AST 5900: Problem Set 7

Gilberto Garcia

May 4, 2024

Problem 1a.

Answer 1a. Attached is the code used for implementing the first order upstream finite difference method. Using the code we create the plot in figure 1.

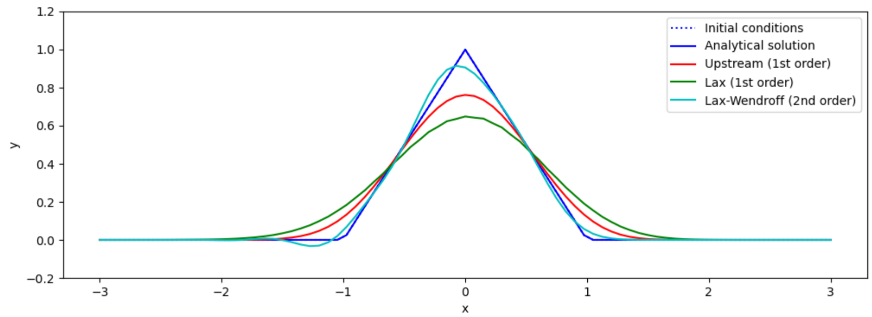


Figure 1: Solving the wave equation using different methods for a triangle wave.

Problem 1b.

Answer 1b. We try different sets of nx and nt values until we arrive at the stable-most solution. That happens when $C_{\max}$=1. So, the range for stable solutions is $0 < C < 1$. Figure 2 shows the solution when $C = 1$.

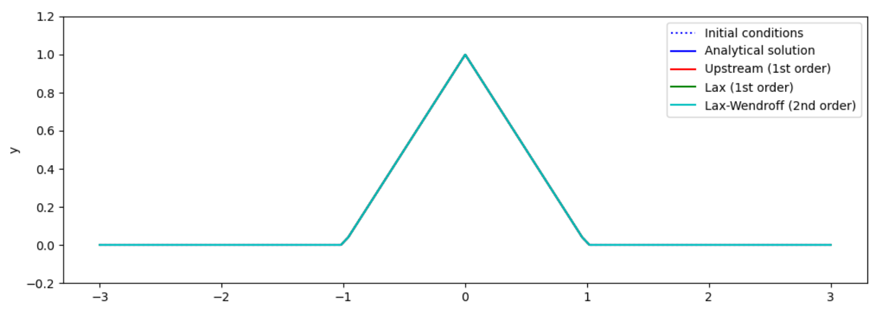


Figure 2: Numerical solutions with C = 1.

Problem 1c.

Answer 1c. The solution for $C=1$ is shown in figure 2. We also plot the solutions for $C<1$ in figure 3 and for $C>1$ in figure 4.

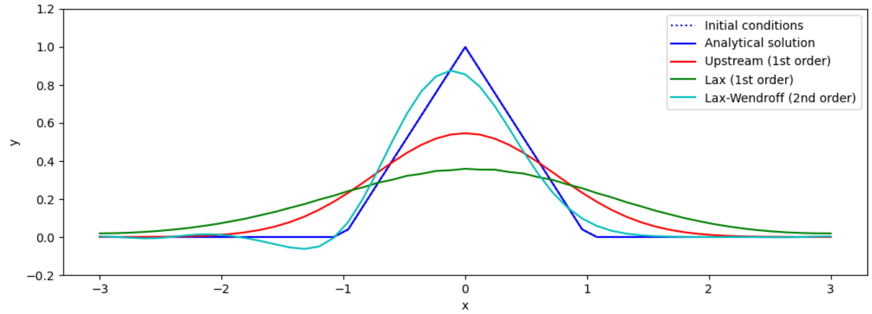


Figure 3: Numerical solutions with C = 0.5.

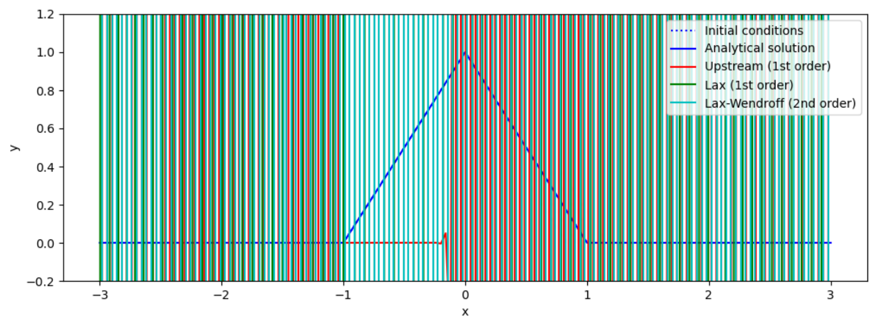


Figure 4: Numerical solutions with C = 1.5.

