Problem 1. *Consider the forward time, centered space discretization*

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}+a\frac{u_{j+1}^n-u_{j-1}^n}{2\Delta x}=b\frac{u_{j-1}^n-2u_j^n+u_{j+1}^n}{\Delta x^2},$$

to the convection-diffusion equation,

$$u_t+au_x=bu_{xx},\ b>0.$$

(a) Let $\nu=a\Delta t/\Delta x$ and $\mu=b\Delta t/\Delta x^2$. Since the solution to the PDE does not grow in time, it seems reasonable to require that the numerical solution not grow in time. Use von Neumann analysis to show that the numerical solution does not grow (in 2-norm) if and only if $\nu^2\leq 2\mu\leq 1$.

Assume $u_j^n=e^{i\xi x_j}$ and $u_j^{n=1}=g(\xi)e^{i\xi x_j}$. Then the scheme is

$$\begin{aligned}
g(\xi)e^{i\xi x_j}-e^{i\xi x_j}+\frac{a\Delta t}{2\Delta x}\big(e^{i\xi(x_j+\Delta x)}-e^{i\xi(x_j-\Delta x)}\big)&=\frac{b\Delta t}{\Delta x^2}\big(e^{i\xi(x_j-\Delta x)}-2e^{i\xi x_j}+e^{i\xi(x_j+\Delta x)}\big)\\
g(\xi)-1+\frac{\nu}{2}\big(e^{i\xi\Delta x}-e^{-i\xi\Delta x}\big)&=\mu\big(e^{-i\xi\Delta x}-2+e^{i\xi\Delta x}\big)\\
g(\xi)-1+i\nu\sin(\xi\Delta x)&=-4\mu\sin^2\left(\frac{\xi\Delta x}{2}\right)\\
\implies g(\xi)=1-2i\nu\sin\left(\frac{\xi\Delta x}{2}\right)\cos\left(\frac{\xi\Delta x}{2}\right)&-4\mu sin^2\left(\frac{\xi\Delta x}{2}\right).
\end{aligned}$$

Since

$$\left\|u^{n+1}\right\|_2=\|g(\xi)u^n\|_2\leq\|g(\xi)\|_\infty\|u^n\|_2,$$

in order for solutions to not grow in the 2-norm, we require that

$$\max_\xi|g(\xi)|\leq 1\implies|g(\xi)|\leq 1\ \forall\xi.$$

Then

$$\begin{aligned}
|g(\xi)|&=\left(1-4\mu\sin^2\left(\frac{\xi\Delta x}{2}\right)\right)^2+\left(-2\nu\sin\left(\frac{\xi\Delta x}{2}\right)\cos\left(\frac{\xi\Delta x}{2}\right)\right)^2\\
&=\left(1-4\mu\sin^2\left(\frac{\xi\Delta x}{2}\right)\right)^2+4\nu^2\sin^2\left(\frac{\xi\Delta x}{2}\right)\cos^2\left(\frac{\xi\Delta x}{2}\right)\\
&=1-8\mu\sin^2\left(\frac{\xi\Delta x}{2}\right)+16\mu^2\sin^4\left(\frac{\xi\Delta x}{2}\right)+4\nu^2\sin^2\left(\frac{\xi\Delta x}{2}\right)\left(1-\sin^2\left(\frac{\xi\Delta x}{2}\right)\right)\\
&=1+(4\nu^2-8\mu)\sin^2\left(\frac{\xi\Delta x}{2}\right)+(16\mu^2-4\nu^2)\sin^4\left(\frac{\xi\Delta x}{2}\right).
\end{aligned}$$

If $\nu^2\leq 2\mu\leq 1$, then

$$4\nu^2-8\mu\leq 0.$$

Also, $\mu \leq 1/2 \implies \mu^2 \leq 1/4$, and $\nu^2 \leq 1$ so

$$16\mu^2 - 4\nu^2 \leq 16/4 - 4\nu^2 \leq 4 - 4 = 0.$$

Thus

$$|g(\xi)| = 1 + (4\nu^2 - 8\mu)\sin^2\left(\frac{\xi\Delta x}{2}\right) + (16\mu^2 - 4\nu^2)\sin^4\left(\frac{\xi\Delta x}{2}\right) \leq 1.$$

So if $\nu^2 \leq 2\mu \leq 1$, numerical solutions do not grow in the 2-norm.

