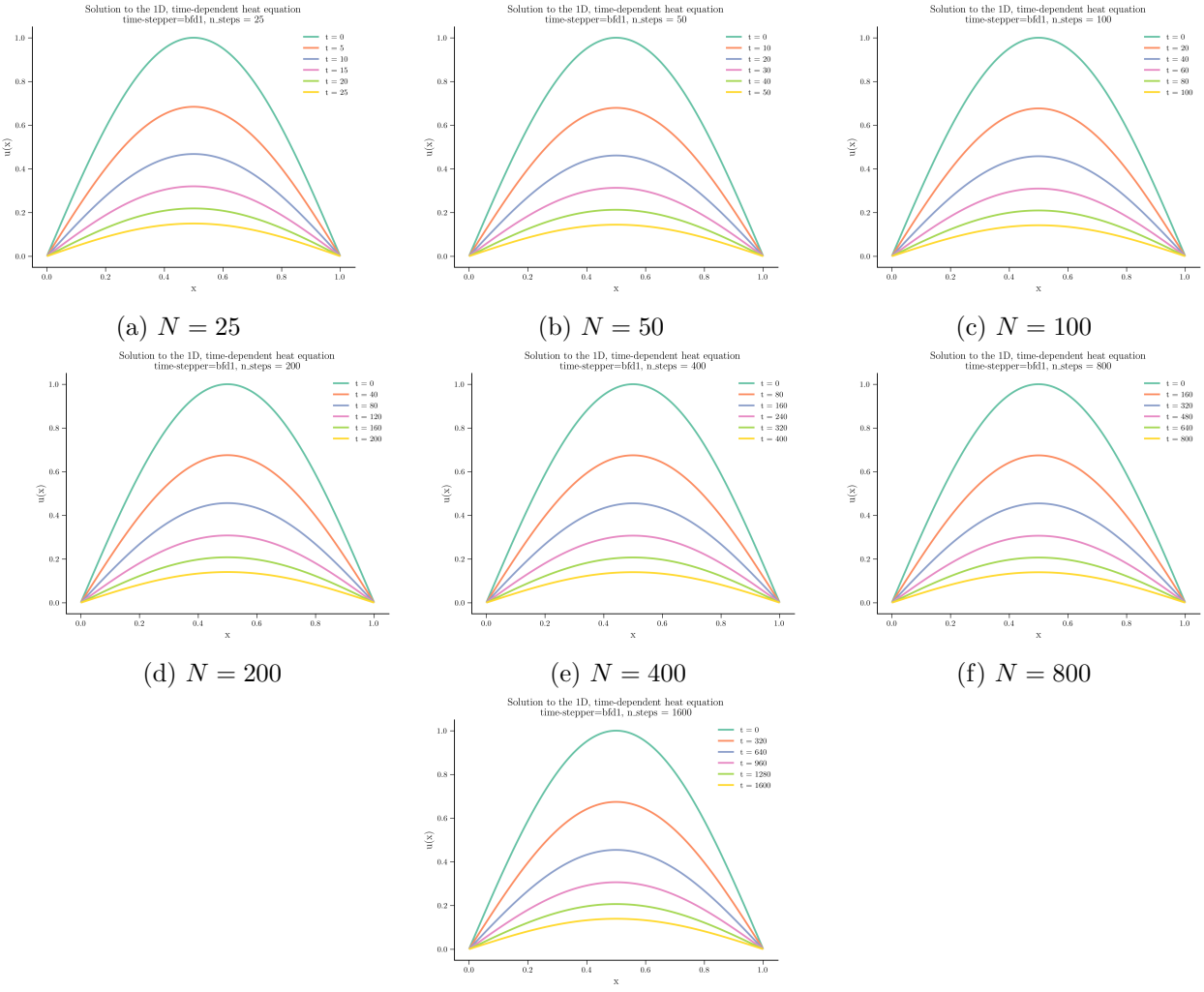
Liam Pohlmann, Alex Shaffer
Email: liamp2@illinois.edu, aws9@illinois.edu

CS 555: Homework # 1
Spring 2026

Problem 1
Convergence of BDF1, BDF2, and Crank-Nicholson.

Solution: For both prescribed initial conditions, BFD1 (Figure 1, Figure 4) and BFD2 (Figure 2 and Figure 5) perform well for a varying number of time steps. Additionally, both methods converged to the expected ratio of 2 and 4 for BFD1 and BFD2, respectively (see Table 1, Table 4 and Table 2, Table 5). The Crank-Nicholson method performed well for the sinusoidal initial condition (Figure 3, Table 3) but very poorly for the constant-value initial condition (Figure 6, Table 6). The solver appears to struggle resolving the solution on the boundaries, resulting in rapid oscillations, though this problem appeared to lessen as the number of timesteps increased.