Problem 2a.

Answer 2a. We examine the error of the numerical solution. We vary nt while keeping nx constant. We produce the absolute error plot in figure 5. We see that reducing the C number increases the error. In order to decrease C, we increase nt values, which, in turn, decreases Δt. So reducing Δt does not improve accuracy.

Problem 2b.

Answer 2b. The modified equation that we got from lecture is

$$\frac{\partial u}{\partial t}+c\frac{\partial u}{\partial x}=(1-C)\frac{c\Delta x}{2}\frac{\partial^2 u}{\partial x^2}+\mathcal{O}(\Delta x^2) \tag{1}$$

The largest order error term is the first term on the left hand side. This term is what defines $C_{\max}=1$ because $C=1$ cancels out the term entirely. In this scenerio, we arrive at the exact solution, as we saw in our answers above. For $C>1$, we get a negative number which would be nonphysical, so our solution is unstable, as shown above. Thus, our range for C is between 0 and 1.

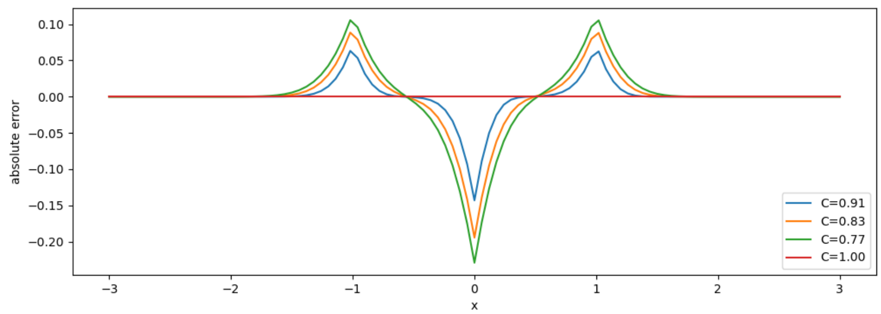


Figure 5: Error plot for various C values.

Problem 2c.

Answer 2c. We keep C and nt constant at 0.9 and 110, respectively while varying nx. We examine how this changes the maximum error in figure 6. We see that the slope is roughly -1 so the error diminishes as we'd expect for a first order method.

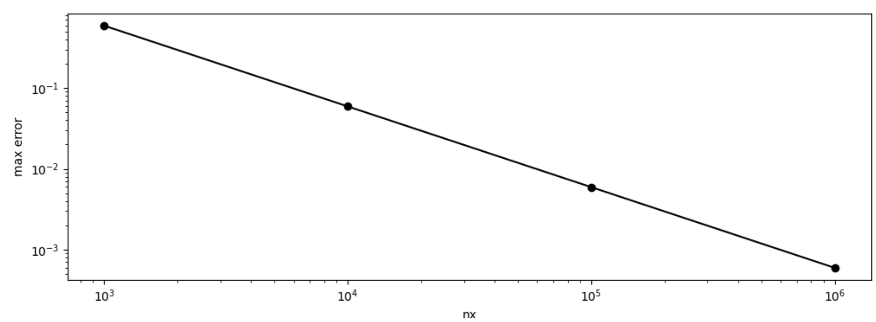


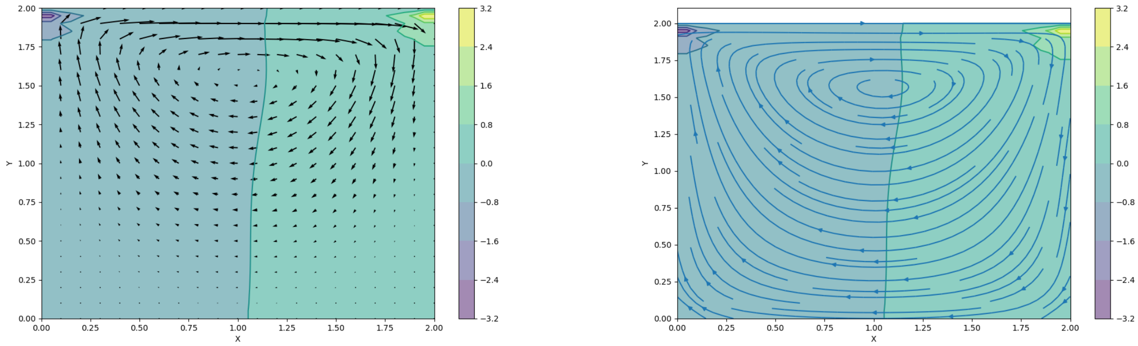
Figure 6: Error plot for various nx values, keeping nt and C constant.

