Problem One:
1.1.2) Write matlab code to solve the system from 1.1.1. Use step sizes $n = 5, 6, 7, 8$. Compare results with the exact solution by estimating the relative errors.

Solution: Create a script `steady_state_solver` which solves the system from 1.1.1 given different values of $n$.

The solutions are plotted in figure 1. None of the approximate solutions appear to converge *exactly* to the analytic solution. However, the accuracy of the solutions improves as $n$ is increased, and the approximation when $n = 8$ is very close.

*Figure 1*
The relative error of the solutions is plotted in figure 2. This is further evidence that as $n$ increases, the accuracy of the approximation also increases.

**Figure 2**
Relative Error for approximations with various $n$

1.2.2)
Write matlab code to solve the three systems obtained in part 1.2.1. Use timesteps of $\frac{1}{t^n}$, for $n = 0, 1, \text{ and } 2$. Use a spatial step size of $\frac{L}{32}$. Obtain numerical results from $t = 0$ to $t = 1$. Comment on whether all step sizes work for all methods. Create three dimensional plots for each method.

**Solution:** For each method, explicit, implicit, and Crank-Nicolson, create a script which approximates the solution to the partial differential equation.

**Explicit Method**
This is the most straightforward method. We can see from the Courant-Friedrichs-Lewy condition that this method will not converge until the time step is sufficiently small. In fact, given that our time intervals are $\frac{1}{t^n}$, we must have $n = 6$ before convergence will be possible.
We present no figures for this method, because the solution diverges. That is to say, plotting the approximate solution with \( n = 0, 1, 2 \) using the explicit method provides no great insight into the nature of the solution.

The code for the implementation of all three methods, explicit, implicit, and Crank-Nicolson, is reserved for the appendix.

**Implicit Method**

In this method, each successive iteration as \( t \) increases involves solving a system of linear equations of the following form:

\[
\begin{pmatrix}
C_1 + f_1(x) \\
\vdots \\
C_N + f_N(x)
\end{pmatrix} = A 
\begin{pmatrix}
C_1^{l+1} \\
\vdots \\
C_N^{l+1}
\end{pmatrix}
\]

The values of \( f(x) \) and the coefficients of the matrix \( A \) are derived from equation 1, below. Note that the terms \( i = 1 \) and \( i = N \) require special care since the boundary conditions are necessary.

\[
c_i^l = \frac{-D_{i-1/2}}{h^2}(\Delta t)c_{i-1}^{l+1} + \left[ \frac{D_{i+1/2} + D_{i-1/2}}{h^2} + k \right] (\Delta t)c_i^{l+1} + c_i^{l+1} - \frac{D_{i+1/2}}{h^2}(\Delta t)c_{i+1}^{l+1} - f_i(\Delta t)
\]

The approximate solutions for \( n = 0, 1 \) and \( 2 \) are presented in figures 3, 4, and 5.

**Figure 3**

Approximate Solution \( n = 0 \)
Figure 4
Approximate Solution $n = 1$

Figure 5
Approximate Solution $n = 2$
Crank-Nicolson
This method is very similar in form to the implicit method, and indeed, it achieves similar results. The Crank-Nicolson provides a slightly better approximation of the solution than the implicit method can offer given identical time steps (e.g. number of iterations required). However, the computation time was notably longer for the Crank-Nicolson method compared to the implicit method.

Figure 6
Approximate Solution $n = 0$

Figure 7
Approximate Solution $n = 1$
1.2.3) Use the matlab built in function \textit{pdepe} to solve the initial boundary value problem. Plot the solution using \textit{surf}.

\textbf{Solution}: We will create three functions, one for the partial differential equation, one for the initial condition, and one for the boundary conditions. Then, we can call \textit{pdepe} and produce a 3-dimensional plot of its solution.

The three functions, titled \textit{pdefun}, \textit{pdeic}, and \textit{pdebc}, respectively, have a format stipulated by the function \textit{pdepe}, and may be found in the appendix.

Our function call is as follows:
\[
sol = \text{pdepe}(0,@pdefun,@pdeic,@pdebc,xlist,tspan)
\]
The resulting solution is plotted in figure 9 below.

