```
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, atol=1e-10, maxiter=100000,
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 the results as a DataFrame
df = pd.DataFrame(results)

# Rename columns for better readability
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'
})

# Reset the index to start from 1 for better readability
df.index = range(1, len(df) + 1)

# Print the DataFrame as plain text
print(df.to_string(index=True))
```

```
C:\Users\jhyang\AppData\Local\Temp\ipykernel_16576\1634977234.py:15:
LinAlgWarning: Ill-conditioned matrix (rcond=2.93284e-20): result may
not be accurate.
```

```
  x_direct = solve(A, b)
C:\Users\jhyang\AppData\Local\Temp\ipykernel_16576\1634977234.py:15:
LinAlgWarning: Ill-conditioned matrix (rcond=8.9205e-21): result may
not be accurate.
  x_direct = solve(A, b)
```

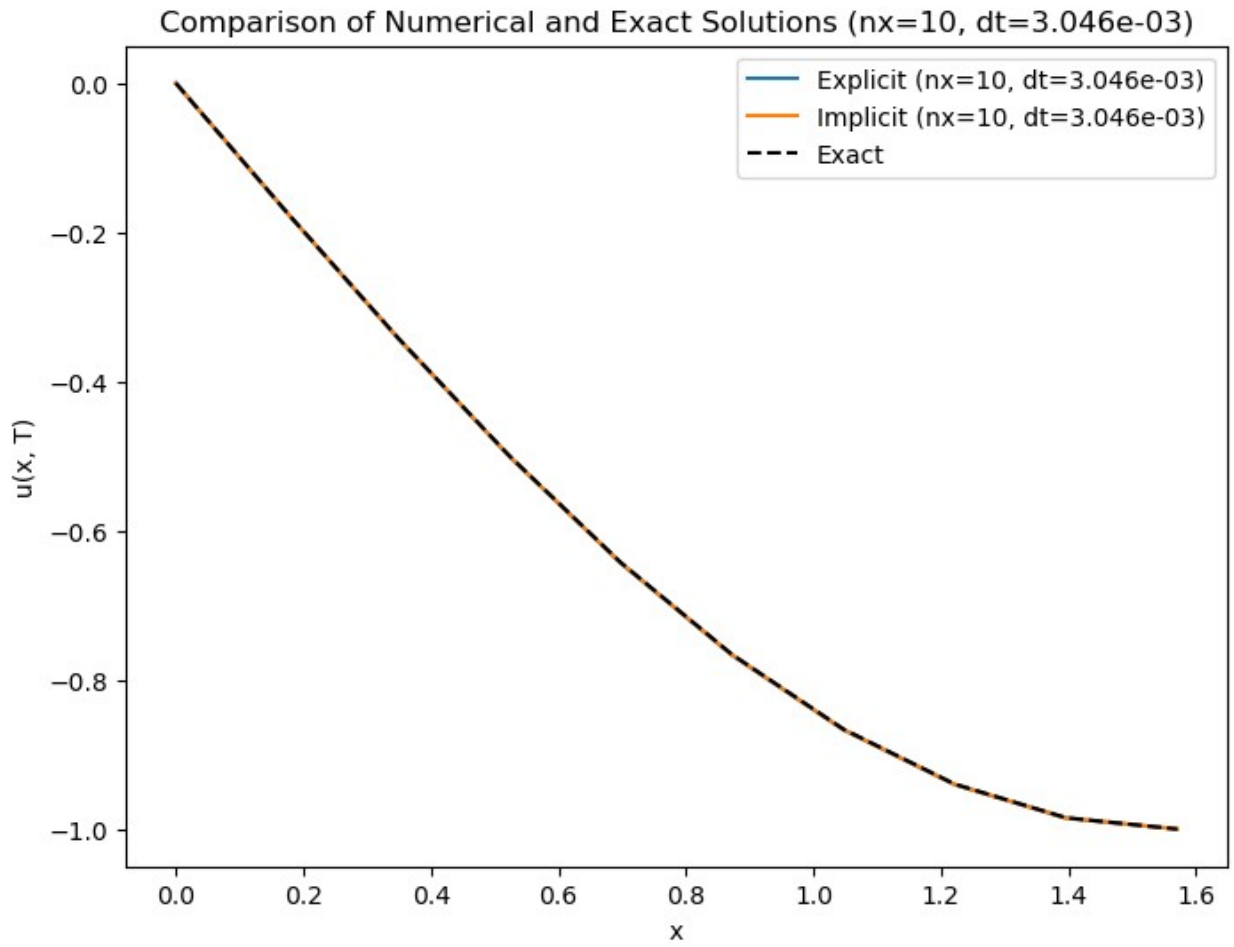
	n	K(A)	Direct Error	PG Error	PG Iter	PCG Error	PCG Iter
1	5	4.77e+05	4.03e-12	4.36e-03	6823	4.22e-02	3
2	9	4.93e+11	5.26e-05	4.42e-03	12489	2.79e-02	4
3	20	1.16e+18	1.60e+02	6.71e-03	13419	3.30e-02	5
4	100	1.08e+19	2.95e+03	6.61e-03	61033	1.32e-01	6

