Numerical Methods HW3

Xiangdong Yang

October 2025

All source code can be found here in my github repository

a Exact Solution

Assume a separated form of $u(x,t)$:

$$u(x,t) = A(x)B(t)$$

Substituting back into the PDE and obtaining:

$$\begin{aligned} A(x)\frac{dB(t)}{dt} &= B(t)\frac{d^2A(x)}{dx^2} \\ \implies \frac{dB(t)}{B(t)dt} &= \frac{d^2A(x)}{A(x)dx^2} = C \end{aligned}$$

where C is some constant.
Solve for $B(t)$:

$$\begin{aligned} \frac{dB(t)}{B(t)} &= Cdt \\ \implies B(t) &= \exp(Ct) \end{aligned}$$

Solve for $A(x)$:

$$\begin{aligned} \frac{d^2A(x)}{dx^2} &= CA(x) \\ \implies A(x) &= c_1\cos(kx) + c_2\sin(kx) \end{aligned}$$

where $-k^2 = C$
Apply the boundary condition:

$$A(0) = c_1 = 0, \quad A(2\pi) = c_2\sin(2\pi k) = 0$$

This implies that $k = n/2$ for some integer n.
Thus the general solution with these boundary conditions is:

$$u(x,t) = \sum_{n=1}^{\infty} a_n \exp(-\frac{n^2}{4}t)\sin(\frac{n}{2}x)$$

Apply the initial condition:

$$u(x,0) = \sum_{i=1}^{\infty} a_n \sin(\frac{n}{2}x) = \sin(mx)$$

which implies that:

$$a_n = \begin{cases} 1, & n = 2m \\ 0, & \text{otherwise} \end{cases}$$

Therefore:

$$u(x,t) = \sin(mx)\exp(-m^2t)$$

b Numerical Solutions

Put $r = \Delta t/\Delta x^2$

b.1 Forward Euler:

The forward Euler (FE) scheme is as follows:

$$u_j^{n+1} = u_j^n + r(u_{j-1}^n - 2u_j^n + u_{j+1}^n)$$

I wrote the following function in Python to implement FE:

```
def forward_euler(max_t,m,r=None,delta_x=None,delta_t=None):
    if delta_t is None:
        delta_t=r*delta_x**2
    if delta_x is None:
        delta_x=np.sqrt(delta_t/r)
    if r is None:
        r=delta_t/delta_x**2
    grid=np.array(np.meshgrid(np.arange(0,2*np.pi+delta_x/2,delta_x),np.arange(0,max_t+
                                                    delta_t/2,delta_t)))
    u=np.zeros((1,grid.shape[1],grid.shape[2])) #time space
    u[:,0,:]=np.sin(m * grid[0,0]) #initial condition
    u[:,:,0]=0 #boundary condition
    u[:,:,-1]=0 #boundary condition
    for n in range(u.shape[1]-1): #time
        for j in range(1,u.shape[2]-1): #space
            u[:,n+1,j]=r*(u[:,n,j+1]-2*u[:,n,j]+u[:,n,j-1])+u[:,n,j]
    return u,grid
```

b.2 Backward Euler

The backward Euler (BE) scheme is as follows:

$$-ru_{j-1}^{n+1} + (1+2r)u_j^{n+1} - ru_{j+1}^{n+1} = u_j^n$$

Suppose there are N grid points in space indexed from 1 to N. Considering the boundary conditions, the system of equations can be written as follows:

$$\implies \begin{bmatrix} 1+2r & -r & & & 0 \\ -r & 1+2r & -r & & \\ & -r & 1+2r & \ddots & \\ & & \ddots & \ddots & -r \\ 0 & & & -r & 1+2r \end{bmatrix} \begin{bmatrix} u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{N-2}^{n+1} \\ u_{N-1}^{n+1} \end{bmatrix} = \begin{bmatrix} u_2^n \\ u_3^n \\ \vdots \\ u_{N-2}^n \\ u_{N-1}^n \end{bmatrix}$$

