Astron 702: Midterm

Due on Thursday, March 27

Townsend 1:20 pm

Elijah Bernstein-Cooper

March 27, 2014

Problem 1

Using the Forward-Time Center-Space (FTCS) scheme to calculate the progression of $y(x,t)$ we use the following

$$y_k^{n+1} = y_k^n - a(t^{n+1} - t^n)\ \frac{y_{k+1}^n - y_{k-1}^n}{x_{k+1} - x_{k-1}} \tag{1}$$

We use this scheme to calculate $y(x,t)$ on the intervals $0 < x < 1$ and $0 < t < 3$ with $\delta x = 0.01$ and $\delta t = 0.005$ with periodic boundary conditions. After running this simulation we find that the FTCS scheme reproduces the analytic result of

$$y(x,t) = \sin[2\pi(x - at)] \tag{2}$$

terribly. See Figure 1 for snapshots of y at different times as a function of x. The numerical progression of $y(x,t)$ diverged.

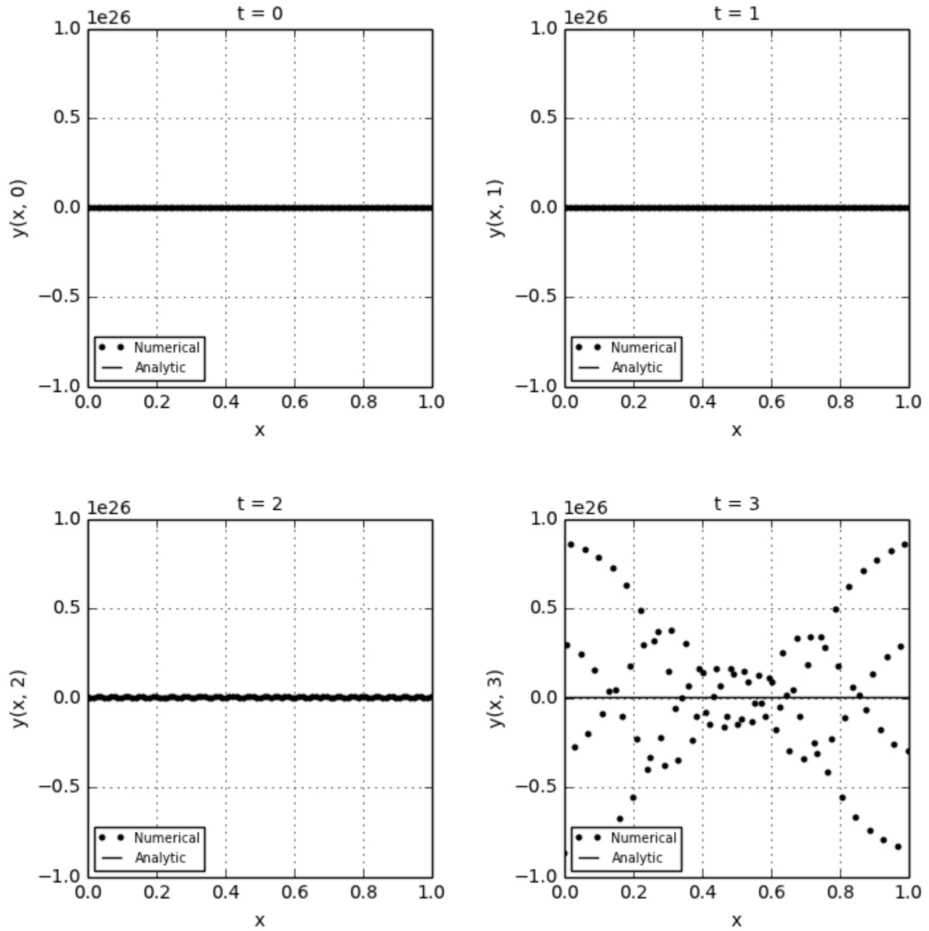

Figure 1: Snapshots of y at different times as a function of x using the FTCS scheme. The FTCS scheme reproduces the analytic results Equation 2 terribly. The numerical progression of $y(x,t)$ diverged.

Problem 2

The Forward-Time Forward-Space (FTFS) scheme can be calculated numerically by

$$y_k^{n+1} = y_k^n - a(t^{n+1} - t^n)\,\frac{y_{k+1}^n - y_k^n}{x_{k+1} - x_k} \tag{3}$$

and the Forward-Time Backward-Space (FTFS) scheme can be calculated numerically by

$$y_k^{n+1} = y_k^n - a(t^{n+1} - t^n)\,\frac{y_k^n - y_{k-1}^n}{x_k - x_{k-1}} \tag{4}$$

We ran the numerical simulation of the progression of $y(x,t)$ using the same parameters and boundaries as in Problem 1. Figures 2 and 3 show the results of the simulations.

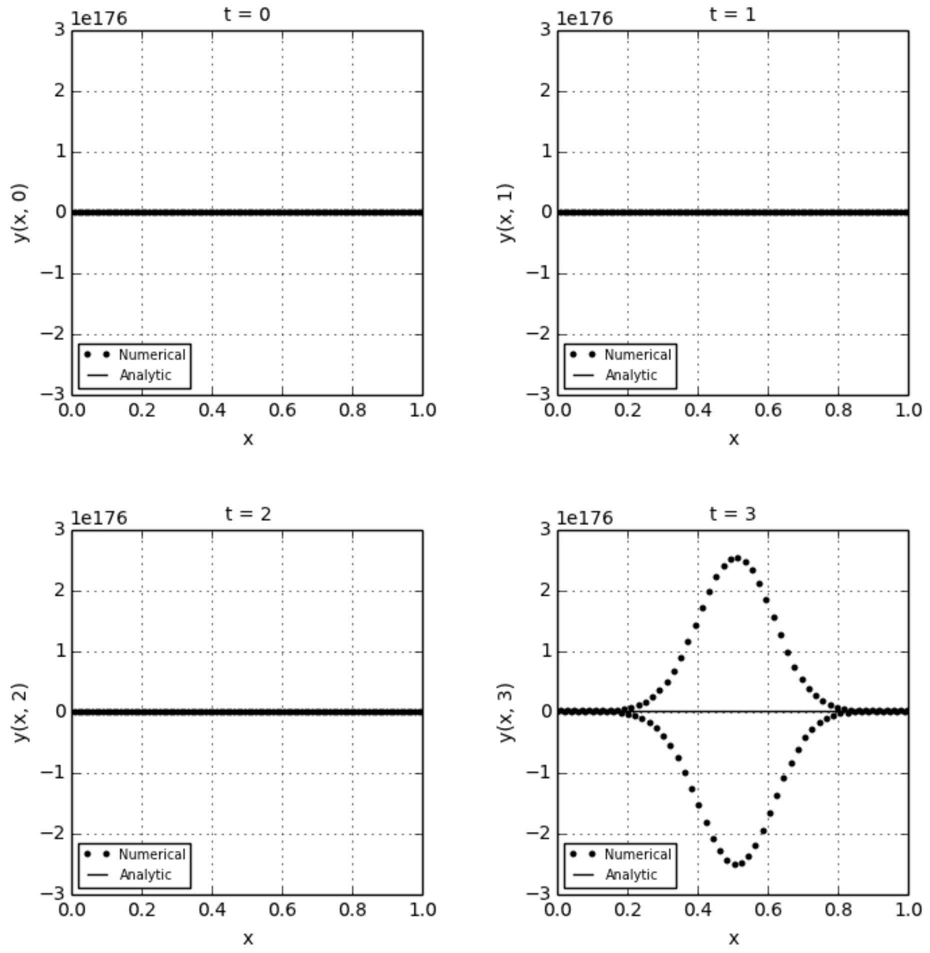

Figure 2: Snapshots of y at different times as a function of x using the FTFS scheme. The FTFS scheme reproduces the analytic results Equation 2 terribly given the parameters in Problem 1. The numerical progression of $y(x,t)$ diverged.

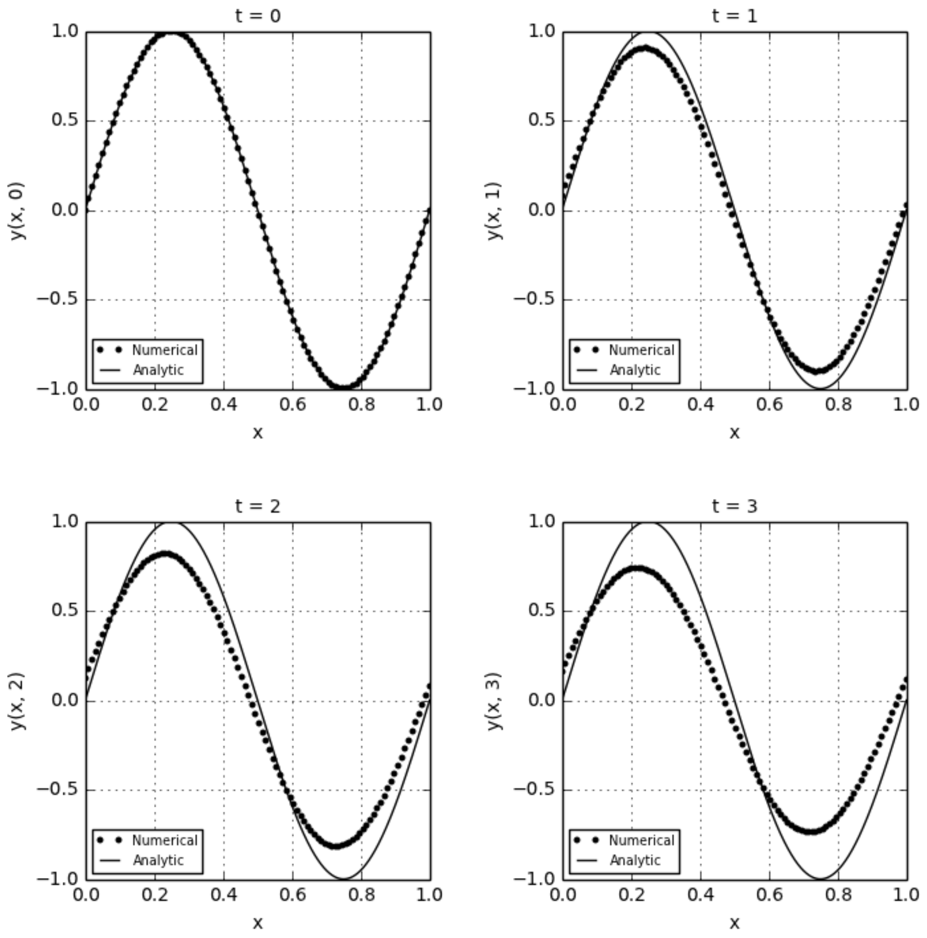

Figure 3: Snapshots of y at different times as a function of x using the FTBS scheme. The FTBS scheme reproduces the analytic results Equation 2 mildly well given the parameters in Problem 1. The numerical progression of $y(x,t)$ is decreasing in amplitude from the analytic result with time. This is an example of numerical diffusion.

Problem 3

We ran the numerical simulation of the progression of $y(x, t)$ using the same parameters and boundaries as in Problem 1 except we set the flow speed, $a = -1$. Figures 4 and 5 show the results of the simulations.

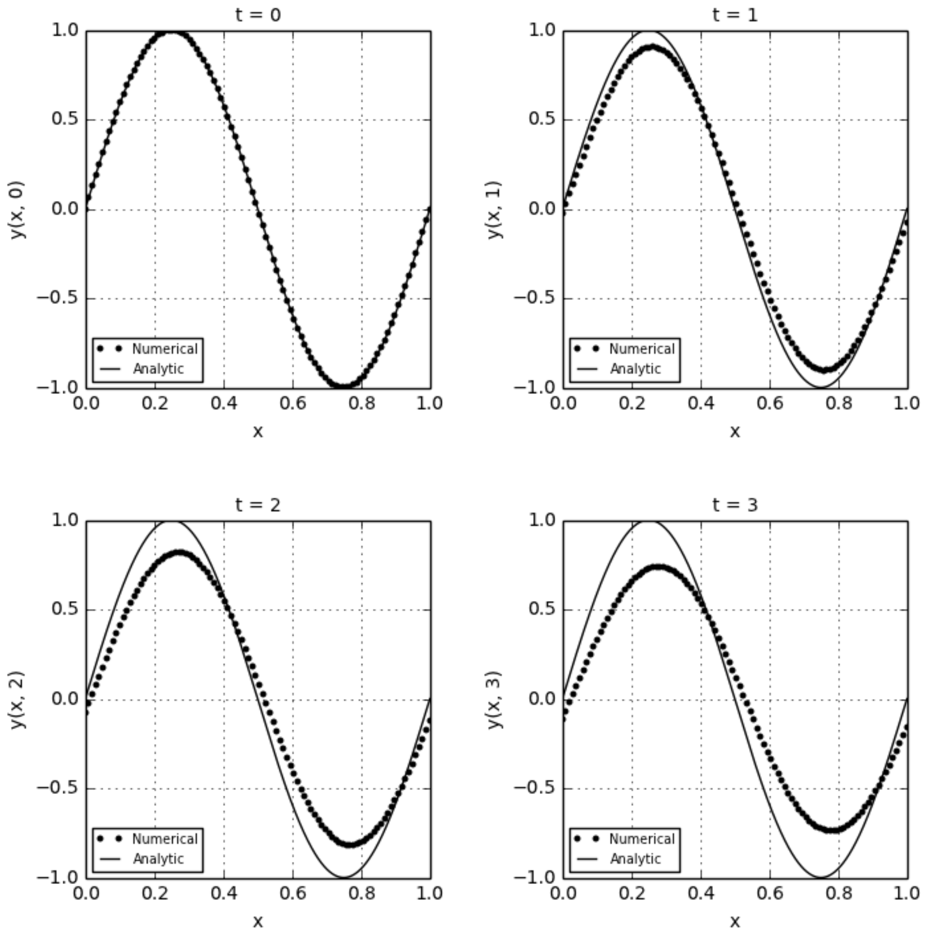

Figure 4: Snapshots of y at different times as a function of x using the FTFS scheme. The FTFS scheme reproduces the analytic results Equation 2 mildly well given the parameters in Problem 1. The numerical progression of $y(x, t)$ is decreasing in amplitude from the analytic result with time. Because the flow speed switched directions the FTFS scheme is now calculating the progression of y from upstream instead of downstream.

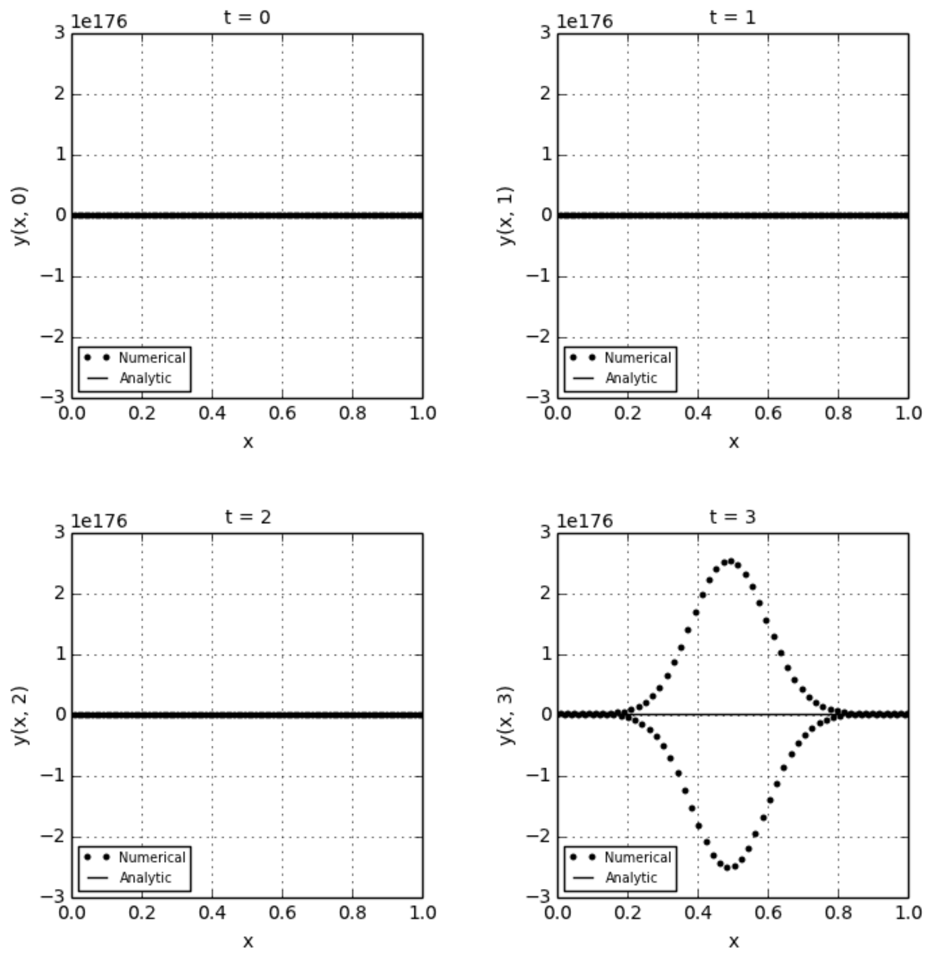

Figure 5: Snapshots of y at different times as a function of x using the FTBS scheme. The FTBS scheme reproduces the analytic results Equation 2 terribly given the parameters in Problem 1. The numerical progression of $y(x, t)$ diverged. Because the flow speed switched directions the FTBS scheme is now calculating the progression of y from downstream instead of upstream.

Problem 4

To apply a von Neumann stability analysis on the Forward-Time Forward-Space (FTFS) scheme: since $x_k = k\Delta x$, the initial state at $t = 0$ can be written as

$$y_k^0 = \sin[2\pi k\Delta x] \tag{5}$$

which we then substitute into the FTFS scheme

$$y_k^{n+1} = y_k^n + a(t^{n+1} - t^n)\ \frac{y_{k+1}^n - y_k^n}{x_{k+1} - x_k} \tag{6}$$

to get the solution after a single time-step

$$y_k^1 = \sin[2\pi k\Delta x] - a\Delta t\frac{\sin[2\pi(k+1)\Delta x] - \sin[2\pi k\Delta x]}{\Delta x} \tag{7}$$

Setting $a\Delta t/\Delta x = \alpha$ we get

$$y_k^1 = \sin[2\pi k\Delta x] - \alpha\sin[2\pi k\Delta x + 2\pi\Delta x] - \sin[2\pi k\Delta x] \tag{8}$$

using the identity

$$\sin[\theta] \pm \sin[\phi] = 2\sin[\frac{\theta \pm \phi}{2}]\cos[\frac{\theta \mp \phi}{2}] \tag{9}$$

$$y_k^1 = \sin[2\pi k\Delta x] - 2\alpha\sin[\frac{2\pi k\Delta x + 2\pi\Delta x - 2\pi k\Delta x}{2}]\cos[\frac{2\pi k\Delta x + 2\pi\Delta x + 2\pi k\Delta x}{2}] \tag{10}$$

$$y_k^1 = \sin[2\pi k\Delta x] - 2\alpha\sin[\pi\Delta x]\cos[2\pi k\Delta x + \pi\Delta x] \tag{11}$$

using the identity

$$\cos[\theta \pm \phi] = \cos[\theta]\cos[\phi] \mp \sin[\theta]\sin[\phi] \tag{12}$$

$$y_k^1 = \sin[2\pi k\Delta x] - 2\alpha\sin[\pi\Delta x][\cos[2\pi k\Delta x]\cos[\pi\Delta x] - \sin[2\pi k\Delta x]\sin[\pi\Delta x]] \tag{13}$$

$$y_k^1 = \sin[2\pi k\Delta x] - 2\alpha\sin[\pi\Delta x][\cos[\pi\Delta x]\sin[\pi\Delta x]\cos[2\pi k\Delta x] - \sin^2[\pi\Delta x]\sin[2\pi k\Delta x]] \tag{14}$$

$$y_k^1 = (1 + 2\alpha\sin^2[\pi\Delta x])\sin[2\pi k\Delta x] + 2\alpha\cos[\pi\Delta x]\sin[\pi\Delta x]\cos[2\pi k\Delta x] \tag{15}$$

using the identities

$$\sin^2[\theta] = \frac{1 - \cos[2\theta]}{2} \qquad \cos[\theta]\sin[\theta] = \frac{\sin[2\theta]}{2} \tag{16}$$

$$y_k^1 = (1 - \frac{2\alpha}{2}(1 - \cos[2\pi\Delta x]))\sin[2\pi k\Delta x] + \frac{2\alpha}{2}\sin[2\pi\Delta x]\cos[2\pi k\Delta x] \tag{17}$$

We define the grid parameter $G \equiv 1 - \cos[2\pi\Delta x]$ thus

$$y_k^1 = (1 - G\alpha)\sin[2\pi k\Delta x] + \alpha\sin[2\pi\Delta x]\cos[2\pi k\Delta x] \tag{18}$$

using the identity

$$a\sin[\theta] + b\cos[\theta] = c\sin[\theta - \phi] \tag{19}$$

Problem 4 continued on next page...

where $c = \sqrt{a^2 + b^2}$ and $\phi = \text{atan2}[b, a]$, we can arrange Eq. 18 to be

$$y_k^1 = A \sin[2\pi\Delta x + \phi] \tag{20}$$

where

$$A = \sqrt{(1 + G\alpha)^2 + \alpha^2 \sin^2[2\pi\Delta x]} \qquad \phi = \text{atan2}[-\alpha \sin[2\pi\Delta x], 1 + G\alpha] \tag{21}$$

We can manipulate G^2 by setting $\theta = 2\pi\Delta x$ and then

$$\begin{aligned} G &\equiv 1 - \cos[\theta] \\ G^2 &= 1 - 2\cos[\theta] + \cos^2[\theta] \\ G^2 &= 1 - 2\cos[\theta] + 1 - \sin^2[\theta] \\ G^2 &= 2 - 2(1 - G) - \sin^2[\theta] \end{aligned}$$

thus

$$\sin^2[\theta] = 2G - G^2 \tag{22}$$

which we can substitute into A, to get

$$A = \sqrt{(1 + G\alpha)^2 + \alpha^2(2G - G^2)} \tag{23}$$

$$A = \sqrt{(1 + 2G\alpha + G^2\alpha^2) + 2\alpha^2 G - \alpha^2 G^2} \tag{24}$$

$$A = \sqrt{1 + 2G\alpha(\alpha + 1)} \tag{25}$$

Finally, for any given number of time steps, n, using the Forward-Time Forward-Space scheme we will have

$$y_k^n = A^n \sin[2\pi\Delta x + n\phi] \tag{26}$$

where

$$A = \sqrt{1 + 2G\alpha(\alpha + 1)} \qquad \phi = \text{atan2}[-\alpha \sin[2\pi\Delta x], (1 + G\alpha)] \tag{27}$$

To apply a von Neumann stability analysis on the Forward-Time Backward-Space (FTBS) scheme: since $x_k = k\Delta x$, the initial state at $t = 0$ can be written as

$$y_k^{n+1} = y_k^n - a(t^{n+1} - t^n)\ \frac{y_k^n - y_{k-1}^n}{x_k - x_{k-1}} \tag{28}$$

to get the solution after a single time-step

$$y_k^1 = \sin[2\pi k\Delta x] + a\Delta t \frac{\sin[2\pi k\Delta x] - \sin[2\pi(k-1)\Delta x]}{\Delta x} \tag{29}$$

Following the same methods as for the FTFS stability analysis, we reveal the progression of amplitudes as a function of time and space:

For any given number of time steps, n, using the Forward-Time Backward-Space scheme we will have

$$y_k^n = A^n \sin[2\pi\Delta x + n\phi] \tag{30}$$

where

$$A = \sqrt{1 + 2G\alpha(\alpha - 1)} \qquad \phi = \text{atan2}[\alpha \sin[2\pi\Delta x], (1 - G\alpha)] \tag{31}$$

Problem 5

Figure 6 analyzes the stability of FTFS and FTBS schemes across a range of G and α.

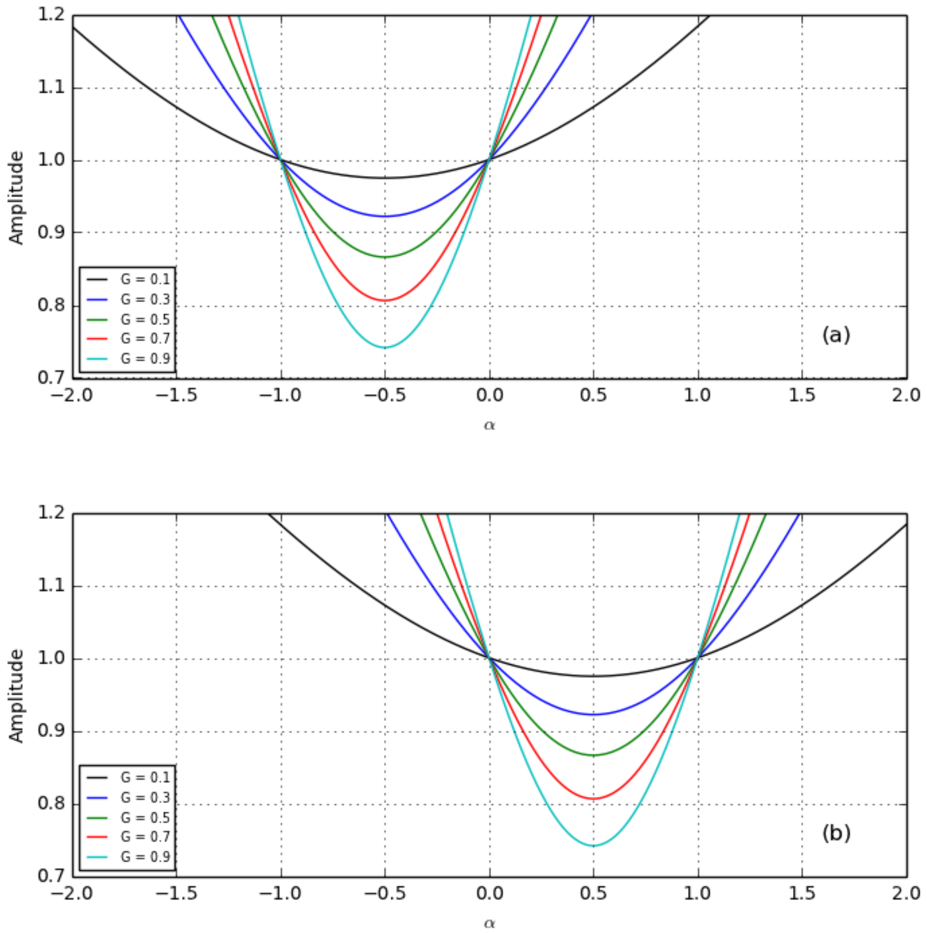

Figure 6: Amplitudes of (a): FTFS scheme and (b): FTBS scheme as a function of α for different G values (see Equations 27 and 31 respectively). The amplitude of the flow will remain finite as a function of time in the FTFS scheme for $0 < \alpha < 1$, and for the FTBS scheme for $-1 < \alpha < 0$. This means that only upstream schemes will provide stable progressions of flow amplitudes for $|\alpha| < 1$.

Problem 6

Figure 7 demonstrates the result of running a upstream scheme with $|\alpha| > 1$. The numerical progression of $y(x,t)$ diverged. α is now > 2 hence the amplitude will diverge from solution because the FTBS scheme is calculating the progression of y from downstream instead of upstream.

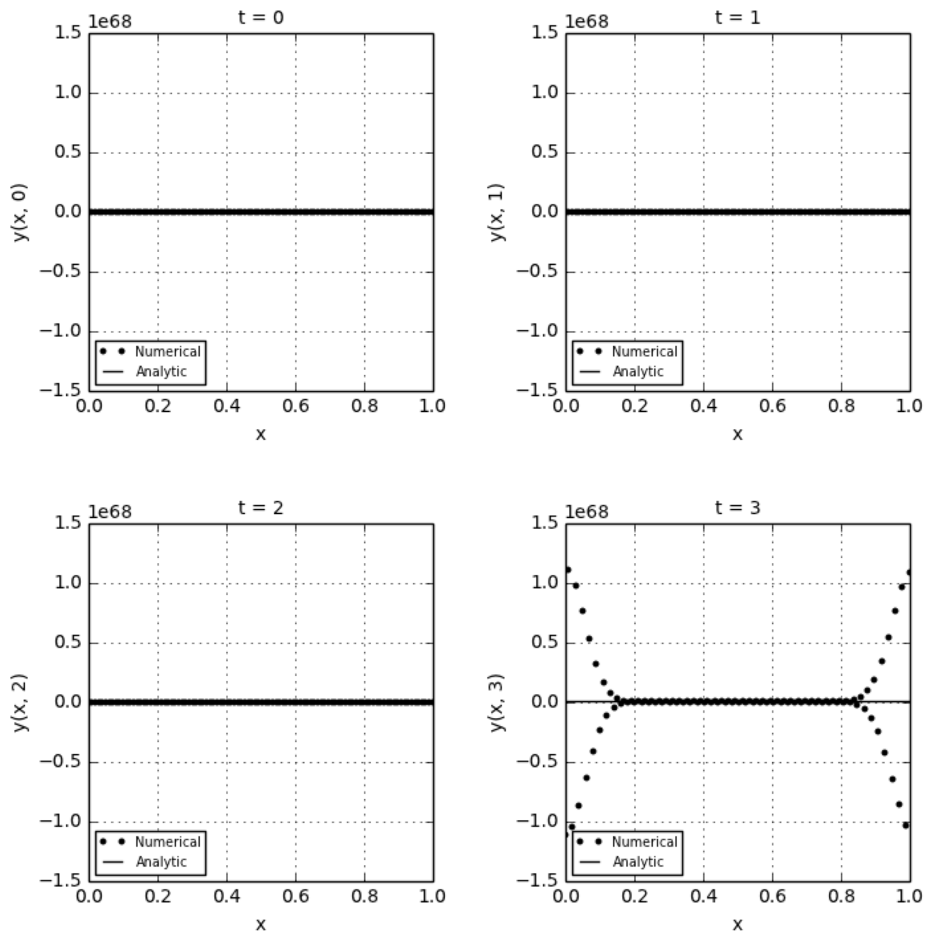

Figure 7: Snapshots of y at different times as a function of x using the FTBS scheme. The FTBS scheme reproduces the analytic results Equation 2 terribly given a flow speed of 1 and $\alpha = 2$. The numerical progression of $y(x,t)$ diverged. Because α is now > 2 the amplitude will diverge from solution. The FTBS scheme is calculating the progression of y from downstream instead of upstream.

Problem 7

Figure 8 demonstrates the affect of choosing different α values in a stable scheme. The numerical diffusion is greater for $\alpha = 0.25$ than for $\alpha = 0.5$ because the simulation with $\alpha = 0.25$ needs more time steps to reach the same time as for the $\alpha = 0.5$ simulation, despite the diffusion amplitude being closer to 1 for $\alpha = 0.25$.

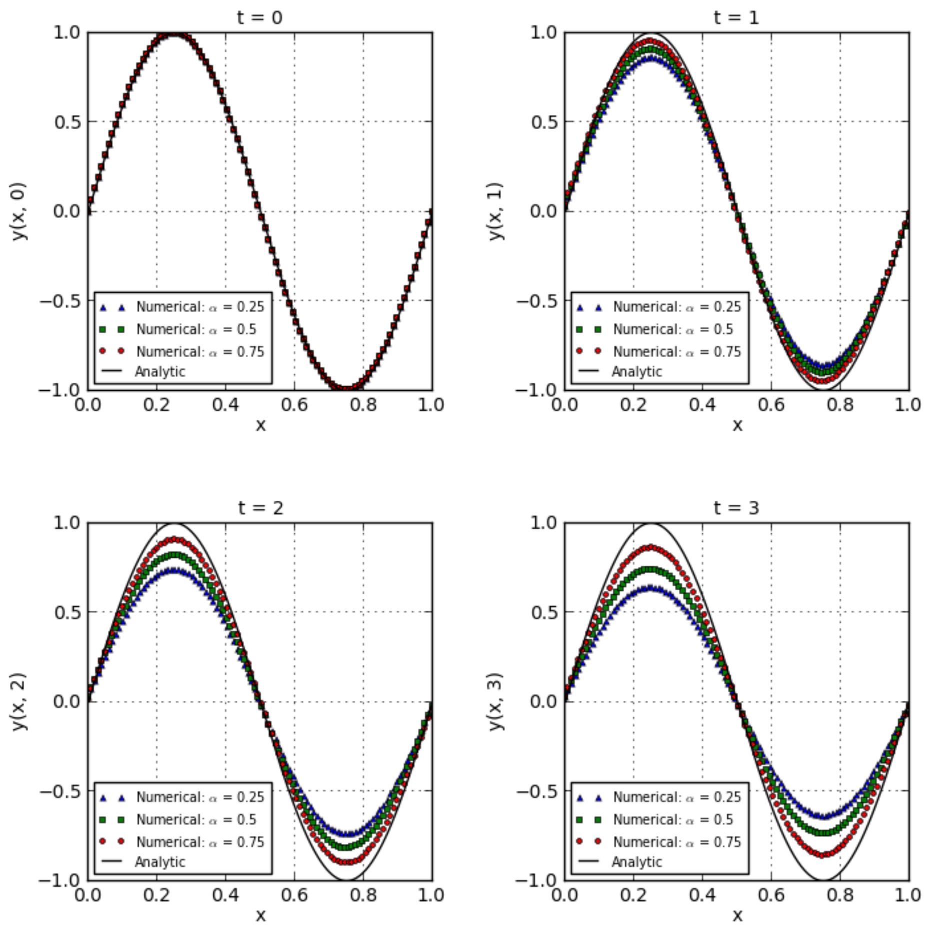

Figure 8: Snapshots of y at different times as a function of x using the FTBS scheme given a flow speed of 1 and and with varying α values.

Problem 8

Our BTCS scheme was derived beginning with

$$y_k^{n+1} = y_k^n + a(t^{n+1} - t^n)\ \frac{y_{k+1}^n - y_k^n}{x_{k+1} - x_k} \tag{32}$$

and next solving for y_k^n

$$y_k^n = y_k^{n+1} - \alpha y_{k+1}^{n+1} + \alpha y_{k-1}^{n+1} \tag{33}$$

whereby we can construct a $K \times K$ matrix M to solve the scheme for forward times

$$M = \begin{pmatrix} 1 & -\alpha & \cdots & \alpha \\ \alpha & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -\alpha & \alpha & \cdots & 1 \end{pmatrix} \tag{34}$$

$$M \begin{pmatrix} y_0^{n+1} \\ y_1^{n+1} \\ \vdots \\ y_{K-1}^{n+1} \end{pmatrix} = \begin{pmatrix} y_0^n \\ y_1^n \\ \vdots \\ y_{K-1}^n \end{pmatrix} \tag{35}$$

which allows us to solve for future time steps from an initial condition.

Figure 9 shows simulation results from the backward-time centered-space (BTCS) scheme.

Problem 8 continued on next page. . .

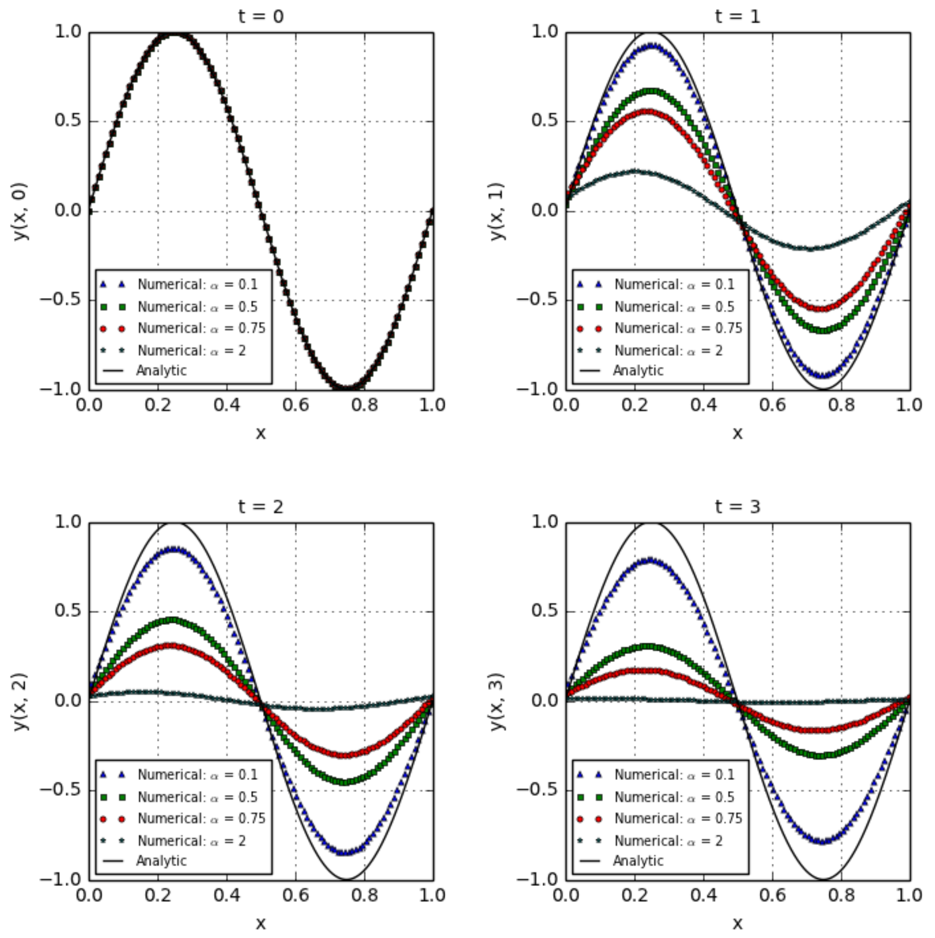

Figure 9: Snapshots of y at different times as a function of x using the BTCS scheme given a flow speed of 1 and and with varying α values. The BTCS scheme is much more stable than the FTBS or FTFS schemes for any choice of α, though numerical diffusion may be prominent.

Problem 8 continued on next page. . .

To apply a von Neumann stability analysis on the BTCS scheme we substitute the initial the initial state at $t = 0$

$$y_k^0 = \sin[2\pi k\Delta x] \tag{36}$$

which we then substitute into the BTCS scheme

$$y_k^{n-1} = y_k^n + a(t^n - t^n)\ \frac{y_{k+1}^n - y_{k-1}^n}{x_{k+1} - x_{k-1}} = y_k^n + \frac{a\Delta t}{2\Delta x}\ (y_{k+1}^n - y_{k-1}^n) \tag{37}$$

to get the solution before a single time-step we assumed an ansatz of

$$y_k^n = A^n e^{ki2\pi\Delta x} \tag{38}$$

and substituted this into the BTCS scheme to get

$$A^{n-1}e^{ki2\pi\Delta x} = A^n e^{ki2\pi\Delta x} + \frac{\alpha}{2}[A^n e^{(k+1)i2\pi\Delta x} + A^n e^{(k-1)i2\pi\Delta x}] \tag{39}$$

$$A^n A^{-1} e^{ki2\pi\Delta x} = A^n e^{ki2\pi\Delta x}[1 + \frac{\alpha}{2}[e^{2\pi\Delta x} + e^{-2\pi\Delta x}]] \tag{40}$$

leading to a solution for the amplification factor A

$$A = \frac{1}{1 + \frac{\alpha}{2}[e^{2\pi\Delta x} - e^{-2\pi\Delta x}]} \tag{41}$$

$$A = \frac{1}{1 + \frac{\alpha}{2}2i\sin[2\pi\Delta x]} \tag{42}$$

whose modulus is

$$A = \frac{1}{\sqrt{1 + \alpha^2\sin^2[2\pi\Delta x]}} \tag{43}$$

in which we substitute Equation 22 to get

$$A = \frac{1}{\sqrt{1 + \alpha^2(2G - G^2)}} \tag{44}$$

Finally, for any given number of time steps, n, using the Backward-Time Center-Space scheme we will have

$$y_k^n = A^n e^{ki2\pi\Delta x} \tag{45}$$

where

$$A = \frac{1}{\sqrt{1 + \alpha^2(2G - G^2)}} \tag{46}$$

For any choice of α and G, $A < 1$. Figure 10 demonstrates this result for the BTCS scheme.

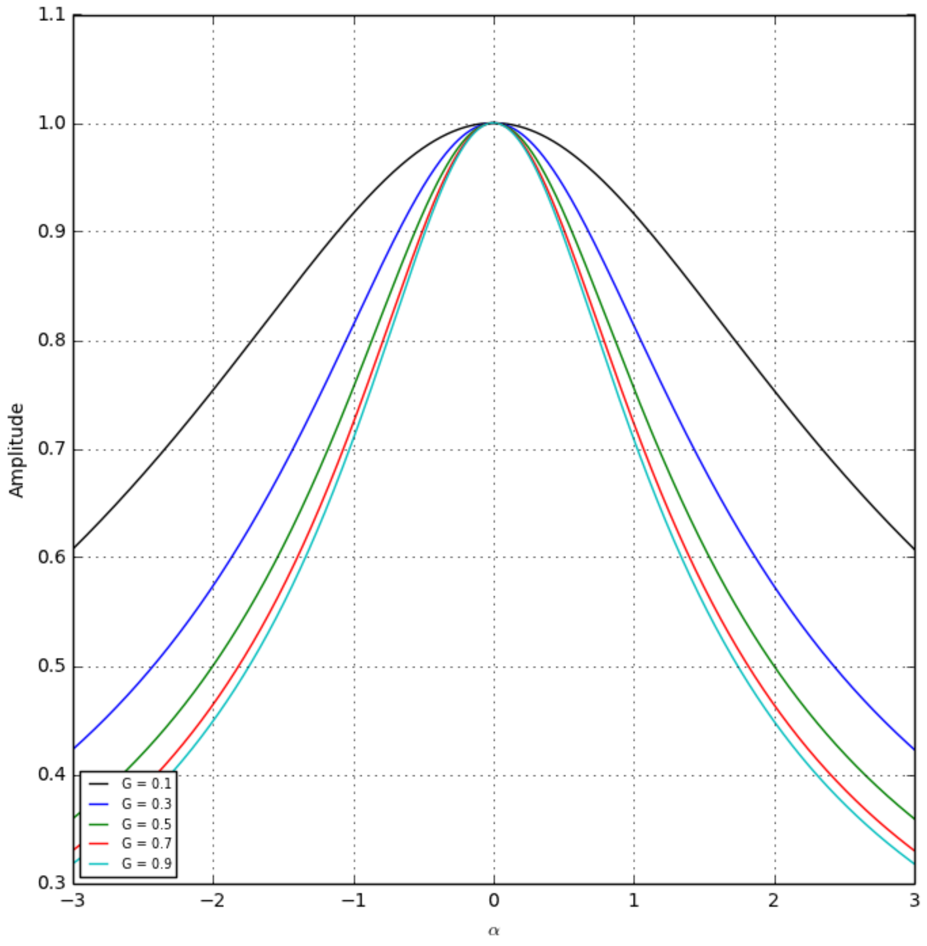

Figure 10: Amplitudes of BTCS scheme as a function of α for different G values (see Equation 46). The amplitude of the flow will remain finite as a function of time in the BTCS scheme for all values of α.

Simulation Code

```
#!/usr/bin/python

import numpy as np

class IVP_simulation():

    def __init__(self, scheme='FTCS', boundary_type='periodic',
       flow_speed=1,
            x_range=(0,1), t_range=(0,3), delta_x=0.01, delta_t
               =0.005,
            initial_condition=0):

        self.scheme = scheme
        self.boundary_type = boundary_type
        self.flow_speed = flow_speed
        self.x_range = x_range
        self.t_range = t_range
        self.delta_x = delta_x
        self.delta_t = delta_t
        self.grid = self.__create_grid(x_range, t_range, delta_x,
            delta_t)
        self.x_grid = self.__calc_x_grid(x_range, delta_x)
        self.t_grid = self.__calc_time_grid(t_range, delta_t)
        self.inital_condition = \
                self.__set_initial_conditions(initial_condition)
        self.t = 0

    def __create_grid(self, x_range, t_range, delta_x, delta_t):
        x_size = (x_range[1] - x_range[0]) / delta_x
        t_size = (t_range[1] - t_range[0]) / delta_t

        return np.empty((x_size, t_size))

    def __set_initial_conditions(self, initial_condition):
        # initial condition at time = 0
        self.__set_y(initial_condition(self.x_grid), 0)

    def __get_y(self, time):
        return self.grid[:,self.__get_time_index(time)]

    def __set_y(self, x_values, time):
        self.grid[:,self.__get_time_index(time)] = x_values

    def __get_time_index(self, time):
        return np.abs(self.t_grid - time).argmin()

    def __calc_x_grid(self, x_range, delta_x):
        x_size = (self.x_range[1] - self.x_range[0]) / self.
           delta_x
        x_grid = np.linspace(self.x_range[0], self.x_range[1],
           x_size)
        return x_grid

    def __calc_time_grid(self, t_range, delta_t):
        time_size = (self.t_range[1] - self.t_range[0]) / self.
           delta_t
        time_grid = np.linspace(self.t_range[0], self.t_range[1],
            time_size)
        return time_grid

    def run_simulation(self, scheme=None, boundary_type=None,
       flow_speed=None,
            x_range=None, t_range=None, delta_x=None, delta_t=
               None,
            initial_condition=None):

        '''
        Parameters
        ----------
        scheme : str
            Scheme for time steps. Options are:
                'FTCS' : Forward Time Center Space
                'FTBS' : Forward Time Backward Space
                'FTFS' : Forward Time Forward Space
        boundary_type : str
            Type of boundary condition solution. Options are:
                'periodic' : periodic
        flow_speed : float
            The flow speed.
        x_range : tuple
            Spatial range.
        t_range : tuple
            Time range.
        delta_x : float
            Spatial resolution.
        delta_t : float
            Time resolution.

        '''

        # Check for set parameters
        change_grid = False
        if scheme is not None:
            self.scheme = scheme
        if boundary_type is not None:
            self.boundary_type = boundary_type
        if flow_speed is not None:
            self.flow_speed = flow_speed
        if x_range is not None:
            self.x_range = x_range
            change_grid = True
        if t_range is not None:
            self.t_range = t_range
            change_grid = True
        if delta_x is not None:
            self.delta_x = delta_x
            change_grid = True
        if delta_t is not None:
            self.delta_t = delta_t
            change_grid = True
        if change_grid:
            self.grid = __create_grid(self.x_range, self.t_range,
                self.delta_x, self.delta_t)
            self.x_grid = self.__calc_x_grid(x_range, delta_x)
            self.t_grid = self.__calc_time_grid(t_range, delta_t)
        if initial_condition is not None:
            self.initial_condition = initial_condition

        # Complete simulation
        if self.scheme.lower() == 'ftcs':
            for i in range(len(self.t_grid)):
                self.step_FTCS()
        elif self.scheme.lower() == 'ftbs':
            for i in range(len(self.t_grid)):
                self.step_FTBS()
        elif self.scheme.lower() == 'ftfs':
            for i in range(len(self.t_grid)):
                self.step_FTFS()
        elif self.scheme.lower() == 'btcs':
            for i in range(len(self.t_grid)):
                self.step_BTCS()

    def step_FTCS(self):

        # get y values at all x at one time
        y = self.__get_y(self.t)

        # initialize next time step
        y_tp1 = np.empty((y.shape))

        for k in range(len(self.x_grid)):
            if k == len(self.x_grid) - 1:
                y_kp1 = y[0]
                x_kp1 = self.x_grid[k] + self.delta_x
                y_km1 = y[k - 1]
                x_km1 = self.x_grid[k - 1]
            elif k == 0:
                y_km1 = y[len(self.x_grid) - 1]
                x_km1 = self.x_grid[k] - self.delta_x
                y_kp1 = y[k + 1]
                x_kp1 = self.x_grid[k + 1]
            else:
                y_kp1 = y[k + 1]
                y_km1 = y[k - 1]
                x_kp1 = self.x_grid[k + 1]
                x_km1 = self.x_grid[k - 1]

            y_tp1[k] = y[k] - self.flow_speed * (self.delta_t) *
              \
                    (y_kp1 - y_km1) / (x_kp1 - x_km1)

        # Set next time step y values, and increase time
        self.__set_y(y_tp1, self.t + self.delta_t)
        self.t += self.delta_t

    def step_FTBS(self):
        # get y values at all x at one time
        y = self.__get_y(self.t)

        # initialize next time step
        y_tp1 = np.empty((y.shape))

        for k in range(len(self.x_grid)):
            if k == len(self.x_grid) - 1:
                y_kp1 = y[0]
                y_k = y[k]
                y_km1 = y[k - 1]
            elif k == 0:
                y_km1 = y[-1]
                y_k = y[k]
                y_kp1 = y[k + 1]
            else:
                y_kp1 = y[k + 1]
                y_km1 = y[k - 1]
                y_k = y[k]

            y_tp1[k] = y[k] - self.flow_speed * (self.delta_t) *
              \
                    (y_k - y_km1) / self.delta_x

        # Set next time step y values, and increase time
        self.__set_y(y_tp1, self.t + self.delta_t)
        self.t += self.delta_t
    def step_FTFS(self):
        # get y values at all x at one time
        y = self.__get_y(self.t)

        # initialize next time step
        y_tp1 = np.empty((y.shape))

        for k in range(len(self.x_grid)):
            if k == len(self.x_grid) - 1:
                y_kp1 = y[0]
                x_kp1 = self.x_grid[k] + self.delta_x
                y_k = y[k]
                x_k = self.x_grid[k]
                y_km1 = y[k - 1]
                x_km1 = self.x_grid[k - 1]
            elif k == 0:
                y_km1 = y[len(self.x_grid) - 1]
                x_km1 = self.x_grid[k] - self.delta_x
                y_k = y[k]
                x_k = self.x_grid[k]
                y_kp1 = y[k + 1]
                x_kp1 = self.x_grid[k + 1]
            else:
                y_kp1 = y[k + 1]
                y_km1 = y[k - 1]
                y_k = y[k]
                x_k = self.x_grid[k]
                x_kp1 = self.x_grid[k + 1]
                x_km1 = self.x_grid[k - 1]

            y_tp1[k] = y[k] - self.flow_speed * (self.delta_t) *
               \
                    (y_kp1 - y_k) / (x_kp1 - x_k)

        # Set next time step y values, and increase time
        self.__set_y(y_tp1, self.t + self.delta_t)
        self.t += self.delta_t

    def step_BTCS(self):

        ''' Backward-time Center-Space scheme. Solves using
            linear algebra.
        '''
        # get y values at all x at one time
        y = self.__get_y(self.t)

        # initialize next time step
        y_tp1 = np.empty((y.shape))
        K = len(self.x_grid)
        M = np.zeros((K,K))
        alpha = self.flow_speed * self.delta_t / self.delta_x

        for i in range(K):
            for j in range(K):
                if i == j:
                    M[i,j] = 1
                    if i==0:
                        M[i+1,j] = -alpha
                        M[-1,j] = alpha
                    elif i==K-1:
                        M[0,j] = -alpha
                        M[i-1,j] = alpha
                    else:
                        M[i+1,j] = -alpha
                        M[i-1,j] = alpha

        #
        y_tp1 = np.linalg.solve(M, y)

        # Set next time step y values, and increase time
        self.__set_y(y_tp1, self.t + self.delta_t)
        self.t += self.delta_t

        return None

    def plot_slice(self, times, limits=None, savedir='./',
       filename=None,
            show=True, title='', additional_sims=None,
               additional_labels=None):

        ''' Plots
        '''

        # Import external modules
        import numpy as np
        import math
        import pyfits as pf
        import matplotlib.pyplot as plt
        import matplotlib
        from mpl_toolkits.axes_grid1 import ImageGrid

        # Set up plot aesthetics
        plt.clf()
        plt.rcdefaults()
        fontScale = 10
        params = {#'backend': 'png',
                  'axes.labelsize': fontScale,
                  'axes.titlesize': fontScale,
                  'text.fontsize': fontScale,
                  'legend.fontsize': fontScale*3/4,
                  'xtick.labelsize': fontScale,
                  'ytick.labelsize': fontScale,
                  'font.weight': 500,
                  'axes.labelweight': 500,
                  'text.usetex': False,
                  'figure.figsize': (8, 8),
                 }
        plt.rcParams.update(params)

        # Create figure
        fig = plt.figure()
        grid = ImageGrid(fig, (1,1,1),
                      nrows_ncols=(2,len(times)/2),
                      ngrids = len(times),
                      direction='row',
                      axes_pad=1,
                      aspect=False,
                      share_all=True,
                      label_mode='All')

        x_analytic = np.linspace(self.x_range[0],self.x_range
           [1],1e5)
        def calc_y(x, time):
            return np.sin(2*np.pi * (x - self.flow_speed * time))

        # save current grid if additional ones being plotted
        grid_save = self.grid

        markers = ['s','o','*']

        for i, time in enumerate(times):
            ax = grid[i]

            if additional_sims is None:
                ax.plot(self.x_grid, self.__get_y(time),
                        color='k',
                        markersize=3,
                        marker='o',
                        linestyle='None',
                        label='Numerical')
            elif additional_sims is not None:
                ax.plot(self.x_grid, self.__get_y(time),
                        #color='k',
                        markersize=3,
                        marker='^',
                        linestyle='None',
                        label='Numerical: %s' % additional_labels
                           [0])
                for j, sim in enumerate(additional_sims):
                    ax.plot(sim.x_grid, sim.__get_y(time),
                            #color='k',
                            markersize=3,
                            marker=markers[j],
                            linestyle='None',
                            label='Numerical: %s' %
                               additional_labels[j+1])

            # plot analytic solution
            y_analytic = calc_y(x_analytic, time)
            ax.plot(x_analytic, y_analytic,
                    color='k',
                    linestyle='-',
                    label='Analytic')

            if limits is not None:
                ax.set_xlim(limits[0],limits[1])
                ax.set_ylim(limits[2],limits[3])

            # Adjust asthetics
            ax.set_xlabel('x',)
            ax.set_ylabel(r'y(x, %s)' % time)
            ax.grid(True)
            ax.legend(loc='lower left')
            ax.set_title('t = %s' % time)

        # reset grid
        self.grid = grid_save

        if filename is not None:
            plt.savefig(savedir + filename,bbox_inches='tight')
        if show:
            fig.show()

    def plot_amplitude(alpha_array = (-1,1), G_values = (1),
       amp_functions = None,
            limits=None, savedir='./', filename=None, show=True,
               title=''):

        ''' Plots

        amp_functions = tuple of functions
        '''

        # Import external modules
        import numpy as np
        import pyfits as pf
        import matplotlib.pyplot as plt
        import matplotlib
        from mpl_toolkits.axes_grid1 import ImageGrid

        # Set up plot aesthetics
        plt.clf()
        plt.rcdefaults()
        fontScale = 10
        params = {#'backend': 'png',
                  'axes.labelsize': fontScale,
                  'axes.titlesize': fontScale,
                  'text.fontsize': fontScale,
                  'legend.fontsize': fontScale*3/4,
                  'xtick.labelsize': fontScale,
                  'ytick.labelsize': fontScale,
                  'font.weight': 500,
                  'axes.labelweight': 500,
                  'text.usetex': False,
                  'figure.figsize': (8, 8),
                 }
        plt.rcParams.update(params)

        size = len(amp_functions)

        # Create figure
        fig = plt.figure()
        grid = ImageGrid(fig, (1,1,1),
                         nrows_ncols=(size,1),
                         ngrids = size,
                         direction='row',
                         axes_pad=1,
                         aspect=False,
                         share_all=True,
                         label_mode='All')

        colors = ['k','b','g','r','c']
        linestyles = ['-','--','-.','-','-']
        letters = ['a','b']

        for i, amp_function in enumerate(amp_functions):
            ax = grid[i]

            for j, G in enumerate(G_values):
                ax.plot(alpha_array, amp_function(alpha_array, G)
                    ,
                        color = colors[j],
                        label = 'G = %s' % G,
                        linestyle = '-')
            if limits is not None:
                ax.set_xlim(limits[0],limits[1])
                ax.set_ylim(limits[2],limits[3])

            # Adjust asthetics
            ax.set_xlabel(r'$\alpha$',)
            ax.set_ylabel(r'Amplitude')
            if size > 1:
                ax.annotate('(%s)' % letters[i],
                        xy = (0.9, 0.1),
                        xycoords='axes fraction',
                        textcoords='axes fraction')
            ax.grid(True)
            ax.legend(loc='lower left')

        if filename is not None:
            plt.savefig(savedir + filename,bbox_inches='tight')
        if show:
            fig.show()

def problem_1():
    def initial_condition(x):
        return np.sin(2 * np.pi * x)

    ftcs_sim_ftbs = IVP_simulation(scheme = 'FTCS',
            boundary_type = 'periodic',
            flow_speed = 1,
            x_range = (0,1),
            t_range = (0,3),
            delta_x = 0.01,
            delta_t = 0.005,
            initial_condition = initial_condition)

    ftcs_sim_ftbs.run_simulation()

    savedir = '/home/elijah/class_2014_spring/fluids/midterm/'
    savedir = '/usr/users/ezbc/Desktop/fluids/midterm/'
    times = [0, 1, 2, 3,]
    ftcs_sim_ftbs.plot_slice(times, savedir=savedir,
                filename='q1_ftcs.png',
                title = 'FTCS simulation slices',
                show=False)

def problem_2():
    def initial_condition(x):
        return np.sin(2 * np.pi * x)

    ftbs_sim = IVP_simulation(scheme = 'FTBS',
            boundary_type = 'periodic',
            flow_speed = 1,
            x_range = (0,1),
            t_range = (0,3),
            delta_x = 0.01,
            delta_t = 0.005,
            initial_condition = initial_condition)

    ftfs_sim = IVP_simulation(scheme = 'FTFS',
            boundary_type = 'periodic',
            flow_speed = 1,
            x_range = (0,1),
            t_range = (0,3),
            delta_x = 0.01,
            delta_t = 0.005,
            initial_condition = initial_condition)

    ftbs_sim.run_simulation()
    ftfs_sim.run_simulation()

    savedir = '/home/elijah/class_2014_spring/fluids/midterm/'
    savedir = '/usr/users/ezbc/Desktop/fluids/midterm/'

    times = [0, 1, 2, 3,]
    ftbs_sim.plot_slice(times, savedir=savedir,
                filename='q2_ftbs.png',
                title = 'FTBS simulation slices',
                show=False)
    ftfs_sim.plot_slice(times, savedir=savedir,
                filename='q2_ftfs.png',
                title = 'FTFS simulation slices',
                show=False)

def problem_3():
    def initial_condition(x):
        return np.sin(2 * np.pi * x)

    ftbs_sim = IVP_simulation(scheme = 'FTBS',
            boundary_type = 'periodic',
            flow_speed = -1,
            x_range = (0,1),
            t_range = (0,3),
            delta_x = 0.01,
            delta_t = 0.005,
            initial_condition = initial_condition)

    ftfs_sim = IVP_simulation(scheme = 'FTFS',
            boundary_type = 'periodic',
            flow_speed = -1,
            x_range = (0,1),
            t_range = (0,3),
            delta_x = 0.01,
            delta_t = 0.005,
            initial_condition = initial_condition)

    ftbs_sim.run_simulation()
    ftfs_sim.run_simulation()

    savedir = '/home/elijah/class_2014_spring/fluids/midterm/'
    savedir = '/usr/users/ezbc/Desktop/fluids/midterm/'
    times = [0, 1, 2, 3,]
    ftbs_sim.plot_slice(times, savedir=savedir,
                filename='q3_ftbs.png',
                title = 'FTBS simulation slices',
                show=False)
    ftfs_sim.plot_slice(times, savedir=savedir,
                filename='q3_ftfs.png',
                title = 'FTFS simulation slices',
                show=False)

def problem_5():

    def amp1(alpha, G):
        return (1 + 2*G*alpha*(alpha + 1))**0.5

    def amp2(alpha, G):
        return (1 + 2*G*alpha*(alpha - 1))**0.5

    amp_tuple = (amp1, amp2)
    alpha_array = np.linspace(-3, 3, 1e4)
    G_values = (0.1, 0.3, 0.5, 0.7, 0.9)

    savedir = '/usr/users/ezbc/Desktop/fluids/midterm/'

    plot_amplitude(alpha_array = alpha_array, G_values = G_values
       ,
            amp_functions = amp_tuple,
            limits = [-2, 2, 0.7, 1.2],
            savedir = savedir,
            filename = 'q5.png',
            title = 'FTFS and FTBS Amplitudes',
            show=False)

def problem_6():
    def initial_condition(x):
        return np.sin(2 * np.pi * x)

    ftbs_sim = IVP_simulation(scheme = 'FTBS',
            boundary_type = 'periodic',
            flow_speed = 1,
            x_range = (0,1),
            t_range = (0,3),
            delta_x = 0.01,
            delta_t = 0.02,
            initial_condition = initial_condition)

    ftbs_sim.run_simulation()

    savedir = '/home/elijah/class_2014_spring/fluids/midterm/'
    savedir = '/usr/users/ezbc/Desktop/fluids/midterm/'
    times = [0, 1, 2, 3,]
    ftbs_sim.plot_slice(times, savedir=savedir,
                filename='q6_ftbs.png',
                title = 'FTBS simulation slices',
                show=False)

def problem_7():
    def initial_condition(x):
        return np.sin(2 * np.pi * x)

    alphas = (0.25, 0.5, 0.75)
    delta_x = 0.01
    flow_speed = 1
    x_range = (0,1)
    t_range = (0,3)

    sim_list = []
    additional_labels = []

    savedir = '/home/elijah/class_2014_spring/fluids/midterm/'
    savedir = '/usr/users/ezbc/Desktop/fluids/midterm/'
    times = [0, 1, 2, 3,]

    for i, alpha in enumerate(alphas):
        delta_t = np.abs(alpha * delta_x)

        ftbs_sim = IVP_simulation(scheme = 'FTBS',
                boundary_type = 'periodic',
                flow_speed = flow_speed,
                x_range = x_range,
                t_range = t_range,
                delta_x = delta_x,
                delta_t = delta_t,
                initial_condition = initial_condition)

        ftbs_sim.run_simulation()
        sim_list.append(ftbs_sim)

        additional_labels.append(r'$\alpha$ = %s' % alpha)

    sim_list[0].plot_slice(times, savedir=savedir,
                filename='q7.png',
                title = 'FTBS simulation slices',
                show=False,
                additional_sims = sim_list[1:],
                additional_labels = additional_labels)

def problem_8():
    def initial_condition(x):
        return np.sin(2 * np.pi * x)

    alphas = (0.1, 0.5, 0.75, 2)
    delta_x = 0.01
    flow_speed = 1
    x_range = (0,1)
    t_range = (0,3)

    sim_list = []
    additional_labels = []

    savedir = '/home/elijah/classes/fluids/midterm/'
    savedir = '/usr/users/ezbc/classes/fluids/midterm/'

    times = [0, 1, 2, 3,]

    for i, alpha in enumerate(alphas):
        delta_t = np.abs(alpha * delta_x)

        if alpha < 0:
            flow_speed = -1

        btcs_sim = IVP_simulation(scheme = 'BTCS',
                boundary_type = 'periodic',
                flow_speed = flow_speed,
                x_range = x_range,
                t_range = t_range,
                delta_x = delta_x,
                delta_t = delta_t,
                initial_condition = initial_condition)

        btcs_sim.run_simulation()

        sim_list.append(btcs_sim)

        additional_labels.append(r'$\alpha$ = %s' % alpha)
    sim_list[0].plot_slice(times, savedir=savedir,
                filename='q8.png',
                title = 'BTCS simulation slices',
                show=False,
                additional_sims = sim_list[1:],
                additional_labels = additional_labels)

    def amp(alpha, G):
        return 1 / (1 + alpha**2*(2*G - G**2))**0.5

    amp_tuple = (amp,)
    alpha_array = np.linspace(-3, 3, 1e4)
    G_values = (0.1, 0.3, 0.5, 0.7, 0.9)


    plot_amplitude(alpha_array = alpha_array, G_values = G_values
       ,
            amp_functions = amp_tuple,
            limits = [-3, 3, 0.3, 1.1],
            savedir = savedir,
            filename = 'q8_amp.png',
            title = 'BTCS Amplitudes',
            show=False)

def main():
    problem_1()
    problem_2()
    problem_3()
    problem_5()
    problem_6()
    problem_7()
    problem_8()

if __name__ == '__main__':
    main()
```