Problem 3.

Answer 3. We follow the code presented here and adjust our parameters so that our Reynold's number is 20. Reynold's number is given by

$$\mathrm{Re} = \frac{uL}{\nu} \tag{2}$$

where u is flow speed, L is length, and ν is kinematic viscosity. We choose $u=1, L=2$, and $\nu=0.1$. We produce the plot shown in figure 7. The code is attached.



(a) (b)

Figure 7: Cavity flow with Re = 20 shown in a (a) vector plot and (b) quiver plot.

Code

```python
'''
Gilberto Garcia
ASTR5900
Homework #7
24 April 2024
'''

#import required libraries
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
plt.rcParams['figure.figsize'] = [12, 4]


#we update the code to match the conditions required for the hw
def wave_equation(nx, nt, c,plot=True):
    # Set spatial parameters
    xmin = -3.0
    xmax = 3.0
    dx = (xmax - xmin) / nx
    print(f"{xmin} <= x <= {xmax} (Spatial domain)")
    print(f"dx = {dx} (Spatial step size)")

    x_points = np.linspace(xmin, xmax, nx + 1)

    # Set temporal parameters
    tmin = 0.
    tmax = 3.
    dt = (tmax - tmin) / nt
    print(f"{tmin} <= x <= {tmax} (Temporal domain)")
```

```python
    print(f"dt = {dt} (Temporal step size)")

    t_points = np.linspace(tmin, tmax, nt + 1)

    # Set speed for linear wave equation
    print(f"c = {c} (Wave speed)")

    # Calculate Courant Number
    C = c * dt / dx
    #C = 0.9 #uncomment for 2C
    print(f"C = {C} (Courant Number)")

    # Define numerical solver step functions
    def upstream_step(u):
        # u_old = u.copy()
        # for x_index in range(1, nx + 1):
        #   u[x] = u_old[x] - C * (u_old[x] - o_old[x - 1])
        u[1:] = u[1:] - C * (u[1:] - u[:-1])

    def upstream_periodic_step(u):
        up = u.copy()
        u[1:] = up[1:] - C * (up[1:] - up[:-1])
        # Periodic boundary condition at x = xmin
        u[0] = up[0] - C * (up[0] - up[-2])
        # Periodic boundary condition at x = xmax
        u[-1] = u[0]

    def lax_step(u):
        u[1:-1] = 0.5 * (u[2:] + u[:-2]) - 0.5 * C * (u[2:] - u[:-2])

    def lax_periodic_step(u):
        up = u.copy()
        u[1:-1] = 0.5 * (up[2:] + up[:-2]) - 0.5 * C * (up[2:] - up[:-2])
        # Periodic boundary condition at x = xmin
        u[0] = 0.5 * (up[1] + up[-2]) - 0.5 * C * (up[1] - up[-2])
        # Periodic boundary condition at x = xmax
        u[-1] = u[0]

    def lax_wendroff_step(u):
        u[1:-1] = u[1:-1] -
            0.5 * C * (u[2:] - u[:-2]) + 0.5 * C ** 2 * (u[2:] - 2 * u[1:-1] + u[:-2])

    def lax_wendroff_periodic_step(u):
        # Periodic boundary condition by wrapping at both ends
        up = np.concatenate(( [u[-2]], u, [u[1]] ))
        u[:] = up[1:-1] - 0.5 *
             C * (up[2:] - up[:-2]) + 0.5 * C ** 2 * (up[2:] - 2 * up[1:-1] + up[:-2])
        return 0

    def triangle(x,t=0):
```

```
        if -1.0 + c * t <= x <= 1.0 + c * t:
            return 1.0 - abs(x - c*t)
        else:
            return 0

    def periodic_triangle(x,t):
        x_wrapped = x
        while  x_wrapped < xmin + c * t:
            x_wrapped += (xmax - xmin)
        while x_wrapped > xmax + c * t:
            x_wrapped -= (xmax - xmin)
        return triangle(x_wrapped, t)

    # Save the initial and final analytical solutions
    u0 = np.array([periodic_triangle(x, tmin) for x in x_points])
    uf = np.array([periodic_triangle(x, tmax) for x in x_points])

  # Initialize numerical solutions
    u_upstream = u0.copy()