Similarly, if we know that numerical solutions do not grow in the 2-norm, i.e., $|g(\xi)| \leq 1$, then it must be that

$$(4\nu^2 - 8\mu)\sin^2\left(\frac{\xi\Delta x}{2}\right) + (16\mu^2 - 4\nu^2)\sin^4\left(\frac{\xi\Delta x}{2}\right) \leq 0.$$

Then since $\sin^2(x)$ and $\sin^4(x)$ are both positive-definite, we must have that

$$4\nu^2 - 8\mu \leq 0 \quad \text{and} \quad 16\mu^2 - 4\nu^2 \leq 0.$$

The former yields $\nu^2 \leq 2\mu$. Then

$$\begin{aligned} 16\mu^2 - 4\nu^2 &\leq 0 \\ 16\mu^2 \leq 4\nu^2 &\leq 8\mu \\ \implies 2\mu^2 &\leq \mu \\ \implies 2\mu &\leq 1. \end{aligned}$$

Hence the stability restriction is indeed $\nu^2 \leq 2\mu \leq 1$.

(b) Suppose that we use the mixed implicit-explicit scheme

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} = b\frac{u_{j-1}^{n+1} - 2u_j^{n+1} + u_{j+1}^{n+1}}{\Delta x^2}.$$

Use von Neumann analysis to derive a stability restriction on the time step.

Let $u_j^n = e^{i\xi x_j}$, assume $u_j^{n+1} = g(\xi)e^{i\xi x_j}$. Then the scheme yields

$$\begin{aligned} g(\xi)e^{i\xi x_j} - e^{i\xi x_j} + \frac{\nu}{2}e^{i\xi x_j}\left(e^{i\xi\Delta x} - e^{-i\xi\Delta x}\right) &= \mu g(\xi)e^{i\xi x_j}\left(e^{-i\xi\Delta x} - 2 + e^{i\xi\Delta x}\right) \\ g - 1 + i\nu\sin(\xi\Delta x) &= -4g\mu\sin^2\left(\frac{\xi\Delta x}{2}\right) \\ g\left(1 + 4\mu\sin^2\left(\frac{\xi\Delta x}{2}\right)\right) &= 1 - 2i\nu\sin\left(\frac{\xi\Delta x}{2}\right)\cos\left(\frac{\xi\Delta x}{2}\right) \\ \implies g(\xi) &= \frac{1 - 2i\nu\sin\left(\frac{\xi\Delta x}{2}\right)\cos\left(\frac{\xi\Delta x}{2}\right)}{1 + 4\mu\sin^2\left(\frac{\xi\Delta x}{2}\right)} \end{aligned}$$

For the scheme to be stable, we need $|g(\xi)| \leq 1$.

$$\frac{1^2 + \left(2\nu \sin\left(\frac{\xi\Delta x}{2}\right)\cos\left(\frac{\xi\Delta x}{2}\right)\right)^2}{\left(1 + 4\mu\sin^2\left(\frac{\xi\Delta x}{2}\right)\right)^2} \leq 1$$

$$1 + 4\nu^2\sin^2\left(\frac{\xi\Delta x}{2}\right)\cos^2\left(\frac{\xi\Delta x}{2}\right) \leq 1 + 8\mu\sin^2\left(\frac{\xi\Delta x}{2}\right) + 16\mu^2\sin^4\left(\frac{\xi\Delta x}{2}\right)$$

$$4\nu\cos^2\left(\frac{\xi\Delta x}{2}\right) \leq 8\mu + 16\mu^2\sin^2\left(\frac{\xi\Delta x}{2}\right)$$

$$4\nu^2\left(1 - \sin^2\left(\frac{\xi\Delta x}{2}\right)\right) \leq 8\mu + 16\mu^2\sin^2\left(\frac{\xi\Delta x}{2}\right).$$