I wrote the following function in Python to implement BE:

```
def backward_euler(max_t,m,r=None,delta_x=None,delta_t=None):
    if delta_t is None:
        delta_t=r*delta_x**2
    if delta_x is None:
        delta_x=np.sqrt(delta_t/r)
    if r is None:
        r=delta_t/delta_x**2
    grid=np.array(np.meshgrid(np.arange(0,2*np.pi+delta_x/2,delta_x),np.arange(0,max_t+
                                                    delta_t/2,delta_t)))
    u=np.zeros((1,grid.shape[1],grid.shape[2])) #time space

    u[:,0,:]=np.sin(m * grid[0,0]) #initial condition
    u[:,:,0]=0 #boundary condition
    u[:,:,-1]=0 #boundary condition
```

```
    A=np.zeros((u.shape[2]-2,u.shape[2]-2))
    np.fill_diagonal(A,1+2*r)
    np.fill_diagonal(A[1:],-r)
    np.fill_diagonal(A[:,1:],-r)

    for n in range(u.shape[1]-1): #time
        b=u[:,n,1:-1].T
        u[:,n+1,1:-1]=solve_tridiagonal(A,b).T
    return u,grid
```

where the solve_tridiagonal function is the tridiagonal matrix algorithm I implemented to solve the system of equations. The code for this function is here at the end of this section

b.3 Crank-Nicolson

The Crank-Nicolson (CN) scheme is as follows:

$$-ru_{j-1}^{n+1} + (2+2r)u_j^{n+1} - ru_{j+1}^{n+1} = ru_{j-1}^n + (2-2r)u_j^n + ru_{j+1}^n$$

Suppose there are N grid points in space indexed from 1 to N. Considering the boundary conditions, the system of equations can be written as follows:

$$\implies \begin{bmatrix} 2+2r & -r & & & 0 \\ -r & 2+2r & -r & & \\ & -r & 2+2r & \ddots & \\ & & \ddots & \ddots & -r \\ 0 & & & -r & 2+2r \end{bmatrix} \begin{bmatrix} u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{N-2}^{n+1} \\ u_{N-1}^{n+1} \end{bmatrix} = \begin{bmatrix} 2-2r & r & & & 0 \\ r & 2-2r & -r & & \\ & r & 2-2r & \ddots & \\ & & \ddots & \ddots & r \\ 0 & & & r & 2-2r \end{bmatrix} \begin{bmatrix} u_2^n \\ u_3^n \\ \vdots \\ u_{N-2}^n \\ u_{N-1}^n \end{bmatrix}$$

```
def crank_nicolson(max_t,m,r=None,delta_x=None,delta_t=None):
    if delta_t is None:
        delta_t=r*delta_x**2
    if delta_x is None:
        delta_x=np.sqrt(delta_t/r)
    if r is None:
        r=delta_t/delta_x**2

    grid=np.array(np.meshgrid(np.arange(0,2*np.pi+delta_x/2,delta_x),np.arange(0,max_t+
                                                    delta_t/2,delta_t)))

    u=np.zeros((1,grid.shape[1],grid.shape[2])) #time space
    u[:,0,:]=np.sin(m * grid[0,0]) #initial condition
    u[:,:,0]=0 #boundary condition
    u[:,:,-1]=0 #boundary condition

    A=np.zeros((u.shape[2]-2,u.shape[2]-2))
    np.fill_diagonal(A,2+2*r)
    np.fill_diagonal(A[1:],-r)
    np.fill_diagonal(A[:,1:],-r)

    B=np.zeros((u.shape[2]-2,u.shape[2]-2))
    np.fill_diagonal(B,2-2*r)
    np.fill_diagonal(B[1:],r)
    np.fill_diagonal(B[:,1:],r)

    for n in range(u.shape[1]-1): #time
        b=B @ u[:,n,1:-1].T
        u[:,n+1,1:-1]=solve_tridiagonal(A,b).T
    return u,grid
```

The solve_tridiagonal function used here is next in this section

TDMA function

The TDMA is split into two parts. The forward_sweep function eliminates all the elements in the lower diagonal and then scale the main diagonal elements to one. The main solve_tridiagonal function then performs back substitution using the output of foward_sweep:

```
def forward_sweep(A,b):
    A_prime=deepcopy(A)
    b_prime=deepcopy(b)
    A_prime[0]/=A[0,0]
    b_prime[0]/=A[0,0]
    for i in range(1,A.shape[0]):
        b_prime[i]=(b_prime[i]-A_prime[i,i-1]/A_prime[i-1,i-1]*b_prime[i-1])/(A_prime[i,i]-
                                                    A_prime[i,i-1]/A_prime[i-1,i-1]*
                                                    A_prime[i-1,i])

        A_prime[i]=(A_prime[i]-A_prime[i,i-1]/A_prime[i-1,i-1]*A_prime[i-1])/(A_prime[i,i]-
                                                    A_prime[i,i-1]/A_prime[i-1,i-1]*
                                                    A_prime[i-1,i])

    return A_prime,b_prime
def solve_tridiagonal(A,b):
    A_prime,b_prime=forward_sweep(A,b)
    x=np.zeros_like(b)
    x[-1]=b_prime[-1]
    for i in reversed(range(b.shape[0]-1)):
        x[i]=b_prime[i]-A_prime[i,i+1]*x[i+1]
    return x
```

c

c.1

The plots are shown below.

