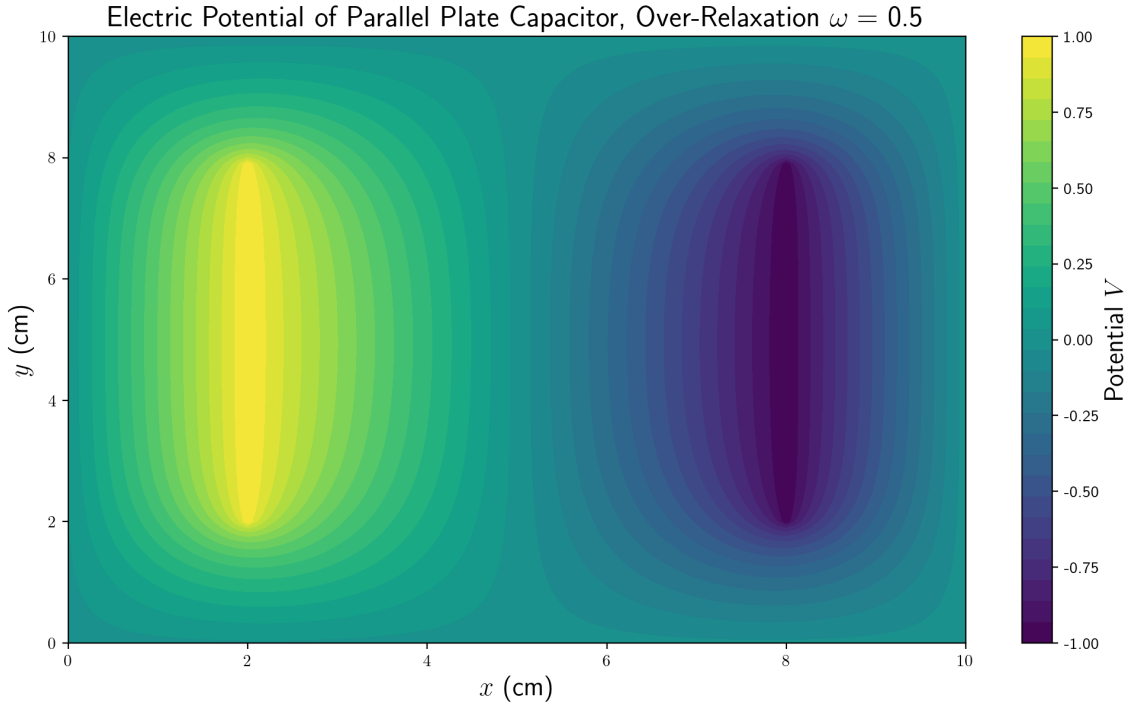
with 943 iterations.

(**b**) In part (b), the same process was written except to include the over-relaxation parameter ω in the `phinew[k, l]` computation not at the boundaries:

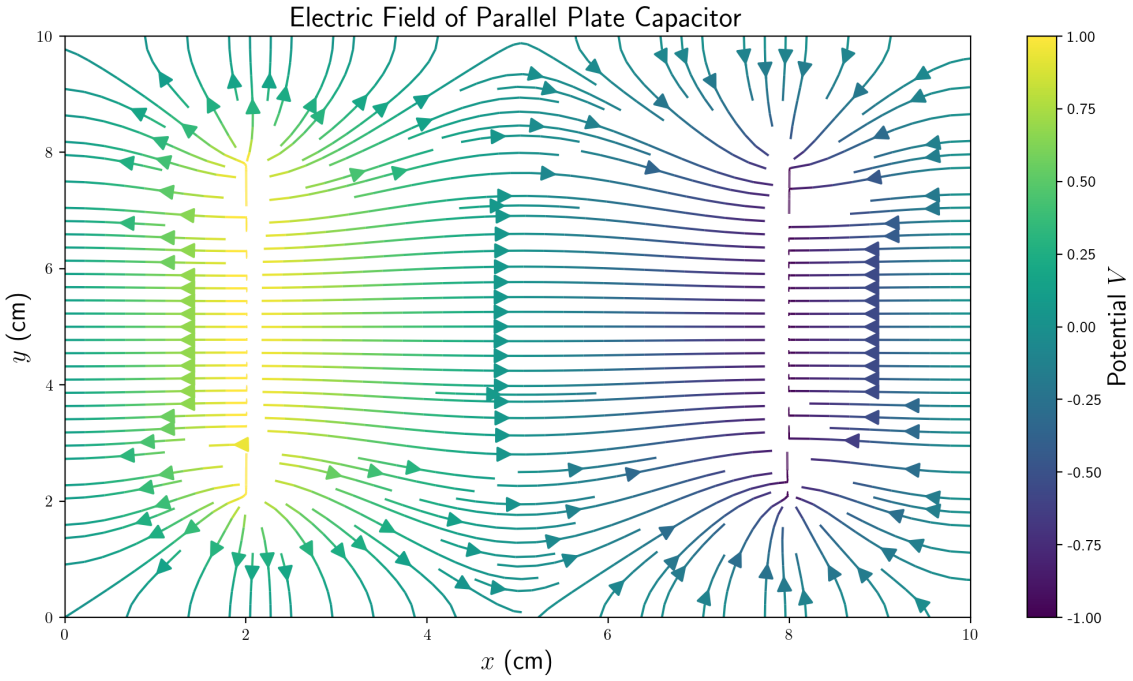
```
else:
    phinew[k, l] = ((1+omega)*(phi[k+1, l] + phinew[k-1, l]  \
                   + phi[k, l+1] + phinew[k, l-1])/4) - (omega*phi[k, l])
```

