Problem 1. *Write programs to solve the advection equation*

$$u_t + au_x = 0,$$

on $[0,1]$ with periodic boundary conditions using upwinding and Lax-Wendroff. For smooth solutions, we expect upwinding to be first-order accurate and Lax-Wendroff to be second-order accurate, but it is not clear what accuracy to expect for nonsmooth solutions.

(a) Let $a = 1$ and solve the problem up to time $t = 1$. Perform a refinement study for both upwinding and Lax-Wendroff with $\Delta t = 0.9a\Delta x$ with a smooth initial condition. Compute the rate of convergence in the 1-norm, 2-norm, and max-norm. Note that the exact solution at time $t = 1$ is the initial condition, and so computing the error is easy.

I used the smooth initial condition $u(x, 0) = \sin(\pi x)$ on [0,1]. I started with an initial number of grid points $N_x = 90$ and initial number of time steps $N_t = 100$ so that

$$\frac{\Delta t}{a\Delta x} = \frac{N_x}{N_t} = 0.9.$$

To implement Upwinding on the periodic domain, I used the iteration matrix

$$S = \begin{pmatrix} 1-\nu & & & \nu \\ \nu & 1-\nu & & \\ & \ddots & \ddots & \\ & & \nu & 1-\nu \end{pmatrix}$$

and followed the scheme

$$u^{n+1} = Su^n$$

for N_t time steps.

To implement Lax-Wendroff on the periodic domain, I used the iteration matrix

$$LW = \begin{pmatrix} 1-\nu^2 & \frac{\nu^2-\nu}{2} & & & & \frac{\nu^2+\nu}{2} \\ \frac{\nu^2+\nu}{2} & 1-\nu^2 & \frac{\nu^2-\nu}{2} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \frac{\nu^2+\nu}{2} & 1-\nu^2 & \frac{\nu^2-\nu}{2} \\ \frac{\nu^2-\nu}{2} & & & & \frac{\nu^2+\nu}{2} & 1-\nu^2 \end{pmatrix}$$

and followed the scheme

$$u^{n+1} = LWu^n$$

for N_t time steps.

From there, I performed a refinement study on both upwinding and Lax-Wendroff methods by doubling N_x (and subsequently N_t). The results are tabulated as follows. As we double the number of grid points, we see that for the upwinding method, the error in the 1-norm and 2-norm are reduced by a factor of 2, so upwinding is indeed first-order in the 1-norm and 2-norm. For the max-norm however, the error is only reduced by a factor of 1.4 ($\approx \sqrt{2}$), so the method is still convergent but less than first-order in the max-norm.

Table 1: Upwinding - 1-Norm Convergence

N_x	$\|u(x,0) - u(x,1)\|_1$	Ratios
90	0.0138114	0
180	0.00694327	1.98918
360	0.00348112	1.99455
720	0.00174294	1.99727
1440	0.000872067	1.99863
2880	0.000436183	1.99932
5760	0.000218129	1.99966
11520	0.000109074	1.99983
23040	5.45392e-05	1.99991
46080	2.72702e-05	1.99996

Table 2: Upwinding - 2-Norm Convergence

Nx	$\|u(x,0) - u(x,1)\|_2$	Ratios
90	0.0153396	0
180	0.00771191	1.98908
360	0.00386653	1.99453
720	0.00193592	1.99726
1440	0.000968623	1.99863
2880	0.000484477	1.99931
5760	0.00024228	1.99966
11520	0.00012115	1.99983
23040	6.05778e-05	1.99991
46080	3.02896e-05	1.99996

Table 3: Upwinding - Max-Norm Convergence

Nx	$\|u(x,0) - u(x,1)\|_\infty$	Ratios
90	0.0216905	0
180	0.0109058	1.40655
360	0.00546805	1.41036
720	0.00273779	1.41228
1440	0.00136984	1.41325
2880	0.000685154	1.41373
5760	0.000342636	1.41397
11520	0.000171333	1.41409
23040	8.567e-05	1.41415
46080	4.28359e-05	1.41418

Table 4: Upwinding- Runtimes

Nx	Runtimes	Runtime Ratios
90	0.002116	0
180	0.003697	1.74716
360	0.007303	1.97539
720	0.015256	2.089
1440	0.034589	2.26724
2880	0.074314	2.14849
5760	0.24084	3.24084
11520	0.771567	3.20365
23040	2.97792	3.85957
46080	11.749	3.94537

For Lax-Wendroff, we see that doubling the number of grid points reduces the error in the 1-norm and 2-norm by a factor of 4, so the method is indeed second-order in the 1-norm and 2-norm. For the max-norm, the error is only reduced by a factor of 2.8, so the method is still convergent but somewhere between first and second-order in the max-norm.

Table 5: Lax-Wendroff - 1-Norm Convergence

N_x	$\|u(x,0)-u(x,1)\|_1$	Ratios
90	0.000617121	0
180	0.000154332	3.99867
360	3.85844e-05	3.99985
720	9.64619e-06	3.99996
1440	2.41155e-06	3.99999
2880	6.02889e-07	4
5760	1.50722e-07	4
11520	3.76805e-08	4
23040	9.42014e-09	4
46080	2.35503e-09	4

Table 6: Lax-Wendroff - 2-Norm Convergence

N_x	$\|u(x,0)-u(x,1)\|_2$	Ratios
90	0.000685479	0
180	0.000171414	3.99897
360	4.28562e-05	3.99975
720	1.07142e-05	3.99994
1440	2.67856e-06	3.99998
2880	6.69641e-07	4
5760	1.6741e-07	4
11520	4.18526e-08	4
23040	1.04631e-08	4
46080	2.61579e-09	4