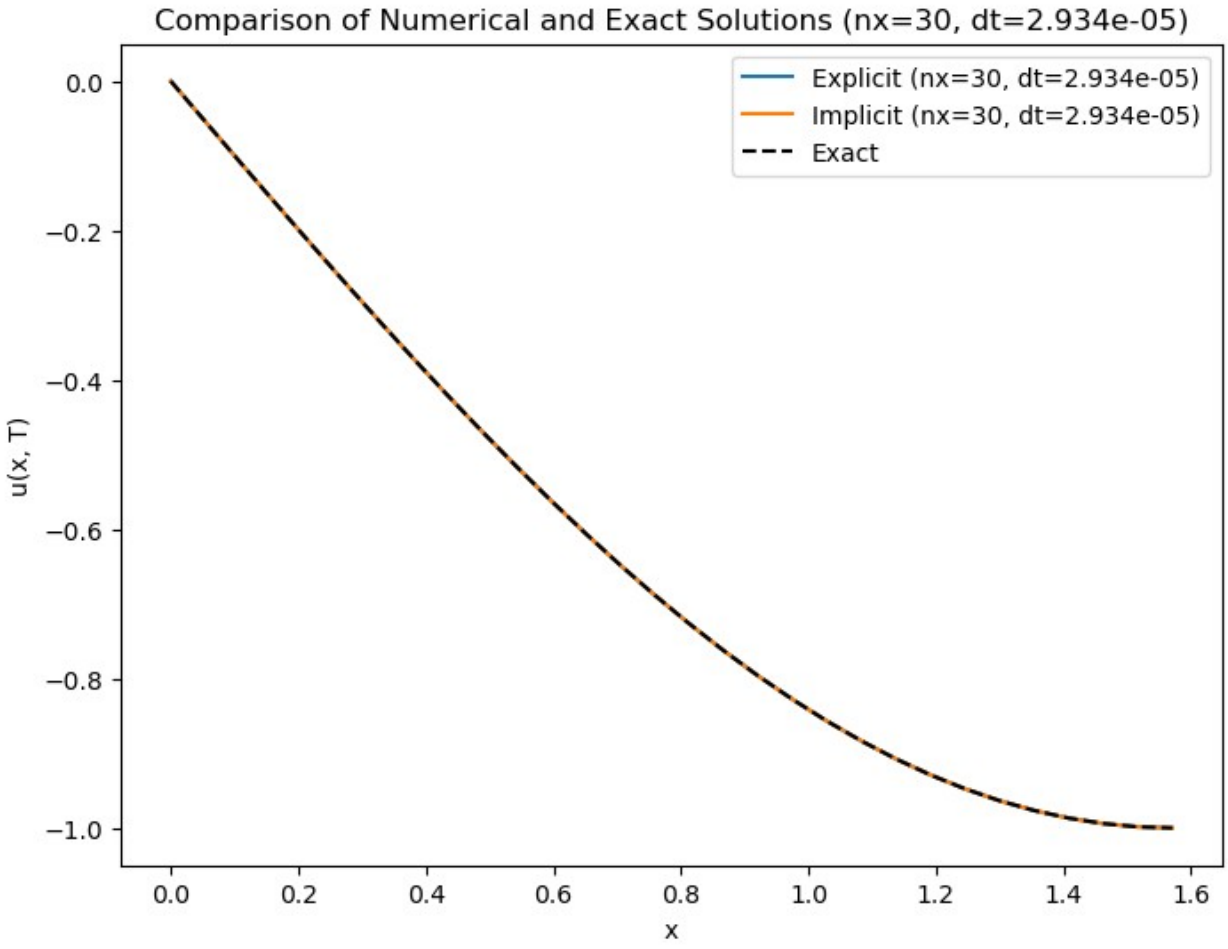
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve

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

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

# Boundary conditions function
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
def source_term(x, t_val):
    return -np.sin(x) * np.sin(t_val) + np.sin(x) * np.cos(t_val)

# Numerical solver
def solve_1d_heat(nx, alpha, method="explicit"):
    dx = L / (nx - 1)
    dt = alpha * dx**2  # Time step size for explicit method
    nt = int(T / dt) + 1
    dt = T / (nt - 1)  # Adjust dt to ensure exact time division

    x = np.linspace(0, L, nx)
    t = np.linspace(0, T, nt)

    u = np.zeros((nt, nx))
    u[0, :] = np.sin(x)  # Initial condition

    if method == "explicit":
        for n in range(0, nt - 1):
            boundary_conditions(u[n, :], t[n])
            for i in range(1, nx - 1):
                u[n + 1, i] = (u[n, i]
                               + alpha * (u[n, i + 1] - 2 * u[n, i] +
u[n, i - 1])
                               + dt * source_term(x[i], t[n]))
            boundary_conditions(u[n + 1, :], t[n + 1])

    elif method == "implicit":
        A = diags([-alpha, 1 + 2 * alpha, -alpha], [-1, 0, 1],
shape=(nx - 2, nx - 2)).toarray()
        for n in range(0, nt - 1):
```

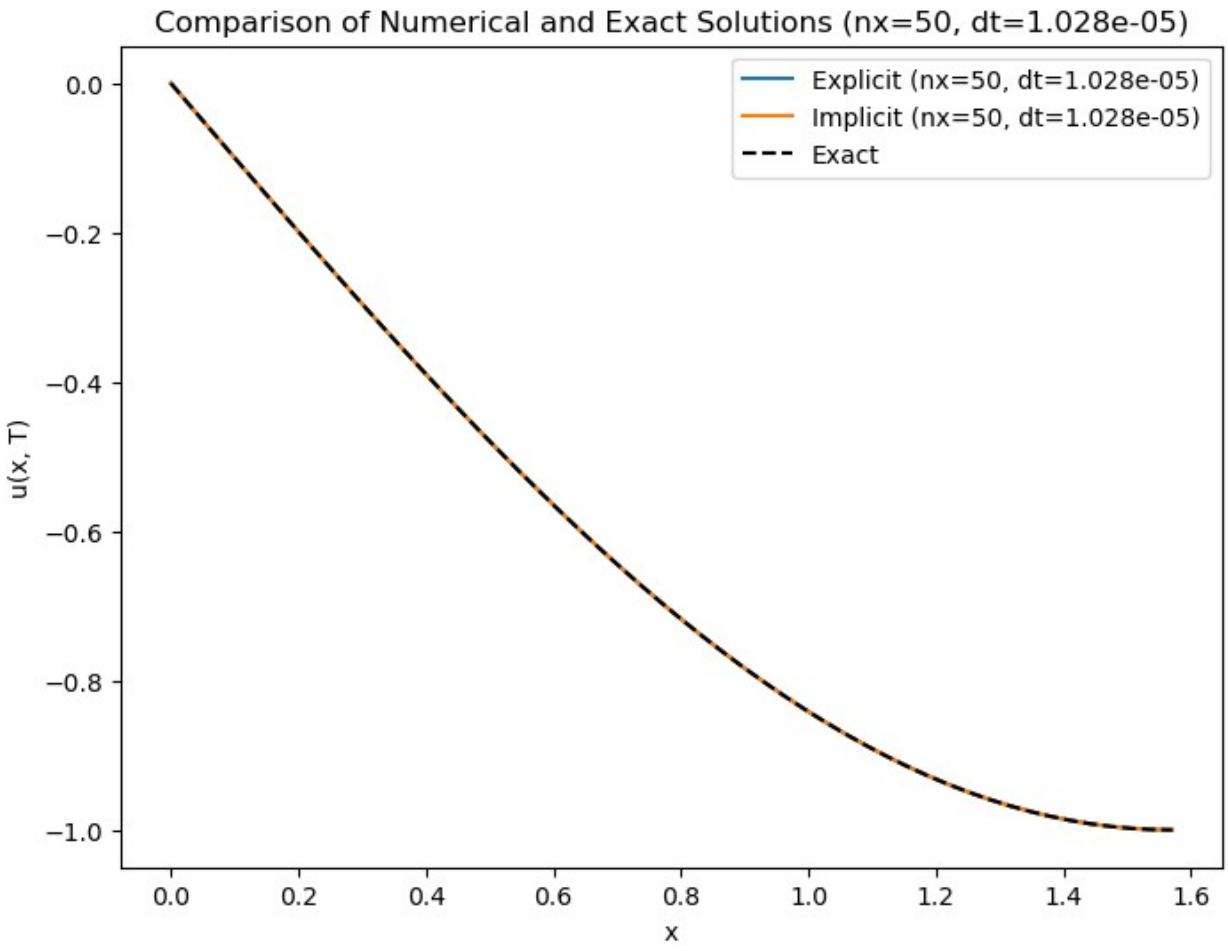
```
            boundary_conditions(u[n, :], t[n])
            b = u[n, 1:-1] + dt * source_term(x[1:-1], t[n])
            b[0] += alpha * 0  # Adjust for boundary condition at left
end
            b[-1] += alpha * np.cos(t[n + 1])  # Adjust for boundary
condition at right end
            u[n + 1, 1:-1] = spsolve(A, b)
            boundary_conditions(u[n + 1, :], t[n + 1])