We are interested in the behavior of the solution at larger time values. As \( t \) continues to grow, the behavior of our solution becomes very stable. This can be seen in figure 10 below, which includes time from \( t = 0 \) to \( t = 50 \).
“So long, and thanks for all the fish.”
- The Ultimate Hitchhiker’s Guide to the Galaxy
**Appendix**

*steady_state_solver.m*

```matlab
L = pi/2
hold on
for(n=5:8)
    h = L/(2^n)
    N = L/h

    fin_matrix = zeros(N,N+1)

    xlist = 0:h:L
    fin_matrix(1,1) =
        (-Dx((xlist(2)+xlist(3))/2)-Dx((xlist(1)+xlist(1))/2))/(h^2)-1
    fin_matrix(1,2) = Dx((xlist(2)+xlist(3))/2)/(h^2)
    fin_matrix(1,N+1)= -Fx(xlist(2))

    for(i2=2:1:N-1)
        fin_matrix(i2,i2+1) = Dx((xlist(i2+1)+xlist(i2+2))/2)/(h^2)
        fin_matrix(i2,i2) =
            -Dx((xlist(i2+1)+xlist(i2+2))/2)/(h^2)-Dx((xlist(i2)+xlist(i2+1))/2)/(h^2)-1
        fin_matrix(i2,i2-1) = Dx((xlist(i2)+xlist(i2+1))/2)/(h^2)
        fin_matrix(i2,N+1) = -Fx(xlist(i2+1))
    end

    fin_matrix(N,N-1) = Dx((xlist(N+1)+xlist(N))/2)/(h^2)
    fin_matrix(N,N) = (-Dx((xlist(N+1)+xlist(N))/2)/(h^2))-1
    fin_matrix(N,N+1) = -Fx(xlist(N+1))

    Sol_mat = rref(fin_matrix)
    Rplot = [ 0 Sol_mat(:,N+1)' ]
    plot(xlist,Rplot)
end
```

**Dx.m**

```matlab
function [ Dxout ] = Dx( x )
%Dx Evaluates the function Dx
if(x<=(pi/3))
    Dxout = 1
end
if(x>(pi/3))
    Dxout = 100
end
end
```
Fx.m
function [ Fxout ] = Fx( x )
% Fx Evaluates the function Fx
if(x<=(pi/3))
    Fxout = 9*sin(3*x)+sin(3*x)
end
if(x>(pi/3))
    Fxout = 9*sin(3*x)+.01*sin(3*x)
end
end

pdebc.m
function [ pl,ql,pr,qr ] = pdebc( x1,u1,xr,ur,t )
pl = u1;
ql = 0;
pr = 0;
qr = pi/2;
end

pdefun.m
function [ c,f,s ] = pdefun( x,t,u,dudx )
if(x<=pi/3)
    f = dudx;
    s = 10*sin(3*x) - u;
    c = 1;
else
    f = 100*dudx;
    s = 9.01*sin(3*x) - u;
    c = 1;
end

pdeic.m
function [ u0 ] = pdeic( x )
u0 = 0;
end

explicit.m
t = 0
delta_t = 1/(10^6)
h = (pi/2)/(32)
xlist = 0:h:(pi/2)
xlistavg = h/2:h:(pi/2)
\[ N = \frac{L}{h} \]

\[ \text{cinit} = \text{zeros}(N+1,1) \]

\[ \text{for}(t=0:\text{delta}_t:1) \]
\[ \text{c\_current}(1) = 0 \]
\[ \text{c\_current}(2) = ((-Dx(xlistavg(2))-Dx(xlistavg(1)))/(h^2)-1)*\text{delta}_t*\text{cinit}(2)+\text{cinit}(2)+((Dx(xlistavg(2))/(h^2))*\text{delta}_t*\text{cinit}(3)+Fx(xlist(2))*\text{delta}_t \]
\[ \text{for}(\text{iter} = 3:1:N-1) \]
\[ \text{c\_current}(\text{iter}) = (Dx(xlistavg(\text{iter}-1))/(h^2))*\text{delta}_t*\text{cinit}(\text{iter}-1)+((-Dx(xlistavg(\text{iter}))-Dx(xlistavg(\text{iter}-1)))/(h^2)-1)*\text{delta}_t*\text{cinit}(\text{iter})+\text{cinit}(\text{iter})+(Dx(xlistavg(\text{iter}))/(h^2))*\text{delta}_t*\text{cinit}(\text{iter}+1)+Fx(xlist(\text{iter}))*\text{delta}_t \]
\[ \text{end} \]
\[ \text{c\_current}(N) = (Dx(xlistavg(N-1))/(h^2))*\text{delta}_t*\text{cinit}(N)+((-Dx(N-1)/(h^2))-1)*\text{delta}_t*\text{cinit}(N)+Fx(xlist(N))*\text{delta}_t \]
\[ \text{c\_final}(\text{int64}(t/\text{delta}_t)+1,:) = \text{c\_current} \]
\[ \text{cinit} = \text{c\_current} \]
\[ \text{end} \]