Code was written in `Python` and included in this report.



(a) $N = 25$ (b) $N = 50$ (c) $N = 100$

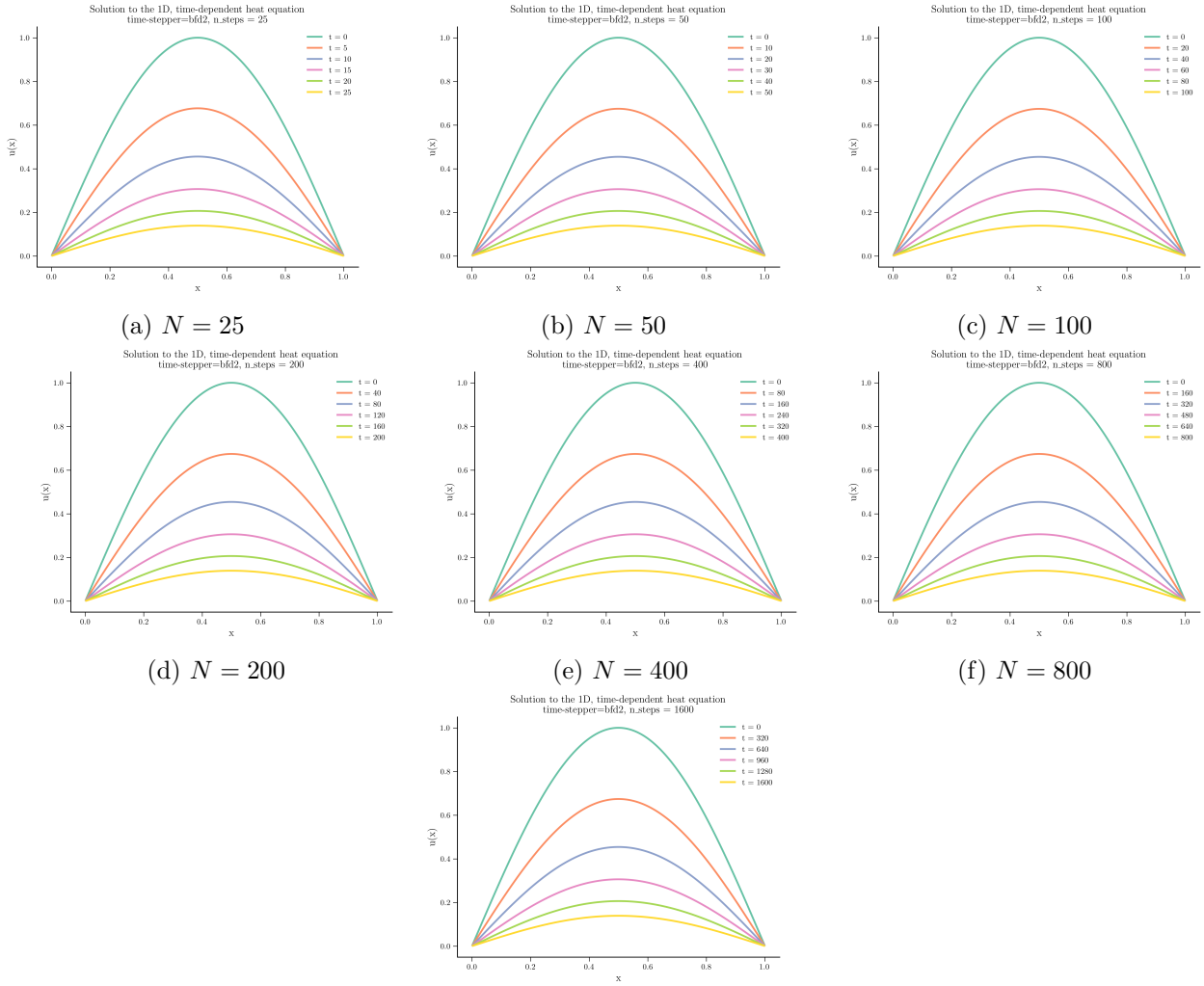
(d) $N = 200$ (e) $N = 400$ (f) $N = 800$

(g) $N = 1600$

Figure 1: BDF1 solution for Problem 1 with sinusoidal initial condition and an increasing number of timesteps.

nt	Relative L2 Error	Ratio
25	7.6865E-02	None
50	3.8698E-02	1.98629517948916
100	1.9415E-02	1.99315253596478
200	9.7243E-03	1.99657867816914
400	4.8663E-03	1.99828978420901
800	2.4342E-03	1.99914584597233
1600	1.2174E-03	1.99957098611054