Table 7: LW - Max-Norm Convergence

N_x	$\|u(x,0)-u(x,1)\|_\infty$	Ratios
90	0.000969175	0
180	0.000242401	2.82788
360	6.06068e-05	2.8283
720	1.51521e-05	2.8284
1440	3.78805e-06	2.82842
2880	9.47015e-07	2.82843
5760	2.36754e-07	2.82843
11520	5.91885e-08	2.82843
23040	1.47971e-08	2.82843
46080	3.69928e-09	2.82843

Table 8: Lax-Wendroff - Runtimes

N_x	Runtimes	Runtime Ratios
90	0.001979	0
180	0.003954	1.99798
360	0.008095	2.04729
720	0.017808	2.19988
1440	0.044824	2.51707
2880	0.106869	2.38419
5760	0.284315	2.66041
11520	0.939381	3.30401
23040	3.94602	4.20066
46080	15.9845	4.0508

(b) Repeat the previous problem with the discontinuous initial condition

$$u(x,0) = \begin{cases} 1 & \text{if } |x-1/2| < 1/4 \\ 0 & \text{otherwise} \end{cases}.$$

Again, I started with an initial number of grid points $N_x = 90$ and initial number of time steps $N_t = 100$ so that

$$\frac{\Delta t}{a\Delta x} = \frac{N_x}{N_t} = 0.9.$$

From there, I performed a refinement study on both upwinding and Lax-Wendroff methods by doubling N_x (and subsequently N_t). The results are tabulated as follows. As we double the number of grid points, we see that for the upwinding method, the errors in the 1-norm is reduced by a factor of 1.4 and the errors in the 2-norm are reduced by a factor of 1.18, so upwinding is less than first-order in the 1-norm and 2-norm. For the max-norm however, the errors are actually growing, so it seems that the method is not convergent for discontinuous initial data in the max-norm.

Table 9: Upwinding - 1-Norm Convergence

N_x	$\|u(x,0)-u(x,1)\|_1$	Ratios
90	0.0527461	0
180	0.0374545	1.40827
360	0.0265402	1.41124
720	0.0187865	1.41272
1440	0.0132911	1.41347
2880	0.00940068	1.41384
5760	0.00664816	1.41403
11520	0.00470127	1.41412
23040	0.00332441	1.41417
46080	0.00235075	1.41419

Table 10: Upwinding - 2-Norm Convergence

Nx	$\|u(x,0)-u(x,1)\|_2$	Ratios
90	0.123893	0
180	0.104568	1.18481
360	0.0880948	1.18699
720	0.074148	1.18809
1440	0.06238	1.18865
2880	0.0524674	1.18893
5760	0.0441248	1.18907
11520	0.0371066	1.18914
23040	0.0312037	1.18917
46080	0.0262395	1.18919

Table 11: Upwinding - Max-Norm Convergence

N_x	$\|u(x,0)-u(x,1)\|_\infty$	Ratios
90	0.45129	0
180	0.465538	0.266128
360	0.475626	0.219853
720	0.482763	0.18248
1440	0.487811	0.152001
2880	0.491381	0.126948
5760	0.493905	0.10623
11520	0.49569	0.089017
23040	0.496953	0.0746683
46080	0.497845	0.0626776

Table 12: Upwinding- Runtimes

N_x	Runtimes	Runtime Ratios
90	0.001942	0
180	0.00373	1.9207
360	0.007954	2.13244
720	0.014647	1.84146
1440	0.034097	2.32792
2880	0.096329	2.82515
5760	0.237519	2.46571
11520	0.863169	3.63411
23040	4.11757	4.77029
46080	19.8843	4.82914

For Lax-Wendroff, we see that doubling the number of grid points reduces the error in the 1-norm by a factor of 1.5 and and in the 2-norm by a factor of 1.24, so the method is less than first-order in the 1-norm and 2-norm. For the max-norm, the error is growing so the method is not convergent for discontinuous initial data in the max-norm.

Table 13: Lax-Wendroff - 1-Norm Convergence

N_x	$\|u(x,0)-u(x,1)\|_1$	Ratios
90	0.0418176	0
180	0.0275851	1.51595
360	0.0183735	1.50135
720	0.0122411	1.50097
1440	0.00813114	1.50546
2880	0.00539299	1.50772
5760	0.003566	1.51234
11520	0.00235739	1.51269
23040	0.00155764	1.51344
46080	0.0010277	1.51567

Table 14: Lax-Wendroff - 2-Norm Convergence

N_x	$\|u(x,0)-u(x,1)\|_2$	Ratios
90	0.106137	0
180	0.0870176	1.21972
360	0.0709873	1.22582
720	0.0577062	1.23015
1440	0.0467847	1.23344
2880	0.0378493	1.23608
5760	0.0305659	1.23828
11520	0.0246465	1.24017
23040	0.0198469	1.24183
46080	0.0159631	1.2433

Table 15: LW - Max-Norm Convergence

N_x	$\|u(x,0) - u(x,1)\|_\infty$	Ratios
90	0.519683	0
180	0.553077	0.191903
360	0.578605	0.150392
720	0.598151	0.118678
1440	0.613178	0.09411
2880	0.624783	0.0748815
5760	0.633786	0.0597193
11520	0.640799	0.0476997
23040	0.646281	0.0381358
46080	0.650578	0.0305066

Table 16: Lax-Wendroff Runtimes

N_x	Runtimes	Runtime Ratios
90	0.001815	0
180	0.00379	2.08815
360	0.008669	2.28734
720	0.018017	2.07833
1440	0.04204	2.33335
2880	0.111989	2.66387
5760	0.445533	3.97836
11520	2.24809	5.04583
23040	11.868	5.27918
46080	57.1154	4.81254

Problem 2. *For solving the heat equation we frequently use Crank-Nicolson, which is trapezoidal rule time integration with a second-order space discretization. The analogous scheme for the linear advection equation is*

$$u_j^{n+1} - u_j^n + \frac{\nu}{4}\big(u_{j+1}^n - u_{j-1}^n\big) + \frac{\nu}{4}\big(u_{j+1}^{n+1} - u_{j-1}^{n+1}\big) = 0,$$

where $\nu = a\Delta t/\Delta x$.