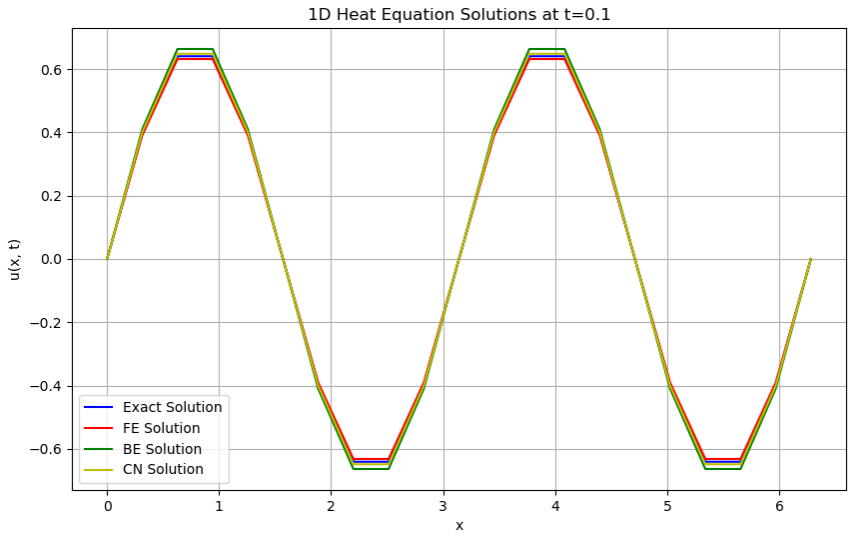

Figure c.1: Numerical and Exact Solutions at $t = 0.1$

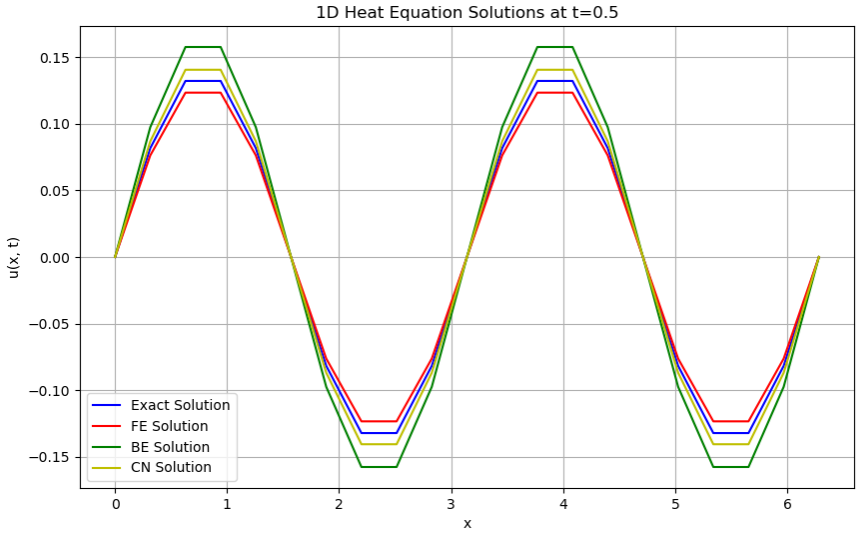

Figure c.2: Numerical and Exact Solutions at $t = 0.5$

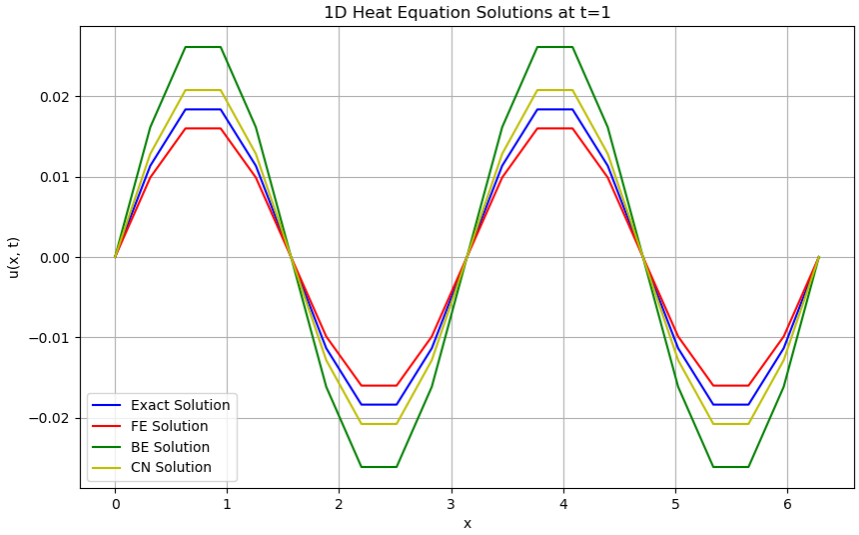

Figure c.3: Numerical and Exact Solutions at $t = 1.0$

Observation: it can be seen from the plots that as time goes on, the numerical solutions diverge more and more from the exact solution. In this particular grid setting, the FE and CN schemes seem to give more accurate solutions, whereas the BE solution performs worse. This, however, does not mean that FE and CN are inherently better than BE. A grid refinement study is necessary to compare these numerical methods.

c.2

c.2.1

To verify this statement, I used the mean absolute error between the numerical and exact solutions on the whole grid as a metric. To perform grid refinement experiments in space, I kept Δt consistent and at a relatively small value of 10^{-7} . The solutions are obtained up to $t = 10^{-5}$, i.e., 100 timesteps. I chose 10 spatial grid sizes from $\left\{2^i \times 10^{-3}\right\}_{i=1}^{10}$ to run the numerical experiments. The resulting plots are shown in figures c.4 to c.6.

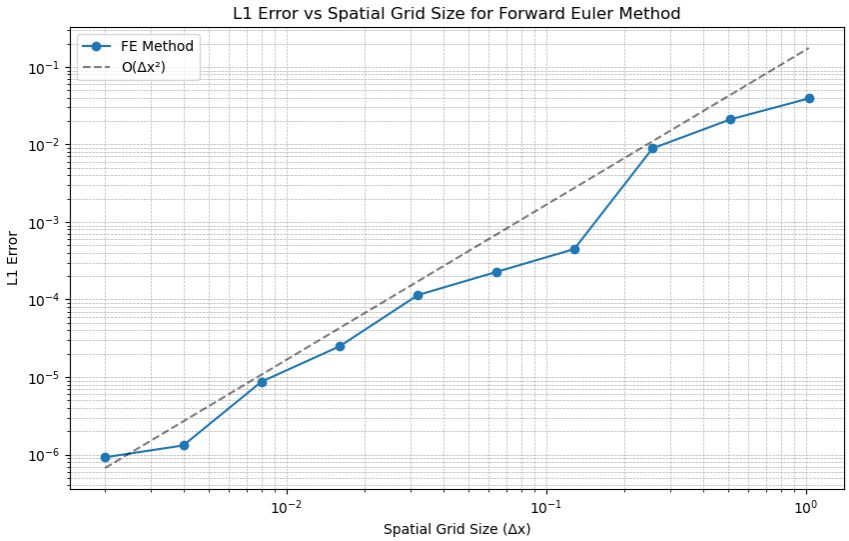

Figure c.4: Scaling of Spatial Error of FE solution

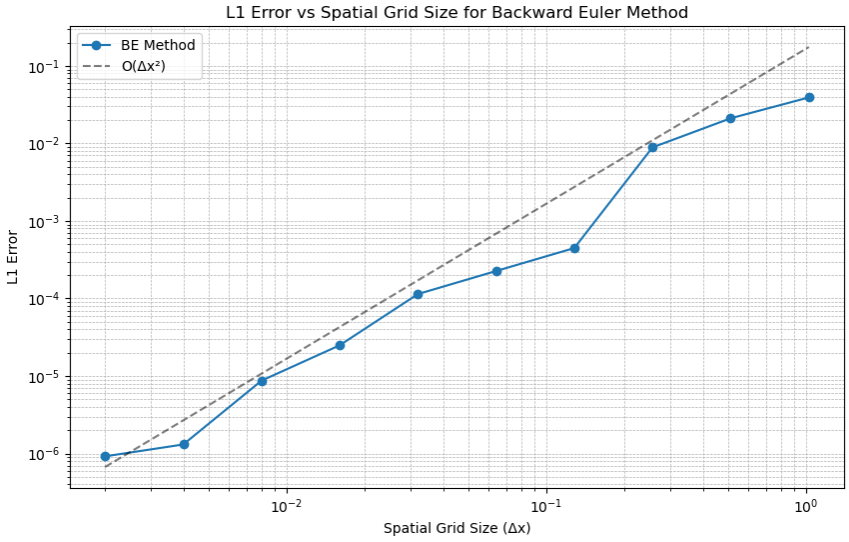

Figure c.5: Scaling of Spatial Error of BE solution

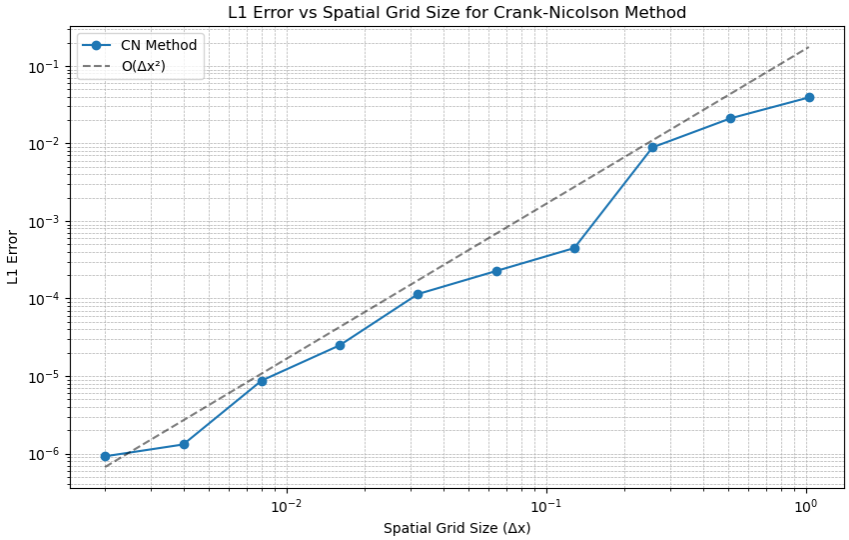

Figure c.6: Scaling of Spatial Error of CN solution

It can be seen in figures c.4 to c.6 that the spatial errors of all three schemes do roughly scale with $O(\Delta x^2)$, consistent with our theoretical analysis.

c.2.2

Keeping $\Delta x = \frac{\pi}{10}$ and running the three numerical schemes for $r = \{0.1, 0.25, 0.49, 0.51, 0.75, 1.0\}$ yields the following results at $t = 10$. For the FE scheme:

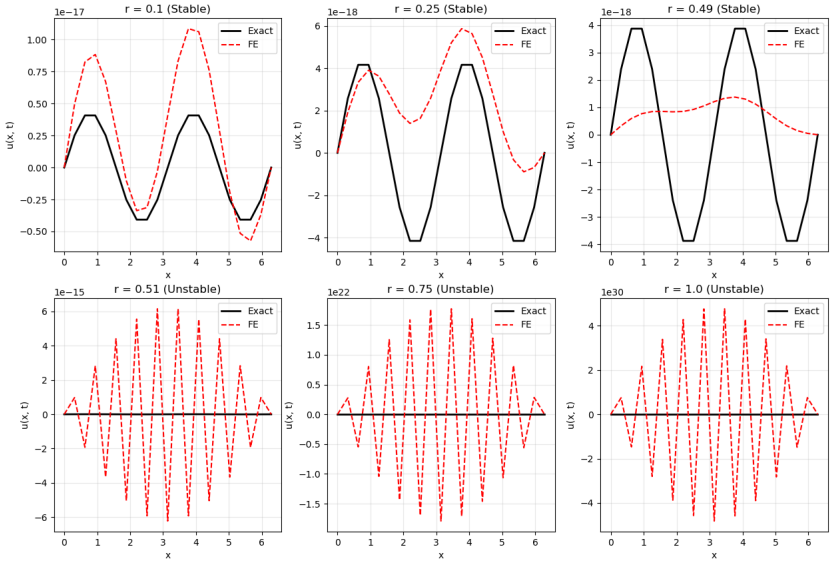

Figure c.7: FE and Exact Solutions for different r values at $t = 10$

For the BE scheme:

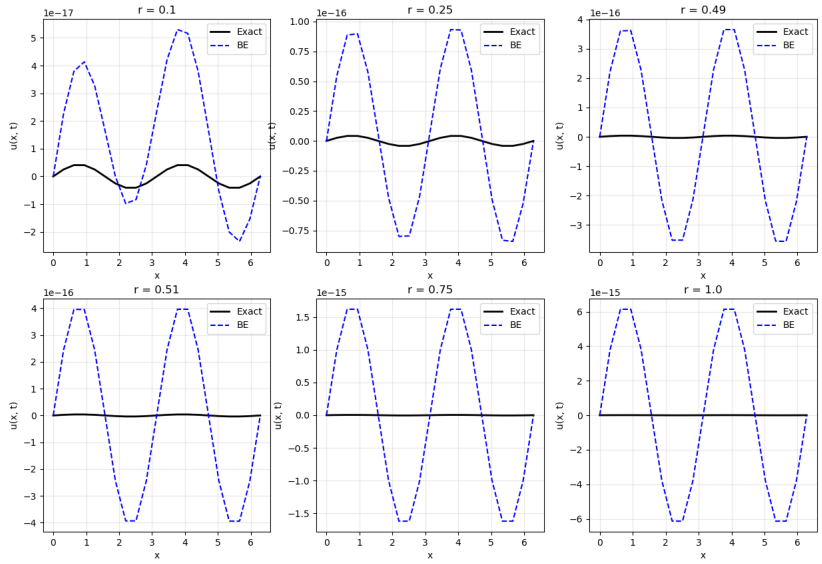

Figure c.8: FE and Exact Solutions for different r values at $t = 10$

For the CN scheme:

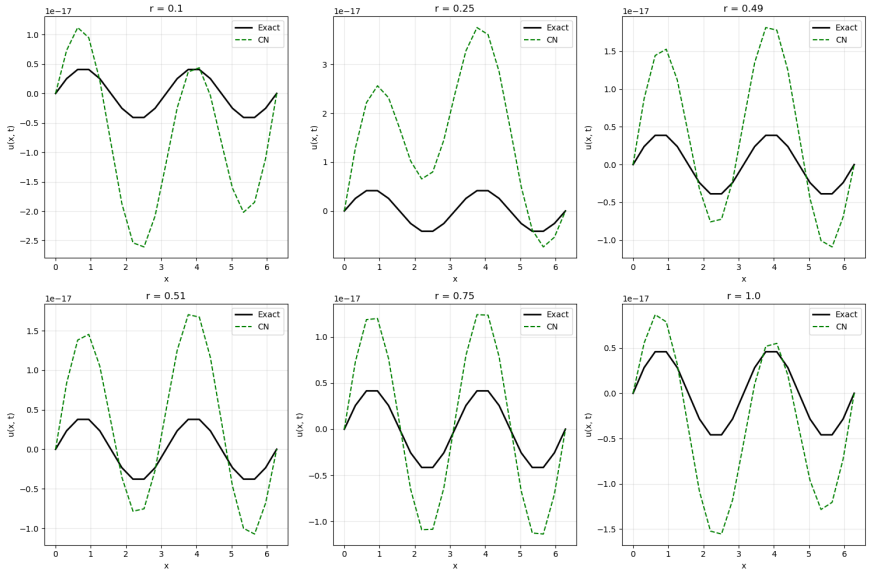

Figure c.9: FE and Exact Solutions for different r values at $t = 10$

It can be seen from figure c.7 that when the FE scheme operates at $r > 0.5$, the solutions grow without bound and have lots of oscillations. For BE and CN schemes in figure c.8 and c.9, for all r values, the solutions stay bounded and do not produce such oscillations. The results are thus consistent with the VN stability analysis, which states that CN and BE schemes are unconditionally stable, but FE is only stable when $0 < r < 0.5$.

c.2.3

To investigate this, I chose to keep $\Delta x = 0.1$ and $r = \{0.4, 0.6, 0.8, 1.0\}$, using the mean absolute error as the metric for accuracy. The $r = 0.4$ can serve as a reference for when $r < 0.5$. Obtaining the results at $t = 1.0$ yields figure c.10 and c.11.