    u_lax = u0.copy()
    u_lax_wendroff = u0.copy()

    # Advance numerical solution by nt timesteps
    for t in t_points[1:]:
        upstream_periodic_step(u_upstream)
        lax_periodic_step(u_lax)
        lax_wendroff_periodic_step(u_lax_wendroff)

    if plot == True:
        # Plot analytical and numerical solutions
        plt.plot(x_points, u0, 'b:', label="Initial conditions")
        plt.plot(x_points, uf, 'b-', label="Analytical solution")
        plt.plot(x_points, u_upstream, 'r-', label="Upstream (1st order)")
        plt.plot(x_points, u_lax, 'g-', label="Lax (1st order)")
        plt.plot(x_points, u_lax_wendroff, 'c-', label="Lax-Wendroff (2nd order)")
        plt.ylim(-0.2, 1.2)
        plt.legend()
        plt.xlabel('x')
        plt.ylabel('y')
        plt.show()

    #calculate relative error:
    abs_err = (u_upstream - uf)
    #return x points, analytical, and 1st order upstream numerical soln
    return x_points,abs_err,C

#1a - centering the propogation waves
wave_equation(nx = 80, nt = 100, c = 2)
#1b - we play around with nx and nt to find where Cmax is not stable
wave_equation(nx = 100, nt = 100, c = 2)
```

```
#we find that c_max of 1 is the upper limit for stability.

#1c - c_max for less than, equal to and greater than Cmax
#less than
wave_equation(nx = 50, nt = 100, c = 2)
#equal to
wave_equation(nx = 100, nt = 100, c = 2)
#greater than
wave_equation(nx = 150, nt = 100, c = 2)

#2a
print()
print('2a')
xpts1,abserr1,C1 = wave_equation(nx = 100, nt = 110, c = 2,plot=False)
xpts2,abserr2,C2 = wave_equation(nx = 100, nt = 120, c = 2,plot=False)
xpts3,abserr3,C3 = wave_equation(nx = 100, nt = 130, c = 2,plot=False)
xpts4,abserr4,C4 = wave_equation(nx = 100, nt = 100, c = 2,plot=False)


plt.plot(xpts1,abserr1,label='C={0:.2f}'.format(C1))
plt.plot(xpts2,abserr2,label='C={0:.2f}'.format(C2))
plt.plot(xpts3,abserr3,label='C={0:.2f}'.format(C3))
plt.plot(xpts4,abserr4,label='C={0:.2f}'.format(C4))
plt.ylabel('absolute error')
plt.xlabel('x')
plt.legend(loc='lower right')
plt.show()

#no, reducing delta t while keeping delta x constant does not improve accuracy.
# bigger nt means smaller delta t but also bigger error so less accuracy.

#2c - we will need to hard code C =0.9 in wave_equation()
#keep C at 0.9. we vary the nx values

nx_vals = [1_000,10_000,100_000,1_000_000]
xpts,err1,c = wave_equation(nx = nx_vals[0], nt = 110, c = 2,plot=False)
xpts,err2,c = wave_equation(nx = nx_vals[1], nt = 110, c = 2,plot=False)
xpts,err3,c = wave_equation(nx = nx_vals[2], nt = 110, c = 2,plot=False)
xpts,err4,c = wave_equation(nx = nx_vals[3], nt = 110, c = 2,plot=False)

#make a list of maximum errors:
max_err = [max(err1),max(err2),max(err3),max(err4)]

#plot nx vs max errors:

plt.loglog(nx_vals,max_err,'k')
plt.scatter(nx_vals,max_err,color='k')
plt.xlabel('nx')
plt.ylabel('max error')
plt.show()
```

```python


###########
### 3 #####
###########


#take the code from
#https
    ://nbviewer.org/github/barbagroup/CFDPython/blob/master/lessons/14_Step_11.ipynb


nx = 41
ny = 41
nt = 500
nit = 50
c = 1
dx = 2 / (nx - 1)
dy = 2 / (ny - 1)
x = np.linspace(0, 2, nx)
y = np.linspace(0, 2, ny)
X, Y = np.meshgrid(x, y)

rho = 1
nu = .1
dt = .001

u = np.zeros((ny, nx))
v = np.zeros((ny, nx))
p = np.zeros((ny, nx))
b = np.zeros((ny, nx))

def build_up_b(b, rho, dt, u, v, dx, dy):

    b[1:-1, 1:-1] = (rho * (1 / dt *
                    ((u[1:-1, 2:] - u[1:-1, 0:-2]) /