(a) Use von Neumann analysis to show that this scheme is unconditionally stable and that $\|u^n\|_2 = \|u^0\|_2$. This scheme is said to be nondissipative- i.e., there is no amplitude error. This seems reasonable because this is a property of the PDE.

Let $u_j^n = e^{i\xi x_j}$, then $u_j^{n+1} = g(\xi)e^{i\xi x_j}$. Then the PDE becomes

$$\begin{aligned} g(\xi)e^{i\xi x_j} + \frac{\nu}{4}g(\xi)\big(e^{i\xi(x_j+\Delta x)} - e^{i\xi(x_j-\Delta x)}\big) &= e^{i\xi x_j} - \frac{\nu}{4}\big(e^{i\xi(x_j+\Delta x)} - e^{i\xi(x_j-\Delta x)}\big) \\ g(\xi)\Big(1 + \frac{\nu}{4}\big(e^{i\xi\Delta x} - e^{-i\xi\Delta x}\big)\Big) &= 1 - \frac{\nu}{4}\big(e^{i\xi\Delta x} - e^{-i\xi\Delta x}\big) \\ g(\xi)\Big(1 + \frac{\nu}{4}i\sin(\xi\Delta x)\Big) &= 1 - \frac{\nu}{4}i\sin(\xi\Delta x). \end{aligned}$$

If we let $\theta = \xi\Delta x$, then we can write the amplification factor $g(\theta)$ as

$$\begin{aligned} g(\theta) &= \frac{1 - \frac{\nu}{2}i\sin(\theta)}{1 + \frac{\nu}{2}i\sin(\theta)} \\ &= \frac{\big(1 - \frac{\nu}{2}i\sin(\theta)\big)^2}{\big(1 + \frac{\nu}{2}i\sin(\theta)\big)\big(1 - \frac{\nu}{2}i\sin(\theta)\big)} \\ &= \frac{1 - \frac{\nu^2}{4}\sin^2(\theta) - i\nu\sin(\theta)}{1 + \frac{\nu^2}{4}\sin^2(\theta)}. \end{aligned}$$

Then we have that

$$|g(\theta)| = \frac{\Big(1 - \frac{\nu^2}{4}\sin^2(\theta)\Big)^2 + (\nu\sin(\theta))^2}{\big(1 + \frac{\nu^2}{4}\sin^2(\theta)\big)^2}$$

$$= \frac{1 - \frac{\nu^2}{2}\sin^2(\theta) + \frac{\nu^4}{16}\sin^4(\theta) + \nu^2\sin(\theta)}{\left(1 + \frac{\nu^2}{4}\sin^2(\theta)\right)^2}$$
$$= \frac{1 + \frac{\nu^2}{2}\sin^2(\theta) + \frac{\nu^4}{16}\sin^4(\theta)}{\left(1 + \frac{\nu^2}{4}\sin^2(\theta)\right)^2}$$
$$= \frac{\left(1 + \frac{\nu^2}{4}\sin^2(\theta)\right)^2}{\left(1 + \frac{\nu^2}{4}\sin^2(\theta)\right)^2} = 1.$$

Then the scheme is unconditionally stable because

$$|g| = 1 \leq 1 + Ct \quad \text{for all } t\text{, for a constant } C.$$

And we have that the scheme is nondissipative since

$$\|u^n\|_2 = \left\|g^n u^0\right\|_2 = |g|^n \left\|u^0\right\|_2 = \left\|u^0\right\|_2.$$

(b) Solve the advection equation on the periodic domain $[0, 1]$ with the initial condition from problem 1(b). Show the solution and comment on your results.

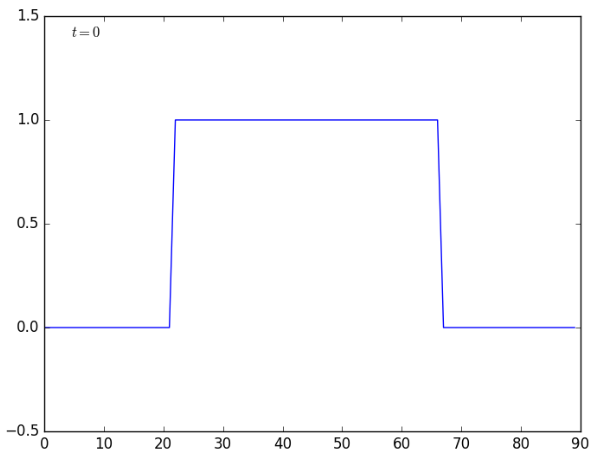
To implement this CN-analogous scheme on the periodic domain, I used the iteration matrix