The above inequality holds for both extreme values of the increasing function $\sin^2\left(\frac{\xi\Delta x}{2}\right)$, 0 and 1. For $\sin^2\left(\frac{\xi\Delta x}{2}\right) = 0$, the inequality becomes

$$4\nu^2 \leq 8\mu$$

or $\nu^2 \leq 2\mu$. For $\sin^2\left(\frac{\xi\Delta x}{2}\right) = 1$, the inequality becomes

$$0 \leq 8\mu + 16\mu^2,$$

which is true since $\mu > 0$. Thus the only stability restriction is that $\nu^2 \leq 2\mu$.

Problem 2. *Write programs to solve*

$$u_t = \Delta u \text{ on } \Omega = (0,1)\times(0,1)$$

$$u = 0 \text{ on } \delta\Omega$$

$$u(x,y,0) = exp(-100((x-0.3)^2 + (y-0.4)^2))$$

to time $t = 1$ using Forward Euler and Crank-Nicolson. For Crank-Nicolson use a fixed time step of $\Delta t = 0.01$, and for Forward Euler use a time step just below the stability requirement.

(a) Time your codes for different grid sizes and compare the time to solve using Forward Euler and Crank-Nicolson.

I altered my Crank-Nicolson code from last assignment to solve the 2D diffusion equation with Dirichlet BC's, implementing the 2D discrete Laplacian instead of the 1D, and changing the setup and initial conditions appropriately. I used a fixed time step of $\Delta t = 0.01$ and varied grid spacings from $h = \Delta x = \Delta y = 2^{-1}, 2^{-2}, \ldots, 2^{-7}$ to solve the diffusion equation.

I coded Forward Euler for the 2D diffusion, and solved it on the same grid spacings h. The time step requirement for FE stability for the 2D diffusion equation is

$$\Delta t \leq \frac{h^2}{4D},$$

and since $D = 1$ in this problem, I used a time step of $\Delta t = \frac{h^2}{5}$. The runtime comparison of the two methods is given in Table 1. FE is faster for coarse grid spacings due to the overhead in solving the linear system in C-N, but C-N easily outperforms FE for finer grid spacings from 2^{-4} and beyond.

Table 1: Runtime (in Seconds) Comparison for C-N, FE for 2-D Diffusion

grid spacings h	Crank-Nicolson Runtimes	Forward Euler Runtimes
2^{-1}	0.134055	0.013954
2^{-2}	0.136421	0.032493
2^{-3}	0.102737	0.125547
2^{-4}	0.151887	0.540384
2^{-5}	0.335359	2.339
2^{-6}	2.63749	13.3055
2^{-7}	14.1191	118.39
2^{-8}	93.7666	2320.99
2^{-9}	610.232	——

(b) In theory, how should the (run)time scale as the grid is refined for each algorithm? How did the (run)time scale with the grid size in practice?

For CN, we used the same number of time steps for each solve, so that doesn't factor into the runtime scaling. Each timestep, however, required a linear solve with an $N^2 \times N^2$ sparse banded matrix. With a dumb algorithm that takes advantage of the banded structure, it should take number of rows $\mathcal{O}(N^2)$ times the first band width spacing $\mathcal{O}(N)$ times the next band width spacing $\mathcal{O}(N)$, for a total of $\mathcal{O}(N^4)$ work. Each $N = 1/h = 2^i$, so the work at each time step should have been 2^{4i}. The next grid spacing level would have $N = 2^{i+1}$ and thus 2^{4i+4} work, for a successive ratio of $2^4 = 16$. Thus the time to solve should scale by a factor 16. In practice for CN, this scaling was never achieved, as shown in Table 2. The practical scaling was around a factor of 6.5, which might indicate an asymptotic scaling factor of 8. This means that the scipy sparse matrix solver I use in the CN routine is better than my $\mathcal{O}(N^4)$ estimate, and it is probably closer to $\mathcal{O}(N^3)$.