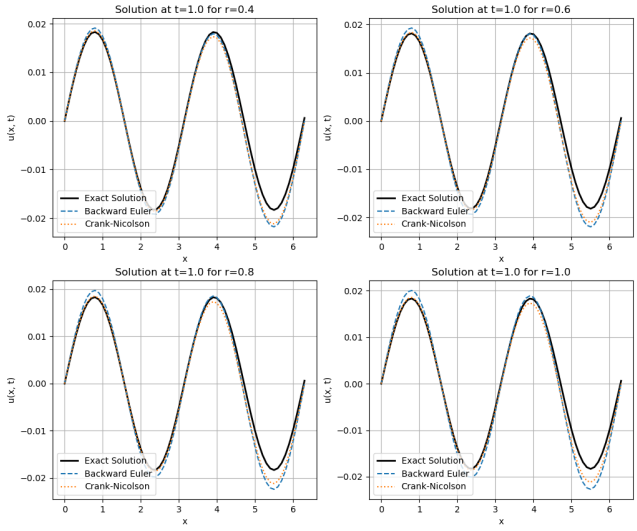

Figure c.10: BE, CN, and Exact Solutions for Different r Values at $t = 1.0$

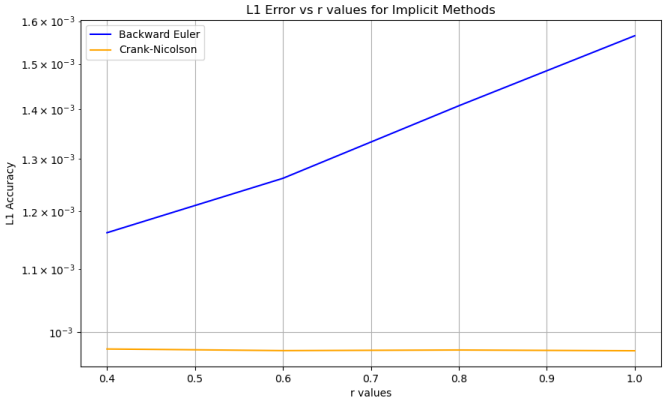

Figure c.11: Mean Absolute Error of BE and CN for Different r Values at $t = 1.0$

From fig c.10, it can be seen that there are hardly any visible differences between the solutions from the implicit methods operating at $r < 0.5$ and those operating at $r > 0.5$. In fig c.11, one can see that the CN scheme shows almost no change in accuracy as r varies. The BE scheme showed some decrease in accuracy, with the error rising from 1.2×10^{-3} to 1.6×10^{-3} as r goes up, but the variation is still very small compared to the scale of the actual solution, whose amplitude goes up to 2×10^{-2}. This small change in error could be due to the fact that Δt also increased as r increased. In conclusion, the accuracy of the implicit schemes suffers no significant impact when operating at $r > 0.5$.

d

d.1

Obtaining the solutions at $t = 0.5$ yields the following plots.

For the Forward Euler scheme:

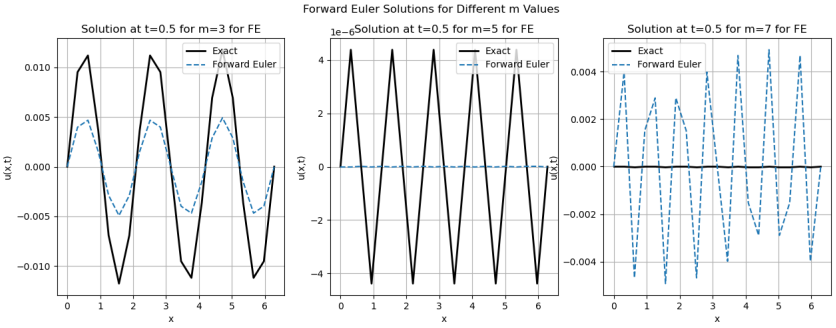

Figure d.1: Exact and FE solutions for m=3 at t=0.5

For the Backward Euler scheme:

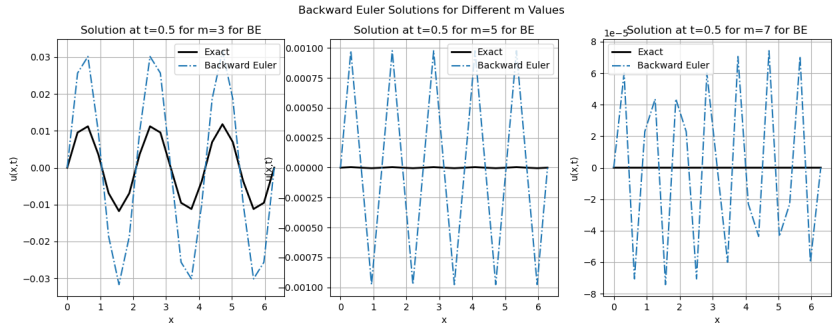

Figure d.2: Exact and FE solutions for m=3 at t=0.5

For the Crank-Nicolson scheme:

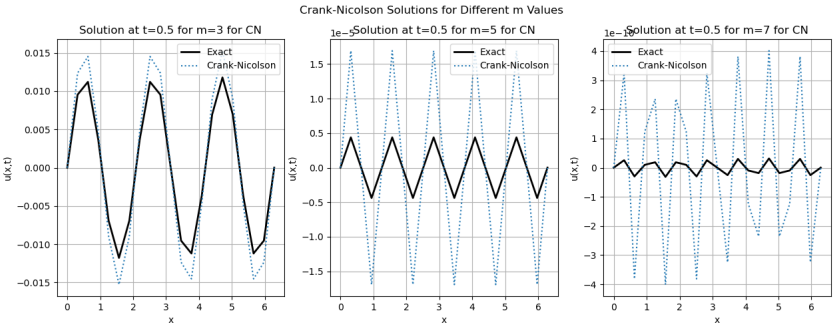

Figure d.3: Exact and FE solutions for m=3 at t=0.5

d.2

The amplification factor for the exact solution G_k^e is given by:

$$G_k^e = \exp(-r\beta_k^2)$$

The amplification factor for Forward Euler G_k^{FE} is given by:

$$G_k^{FE} = 1 - 4r\sin^2\frac{\beta_k}{2}$$

The amplification factor for Backward Euler G_k^{BE} is given by:

$$G_k^{BE} = \frac{1}{1 + 4r\sin^2\frac{\beta_k}{2}}$$

The amplification factor for Crank Nicolson G_k^{CN} is given by:

$$G_k^{CN} = \frac{1 - 2r\sin^2\frac{\beta_k}{2}}{1 + 2r\sin^2\frac{\beta_k}{2}}$$

Put $m_1 = 3, m_2 = 5, m_3 = 7$ and the corresponding β becomes $\beta_{m_1} = \frac{3}{10\pi}, \beta_{m_2} = \frac{5}{10\pi}, \beta_{m_3} = \frac{7}{10\pi}$. Plugging the values into the formulas for the amplification factors yields the following table:

	G_k^e	G_k^{FE}	G_k^{BE}	G_k^{CN}
m_1	0.64	0.59	0.71	0.66
m_2	0.29	0	0.5	0.33
m_3	0.09	-0.59	0.39	0.11

Table d.1: Values of Amplification Factor for Different Solutions

A visualization of the amplification factors can be found in fig d.4:

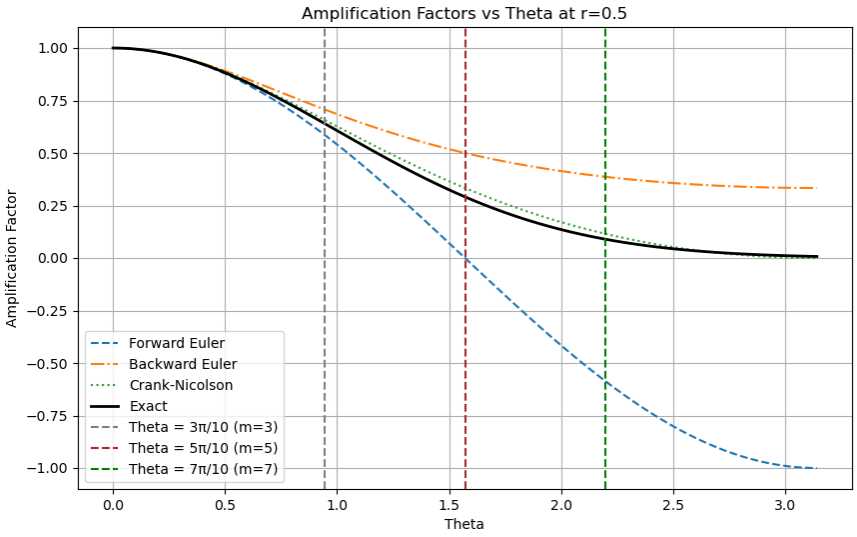

Figure d.4: Visualization of Amplification Factors for Different Solutions for $r = 0.5$

It can be observed that the plots in section d.1 showed what one would expect from the amplification factors.

Take the FE method for example. Even though the frequencies (wavenumbers) of the solution are different in m_1 and m_2, the magnitudes of these two FE solutions are roughly the same, as supported by the fact that $|G_{m_1}^{FE}| = |G_{m_3}^{FE}| = 0.59$. The FE solution of m_2 stays at zero because $G_{m_2}^{FE} = 0$. The lack of smoothing property for the FE scheme at $r = 0.5$ therefore results in poor accuracy.

For both the BE and CN schemes, it can be observed in fig d.2 and d.3 that as frequency increases, the magnitude of the numerical solutions decreases, but also that CN solutions decrease at a faster rate than BE solutions. These are in line with the fact that the amplification factor of CN decreases faster than that of BE.

Moreover, it can also be concluded from the figures that higher modes are more difficult to predict.