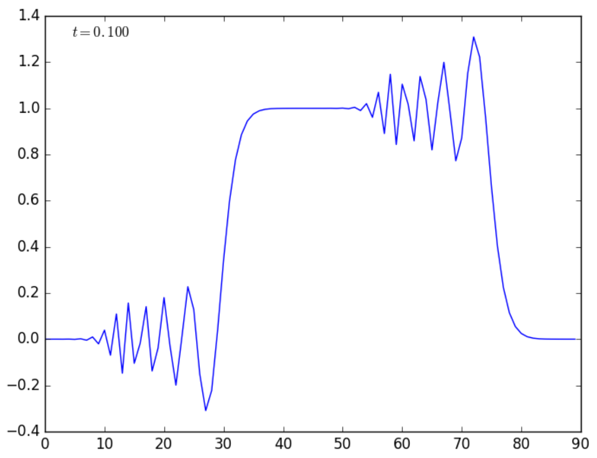
$$CN = \begin{pmatrix} 1 & \frac{-\nu}{4} & & & & \frac{\nu}{4} \\ \frac{\nu}{4} & 1 & \frac{-\nu}{4} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \frac{\nu}{4} & 1 & \frac{-\nu}{4} \\ \frac{-\nu}{4} & & & & \frac{\nu}{4} & 1 \end{pmatrix}$$

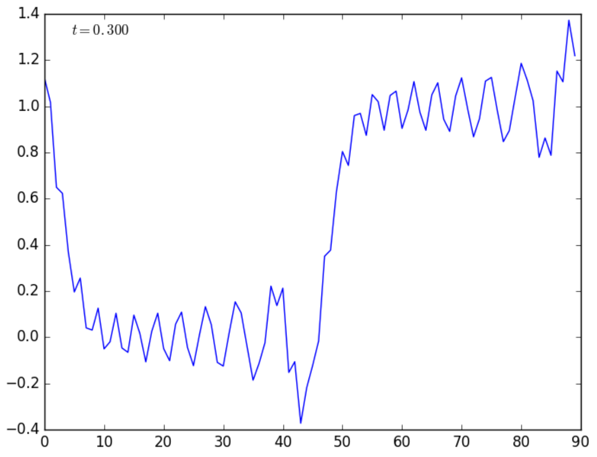
and did the following scheme for N_t time steps

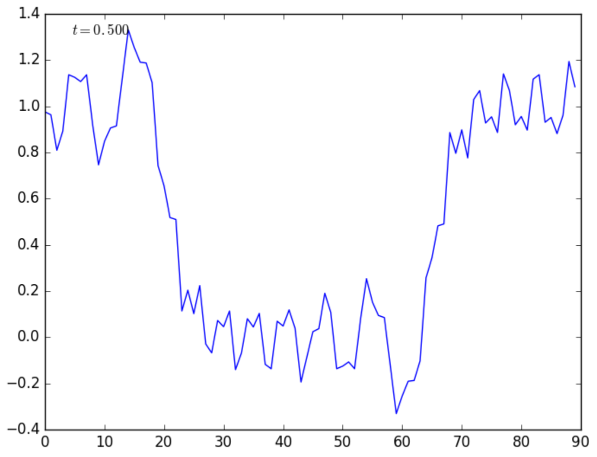
$$(CN)^T u^{n+1} = CNu^n.$$

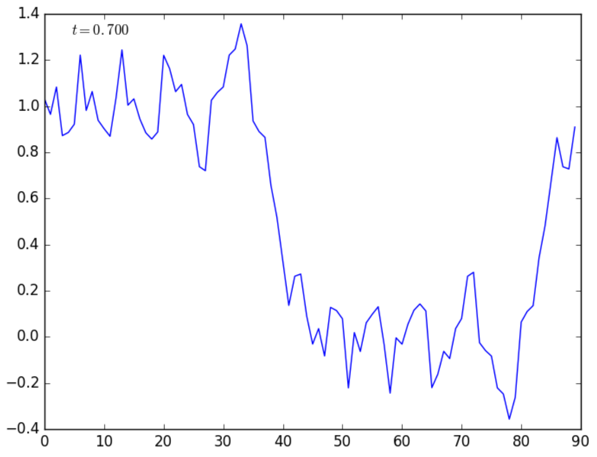
With the discontinuous initial data, things go very wrong very quickly. The high frequencies quickly lag far behind, and the numerical solution doesn't even resemble the real solution after only a few time steps. Notable though is that the amplitude of the solution doesn't grow/shrink much, due to the non-dissipative nature of the scheme. Following are several plots of the numerical solution using this scheme for $\nu = 0.9$, $N_t = 100$, and $N_x = 90$ ($\Delta x = 1/Nx$, $\Delta t = 1/Nt$):

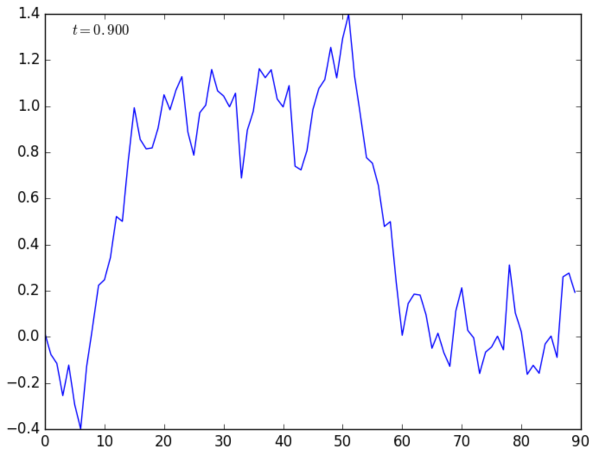

$t = 0$


$t = 0.100$


$t = 0.300$


$t = 0.500$


$t = 0.700$


$t = 0.900$

(c) Compute the relative phase error as $\arg(g(\theta))/(-\nu\theta)$, where g is the amplification factor and $\theta = \xi\Delta x$, and plot it for $\theta \in [0, \pi]$. How does the relative phase error and lack of amplitude error relate to the numerical solutions you observed in part (b).

