```python
import numpy as np
from scipy.linalg import hilbert
from scipy.sparse.linalg import cg
from scipy.linalg import solve
from numpy.linalg import cond
import pandas as pd

# Define the matrix dimensions to be tested
n_values = [5, 9, 20, 100]
results = []

for n in n_values:
    # Create Hilbert matrix and exact solution
    A = hilbert(n)
    x_exact = np.ones(n)
    b = A @ x_exact  # Generate b based on known solution x

    # 1. Compute the condition number
    condition_number = cond(A)

    # 2. Solve using the direct method
    x_direct = solve(A, b)
    error_direct = np.linalg.norm(x_exact - x_direct)

    # 3. Solve using Preconditioned Gradient Descent (PG) method with
iteration count
    def preconditioned_gradient_descent(A, b, M, x0=None, tol=1e-7,
max_iterations=100000):
        n = len(b)
        x = np.zeros_like(b) if x0 is None else x0
        iteration_count = 0  # Initialize iteration counter
        r = b - A @ x  # Initial residual
        while iteration_count < max_iterations and np.linalg.norm(r) >
tol:
            z = M @ r  # Apply preconditioner
            alpha = (r @ z) / (z @ (A @ z))  # Compute step size
            x += alpha * z  # Update solution
            r -= alpha * (A @ z)  # Update residual
            iteration_count += 1  # Increment iteration count
        return x, iteration_count

    # Diagonal preconditioner for PG
    M_pg = np.diag(1 / np.diag(A))
    x_pg, pg_iterations = preconditioned_gradient_descent(A, b, M_pg)
    error_pg = np.linalg.norm(x_exact - x_pg)

    # 4. Solve using PCG method with a diagonal preconditioner and
custom iteration counter
    M_pcg = np.diag(1 / np.diag(A))  # Preconditioner matrix as the
inverse of the diagonal entries of A
```

```
    # Custom iteration counter using a mutable list
    iteration_count = [0]
    def iteration_callback(xk):
        iteration_count[0] += 1

    # Use CG with preconditioning and capture the iteration
information
    x_pcg, pcg_info = cg(A, b, M=M_pcg, tol=1e-10, maxiter=10000,
callback=iteration_callback)
    error_pcg = np.linalg.norm(x_exact - x_pcg)

    # Set the PCG iteration count based on convergence information
    pcg_iterations = iteration_count[0] if pcg_info == 0 else "Reached
Maximum Iterations"

    # Save results
    results.append({
        'n': n,
        'K(A)': f"{condition_number:.2e}",
        'Direct Error': f"{error_direct:.2e}",
        'PG Error': f"{error_pg:.2e}",
        'PG Iter': pg_iterations,
        'PCG Error': f"{error_pcg:.2e}",
        'PCG Iter': pcg_iterations
    })

# Display results in a nicely formatted table
df = pd.DataFrame(results)
df = df.rename(columns={
    'n': 'n', 'K(A)': 'K(A)', 'Direct Error': 'Direct Error', 'PG
Error': 'PG Error',
    'PG Iter': 'PG Iter', 'PCG Error': 'PCG Error', 'PCG Iter': 'PCG
Iter'
})
df.index = range(1, len(df) + 1)
styled_df = df.style.set_table_styles(
    [{'selector': 'th', 'props': [('font-weight', 'bold')]}]
).set_caption("Results for Solving Hilbert Matrix Linear Systems")

# For Jupyter Notebook, display styled DataFrame
styled_df
```

```
C:\Users\jhyang\AppData\Local\Temp\ipykernel_17396\3310942156.py:46:
DeprecationWarning: 'scipy.sparse.linalg.cg' keyword argument `tol` is
deprecated in favor of `rtol` and will be removed in SciPy v1.14.0.
