Question 3:

Refer to the previous pages to see the discretization of the equations for question 3.

1. Using a forward discretization (Euler explicit) for the time derivatives, and central discretization for the first and second order spatial derivatives, evaluate the solution at t = 1.0 using $\Delta t = 5 \times 10^{-3}$.

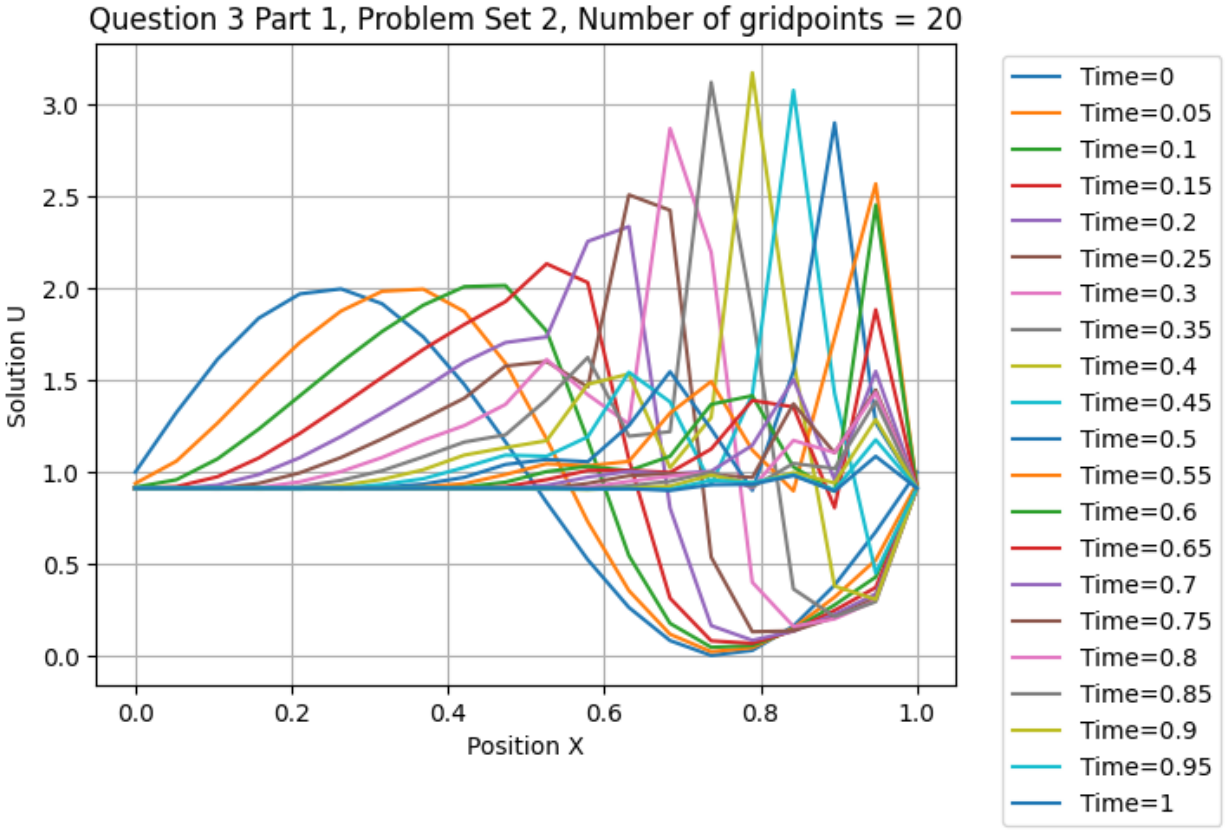

Figure 1: Burgers Equation Discretization using 20 grid points

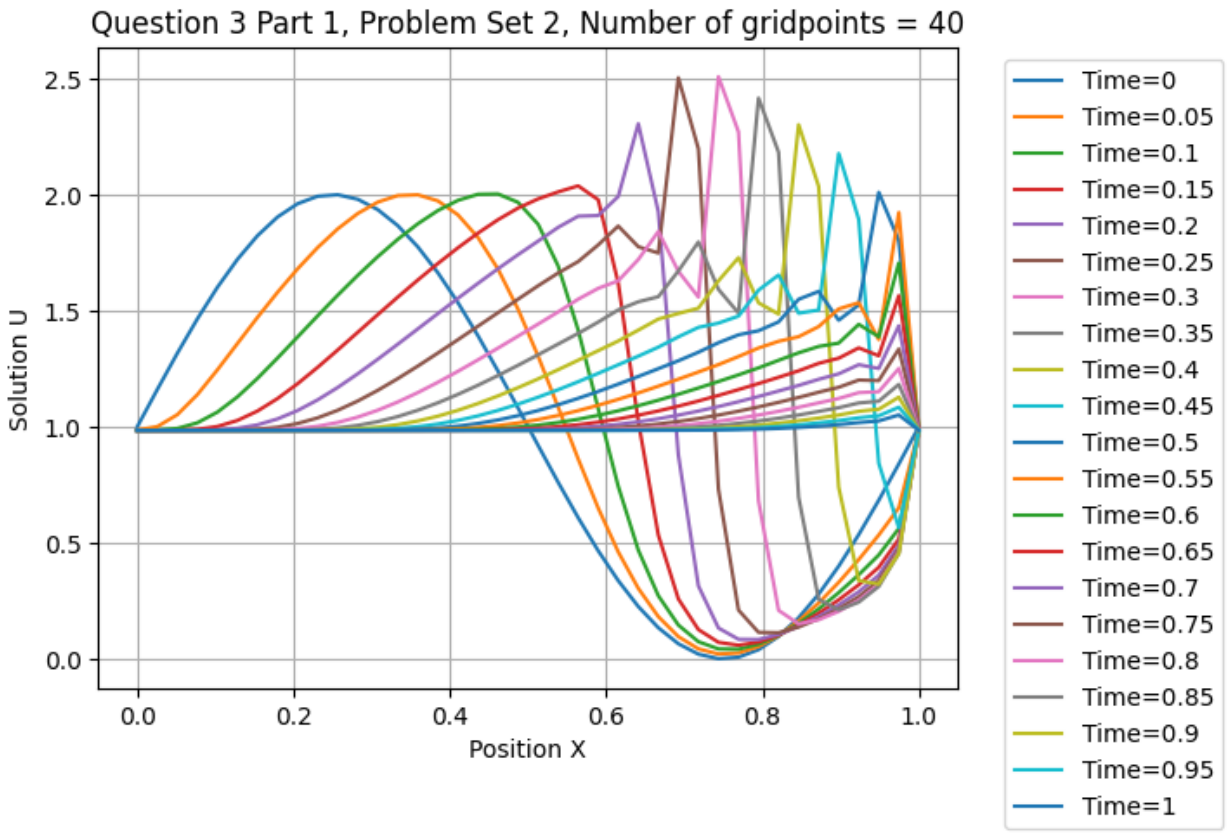

Figure 2: Burgers Equation Discretization using 40 grid points

The code will be included below. In order to calculate the above graph, the periodic boundary condition must be applied. In order to solve the initial point to apply this boundary condition, there are 2 methods. The first would be using the forward differencing method for just the first point (as there are no points before the first point). The second method would be using the last point as a "ghost point", in order to subtract the previous point using central difference method. The second method was chosen in this case. The snippet of code shown below demonstrates how this works.

```
#  Solve initial point for periodic boundary condition
u[0, n + 1] = ((dt * nu) / (dx ** 2)) * (u[1, n] - 2 * u[0, n] + u[-1, n]) -
(dt / (2 * dx)) * u[0, n] * (u[1, n] - u[-1, n]) + u[0, n]
u[-1, n + 1] = u[0, n + 1]
```

2. For each of the grids, conduct numerical experiments to determine the maximum value of Δt for which the solution remains stable. Present your results in Tabular form.