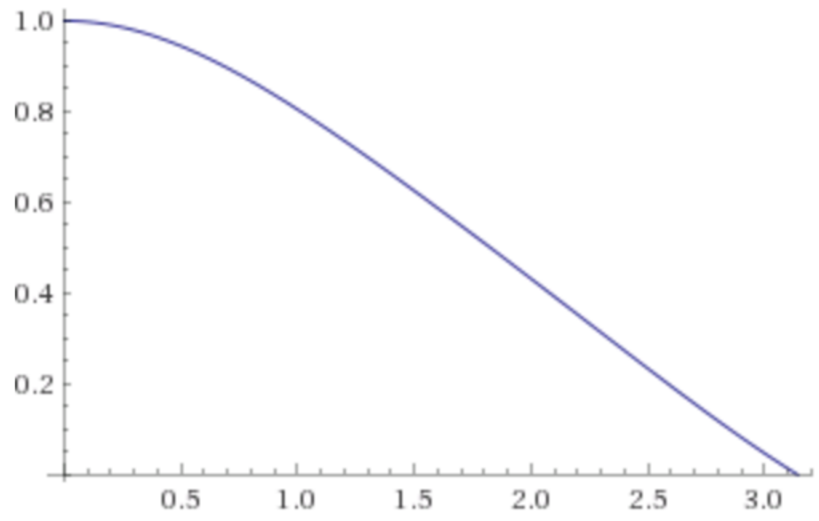
The phase of g is given by

$$\arg(g(\theta)) = \arctan\left(\frac{\text{Im}(g)}{\text{Re}(g)}\right) = \arctan\left(\frac{-\nu\sin\theta}{1 - \frac{\nu^2}{4}\sin^2\theta}\right)$$

Then the relative phase of g is given by

$$\arg(g(\theta))/(-v\theta) = \arctan\left(\frac{\text{Im}(g)}{\text{Re}(g)}\right) = \arctan\left(\frac{-\nu\sin\theta}{1 - \frac{\nu^2}{4}\sin^2\theta}\right)/(-v\theta).$$

The following figure shows a plot of the relative phase for $\nu = 0.9$, $\theta \in [0, \pi]$:



What we see is that for most frequencies we have relative phase lag since $\arg(g(\theta))/(-v\theta) < 1$ for most θ. The higher the frequency, the higher the relative phase lag, and the highest frequencies are almost completely stationary standing waves. The lack of amplitude error in conjunction with this phase lag causes the unphysical behavior we observed in the numerical solution in part b. In contrast to Upwinding and Lax-Wendroff, in this scheme the amplitudes of the high frequencies aren't damped at all due to the non-dissipative nature of this scheme (as we showed in part a). Then since the amplitudes are the high frequencies aren't damped, the enormous phase lag completely dominates the numerical solution, and the numerical solution quickly does not even resemble the real solution.