Table 2: C-N Runtime Scaling

grid spacings h	Crank-Nicolson Runtimes	Successive Ratios
2^{-1}	0.134055	——
2^{-2}	0.136421	1.01765
2^{-3}	0.102737	0.753088
2^{-4}	0.151887	1.47841
2^{-5}	0.335359	2.20795
2^{-6}	2.63749	7.86469
2^{-7}	14.1191	5.35322
2^{-8}	93.7666	6.64112
2^{-9}	610.232	6.50799

For FE, the time step size was $\Delta t = h^2/5$, for grid spacings $h = 2^{-1}, \ldots, 2^{-7}$. Thus at each grid level $h = 2^{-i}$, there were $5 \cdot 2^{2i}$ time steps total. Each timestep requires multiplication by a $2^{2i} \times 2^{2i}$ matrix, so 2^{2i} work. At the next level, there are then a factor of 2^2 more time steps and a factor of 2^2 more work involved at each step, so the runtime should scale by a factor of 16.

In practice for FE, the runtime scaling factor increases as the runtime increases, but I did not run the simulation long enough to achieve the asymptotic scaling factor of 16. The ratios jump from 2.33 to 4 to 9 to 20, so maybe the next run would have settled down closer to 16. The runtime scaling factors are shown in Table 3.

Table 3: FE Runtime Scaling

grid spacing h	Forward Euler run time (seconds)	Ratio
2^{-1}	0.013954	——
2^{-2}	0.032493	2.33
2^{-3}	0.125547	3.86
2^{-4}	0.540384	4.30
2^{-5}	2.339	4.3284
2^{-6}	13.3055	5.68853
2^{-7}	118.39	8.89785
2^{-8}	2320.99	19.6046

(c) For this problem we could use an FFT-based Poisson solver which will perform the direct solve in $\mathcal{O}(N \log(N))$, where N is the total number of grid points. We could also use multigrid and perform the solve in $\mathcal{O}(N)$ time. How should the time scale as the grid is refined for Crank-Nicolson if we used an $\mathcal{O}(N)$ solver?

Since there is no time step restriction in Crank-Nicolson, we can keep the same time step size across finer grids. Then with an $\mathcal{O}(N)$ solver, the runtime should scale like the grid size, i.e., $\mathcal{O}(N)$.