The code was written in the form of a function to allow quick looping through a variety of timesteps. This can be seen below.

```
numx = [20, 40]  # Number of discretization points
dom_len = 1.0  # Domain size
dt = np.arange(5e-3, 4e-2 + 5e-3, 1e-3)
tfinal = 1.0
tinitial = 0.0
for t in dt:
```

```
    for num in numx:
        print(t, num)
        burgers(num, dom_len, tfinal, tinitial, t)
```

This allowed a variety of timesteps between 5e-3 and 4.5e-2 to be tested. Using this method, an overflow in double scalars was encountered for both 40 points and 20 points. A runtime in double scalars means that a number was encountered which is larger than a double scalar can contain (a double scalar contains enough memory to encapsulate values between -1.79769313486e+308 and 1.79769313486e+308). This indicates that at these timesteps the solution blew up to infinity, as the timesteps were too large.

Number of Points	Timestep at which failure occurred
20	0.031
40	0.014

Below the graphs that were created 1 timestep before failure occurred can be seen, and below the subsequent graphs for the smallest timestep used.

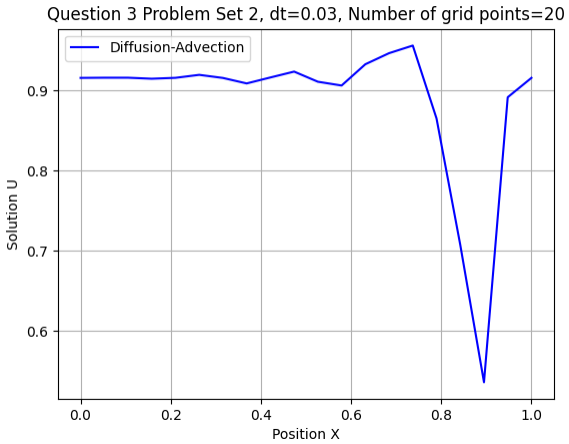

Figure 3: Final timestep before failure occurred with 20 grid points

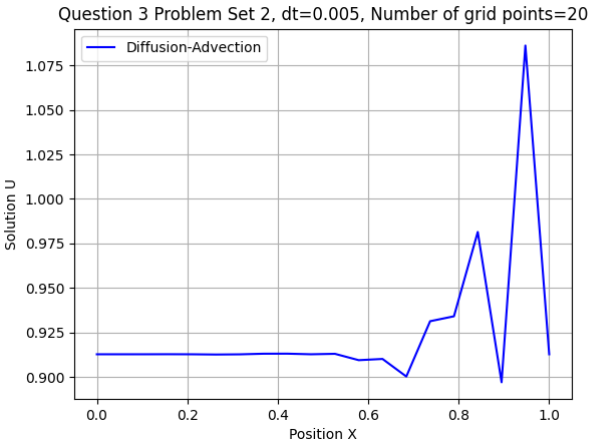

Figure 4: Smallest timestep tested with 20 grid points

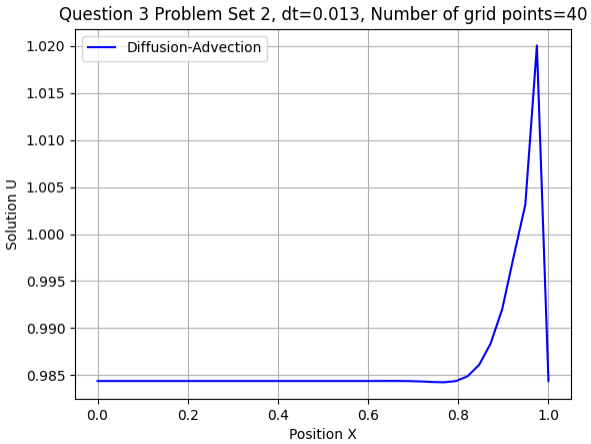

Figure 5: Final timestep before failure occurred with 40 grid points

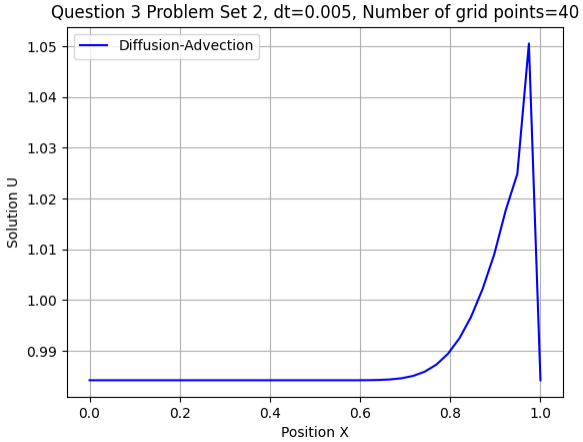

Figure 6: Smallest timestep tested with 40 grid points

Code:

```
# Import packages
import numpy as np
import matplotlib.pyplot as plt
import os
# User defined packages

# Path to save files
current_path = os.getcwd()
plot_folder = current_path + '/plots'

def burgers(numx, dom_len, tfinal, tinitial, dt):
    dx = dom_len / (numx - 1)  # Spatial step size
    x = np.arange(0, dom_len + dx, dx)  # Position vector
    t = np.arange(tinitial, tfinal + dt, dt)  # tfinal + dt is needed to
include tfinal in interval
    u = np.zeros((numx, len(t)))  # Solution vector
    nu = 0.01
    # Initial conditions
```

```