The Python code I used for number 1 is the following:

```
def upwinding_matrix(N, nu):
  #set sparse matrix S for upwinding method
  #u_j^n+1 = (1-v)u_j^n + v u_j-1^n
  #on [0,1] with periodic BCs
  #set back-diagonal components
  back_diag = nu*np.ones(N)
  #set diagonal components
  diag = (1-nu)*np.ones(N)
  #set corner component (from periodic domain)
  corner = nu*np.ones(N)

  # Generate the matrix
  A = np.vstack((back_diag, diag, corner))
  S = scipy.sparse.dia_matrix((A,[-1,0, (N-1)]),shape=(N,N))
  return S

def upwinding_method(u0, S, Nt):
  #upwinding method on [0,1] w/ periodic BCs
  #with time step delT up to time Tf by doing Nt=Tf/delT steps
  #using scheme matrix S(delX, nu)

  #start iteration with initial given u0
  u_old = u0+0
  #setup plots
  plt.axis([0, S.shape[0], -1, 1])
  plt.ion()
  #do Nt steps
  for t in range(Nt):
    #advance u w/ upwinding scheme
    u_next = S.dot(u_old)
    #update u_old
    u_old = u_next+0
    #plot
    plt.plot(u_next)
    # plt.show()
    plt.pause(0.05)
  plt.close()

  return u_next

  def LW_matrix(N, nu):
  #set sparse matrix LW for LW method
  #u_j^n+1 = (1-v^2)u_j^n + (v^2-v)/2(u_j+1^n) + (v^2+v)/2(u_j-1^n)
  #on [0,1] with periodic BCs

  #set sub-diagonal components
  sub_diag = (nu**2 +nu)/2*np.ones(N)
  #set above-diagonal components
  above_diag = (nu**2 -nu)/2*np.ones(N)
  #set diagonal components
  diag = (1-nu**2)*np.ones(N)
  #set right corner component (from periodic domain)
  right_corner = (nu**2+nu)/2*np.ones(N)
  #set left corner component (from periodic domain)
  left_corner = (nu**2-nu)/2*np.ones(N)

  # Generate the matrix
  A = np.vstack((left_corner, sub_diag, diag, above_diag, right_corner))
  LW = scipy.sparse.dia_matrix((A,[-(N-1),-1,0, 1,(N-1)]),shape=(N,N))
  return LW

def LW_method(u0, LW, Nt):
  #LW method on [0,1] w/ periodic BCs
  #with time step delT up to time Tf with number of steps Nt=Tf/delT
  #using scheme matrix LW(delX, nu)

  #start iteration with initial given u0
  u_old = u0+0

  #setup plots
  plt.axis([0, LW.shape[0], -1, 1])
  plt.ion()
  #do Nt = Tf/delT time steps
  for t in range(Nt):
    #advance u w/ LW scheme
    u_next = LW.dot(u_old)
    #update u_old
    u_old = u_next+0
    #plot
    plt.plot(u_next)
    # plt.show()
    plt.pause(0.05)
  plt.close()

  return u_next

 def refinement_study_up():
  #refinement study for advection eqn on [0,1] w/ periodic BCs
  #for upwinding scheme

  #set vector of number of grid points Nx
  Nx = [90*(2**i) for i in range(0,10)]

  #advection speed
  a = 1
```

```
#final time
Tf=1

#dont plot
plotting=0

#record errors + ratios, run times and runtime ratios
errors_norm1 = np.zeros(len(Nx))
norm1_ratios = np.zeros(len(Nx))
errors_norm2 = np.zeros(len(Nx))
norm2_ratios = np.zeros(len(Nx))
errors_normmax = np.zeros(len(Nx))
normmax_ratios = np.zeros(len(Nx))
times = np.zeros(len(Nx))
time_ratios = np.zeros(len(Nx))


for i in tqdm(range(len(Nx))):
  #get delX
  delX = 1/Nx[i]
  #get number of time steps and time step
  Nt = int(Nx[i]/0.9)
  delT = Tf/Nt
  #get Courant number
  nu = a*delT/delX

  #make upwinding method
  S = upwinding_matrix(Nx[i],nu)
  # print(S.toarray())
  #make smooth initial condition
  grid_X = [delX*j for j in range(Nx[i])]
  u0 = np.asarray([sin(2*pi*x) for x in grid_X])

  toc=clock()
  u = upwinding_method(u0, S, Nt)
  tic=clock()
  error = u0-u
  errors_norm1[i] = delX*sum(abs(error))
  errors_norm2[i] = (delX*sum(error**2))**(1/2)
  errors_normmax[i] = max(abs(error))
  times[i]=tic-toc
  if i>0:
    norm1_ratios[i] = errors_norm1[i-1]/errors_norm1[i]
    norm2_ratios[i] = errors_norm2[i-1]/errors_norm2[i]
    normmax_ratios[i] = errors_norm2[i-1]/errors_normmax[i]
    time_ratios[i] = (tic-toc)/times[i-1]
  table = [[Nx[i], errors_norm1[i], errors_norm2[i], errors_normmax[i], times[i], time_ratio
  print(tabulate(table, headers=["Nx", "1-norm error", "2-norm error", "max-norm error", "ru
  norm1_table = [[Nx[i], errors_norm1[i], norm1_ratios[i]] for i in range(len(Nx))]
  print(tabulate(norm1_table, headers=["Nx", "1-norm error", "Ratios"], tablefmt="latex"))
  norm2_table = [[Nx[i], errors_norm2[i], norm2_ratios[i]] for i in range(len(Nx))]
  print(tabulate(norm2_table, headers=["Nx", "2-norm error", "Ratios"], tablefmt="latex"))
  normmax_table = [[Nx[i], errors_normmax[i], normmax_ratios[i]] for i in range(len(Nx))]
  print(tabulate(normmax_table, headers=["Nx", "Max-norm error", "Ratios"], tablefmt="latex"
  time_table = [[Nx[i], times[i], time_ratios[i]] for i in range(len(Nx))]
  print(tabulate(time_table, headers=["Nx", "Runtimes", "Runtime Ratios"], tablefmt="latex")

def refinement_study_LW():
  #refinement study for advection eqn on [0,1] w/ periodic BCs
  #for Lax-Wendroff scheme

  #set vector of number of grid points Nx
  Nx = [90*(2**i) for i in range(0,10)]

  #advection speed
  a = 1
  #final time
  Tf=1

  #dont plot
  plotting=0

  #record errors + ratios, run times and runtime ratios
  errors_norm1 = np.zeros(len(Nx))
  norm1_ratios = np.zeros(len(Nx))
  errors_norm2 = np.zeros(len(Nx))
  norm2_ratios = np.zeros(len(Nx))
  errors_normmax = np.zeros(len(Nx))
  normmax_ratios = np.zeros(len(Nx))
  times = np.zeros(len(Nx))
  time_ratios = np.zeros(len(Nx))


  for i in tqdm(range(len(Nx))):
    #get delX
    delX = 1/Nx[i]
    #get number of time steps and time step
    Nt = int(Nx[i]/0.9)
    delT = Tf/Nt
    #get Courant number
    nu = a*delT/delX
    #make LW method
    LW = LW_matrix(Nx[i],nu)
    # print(S.toarray())
    #make smooth initial condition
    grid_X = [delX*j for j in range(Nx[i])]
    u0 = np.asarray([sin(2*pi*x) for x in grid_X])

    toc=clock()
    u = LW_method(u0, LW, Nt)
    tic=clock()
    error = u0-u
    errors_norm1[i] = delX*sum(abs(error))
    errors_norm2[i] = (delX*sum(error**2))**(1/2)
    errors_normmax[i] = max(abs(error))