The plots were nearly identical to that of part (a).
However, the most crucial observation was that over-relaxation reduces the number of iterations each computation takes, since each grid cell is essentially being 'overshot' by the correct value. It was found that the number of iterations for $\omega = 0.1$ was around 786, while for $\omega = 0.5$ it was 367. This is because nearby values are averaged (hence the $-\omega\varphi(x, y)$ term), accelerating convergence.





(c) Since $\mathbf{E}(x, y) = -\nabla\varphi(x, y)$ in electrostatics, `np.gradient(phinew)` was used to find the gradient of the computed potential. The E_x and E_y components were extracted and plotted over the meshgrid specified in part (a) as a streamplot. Since one may create color gradients of streamlines using scalar fields (ie, $\varphi(x, y)$), the streamlines were color-coded in accordance to the value of the electric potential. Again, a colorbar was added and a plot was created.


Electric Field of Parallel Plate Capacitor
y (cm)
x (cm)
Potential V
0
2
4
6
8
10
1.00
0.75
0.50
0.25
0.00
-0.25
-0.50
-0.75
-1.00

Problem 2

(**a**) We begin with the relations of surface fluid flow and the flux-conservative class relation

$$\frac{\partial \mathbf{w}}{\partial t} = -\frac{\partial \mathbf{F}(\mathbf{w})}{\partial x} \tag{2.1}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = -g\frac{\partial \eta}{\partial x} \tag{2.2}$$

$$\frac{\partial \eta}{\partial t} + \frac{\partial (uh)}{\partial x} = 0 \tag{2.3}$$

If we let $\mathbf{w} = (u, \eta)$ to be a 2-component vector, with $w^0 = u$ and $w^1 = \eta$, then we may determine the components of $\mathbf{F}$ in accordance to equation (2.1). We arrange (2.2) as

$$\frac{\partial u}{\partial t} = -\left[u\frac{\partial u}{\partial x} + g\frac{\partial \eta}{\partial x}\right] \tag{2.4}$$

and noting that, by the chain rule, $u\dfrac{\partial u}{\partial x} = \dfrac{1}{2}\dfrac{\partial (u^2)}{\partial x}$, we find that

$$\frac{\partial u}{\partial t} = -\frac{\partial}{\partial x}\left[\frac{1}{2}u^2 + g\eta\right] \tag{2.5}$$

$$\implies \frac{\partial w^0}{\partial t} = -\frac{\partial F^0(u,\eta)}{\partial x} \tag{2.6}$$

hence $F^0(u, \eta) = \dfrac{1}{2}u^2 + g\eta$. In a similar fashion, from (2.3),

$$\frac{\partial \eta}{\partial t} = -\frac{\partial (uh)}{\partial x} \tag{2.7}$$

$$= -\frac{\partial}{\partial x}[(\eta - \eta_b)u] \tag{2.8}$$

$$\implies \frac{\partial w^1}{\partial t} = -\frac{\partial F^1(u,\eta)}{\partial x} \tag{2.9}$$

with $F^1(u, \eta) = (\eta - \eta_b)u$. Therefore we can write (2.2) and (2.3) in a flux-conservative form with

$$\mathbf{F}(u, \eta) = \left[\frac{1}{2}u^2 + g\eta,\ (\eta - \eta_b)u\right]. \tag{2.10}$$

(**b**) We implement (2.10) in python by discretizing the equations for u and η. Letting u^j and η^j be arrays of length 50 (since $L = [0, 1]$ and $dx = 0.02$m), we timestep forward using an iterable counter in a while loop.