    for j in range(numx):
        u[j, 0] = 1 + np.sin(np.pi * 2 * dx * j)  # Apply initial conditions
    for n in range(len(t) - 1):  # Time loop
        #  Solve initial point for periodic boundary condition
        u[0, n + 1] = ((dt * nu) / (dx ** 2)) * (u[1, n] - 2 * u[0, n] + u[-
1, n]) - (dt / (2 * dx)) * u[0, n] * (u[1, n] - u[-1, n]) + u[0, n]
        u[-1, n + 1] = u[0, n + 1]

        for j in range(1, numx - 1):  # Spatial loop
            #  Solve solution vector
            u[j, n + 1] = ((dt * nu) / (dx ** 2)) * (u[j+1, n] - 2 * u[j, n]
+ u[j - 1, n]) - (dt / (2 * dx)) * u[j, n] * (u[j + 1, n] - u[j - 1, n]) +
u[j, n]
    # For question 1
    # for i in range(len(u[0, :])):
    #     print(i)
    #     if i % 10 == 0:
    #         print("Yes")
    #         plt.plot(x, u[:, i], label='Time=%g' % t[i])
    # plt.grid()
    # plt.legend(loc='upper left', bbox_to_anchor=(1.04, 1))
    # plt.title("Question 3 Part 1, Problem Set 2, Number of gridpoints = %i"
% numx)
    # plt.xlabel('Position X')
    # plt.ylabel("Solution U")
    # save_folder = plot_folder + '_Part 1_Num_X=%i' % numx
    # if not os.path.exists(save_folder):
    #     os.makedirs(save_folder, exist_ok=True)
    # plt.savefig(save_folder + '/Numx=%i' % numx + '.png',
bbox_inches="tight")
    # plt.close()
    # For question 2
    plt.plot(x, u[:, -1], label='Diffusion-Advection', color='b')
    plt.grid()
    plt.legend(loc='best')
    plt.title("Question 3 Problem Set 2, dt=%g, Number of grid points=%i" %
(dt, numx))
    plt.xlabel("Position X")
    plt.ylabel("Solution U")
    save_folder = plot_folder + '_Part 2_Num_X=%i' % numx
    if not os.path.exists(save_folder):
        os.makedirs(save_folder, exist_ok=True)
    plt.savefig(save_folder + '/dt=%g_Numx=%i' % (dt, numx) + '.png')
    # plt.show()
    plt.close()
    # return x, u


numx = [20, 40]  # Number of discretization points
dom_len = 1.0  # Domain size
dt = np.arange(5e-3, 4e-2 + 5e-3, 1e-3)
tfinal = 1.0
tinitial = 0.0
for t in dt:
    for num in numx:
        print(t, num)
        burgers(num, dom_len, tfinal, tinitial, t)
```