    times[i]=tic-toc
    if i>0:
      norm1_ratios[i] = errors_norm1[i-1]/errors_norm1[i]
      norm2_ratios[i] = errors_norm2[i-1]/errors_norm2[i]
      normmax_ratios[i] = errors_norm2[i-1]/errors_normmax[i]
      time_ratios[i] = (tic-toc)/times[i-1]

  table = [[Nx[i], errors_norm1[i], errors_norm2[i], errors_normmax[i], times[i], time_ratio
  print(tabulate(table, headers=["Nx", "1-norm error", "2-norm error", "max-norm error", "ru
  norm1_table = [[Nx[i], errors_norm1[i], norm1_ratios[i]] for i in range(len(Nx))]
  print(tabulate(norm1_table, headers=["Nx", "1-norm error", "Ratios"], tablefmt="latex"))
  norm2_table = [[Nx[i], errors_norm2[i], norm2_ratios[i]] for i in range(len(Nx))]
  print(tabulate(norm2_table, headers=["Nx", "2-norm error", "Ratios"], tablefmt="latex"))
  normmax_table = [[Nx[i], errors_normmax[i], normmax_ratios[i]] for i in range(len(Nx))]
  print(tabulate(normmax_table, headers=["Nx", "Max-norm error", "Ratios"], tablefmt="latex"
  time_table = [[Nx[i], times[i], time_ratios[i]] for i in range(len(Nx))]
  print(tabulate(time_table, headers=["Nx", "Runtimes", "Runtime Ratios"], tablefmt="latex")

def refinement_study_up():
  #refinement study for advection eqn on [0,1] w/ periodic BCs
  #for upwinding scheme

  #set vector of number of grid points Nx
  Nx = [90*(2**i) for i in range(0,10)]

  #advection speed
  a = 1
  #final time
  Tf=1

  #dont plot
```

```
plotting=0

#record errors + ratios, run times and runtime ratios
errors_norm1 = np.zeros(len(Nx))
norm1_ratios = np.zeros(len(Nx))
errors_norm2 = np.zeros(len(Nx))
norm2_ratios = np.zeros(len(Nx))
errors_normmax = np.zeros(len(Nx))
normmax_ratios = np.zeros(len(Nx))
times = np.zeros(len(Nx))
time_ratios = np.zeros(len(Nx))


for i in tqdm(range(len(Nx))):
  #get delX
  delX = 1/Nx[i]
  #get number of time steps and time step
  Nt = int(Nx[i]/0.9)
  delT = Tf/Nt
  #get Courant number
  nu = a*delT/delX

  #make upwinding method
  S = upwinding_matrix(Nx[i],nu)
  # print(S.toarray())
  #make discontinuou initial condition
  u0 = np.r_[[0 for j in range(int(Nx[i]/4))], [1 for j in range(int(Nx[i]/4),int(3*Nx[i]/

  toc=clock()
  u = upwinding_method(u0, S, Nt)
  tic=clock()
  error = u0-u
  errors_norm1[i] = delX*sum(abs(error))
  errors_norm2[i] = (delX*sum(error**2))**(1/2)
  errors_normmax[i] = max(abs(error))
  times[i]=tic-toc
  if i>0:
    norm1_ratios[i] = errors_norm1[i-1]/errors_norm1[i]
    norm2_ratios[i] = errors_norm2[i-1]/errors_norm2[i]
    normmax_ratios[i] = errors_norm2[i-1]/errors_normmax[i]
    time_ratios[i] = (tic-toc)/times[i-1]

table = [[Nx[i], errors_norm1[i], errors_norm2[i], errors_normmax[i], times[i], time_ratio
print(tabulate(table, headers=["Nx", "1-norm error", "2-norm error", "max-norm error", "ru
norm1_table = [[Nx[i], errors_norm1[i], norm1_ratios[i]] for i in range(len(Nx))]
print(tabulate(norm1_table, headers=["Nx", "1-norm error", "Ratios"], tablefmt="latex"))
```

```
  norm2_table = [[Nx[i], errors_norm2[i], norm2_ratios[i]] for i in range(len(Nx))]
  print(tabulate(norm2_table, headers=["Nx", "2-norm error", "Ratios"], tablefmt="latex"))
  normmax_table = [[Nx[i], errors_normmax[i], normmax_ratios[i]] for i in range(len(Nx))]
  print(tabulate(normmax_table, headers=["Nx", "Max-norm error", "Ratios"], tablefmt="latex"
  time_table = [[Nx[i], times[i], time_ratios[i]] for i in range(len(Nx))]
  print(tabulate(time_table, headers=["Nx", "Runtimes", "Runtime Ratios"], tablefmt="latex")

def refinement_study_LW():
  #refinement study for advection eqn on [0,1] w/ periodic BCs
  #for Lax-Wendroff scheme

  #set vector of number of grid points Nx
  Nx = [90*(2**i) for i in range(0,10)]

  #advection speed
  a = 1
  #final time
  Tf=1

  #dont plot
  plotting=0

  #record errors + ratios, run times and runtime ratios
  errors_norm1 = np.zeros(len(Nx))
  norm1_ratios = np.zeros(len(Nx))
  errors_norm2 = np.zeros(len(Nx))
  norm2_ratios = np.zeros(len(Nx))
  errors_normmax = np.zeros(len(Nx))
  normmax_ratios = np.zeros(len(Nx))
  times = np.zeros(len(Nx))
  time_ratios = np.zeros(len(Nx))


  for i in tqdm(range(len(Nx))):
    #get delX
    delX = 1/Nx[i]
    #get number of time steps and time step
    Nt = int(Nx[i]/0.9)
    delT = Tf/Nt
    #get Courant number
    nu = a*delT/delX

    #make LW method
    LW = LW_matrix(Nx[i],nu)

    #make discontinuou initial condition
    u0 = np.r_[[0 for j in range(int(Nx[i]/4))], [1 for j in range(int(Nx[i]/4),int(3*Nx[i]/


    toc=clock()
    u = LW_method(u0, LW, Nt)
    tic=clock()
    error = u0-u
    errors_norm1[i] = delX*sum(abs(error))
    errors_norm2[i] = (delX*sum(error**2))**(1/2)
    errors_normmax[i] = max(abs(error))
    times[i]=tic-toc
    if i>0:
      norm1_ratios[i] = errors_norm1[i-1]/errors_norm1[i]
      norm2_ratios[i] = errors_norm2[i-1]/errors_norm2[i]
      normmax_ratios[i] = errors_norm2[i-1]/errors_normmax[i]
      time_ratios[i] = (tic-toc)/times[i-1]

  table = [[Nx[i], errors_norm1[i], errors_norm2[i], errors_normmax[i], times[i], time_ratio
  print(tabulate(table, headers=["Nx", "1-norm error", "2-norm error", "max-norm error", "ru
  norm1_table = [[Nx[i], errors_norm1[i], norm1_ratios[i]] for i in range(len(Nx))]
  print(tabulate(norm1_table, headers=["Nx", "1-norm error", "Ratios"], tablefmt="latex"))
  norm2_table = [[Nx[i], errors_norm2[i], norm2_ratios[i]] for i in range(len(Nx))]
  print(tabulate(norm2_table, headers=["Nx", "2-norm error", "Ratios"], tablefmt="latex"))
  normmax_table = [[Nx[i], errors_normmax[i], normmax_ratios[i]] for i in range(len(Nx))]
  print(tabulate(normmax_table, headers=["Nx", "Max-norm error", "Ratios"], tablefmt="latex"
  time_table = [[Nx[i], times[i], time_ratios[i]] for i in range(len(Nx))]
  print(tabulate(time_table, headers=["Nx", "Runtimes", "Runtime Ratios"], tablefmt="latex")
```