Until then, if set, it will override `rtol`.
  x_pcg, pcg_info = cg(A, b, M=M_pcg, tol=1e-10, maxiter=10000,
callback=iteration_callback)
C:\Users\jhyang\AppData\Local\Temp\ipykernel_17396\3310942156.py:15:
```

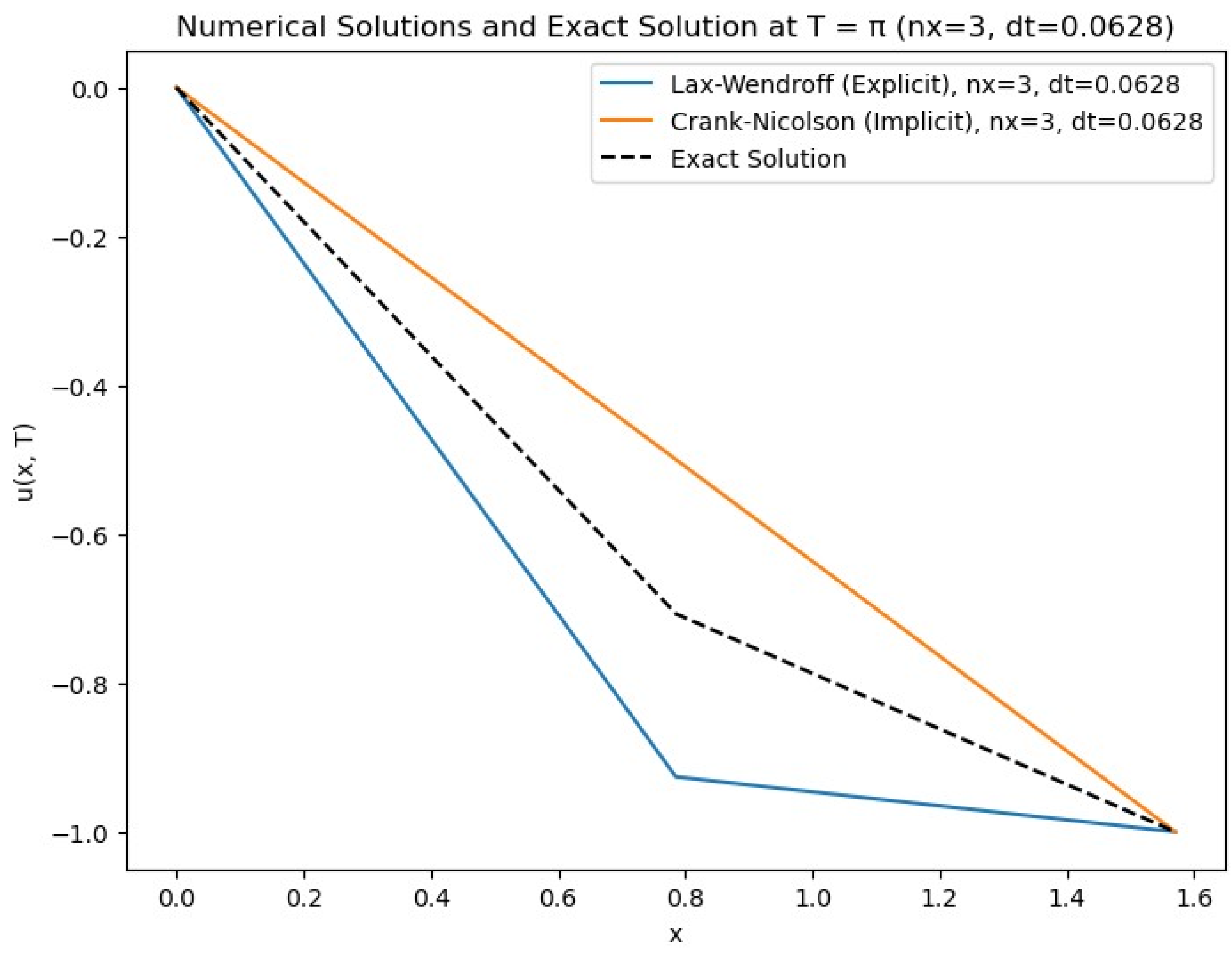
```
LinAlgWarning: Ill-conditioned matrix (rcond=5.62878e-20): result may
not be accurate.