[If j = 0 or j = J-1 (boundaries):]

$$u^j_{\text{new}} = u_{0t} - (dt/dx)(0.5{u^{j+1}}^2 + g\eta^{j+1} - 0.5{u^j}^2 - g\eta^j) \qquad [j = 0] \tag{2.11}$$

$$u^j_{\text{new}} = u_{Lt} - (dt/dx)(0.5{u^j}^2 + g\eta^j - 0.5{u^{j-1}}^2 - g\eta^{j-1}) \qquad [j = J-1] \tag{2.12}$$

[Else (interior):]

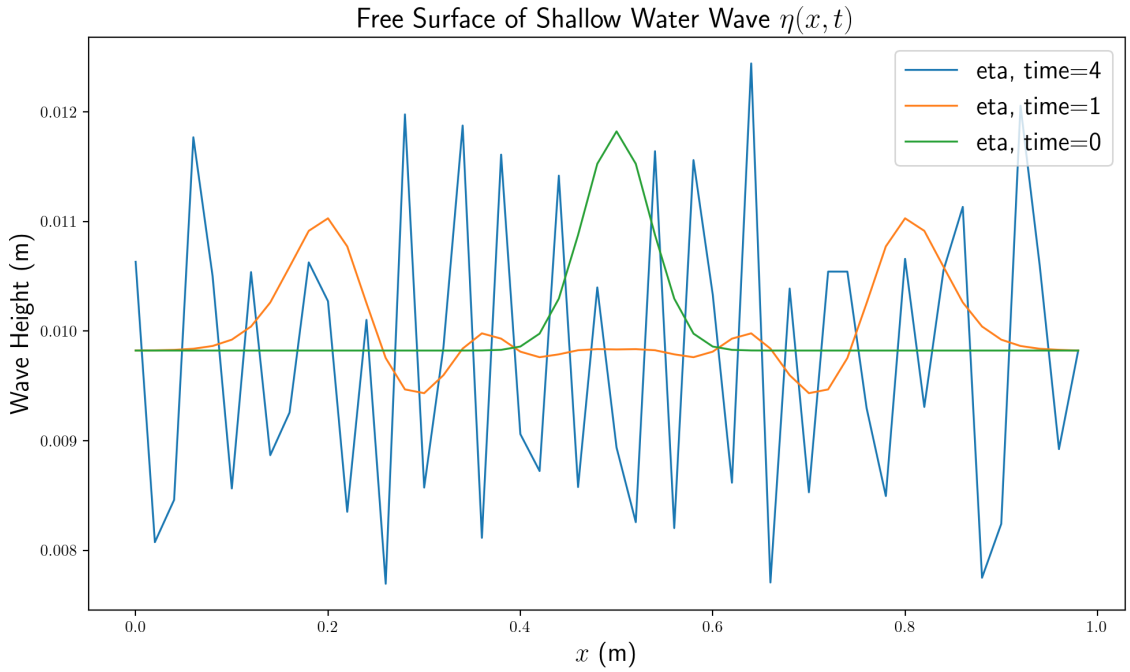
$$u^j_{\text{new}} = u^j - \beta(0.5{u^{j+1}}^2 + g\eta^{j+1} - 0.5{u^{j-1}}^2 - g\eta^{j-1}) \tag{2.13}$$

$$\eta^j_{\text{new}} = \eta^j - \beta((\eta^{j+1} - \eta_b^{j+1})u^{j+1} - (\eta^{j-1} - \eta_b^{j-1})u^{j-1}) \tag{2.14}$$

where $\beta = \dfrac{dt}{2dx}$. This formula was derived as in the lab notebook. Here, u_{0t} and u_{Lt} are the edge boundary conditions on u, which are zero. We define η_{new} and u_{new} as the currently evaluated arrays, which are then copied to u and η with `np.copy()`. At the end of each iteration, the counter was increased. The loop stopped once $n = N + 1$, where N was the total number of specified timesteps. With the initial η^j at $t = 0$ (that is, the $n = 0$ iterator input) condition being

$$\eta(x, 0) = H + Ae^{-(x-\mu)^2}/\sigma^2 - \langle Ae^{-(x-\mu)^2}/\sigma^2 \rangle \tag{2.15}$$

with the conditions of H, A, μ, and σ being as specified in the lab notebook (I just defined (2.15) as a seperate function in the code), the while loop was ran as a function call and the resulting u and η arrays were plotted. Since the function did not keep track of each timestep, each timestep needed to be evaluated with a seperate function call. This was just done in a for loop.