The following Python code was used for problem 2.

```
#Problem2.py
#Carter Johnson

#Runtime comparison of Forward Euler and Crank-Nicolson
#for 2-D diffusion eqn

#u_t = Delta u  on Omega = (0,1)x(0,1)
#u = 0 on dOmega
#u(x,y,0)=exp(-100((x-0.3)^2+(y-0.4)^2))

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 time import clock


def run_comparison():
    #run comparison between Crank-Nicolson and Forward euler on the problem
    #set up the vectors and parameters for the methods and run
    #using diffusion coefficient, initial condition, forcing function from problem 2

    #set vector of grid spacings
    h = [2**(-i) for i in range(1,10)]

    #set time steps for CN, forcing function for CN
    CN_del_t = 0.01
    Nt_CN = int(1/CN_del_t)
    f_CN = np.zeros(Nt_CN+1)

    #diffusion coefficient
    D = 1

    #Don't plot
    plotting=0

    #record run times and runtime ratios
    CN_times = np.zeros(len(h))
    FE_times = np.zeros(len(h))
    CN_ratios = np.zeros(len(h))
    FE_ratios = np.zeros(len(h))
```

```
    for i in tqdm(range(len(h))):

        FE_del_t = h[i]**2/5

        #make vector of forcing function for FE at all times, also get grid points for level
        Nx = int(1/h[i])-1
        Ny = int(1/h[i])-1
        Nt_FE = int(1/FE_del_t)
        X = [j*h[i]for j in range(1, Nx+1)]
        Y = [j*h[i] for j in range(1, Ny+1)]

        #f =0
        f_FE = np.zeros(Nt_FE+1)

        #initial condition u(x,y,0)=exp(-100((x-0.3)^2+(y-0.4)^2))
        u = [[exp(-100*((x-0.3)**2+(y-0.4)**2)) for x in X] for y in Y]
        u = np.asarray(u)


        toc=clock()
        u = crank_nicolson_method(h[i], CN_del_t, u, f_CN, D, X, Y, plotting)
        tic=clock()
        if i>0:
            CN_ratios[i]=(tic-toc)/CN_times[i-1]
        CN_times[i]=tic-toc

        if i<=7:
            toc=clock()
            u = forward_euler_method(h[i], FE_del_t, u, f_FE, D, X, Y, plotting)
            tic=clock()
            if i>0:
                FE_ratios[i]=(tic-toc)/FE_times[i-1]
            FE_times[i]=tic-toc

    runtime_table = [[h[i], CN_times[i], CN_ratios[i], FE_times[i], FE_ratios[i]] for i in r
    print(tabulate(runtime_table, headers=["grid spacings", "Crank-Nicolson Runtimes", "CN R
    return u

def sparse_matrices(h):
    #set sparse matrix L, the discrete 2-D Laplacian
    #for 5-pt 2nd order approximation

    #Set number of grid points
    N = 1/h - 1
```

```
    #set off-diagonal Laplacian components
    off_diag = 1*np.ones(N)
    #set diagonal Laplacian components
    diag = (-2)*np.ones(N)

    # Generate the diagonal and off-diagonal matrices
    A = np.vstack((off_diag, diag, off_diag))
    lap1D = scipy.sparse.dia_matrix((A,[-1,0,1]),shape=(N,N))
    speye = scipy.sparse.identity(N)
    #put diagonals together into sparse matrix format
    L = (1/(h**2))*(scipy.sparse.kron(lap1D,speye) + scipy.sparse.kron(speye,lap1D))

    #make identity matrix of same size as L
    I = scipy.sparse.identity(N**2)
    return L, I

def crank_nicolson_time_step(h, del_t, u, L, f, I):
    #one time step of crank-nicolson solver
    N = 1/h -1

    #(I + del_t/2 L)u^n
    A = (I + (del_t/2) * L)
    RHS_terms = A.dot(u.flatten(order='C')) + 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)
    u_next = np.reshape(u_next, (N, N))

    return u_next

def crank_nicolson_method(h, del_t, u, f, D, x, y, plotting):

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

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

    if plotting==1:
        #set up plotting
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        # `plot_surface` expects `x` and `y` data to be 2D
```

```
        X, Y = np.meshgrid(x, y)
        #keep z limits fixed
        ax.set_zlim(0, 1)
        plt.ion()
        #plot first frame, u(x,y,0)
        frame = ax.plot_surface(X, Y, u)
        plt.pause(0.05)

    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(h, del_t, u, D*L, f_half, I)

        if plotting==1:
            #plot current u
            ax.collections.remove(frame)
            frame = ax.plot_surface(X, Y, u)
            plt.pause(0.05)

    return u

def forward_euler_time_step(h, del_t, u, L, f, I):
    #one time step of crank-nicolson solver
    N = 1/h -1

    #u^n+1 = (I + del_t L)u^n + del_t*f
    A = (I + (del_t) * L)
    u_next = A.dot(u.flatten(order='C')) + del_t*f
    u_next = np.reshape(u_next, (N, N))

    return u_next

def forward_euler_method(h, del_t, u, f, D, x, y, plotting):

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

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

    if plotting==1:
        #set up plotting
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        # `plot_surface` expects `x` and `y` data to be 2D
```

```
        X, Y = np.meshgrid(x, y)
        #keep z limits fixed
        ax.set_zlim(0, 1)
        plt.ion()
        #plot first frame, u(x,y,0)
        frame = ax.plot_surface(X, Y, u)
        plt.pause(0.05)

    for t in range(0,Nt):
        #solve for next u
        u = forward_euler_time_step(h, del_t, u, D*L, f[t], I)

        if plotting==1:
            #plot current u
            ax.collections.remove(frame)
            frame = ax.plot_surface(X, Y, u)
            plt.pause(0.05)

    return u
```