  x_direct = solve(A, b)
C:\Users\jhyang\AppData\Local\Temp\ipykernel_17396\3310942156.py:15:
LinAlgWarning: Ill-conditioned matrix (rcond=3.29506e-21): result may
not be accurate.
  x_direct = solve(A, b)

<pandas.io.formats.style.Styler at 0x233e9bb2bd0>
```

Results for Solving Hilbert Matrix Linear Systems

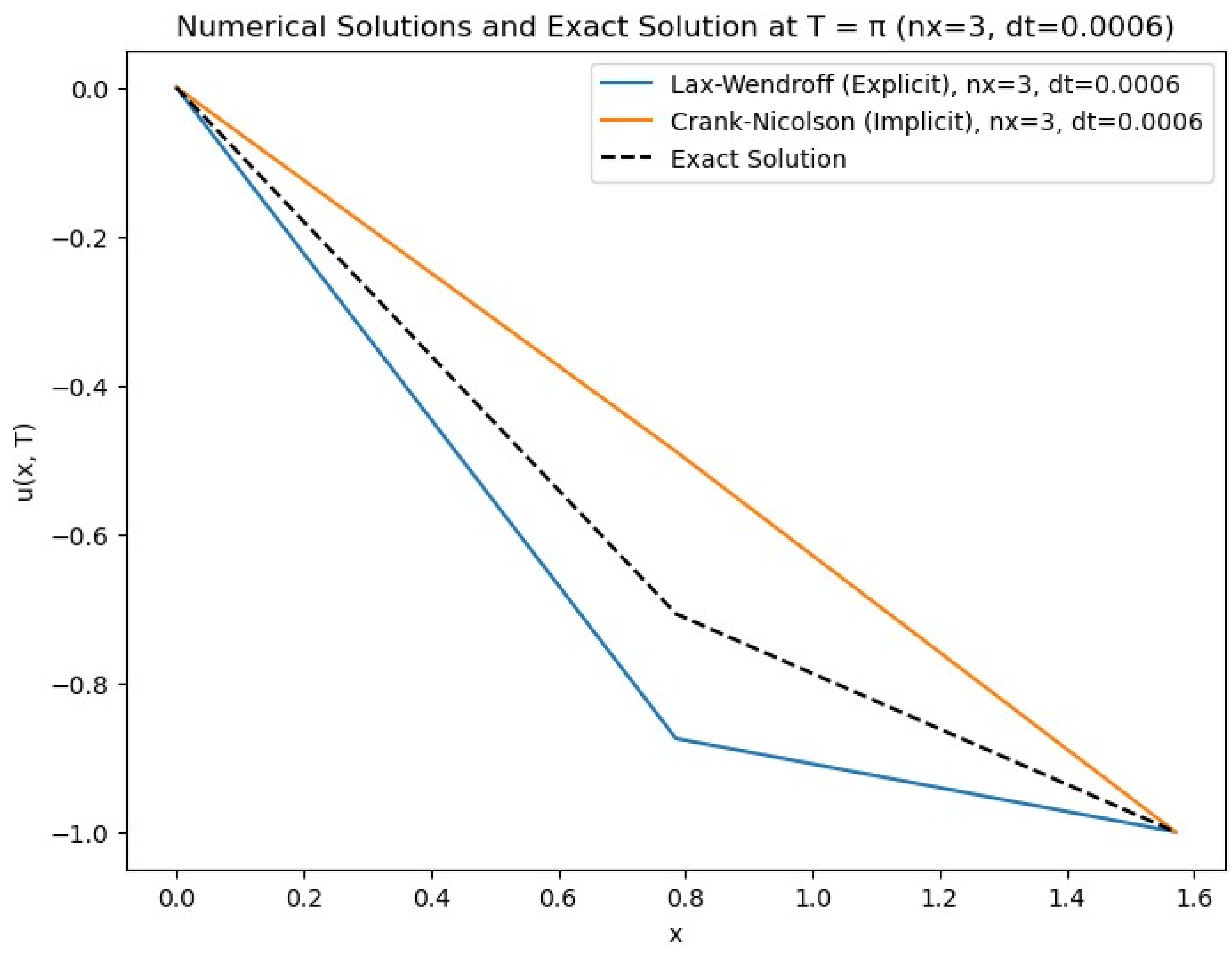
	n	K(A)	Direct Error	PG Error	PG Iter	PCG Error	PCG Iter
1	5	4.77e+05	8.46e-12	4.36e-03	6823	4.03e-12	7
2	9	4.93e+11	5.44e-05	4.42e-03	12489	7.55e-04	8
3	20	1.16e+18	2.35e+02	6.71e-03	13419	5.09e-04	13
4	100	1.08e+19	1.87e+03	6.61e-03	61033	1.18e-03	23

```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters
L = np.pi / 2  # Spatial domain length
T = np.pi  # End time of the simulation

# Define different values of nx and dt for comparison
grid_settings = [
    {'nx': 3, 'alpha': 0.1},
    {'nx': 3, 'alpha': 0.001},
    {'nx': 10, 'alpha': 0.01},
    {'nx': 20, 'alpha': 0.01}
]

# Exact solution function for comparison
def exact_solution(x, t_val):
    return np.sin(x) * np.cos(t_val)

# Boundary conditions function, applying Dirichlet boundary conditions
def boundary_conditions(u, t_val):
    u[0] = 0  # u(0, t) = 0
    u[-1] = np.cos(t_val)  # u(L, t) = cos(t)

# Source term function as given in the equation
def source_term(x, t_val):
    return -np.sin(x) * np.sin(t_val) + np.sin(x) * np.cos(t_val)

# Main loop for testing different grid resolutions and stability
factors
for setting in grid_settings:
    nx = setting['nx']  # Number of spatial grid points
    alpha = setting['alpha']  # Stability factor for explicit method

    # Calculate spatial and temporal step sizes
    dx = L / (nx - 1)  # Spatial step size
    dt = alpha * dx**2  # Time step size based on stability condition
for explicit method
    nt = int(T / dt) + 1  # Calculate number of time steps to reach T
    dt = T / (nt - 1)  # Recalculate dt to ensure it exactly divides T

    # Generate spatial and temporal grids
    x = np.linspace(0, L, nx)  # Spatial grid points
    t = np.linspace(0, T, nt)  # Temporal grid points

    # Initialize arrays to store solutions for explicit and Crank-
Nicolson methods
    u_explicit = np.zeros((nt, nx))  # Solution array for explicit
method
    u_crank_nicolson = np.zeros((nt, nx))  # Solution array for Crank-
```

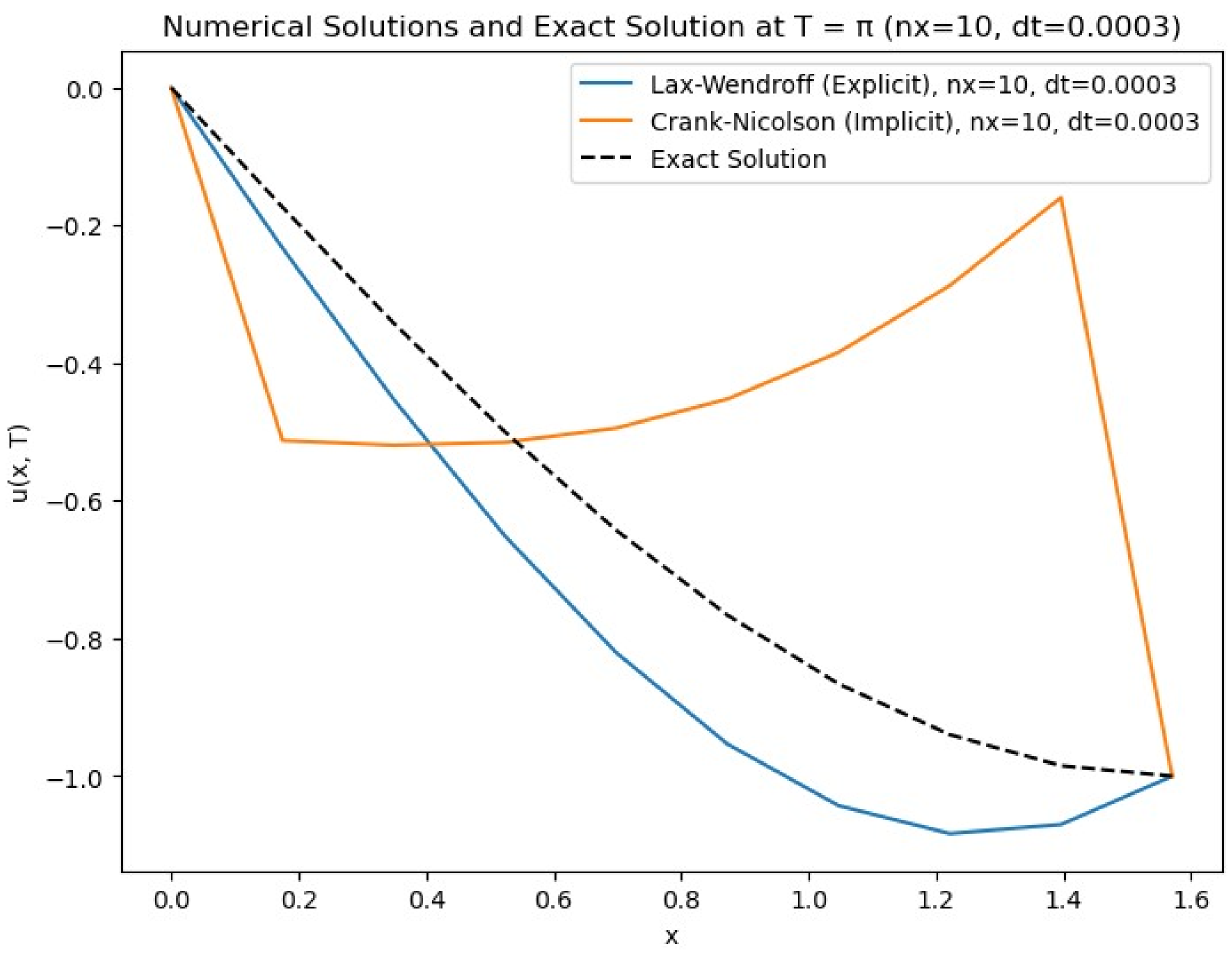
```python
Nicolson method

    # Set initial condition for both methods
    u_explicit[0, :] = np.sin(x)
    u_crank_nicolson[0, :] = np.sin(x)

    # Lax-Wendroff (Explicit) method with updated stability condition
    for n in range(0, nt - 1):
        # Apply boundary conditions for current time step
        boundary_conditions(u_explicit[n, :], t[n])
        for i in range(1, nx - 1):
            # Lax-Wendroff scheme to update interior points
            u_explicit[n + 1, i] = (u_explicit[n, i]
                                    + 0.5 * alpha * (u_explicit[n, i +
1] - 2 * u_explicit[n, i] + u_explicit[n, i - 1])
                                    + dt * source_term(x[i], t[n])
                                    + 0.5 * (alpha ** 2) *
(u_explicit[n, i + 1] - u_explicit[n, i - 1]))
        # Apply boundary conditions for next time step
        boundary_conditions(u_explicit[n + 1, :], t[n + 1])

    # Crank-Nicolson (Implicit) method with improved matrix filling
and boundary handling
    A = np.zeros((nx - 2, nx - 2))  # Tridiagonal matrix for Crank-
Nicolson
    B = np.zeros((nx - 2, nx - 2))  # Tridiagonal matrix for previous
time step
    b = np.zeros(nx - 2)  # Right-hand side vector

    # Fill matrices A and B for Crank-Nicolson scheme
    for i in range(nx - 2):
        A[i, i] = 1 + alpha  # Main diagonal of A
        B[i, i] = 1 - alpha  # Main diagonal of B
        if i > 0:
            A[i, i - 1] = -0.5 * alpha  # Lower diagonal of A
            B[i, i - 1] = 0.5 * alpha  # Lower diagonal of B
        if i < nx - 3:
            A[i, i + 1] = -0.5 * alpha  # Upper diagonal of A
            B[i, i + 1] = 0.5 * alpha  # Upper diagonal of B

    # Time-stepping loop for Crank-Nicolson
    for n in range(0, nt - 1):
        # Apply boundary conditions for the current time step
        boundary_conditions(u_crank_nicolson[n, :], t[n])
        # Set up the right-hand side vector b for Crank-Nicolson
        b = B @ u_crank_nicolson[n, 1:-1] + dt * source_term(x[1:-1],
t[n])
        # Adjust for boundary condition at right end
        b[0] += 0.5 * alpha * np.cos(t[n + 1])
        # Solve the tridiagonal system for the next time step
        u_crank_nicolson[n + 1, 1:-1] = np.linalg.solve(A, b)
        # Apply boundary conditions for the next time step
        boundary_conditions(u_crank_nicolson[n + 1, :], t[n + 1])

    # Calculate the exact solution at final time T for comparison
    u_exact = exact_solution(x, T)

    # Calculate relative errors for both methods
    norm_u_exact = np.linalg.norm(u_exact)  # Norm of the exact
solution
    error_explicit = np.linalg.norm(u_explicit[-1, :] - u_exact) /
norm_u_exact if norm_u_exact != 0 else np.nan
    error_crank_nicolson = np.linalg.norm(u_crank_nicolson[-1, :] -
u_exact) / norm_u_exact if norm_u_exact != 0 else np.nan

    # Plot results: Numerical solutions and exact solution at T
    plt.figure(figsize=(8, 6))
    plt.plot(x, u_explicit[-1, :], label=f'Lax-Wendroff (Explicit),
nx={nx}, dt={dt:.4f}')
    plt.plot(x, u_crank_nicolson[-1, :], label=f'Crank-Nicolson
(Implicit), nx={nx}, dt={dt:.4f}')
    plt.plot(x, u_exact, 'k--', label='Exact Solution')