Note that the solution is stable for $t = 0$ (the initial condition) and $t = 1$, but diverges into mess at $t = 4$.

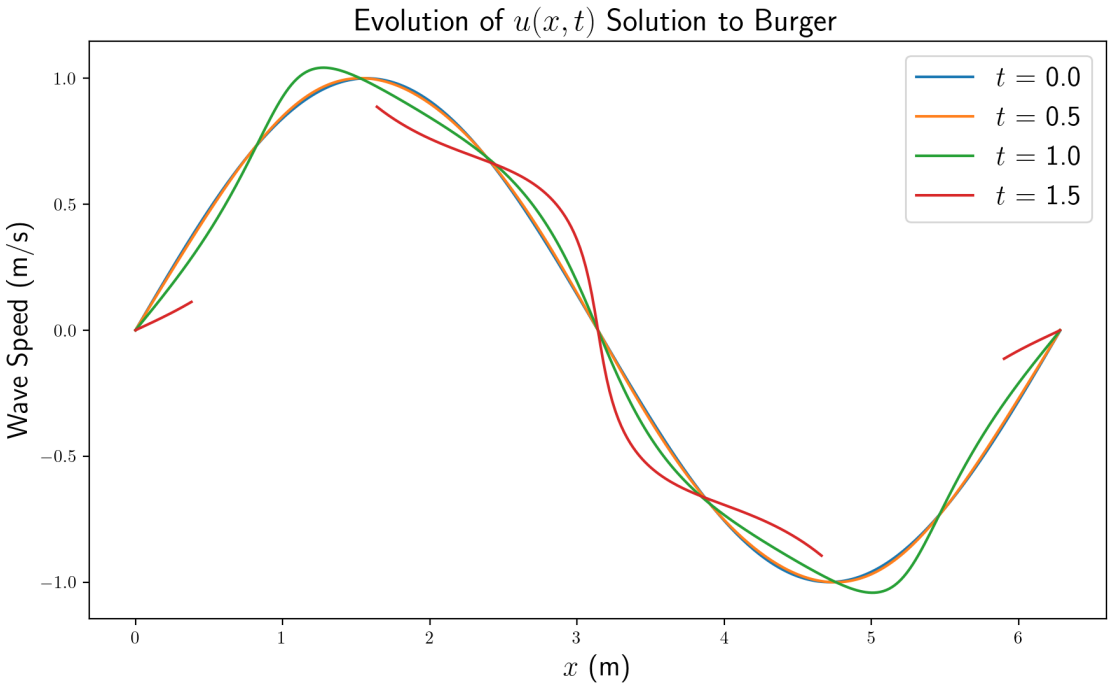
Problem 3

(a) This problem follows very similarly from that of 2 (b), however we now write our code as multi-dimensional arrays to keep track of the spatial and temporal steps. We write the function `burger_solve` to take in dx, dt, ϵ values, as well as the time / space lengths and initial conditions specified on the boundary. That is, $u(t,0) = u(t,L_x) = 0$. $u(0,x)$ was written to be a sine wave across the domain $[0, 2\pi]$ at $dx = 0.02$ spacing (see part (b)).
u was defined to be a $J \times N$ zero-array (space $\times$ time) and the $t = 0$ condition was implemented via $u[:,0] = u_{x0}$. Each component was then iterated over:

```
for n in range(1, N-1):
    for j in range(0, J):
        if j == 0:
            u[n, j] = u_0t
        if j == J-1:
            u[n, j] = u_Lt
        else:
            u[n+1, j] = u[n-1, j] - (beta/2)*((u[n, j+1]**2) - (u[n, j-1]**2))
```

where the first two conditions implement the boundary conditions, and the last equation is responsible for the forward timestep. The n loop must go before the j loop, since we must evaluate each j prior to stepping forward in time. Once this was complete, the whole u array was returned.

(b) The code can be ran by calling the function. It was important to make sure the input array u_{x0} was the same length as the x array, else the array broadcasting would have different shapes. Once the function was ran and the array was returned, each timestep could be accessed by taking the slice $u[k,:]$ where `k = int(t/dt)` is the index of the time t. This was again implemented into a for loop and the results were plotted.



As time progressed, we observe higher sine modes being added, and a steepening of the wave, as expected. This solution is realistic, however we observe again an instability as the solution progressed. It was found that (by mistake, actually) that larger step sizes in x preserve the shape of the wave, and it was assumed that this was because smaller step sizes dx can allow the solution to diverge too quickly since β is smaller. The plot for $dx = 0.02$ is shown above, but I also tried $dx = 0.1$: