Problem 1. *Consider the advection equation*

$$u_t + au_x = 0$$

on the interval [0,1) with periodic boundary conditions. Space is discretized as $x_j = j\Delta x$ for $j = 0, \dots, N-1$, so that $\Delta x = 1/N$. Discretize the spatial derivative with the second-order centered difference operator.

(a) For simplicity, assume N is odd. The eigenvectors of the centered difference operator are

$$v_j^k = \exp(2\pi i k x_j),$$

for $k = -(N-1)/2, \dots, (N-1)/2$. Compute the eigenvalues.

$\implies$ With centered difference operator

$$Du_j = \frac{u_{j+1} - u_{j-1}}{2\Delta x},$$

we have that

$$\begin{aligned} Dv_j^k &= \lambda_k v_j^k \\ \frac{v_{j+1}^k - v_{j-1}^k}{2\Delta X} &= \lambda_k v_j^k \\ \frac{e^{2\pi i k (x_j + \Delta x)} - e^{2\pi i k (x_j - \Delta x)}}{2\Delta x} &= \lambda_k v_j^k \\ \left(\frac{e^{2\pi i k \Delta x} - e^{-2\pi i k \Delta x}}{2\Delta x} \right) v_j^k &= \lambda_k v_j^k \\ \left(\frac{i \sin(2\pi k \Delta x)}{\Delta x} \right) v_j^k &= \lambda_k v_j^k. \end{aligned}$$

Hence the eigenvalues are

$$\lambda_k = \frac{i \sin(2\pi k \Delta x)}{\Delta x}.$$

(b) Derive a time step restriction on a method-of-lines approach which uses classical fourth-order Runge-Kutta for time stepping.

The fourth-order Runge-Kutta scheme applied to $y' = \lambda y$ is

$$\begin{aligned} y_1^* &= y^n \\ y_2^* &= y^n \left(1 + \frac{\Delta t}{2} \lambda \right) \\ y_3^* &= y^n \left(1 + \frac{\Delta t}{2} \lambda \left(1 + \frac{\Delta t}{2} \lambda \right) \right) \\ y_4^* &= y^n \left(1 + \Delta t \lambda \left(1 + \frac{\Delta t}{2} \lambda \left(1 + \frac{\Delta t}{2} \lambda \right) \right) \right) \end{aligned}$$

$$\begin{aligned} y^{n+1} &= y^n + \frac{\Delta t\lambda}{6}(y_1^* + 2y_2^* + 2y_3^* + y_4^*) \\ &= y^n\left(1 + \Delta t\lambda + \frac{(\Delta t\lambda)^2}{2} + \frac{(\Delta t\lambda)^3}{3!} + \frac{(\Delta t\lambda)^4}{4!}\right). \end{aligned}$$

So for $z = \Delta t\lambda$, the region of stability for the 4th order Runge-Kutta scheme is

$$\left|1 + z + z^2/2 + z^3/3! + z^4/4!\right| \leq 1.$$

This translates to

$$\text{Re}(z) = 0,\ -2\sqrt{2} \leq \text{Im}(z) \leq 2\sqrt{2}.$$

If z is pure imaginary, then this is equivalent to

$$|z| \leq 2\sqrt{2}.$$

The discrete-space advection equation,

$$u_t = -aD_c u,$$

has eigenvalues

$$\lambda_k = -a\frac{i\sin(2\pi k\Delta x)}{\Delta x}.$$

Since z is pure imaginary,

$$z = -a\Delta t\lambda_k = -ai\frac{\Delta t}{\Delta x}\sin(2\pi k\Delta x) \implies |z| \leq \frac{a\Delta t}{\Delta x}.$$

Combining this with $|z| \leq 2\sqrt{2}$, we obtain the requirement that

$$\Delta t \leq \frac{2\sqrt{2}\Delta x}{a}$$

for z to be in the region of stability and the scheme to be stable.

Problem 2. *Consider the following PDE*

$$u_t = 0.01u_{xx} + 1 - \exp(-t),\ \ 0 < x < 1$$
$$u(0,t) = 0,\ \ u(1,t) = 0$$
$$u(x,0) = 0.$$

Write a program to solve the problem using Crank-Nicolson up to time $t = 1$, and perform a refinement study that demonstrates that the method is second-order accurate in space and time.

I implemented the Crank-Nicolson method to solve the above problem, and carried out a refinement study to demonstrate that the method is second-order accurate in space and time. Without the solution to the above problem, I considered the ratios of successive differences as opposed to errors. For $\Delta x[0] = 2^{-2}, \Delta t[0] = 2^{-1}$, I used Crank-Nicolson to solve the problem for $u(x,1)$. Then I halved Δx and Δt to get $\Delta x[1], \Delta t[1]$, and compared the next computed $u(x,1)$, u_{new} with the previous $u(x,1)$, u_{old}. I compared them pointwise with a simple restriction on u_{new} to match with the points of u_{old}, and took the discrete 1-norm of the difference:

$$d_1 = \Delta x[0] \|\text{restrict}(u_{new}) - u_{old}\|_1$$

To demonstrate second-order accuracy, I then computed the ratios of these successive differences:

$$\frac{d_1[i]}{d_1[i+1]} = 2\frac{\|\text{restrict}(u[i]) - u[i-1]\|_1}{\|\text{restrict}(u[i+1]) - u[i]\|_1}.$$

The refinement study, as shown in Table 1, shows that halving both space and time steps results in a four-fold decrease in successive differences, suggesting that the method itself is second-order accurate in space and time.

Table 1: Refinement Study for Crank-Nicolson, Successive Differences Approach

Δx	Δt	d_1	$d_1[i]/d_1[i+1]$
2^{-2}	2^{-1}	—	—
2^{-3}	2^{-2}	0.0132258	—
2^{-4}	2^{-3}	0.00575532	3.30498
2^{-5}	2^{-4}	0.00174141	3.82538
2^{-6}	2^{-5}	0.000455224	3.95694
2^{-7}	2^{-6}	0.000115045	3.98928
2^{-8}	2^{-7}	2.88385e-05	3.99732
2^{-9}	2^{-8}	7.21445e-06	3.99933
2^{-10}	2^{-9}	1.80391e-06	3.99983

To further confirm the accuracy of my program, I also performed a refinement study on the problem

$$u_t = u_{xx},\ 0 < x < 1$$
$$u(0,t) = 0,\ \ u(1,t) = 0$$

$$u(x{,}0) = \sin(\pi x),$$

which has analytic solution $u_{sol}(x,t) = e^{-\pi^2 t}\sin(\pi x)$.

I solved this problem using the Crank-Nicolson method for $u(x,1)$ for the space and time steps given in Table 2, and compared it with the analytic solution at $t = 1$ sampled on the corresponding grid points to obtain the error. Again, I used the discrete 1-norm to compare the solutions

$$e = \Delta x \|u - u_{sol}\|_1.$$

Table 2: Refinement Study for Crank-Nicolson, Errors compared to Analytic solution

Δx	Δt	e	$\frac{e[i]}{e[i+1]}$
2^{-2}	2^{-1}	0.0973895	—
2^{-3}	2^{-2}	2.60705e-05	0.995771
2^{-4}	2^{-3}	2.61812e-05	2.88008
2^{-5}	2^{-4}	9.09046e-06	3.70995
2^{-6}	2^{-5}	2.45029e-06	3.92717
2^{-7}	2^{-6}	6.23933e-07	3.98178
2^{-8}	2^{-7}	1.56697e-07	3.99544
2^{-9}	2^{-8}	3.9219e-08	3.99886
2^{-10}	2^{-9}	9.80753e-09	3.9997

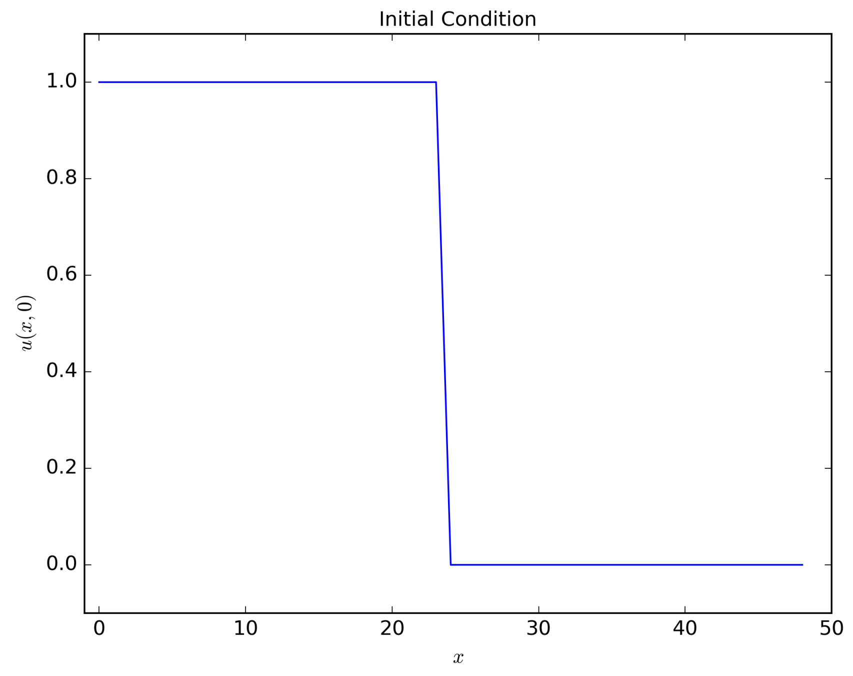
Again, the reduction in Δx and Δt by a factor of 2 led to a reduction in error by a factor of about 4, so my Crank-Nicolson method is indeed second-order accurate in space and time.

Problem 3. *Consider the following PDE*

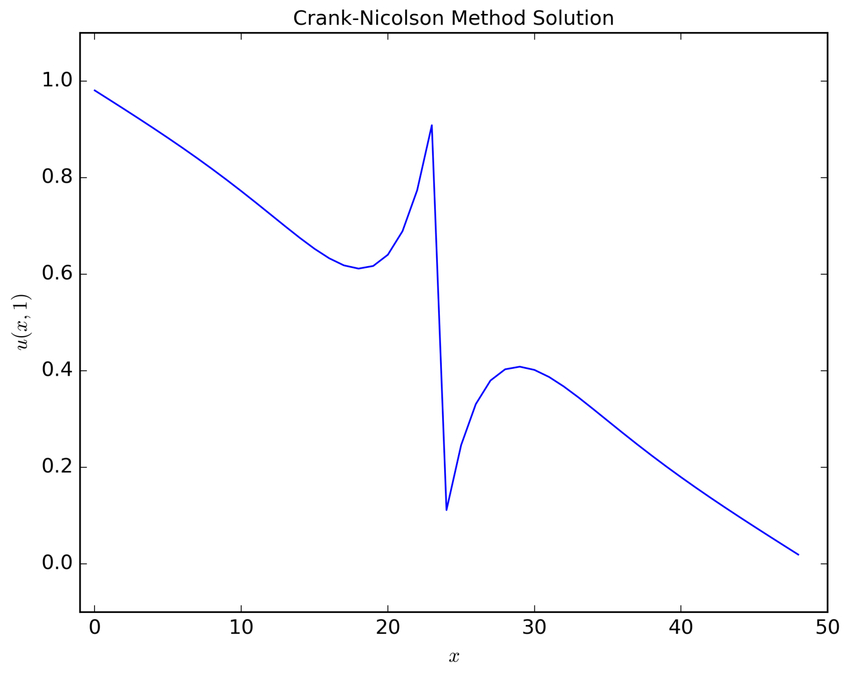
$$u_t = u_{xx}, \quad 0 < x < 1$$
$$u(0,t) = 1, \quad u(1,t) = 0$$
$$u(x,0) = \begin{cases} 1, & \text{if } x < 0.5 \\ 0, & \text{if } x \geq 0.5. \end{cases}$$

(a) Use Crank-Nicolson with grid spacing $\Delta x = 0.02$ and time step $\Delta t = 0.1$ to solve the problem up to time $t = 1$. Comment on your results. What is wrong with this solution?

The initial condition for this problem is given in the following figure:



With Crank-Nicolson with grid spacing $\Delta x = 0.02$ and time step $\Delta t = 0.1$, the solution at time $t = 1$ develops a singularity about the discontinuity at $x = 0.5$ from the initial condition. This is wrong, as the solution should smooth out and become a straight line between the boundary conditions.



(b) Give a mathematical argument to explain the unphysical behavior you observed in the numerical solution.

The Crank-Nicolson scheme is

$$u^{n+1} = \left(I - \frac{\Delta t}{2}L\right)^{-1}\left(I + \frac{\Delta t}{2}L\right)u^n + b = Bu^n + b.$$

Then

$$\|B\|_2 = \max_k \left|\frac{1+\Delta t\lambda_k/2}{1-\Delta t\lambda_k/2}\right|,$$

where λ_k are the eigenvalues of the difference operator L:

$$\lambda_k = \frac{-4D}{\Delta x^2}\sin^2\left(\frac{k\pi\Delta x}{2}\right), \quad k = 1\ldots, N.$$

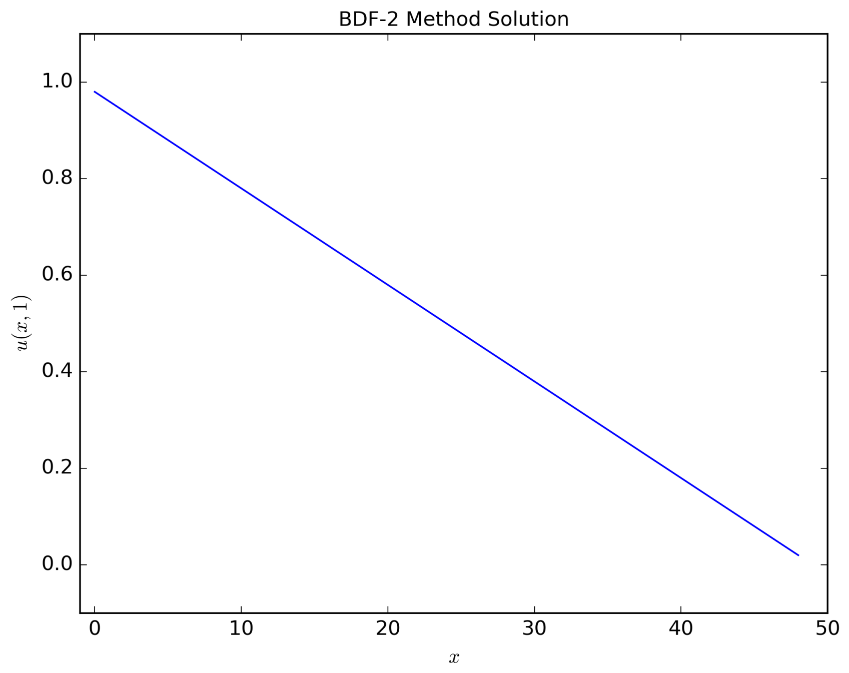
Then for $D = 1$, $\Delta t = 0.1, \Delta x = 0.02$, $N = 1/\Delta x - 1 = 49$ as in part (a),

$$\|B\|_2 = \max_{k=1,\ldots,49}\left|\frac{1-500\sin^2(0.01\pi k)}{1+500\sin^2(0.01\pi k)}\right| = \left|\frac{499}{501}\right| \approx 1.$$

Thus while the scheme is stable, it is very close to unstable so solutions that are supposed to decay will decay very slowly. The number of eigenvalues of B that are close to -1, however, induce oscillations at discontinuities, since successive iterations will have opposite signs. Since the scheme is stable, the oscillations will eventually dampen, however, but very slowly due to the fact that the norm is close to 1 in magnitude.

(c) Repeat the simulation using BDF2, and discuss why the unphysical behavior is not present in the numerical solution for any time step.

I implemented BDF-2 with a starting step of Backwards Euler to get $u[1]$ from the initial condition. With BDF-2 and the same grid spacing $\Delta x = 0.02$, time step $\Delta t = 0.1$, the solution at time $t = 1$ looks smooth and linear as it should.



Starting BDF-2 with a single backwards Euler step, as I did in my implementation, we have

$$u^1 = (I - \Delta t L)^{-1} u^0 = \tilde{B} u^0.$$

Then

$$\left\|\tilde{B}\right\|_2 = \max_{k=1,\dots,N} \left| \frac{1}{1 + \frac{4\Delta t D}{\Delta x^2}\sin^2(\frac{k\pi\Delta x}{2})} \right| < 1,$$

so

$$\left\|u^1\right\|_2 \leq \left\|u^0\right\|_2.$$

The BDF-2 scheme is

$$\begin{aligned}(3I - 2\Delta t L)u^{n+1} &= 4u^n - u^{n-1}\\ u^{n+1} &= 4(3I - 2\Delta t L)^{-1}u^n - (3I - 2\Delta t L)^{-1}u^{n-1}.\\ u^{n+1} &= 4Bu^n - Bu^{n-1}.\end{aligned}$$

Then

$$\|B\|_2 = \max_{k=1,\dots,N} \left| \frac{1}{3 + \frac{8\Delta t D}{\Delta x^2}\sin^2(\frac{k\pi\Delta x}{2})} \right|,$$

so

$$0 \leq 3\|B\|_2 < 1.$$

Then
$$\|u^2\|_2 \leq 4\|B\|_2\|u^1\|_2 - \|B\|_2\|u^0\|_2 \leq 3\|B\|_2\|u^0\|_2 \leq \|u^0\|_2.$$
Similarly, for any step we have that
$$\|u^{n+1}\|_2 \leq \|u^{n-1}\|_2.$$
Thus the BDF-2 method is stable for any Δt, Δx, and since all its eigenvalues are nonnegative there are no oscillations in the solution.

Code

The code used for the refinement study in Problem 2 is as follows:

```
#Refinement_study.py

from __future__ import division
import numpy as np
from numpy import exp, sin, pi
from numpy.linalg import norm
import matplotlib.pyplot as plt
from tabulate import tabulate
from tqdm import tqdm
from crank_nicolson import crank_nicolson_method
from bdf2 import bdf2_method

def succ_diff_refinement_study():
  #perform a refinement study to demonstrate Crank-Nicolson
  #is second-order accurate in space and time
  #using successive differences
  #since no analytic soln

  #max number of del_x,del_t values to examine
  refine_MAX = 10

  #loop through del_x values
  del_x = [2**(-2-i) for i in range(0,refine_MAX)]
  del_t = [2**(-1-i) for i in range(0, refine_MAX)]

  #set container for successive differences
  diffs = np.zeros(refine_MAX)

  #get u(x,1) through Crank-Nicolson:
  u_new = run_problem(del_x[0], del_t[0])

  #loop over finer del_x, take successive differences
  for i in tqdm(range(1,refine_MAX)):
    #store previous iterate
    u_old = u_new + 0

    #get next u(x,1) through Crank-Nicolson
    u_new = run_problem(del_x[i], del_t[i])

    #calculate successive difference between u(x,1) new and old
    diffs[i] = del_x[i-1]*norm(restriction(u_new, del_x[i]) - u_old,1)

  #tabulate refinement study results
  two_norm_table = [[del_x[i], del_t[i], diffs[i], diffs[i]/diffs[i+1]] for i in range(refin
  print(tabulate(two_norm_table, headers=['delta x', 'delta t', 'diffs', 'diff ratios'], tab

def errors_refinement_study():
  #perform a refinement study to demonstrate Crank-Nicolson
  #is second-order accurate in space and time
  #using error analysis with known analytic soln

  #max number of del_x,del_t values to examine
  refine_MAX = 10

  #loop through del_x values
  del_x = [2**(-2-i) for i in range(0,refine_MAX)]
  del_t = [2**(-1-i) for i in range(0, refine_MAX)]

  #set container for successive differences
  errors = np.zeros(refine_MAX)

  #loop over finer del_x and del_t
  for i in tqdm(range(0,refine_MAX)):
    #get approx u(x,1) through Crank-Nicolson
    [u_approx, u_sol] = run_test(del_x[i], del_t[i])

    #calculate error between u(x,1) approx and known solution
    errors[i] = del_x[i]*norm(u_approx - u_sol,1)

  #tabulate refinement study results
  two_norm_table = [[del_x[i], del_t[i], errors[i], errors[i]/errors[i+1]] for i in range(re
  print(tabulate(two_norm_table, headers=['delta x', 'delta t', 'errors', 'error ratios'], t

def restriction(u_f, h):
  #simple restriction operation
  h2 = 2*h
  n2 = int(1/h2)-1
  u_c = np.zeros(n2, dtype=float)
  #loop over coarse mesh
  for i in range(0,n2):
    u_c[i] = u_f[2*i+1]

  return u_c

def run_test(del_x, del_t):
  #set up the vectors and parameters for Crank-Nicolson method and run
  #using diffusion coefficient, initial condition of test problem
  #u_t = u_xx
  #u(0,t)=u(1,t)=0
  #u(x,0)=sin(pi x)
  #which has solution u(x,t)=e^{-pi^2 t}sin(pi x)

  #make vector of forcing function at all times
  Nx = int(1/del_x)-1
  Nt = int(1/del_t)
  x = [i*del_x for i in range(1, Nx+1)]
  t = [i*del_t for i in range(Nt+1)]

  #f = 0
  f = [0*t for t in t]

  #initial condition u(x,0)=sin(pi x)
  u = [sin(pi*x) for x in x]

  #known solution u(x,t)=e^{-pi^2(0.01)t}sin(pi x) at t=1:
  u_sol = [exp(-pi**2)*sin(pi*x) for x in x]

  #diffusion coefficient
  D = 1
  u = crank_nicolson_method(del_x, del_t, u, f, D)
  return u, u_sol

def run_problem(del_x, del_t):
  #set up the vectors and parameters for Crank-Nicolson method and run
  #using diffusion coefficient, initial condition, forcing function from problem 2

  #make vector of forcing function at all times
  Nx = int(1/del_x)-1
  Nt = int(1/del_t)
  # x = [i*del_x for i in range(0, Nx+1)]
  t = [i*del_t for i in range(0, Nt+1)]

  #f = 1-exp(-t)
  neg_t = [-t for t in t]
  f = 1-exp(neg_t)

  #initial condition u(x,0)=0
  u = np.zeros(Nx)

  #diffusion coefficient
  D = 0.01
  u = crank_nicolson_method(del_x, del_t, u, f, D)
  return u

if __name__ == '__main__':
  errors_refinement_study()
  succ_diff_refinement_study()

#Crank-Nicolson.py

from __future__ import division
import numpy as np
import scipy.sparse as sparse
import scipy.sparse.linalg

def sparse_matrices(del_x):
  #set sparse matrix L, the discrete Laplacian
  #for 3-pt 2nd order approximation

  #Set number of grid points
  N = 1/del_x - 1

  #set off-diagonal Laplacian components
  offdiag = (1/(del_x**2))*np.ones(N)
  #set diagonal Laplacian components
  diag = np.ones(N)*(-2/(del_x**2))

  #put diagonals together into sparse matrix format
  data = np.vstack((offdiag, diag,offdiag))
  L = sparse.dia_matrix((data, [-1, 0,1]), shape = (N,N))

  #create identity matrix of same size as L
  I = sparse.identity(N)

  return L, I

def crank_nicolson_time_step(del_t, u, L, f, I):
  #one time step of crank-nicolson solver

  #(I + del_t/2 L)u^n
  A = (I + (del_t/2) * L)
  RHS_terms = A.dot(u) + del_t*f

  #make LHS matrix, put in CSC form for solver
  LHS_matrix = scipy.sparse.csc_matrix(I-(del_t/2)*L)

  #solve (I-del_t/2 L)u^n+1 = (I + del_t/2 L)u^n + del_t f^n+1/2
  u_next = scipy.sparse.linalg.spsolve(LHS_matrix, RHS_terms)

  return u_next

def crank_nicolson_method(del_x, del_t, u, f, D):

  #create sparse matrices for crank-nicolson method
  [L, I] = sparse_matrices(del_x)

  #calculate number of time points after 0 up to 1 (inclusive)
  Nt = int(1/del_t)

  for t in range(0,Nt):
    #take half point of f for solve
    f_half = (f[t]+f[t+1])/2
    #solve for next u
    u = crank_nicolson_time_step(del_t, u, D*L, f_half, I)

  return u
```

The code used for Problem 3 include some of the above methods and the following:

```
#Discontinuous_ic_issue.py

#Illustrate a problem with Crank-Nicolson method
#on a diffusion problem with discontinuous initial condition

from __future__ import division
import numpy as np
from numpy import exp
import matplotlib.pyplot as plt
from crank_nicolson import crank_nicolson_method
from bdf2 import bdf2_method

def illustrate_issue():
  #set up the vectors and parameters for Crank-Nicolson method and run
  #using diff coefficient, initial condition from problem 3
  #no forcing function, but include left BC as forcing function at x=0
  del_x = 0.02
  del_t = 0.1
  #make matrix of forcing function f=0 at all times and spaces
  Nx = int(1/del_x)-1
  Nt = int(1/del_t)
  f = np.zeros((Nt+1,Nx))
  #include LHS BC u(0,t)=1
  f[:,0] = 1/(del_x**2)*np.ones(Nt+1)

  #initial condition u(x,0)=1 if x<0.5, 0 if x>=0.5
  u = np.zeros(Nx)
  for i in range(int(Nx/2)):
    u[i] = 1

  #plot IC
  plt.plot(u)
  plt.title(r"Initial Condition", fontsize=12)
  plt.axis([-1,50, -0.1, 1.1])
  plt.xlabel(r"$x$")
  plt.ylabel("$u(x,0)$")
  plt.savefig("problem3_initial_condition.png", dpi=300)
  plt.show()
  plt.close()

  #diffusion coefficient
  D = 1
  u = crank_nicolson_method(del_x, del_t, u, f, D)

  #plot u(x,1)
  plt.plot(u)
  plt.title(r"Crank-Nicolson Method Solution", fontsize=12)
  plt.axis([-1,50, -0.1, 1.1])
  plt.xlabel(r"$x$")
  plt.ylabel("$u(x,1)$")
  plt.savefig("problem3_crank_nicolson_issue.png", dpi=300)
  plt.show()
  plt.close()

def fix_issue():
  #set up the vectors and parameters for BDF-2 method and run
  #using diff coefficient, initial condition from problem 3
  #no forcing function, but include left BC as forcing function at x=0
  del_x = 0.02
  del_t = 0.1

  #make matrix of forcing function f=0 at all times and spaces
  Nx = int(1/del_x)-1
  Nt = int(1/del_t)
  f = np.zeros((Nt+1,Nx))
  #include RHS BC u(0,t)=1
  f[:,0] = 1/(del_x**2)*np.ones(Nt+1)

  #initial condition u(x,0)=1 if x<0.5, 0 if x>=0.5
  u = np.zeros(Nx)
  for i in range(int(Nx/2)):
    u[i] = 1

  #plot IC
  plt.plot(u)
  plt.show()
  plt.close()

  #diffusion coefficient
  D = 1
  u = bdf2_method(del_x, del_t, u, f, D)

  #plot u(x,1)
  plt.plot(u)
  plt.title(r"BDF-2 Method Solution", fontsize=12)
  plt.axis([-1,50, -0.1, 1.1])
  plt.xlabel(r"$x$")
  plt.ylabel("$u(x,1)$")
  plt.savefig("problem3_bdf2_fixedissue.png", dpi=300)
  plt.show()
  plt.close()

if __name__ == '__main__':
  illustrate_issue()
  fix_issue()

#BDF2.py


#1-D Diffusion equation solver using BDF-2 routine
#3-pt 2nd order spatial discretization
#with a 2nd order 3-pt approximation for time derivative
#for Dirichlet BC's

from __future__ import division

import numpy as np

import scipy.sparse as sparse
import scipy.sparse.linalg

def sparse_matrices(del_x):
  #set sparse matrix L, the discrete Laplacian
  #for 3-pt 2nd order approximation

  #Set number of grid points
  N = 1/del_x - 1

  #set off-diagonal Laplacian components
  offdiag = (1/(del_x**2))*np.ones(N)
  #set diagonal Laplacian components
  diag = np.ones(N)*(-2/(del_x**2))

  #put diagonals together into sparse matrix format
  data = np.vstack((offdiag, diag,offdiag))
  L = sparse.dia_matrix((data, [-1, 0,1]), shape = (N,N))

  #create identity matrix of same size as L
  I = sparse.identity(N)

  return L, I

def bdf2_time_step(del_t, u_new, u_old, L, f, I):
  #one time step of BDF-2 solver

  #use past two iterates on RHS
  #4u^n -u^n-1 + 2del_t f^n+1
  rhs_terms = 4*u_new  - u_old + 2*del_t*f

  #make LHS matrix, put in CSC form for solver
  LHS_matrix = scipy.sparse.csc_matrix(3*I-2*del_t*L)

  #solve (3I-2del_t L)u^n+1 = 4u^n -u^n-1 + 2del_t f^n+1
  u_next = scipy.sparse.linalg.spsolve(LHS_matrix, rhs_terms)

  return u_next

def back_euler_step(del_t, u, L, f, I):
  #one time step of Backward-Euler solver

  #use last iterate on RHS
  #4u^n -u^n-1 + 2del_t f^n+1
  rhs_terms = u + del_t*f

  #make LHS matrix, put in CSC form for solver
  LHS_matrix = scipy.sparse.csc_matrix(I-del_t*L)

  #solve (3I-2del_t L)u^n+1 = 4u^n -u^n-1 + 2del_t f^n+1
  u_next = scipy.sparse.linalg.spsolve(LHS_matrix, rhs_terms)

  return u_next

def bdf2_method(del_x, del_t, u, f, D):

  #create sparse matrices for crank-nicolson method
  [L, I] = sparse_matrices(del_x)
  #add diffusion coefficient to Laplacian
  L = D*L

  #calculate number of time points between 0 and 1
  Nt = int(1/del_t)

  #solve for first step using Backward-Euler
  u_old = u + 0
  u_new = back_euler_step(del_t, u_old, D*L, f[1], I)

  #step through time looking backwards
  for t in range(1,Nt-1):

    #solve for next u using previous two iterates u_new and u_old
    u = bdf2_time_step(del_t, u_new, u_old, D*L, f[t+1], I)

    #store previous iterates
    u_old = u_new+0
    u_new = u+0

  return u
```