    return x, t, u, dx

# Initialize a table to store results
results_table = []

# Error and plotting
grid_settings = [
    {'nx': 10, 'alpha': 0.1},
    {'nx': 30, 'alpha': 0.01},
    {'nx': 50, 'alpha': 0.01},
    {'nx': 70, 'alpha': 0.01}
]

for setting in grid_settings:
    nx, alpha = setting['nx'], setting['alpha']
    x, t, u_explicit, dx = solve_1d_heat(nx, alpha, method="explicit")
    _, _, u_implicit, _ = solve_1d_heat(nx, alpha, method="implicit")

    u_exact = exact_solution(x, T)
    norm_u_exact = np.linalg.norm(u_exact)

    error_explicit = np.linalg.norm(u_explicit[-1, :] - u_exact) /
norm_u_exact
    error_implicit = np.linalg.norm(u_implicit[-1, :] - u_exact) /
norm_u_exact

    plt.figure(figsize=(8, 6))
    plt.plot(x, u_explicit[-1, :], label=f'Explicit (nx={nx},
dt={alpha*dx**2:.3e})')
    plt.plot(x, u_implicit[-1, :], label=f'Implicit (nx={nx},
dt={alpha*dx**2:.3e})')
    plt.plot(x, u_exact, 'k--', label='Exact')
    plt.xlabel('x')
    plt.ylabel('u(x, T)')
    plt.title(f'Comparison of Numerical and Exact Solutions (nx={nx},
dt={alpha*dx**2:.3e})')
    plt.legend()
    plt.show()

    # Save results to the table
    results_table.append({
```

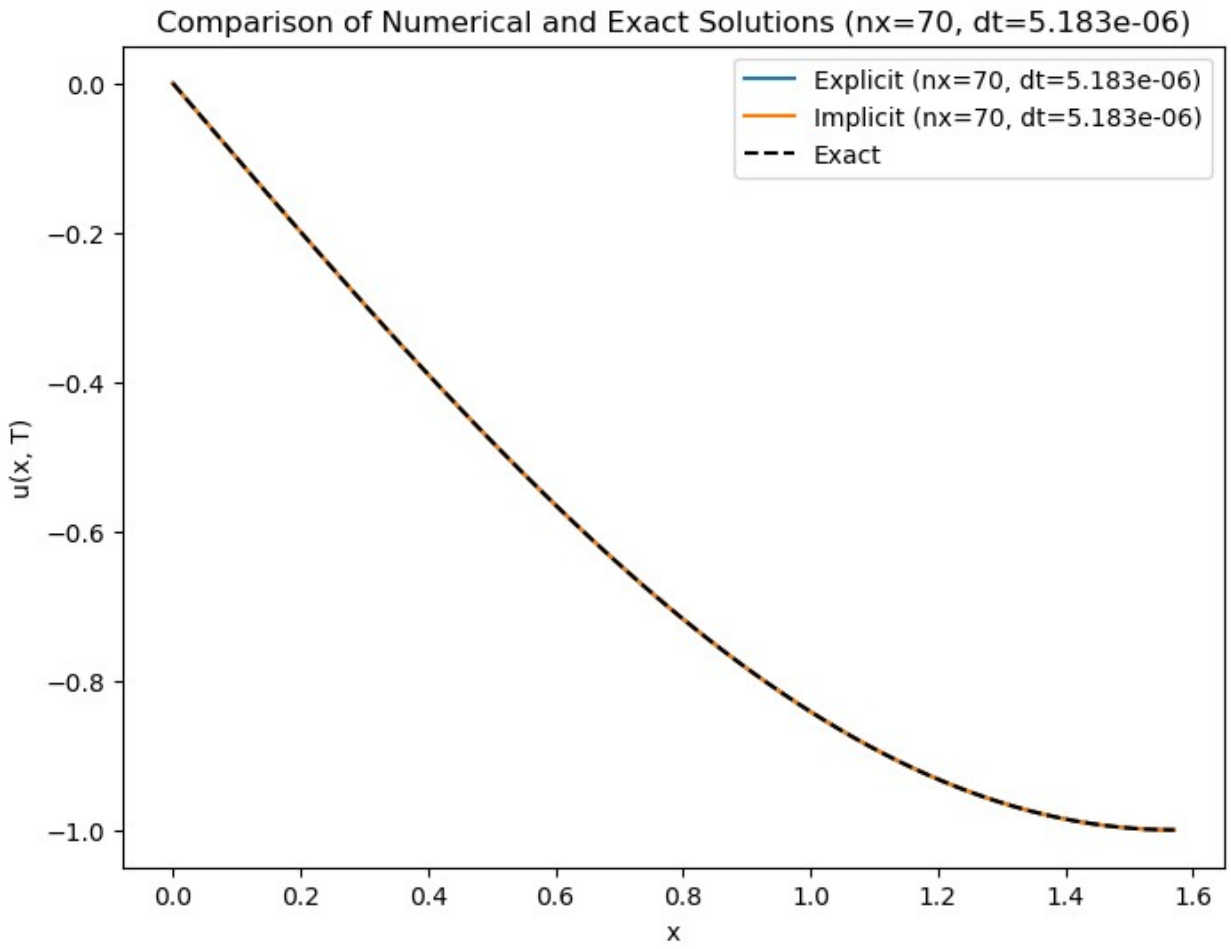
```
        "nx": nx,
        "alpha": alpha,
        "dx": dx,
        "dt": alpha * dx**2,
        "Relative Error (Explicit)": error_explicit,
        "Relative Error (Implicit)": error_implicit
    })

# Convert results to DataFrame and display
results_df = pd.DataFrame(results_table)
print(results_df)
```

```
C:\Users\jhyang\AppData\Local\Temp\ipykernel_9248\3979193567.py:47:
SparseEfficiencyWarning: spsolve requires A be CSC or CSR matrix
format
  u[n + 1, 1:-1] = spsolve(A, b)
```




Comparison of Numerical and Exact Solutions (nx=30, dt=2.934e-05)
Explicit (nx=30, dt=2.934e-05)
Implicit (nx=30, dt=2.934e-05)
Exact
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
u(x, T)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
x


Comparison of Numerical and Exact Solutions (nx=50, dt=1.028e-05)
Explicit (nx=50, dt=1.028e-05)
Implicit (nx=50, dt=1.028e-05)
Exact
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
u(x, T)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
x



	nx	alpha	dx	dt	Relative Error (Explicit)	Relative Error (Implicit)
0	10	0.10	0.174533	0.003046	0.000835	0.000688
1	30	0.01	0.054165	0.000029	0.000052	0.000051
2	50	0.01	0.032057	0.000010	0.000019	0.000018
3	70	0.01	0.022765	0.000005	0.000009	0.000009