\textit{implicit.m}

\[ t = 0 \]
\[ \text{delta}_t = 1/(10^1) \]
\[ h = (\pi/2)/(32) \]
\[ xlist = 0:h:(\pi/2) \]
\[ xlistavg = h/2:h:(\pi/2) \]
\[ N = \frac{L}{h} \]
\[ \text{cinit} = \text{zeros}(N+1,1) \]
\[ \text{Augmat} = \text{zeros}(N,N+1) \]
\[ \text{for}(t=0:\text{delta}_t:1) \]
\[ \text{Augmat}(1,1) = (((Dx(xlistavg(2))+Dx(xlistavg(1)))/(h^2)+1)*\text{delta}_t)+1 \]
\[ \text{Augmat}(1,2) = (-Dx(xlistavg(2))/(h^2))*\text{delta}_t \]
\[ \text{Augmat}(1,N+1) = \text{cinit}(2)+Fx(xlist(2))*\text{delta}_t \]
\[ \text{for}(i=2:1:N-1) \]
\[ \text{Augmat}(i,i-1) = (-Dx(xlistavg(i))/(h^2))*\text{delta}_t \]
\[ \text{Augmat}(i,i) = (((Dx(xlistavg(i+1))+Dx(xlistavg(i)))/(h^2))*\text{delta}_t)+1 \]
\[ \text{Augmat}(i,i+1) = (-Dx(xlistavg(i+1))/(h^2))*\text{delta}_t \]
\[ \text{Augmat}(i,N+1) = \text{cinit}(i+1)+Fx(xlist(i+1))*\text{delta}_t \]
end

Augmat(N,N-1) = ((-Dx(xlistav(N))/(h^2))*delta_t
Augmat(N,N) = ((Dx(xlistav(N))/(h^2))+1)*delta_t+1
Augmat(N,N+1) = cinit(N)+Fx(xlist(N))*delta_t

solved_augmat = rref(Augmat)
sol = solved_augmat(:,N+1)

final_implicit_matrix(int64(t/delta_t)+1,:) = sol
cinit = [ 0 sol' ]
end

crank-nicolson.m
t = 0
L=pi/2
delta_t = 1/(10^2)
h = (pi/2)/(32)
xlist = 0:h:(pi/2)
xlistav = h/2:h:(pi/2)
N = L/h
cinit = zeros(N+1,1)

Augmat = zeros(N,N+1)
for(t=0:delta_t;1)
    Augmat(1,1) = (((-Dx(xlistav(2))-Dx(xlistav(1)))/(h^2)-1)*(-delta_t/2)+1
    Augmat(1,2) = (Dx(xlistav(2))/(h^2))((-delta_t/2)
    Augmat(1,N+1) = (((-Dx(xlistav(2))-Dx(xlistav(1)))/(h^2)-1)*(delta_t/2)*cininit2)+(Dx(xlistav(2))/(h^2))*(delta_t/2)*cininit3+Fx(xlist(2))*(delta_t/2)+cininit2

for(i=2:1:N)
    Augmat(i,i-1) = Dx(xlistav(i))/(h^2)*(-delta_t/2)
    Augmat(i,i) = (((-Dx(xlistav(i+1))-Dx(xlistav(i)))/(h^2)-1)*(-delta_t/2)+1
    Augmat(i,i+1) = (Dx(xlistav(i+1))/(h^2))*(-delta_t/2)
    Augmat(i,N+1) = (Dx(xlistav(i))/(h^2))*(delta_t/2)*cininiti+(((-Dx(xlistav(i+1))-Dx(xlistav(i)))/(h^2)-1)*(delta_t/2)*cininiti+1+(Dx(xlistav(i+1))/(h^2))*(delta_t/2)*cininiti2)+(Fx(xlist(i+1))*(delta_t/2)+cininiti+1)
end

Augmat(N,N-1) = Dx(xlistav(N))/(h^2)*(-delta_t/2)
Augmat(N,N) = ((-Dx(xlistav(N)))/(h^2)-1)*(-delta_t/2)+1
Augmat(N,N+1) =
(Dx(xlistav(N))/(h^2))*(delta_t/2)*cinit(N)+((-Dx(xlistav(N)))/(h^2)-1)*(delta_t/2)*cinit(N+1)+Fx(xlist(N+1))*(delta_t/2)+cinit(N+1)

solved_augmat = rref(Augmat)
sol = solved_augmat(:,N+1)

final_crank_matrix(int64(t/delta_t)+1,:) = sol
cinit = [ 0 sol' ]
end

References