Python Code for Problem 2:

```
def CN_matrix(N, nu):
  #set sparse matrix CN for CN-analogous method
  #u_j^n+1 + v/4(u_j+1^n+1-u_j-1^n+1) =  u_j^n - v/4(u_j+1^n - u_j-1^n)
  #on [0,1] with periodic BCs

  #set sub-diagonal components
  sub_diag = (nu/4)*np.ones(N)
  #set above-diagonal components
  above_diag = (-nu/4)*np.ones(N)
  #set diagonal components
  diag = np.ones(N)
  #set right corner component (from periodic domain)
  right_corner = (nu/4)*np.ones(N)
  #set left corner component (from periodic domain)
  left_corner = (-nu/4)*np.ones(N)
  # Generate the matrix
  A = np.vstack((left_corner, sub_diag, diag, above_diag, right_corner))
  CN = scipy.sparse.dia_matrix((A,[-(N-1),-1,0, 1,(N-1)]),shape=(N,N))
  return CN

def CN_method(u0, CN, Nt):
  #CN-analogous method on [0,1] w/ periodic BCs
  #with time step delT up to time Tf by doing Nt=Tf/delT steps
  #using scheme matrix S(delX, nu)

  #start iteration with initial given u0
  u_old = u0+0
  #setup plots
  fig = plt.figure()
  ax = fig.add_subplot(111)
  ax.set_ylim(min(u0)-0.5, max(u0)+0.5)
  plt.ion()
  text = ax.text(0.05,0.95, r"$t=0$", transform=ax.transAxes)
  ax.plot(u_old)
  frame_no=1
  filename='CN_advection_fig0'+str(frame_no)+'.png'
  savefig(filename)
  plt.pause(0.5)

  #do Nt steps
  for t in range(Nt):
    #advance u w/ CN-type scheme
    RHS_terms = CN.dot(u_old)
    LHS_matrix = scipy.sparse.csc_matrix(CN.T)
    u_next = scipy.sparse.linalg.spsolve(LHS_matrix, RHS_terms)
    #update u_old
    u_old = u_next+0
    #plot
    ax.clear()
    text = ax.text(0.05,0.95, r"$t=%.3f$" % (t/Nt), transform=ax.transAxes)
    frame = ax.plot(u_next)
    plt.pause(0.5)
    if t%10==0:
      frame_no=frame_no+1
      if frame_no<10:
        filename='CN_advection_fig0'+str(frame_no)+'.png'
      else:
        filename='CN_advection_fig'+str(frame_no)+'.png'
      savefig(filename)

  return u_next
```

```
if __name__ == '__main__':
  #system parameters
  a=1
  Tf = 1
  delX = 0.01
  Nx = 90
  Nt = 100
  delX = 1/Nx
  delT = Tf/Nt
  nu = a*delT/delX
  print(nu)
  #make upwinding method
  CN = CN_matrix(Nx,nu)
  print(CN.toarray())
  #make smooth initial condition
  # grid_X = [delX*i for i in range(Nx)]
  # # print(grid_X)
  # u0 = np.asarray([sin(2*pi*x) for x in grid_X])

  #make disconts initial cond
  u0 = np.r_[[0 for j in range(int(Nx/4))], [1 for j in range(int(Nx/4),int(3*Nx/4))], [0 fo

  # print(u0)
  #solve advection eqn using upwinding method up to Tf=1
  u = CN_method(u0, CN, Nt)
  # print(u)
  print(u0-u)
```