    plt.xlabel('x')
    plt.ylabel('u(x, T)')
    plt.legend()
    plt.title(f'Numerical Solutions and Exact Solution at T = π
(nx={nx}, dt={dt:.4f})')
    plt.show()

    # Print errors for comparison
    print(f"For nx={nx}, dt={dt:.4f}:")
    print(f"Relative error for Lax-Wendroff Method (Explicit):
{error_explicit:.2e}")
    print(f"Relative error for Crank-Nicolson Method (Implicit):
{error_crank_nicolson:.2e}")
    print("-----------------------------------------------------")
```



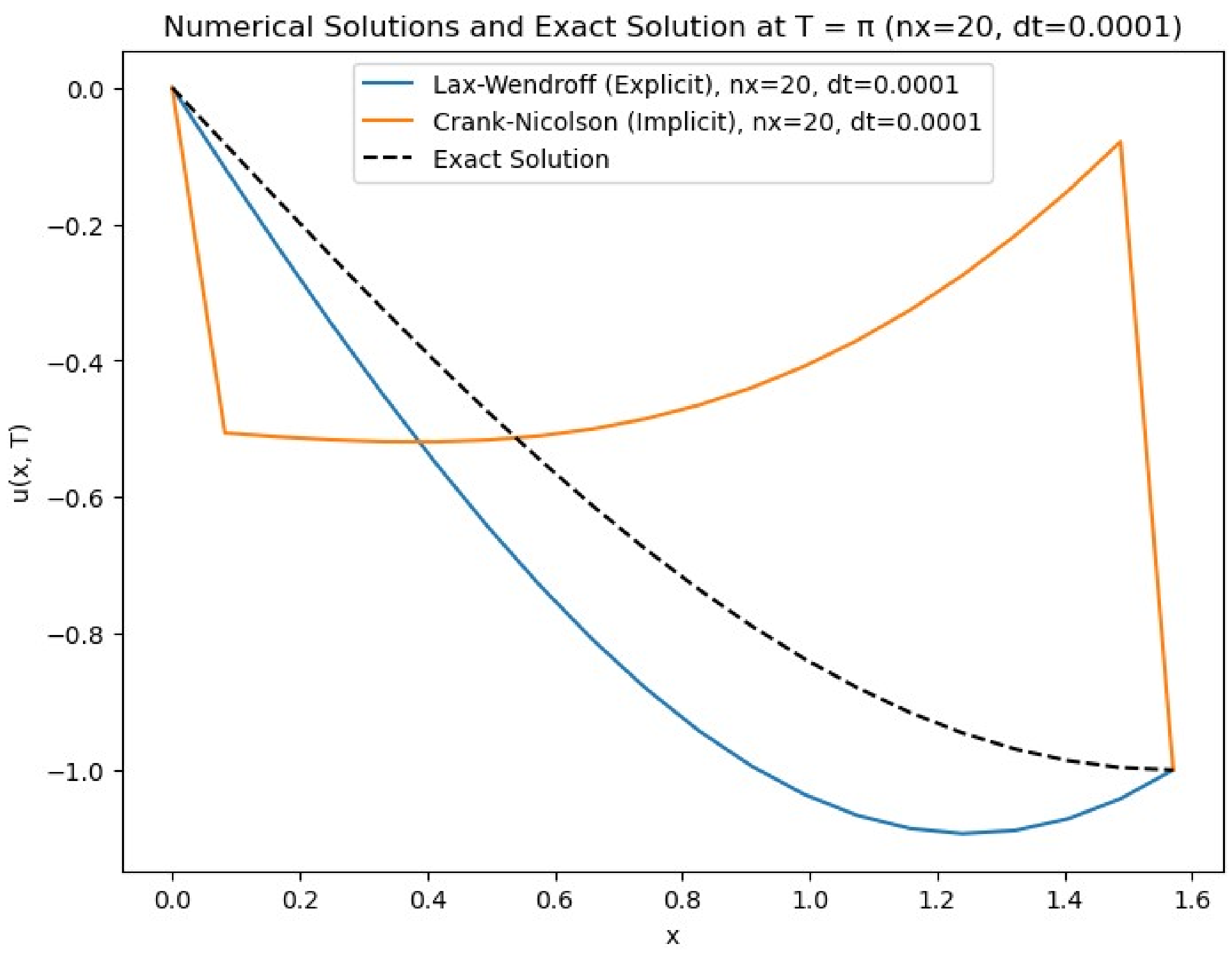
```
For nx=3, dt=0.0628:
Relative error for Lax-Wendroff Method (Explicit): 1.79e-01
Relative error for Crank-Nicolson Method (Implicit): 1.69e-01
----------------------------------------------------
```



```
For nx=3, dt=0.0006:
Relative error for Lax-Wendroff Method (Explicit): 1.37e-01
Relative error for Crank-Nicolson Method (Implicit): 1.78e-01
----------------------------------------------------
```



```
For nx=10, dt=0.0003:
Relative error for Lax-Wendroff Method (Explicit): 1.81e-01
Relative error for Crank-Nicolson Method (Implicit): 5.67e-01
--------------------------------------------------
```



```
For nx=20, dt=0.0001:
Relative error for Lax-Wendroff Method (Explicit): 2.07e-01
Relative error for Crank-Nicolson Method (Implicit): 6.34e-01
-----------------------------------------------------
```