                     (2 * dx) + (v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy)) -
                    ((u[1:-1, 2:] - u[1:-1, 0:-2]) / (2 * dx))**2 -
                      2 * ((u[2:, 1:-1] - u[0:-2, 1:-1]) / (2 * dy) *
                           (v[1:-1, 2:] - v[1:-1, 0:-2]) / (2 * dx))-
                          ((v[2:, 1:-1] - v[0:-2, 1:-1]) / (2 * dy))**2))

    return b


def pressure_poisson(p, dx, dy, b):
    pn = np.empty_like(p)
    pn = p.copy()

    for q in range(nit):
```

```
        pn = p.copy()
        p[1:-1, 1:-1] = (((pn[1:-1, 2:] + pn[1:-1, 0:-2]) * dy**2 +
                          (pn[2:, 1:-1] + pn[0:-2, 1:-1]) * dx**2) /
                          (2 * (dx**2 + dy**2)) -
                          dx**2 * dy**2 / (2 * (dx**2 + dy**2)) *
                          b[1:-1,1:-1])

        p[:, -1] = p[:, -2] # dp/dx = 0 at x = 2
        p[0, :] = p[1, :]  # dp/dy = 0 at y = 0
        p[:, 0] = p[:, 1]  # dp/dx = 0 at x = 0
        p[-1, :] = 0       # p = 0 at y = 2

    return p


def cavity_flow(nt, u, v, dt, dx, dy, p, rho, nu):
    un = np.empty_like(u)
    vn = np.empty_like(v)
    b = np.zeros((ny, nx))

    for n in range(nt):
        un = u.copy()
        vn = v.copy()

        b = build_up_b(b, rho, dt, u, v, dx, dy)
        p = pressure_poisson(p, dx, dy, b)

        u[1:-1, 1:-1] = (un[1:-1, 1:-1]-
                         un[1:-1, 1:-1] * dt / dx *
                        (un[1:-1, 1:-1] - un[1:-1, 0:-2]) -
                         vn[1:-1, 1:-1] * dt / dy *
                        (un[1:-1, 1:-1] - un[0:-2, 1:-1]) -
                         dt / (2 * rho * dx) * (p[1:-1, 2:] - p[1:-1, 0:-2]) +
                         nu * (dt / dx**2 *
                        (un[1:-1, 2:] - 2 * un[1:-1, 1:-1] + un[1:-1, 0:-2]) +
                         dt / dy**2 *
                        (un[2:, 1:-1] - 2 * un[1:-1, 1:-1] + un[0:-2, 1:-1])))

        v[1:-1,1:-1] = (vn[1:-1, 1:-1] -
                        un[1:-1, 1:-1] * dt / dx *
                       (vn[1:-1, 1:-1] - vn[1:-1, 0:-2]) -
                        vn[1:-1, 1:-1] * dt / dy *
                       (vn[1:-1, 1:-1] - vn[0:-2, 1:-1]) -
                        dt / (2 * rho * dy) * (p[2:, 1:-1] - p[0:-2, 1:-1]) +
                        nu * (dt / dx**2 *
                       (vn[1:-1, 2:] - 2 * vn[1:-1, 1:-1] + vn[1:-1, 0:-2]) +
                        dt / dy**2 *
                       (vn[2:, 1:-1] - 2 * vn[1:-1, 1:-1] + vn[0:-2, 1:-1])))

        u[0, :]  = 0
```

```
        u[:, 0]  = 0
        u[:, -1] = 0
        u[-1, :] = 1   # set velocity on cavity lid equal to 1
        v[0, :]  = 0
        v[-1, :] = 0
        v[:, 0]  = 0
        v[:, -1] = 0


    return u, v, p


u = np.zeros((ny, nx))
v = np.zeros((ny, nx))
p = np.zeros((ny, nx))
b = np.zeros((ny, nx))
nt = 700
u, v, p = cavity_flow(nt, u, v, dt, dx, dy, p, rho, nu)



fig = plt.figure(figsize=(11,7), dpi=100)
# plotting the pressure field as a contour
plt.contourf(X, Y, p, alpha=0.5, cmap=cm.viridis)
plt.colorbar()
# plotting the pressure field outlines
plt.contour(X, Y, p, cmap=cm.viridis)
# plotting velocity field
plt.quiver(X[::2, ::2], Y[::2, ::2], u[::2, ::2], v[::2, ::2])
plt.xlabel('X')
plt.ylabel('Y')
plt.show()


fig = plt.figure(figsize=(11, 7), dpi=100)
plt.contourf(X, Y, p, alpha=0.5, cmap=cm.viridis)
plt.colorbar()
plt.contour(X, Y, p, cmap=cm.viridis)
plt.streamplot(X, Y, u, v)
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```