Table 1: BFD1 Sinusoidal Initial Condition Output Table



(a) $N = 25$ (b) $N = 50$ (c) $N = 100$

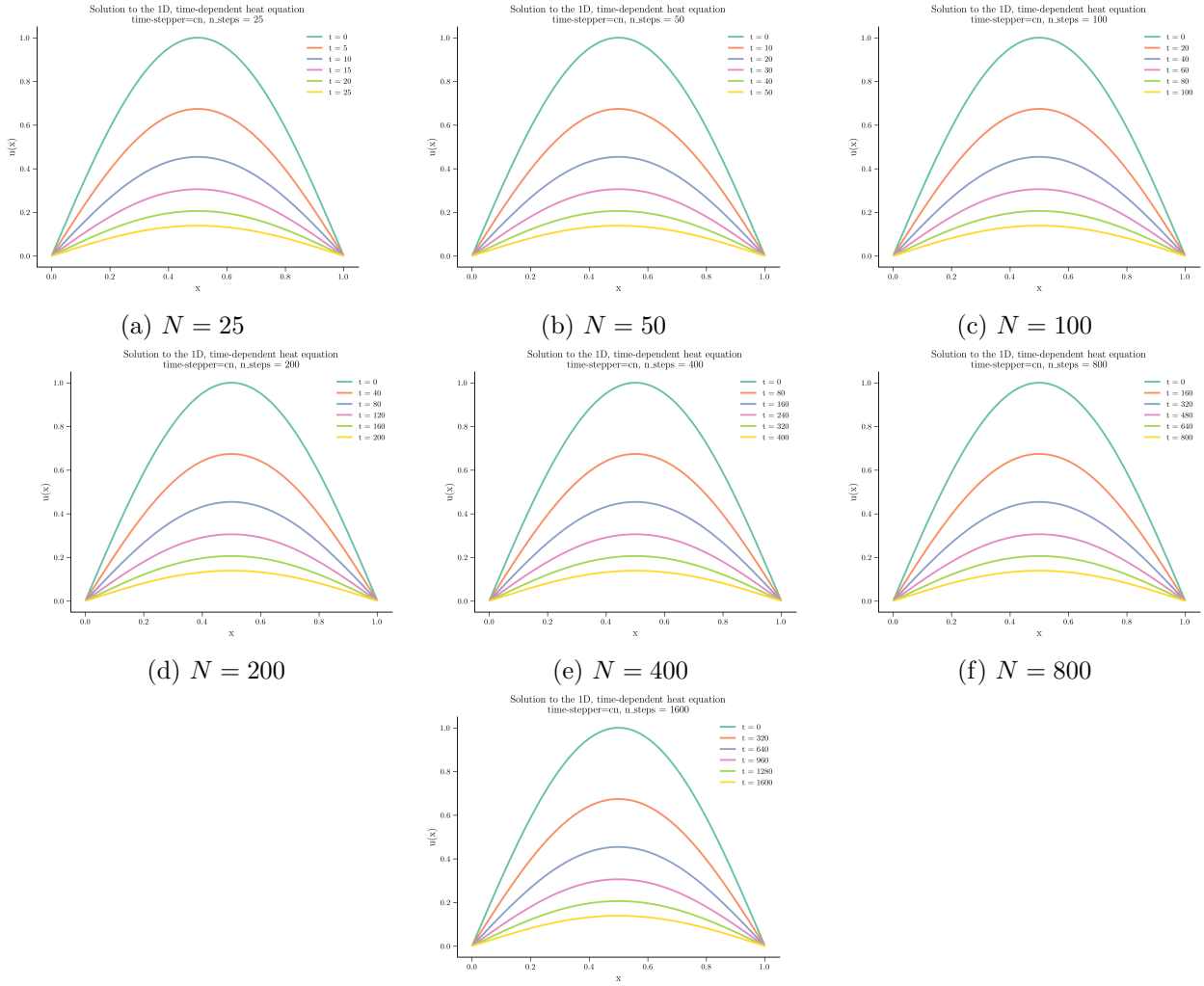
(d) $N = 200$ (e) $N = 400$ (f) $N = 800$

(g) $N = 1600$

Figure 2: BDF2 solution for Problem 1 with sinusoidal initial condition and an increasing number of timesteps.

nt	Relative L2 Error	Ratio
25	5.9969E-04	None
50	1.4517E-04	4.13089413461047
100	3.5991E-05	4.0335432954678
200	8.9772E-06	4.00916835052149
400	2.2414E-06	4.00523917104543
800	5.6085E-07	3.9963662517768
1600	1.3945E-07	4.02176317200121

Table 2: BFD2 Sinusoidal Initial Condition Output Table



(a) $N = 25$ (b) $N = 50$ (c) $N = 100$

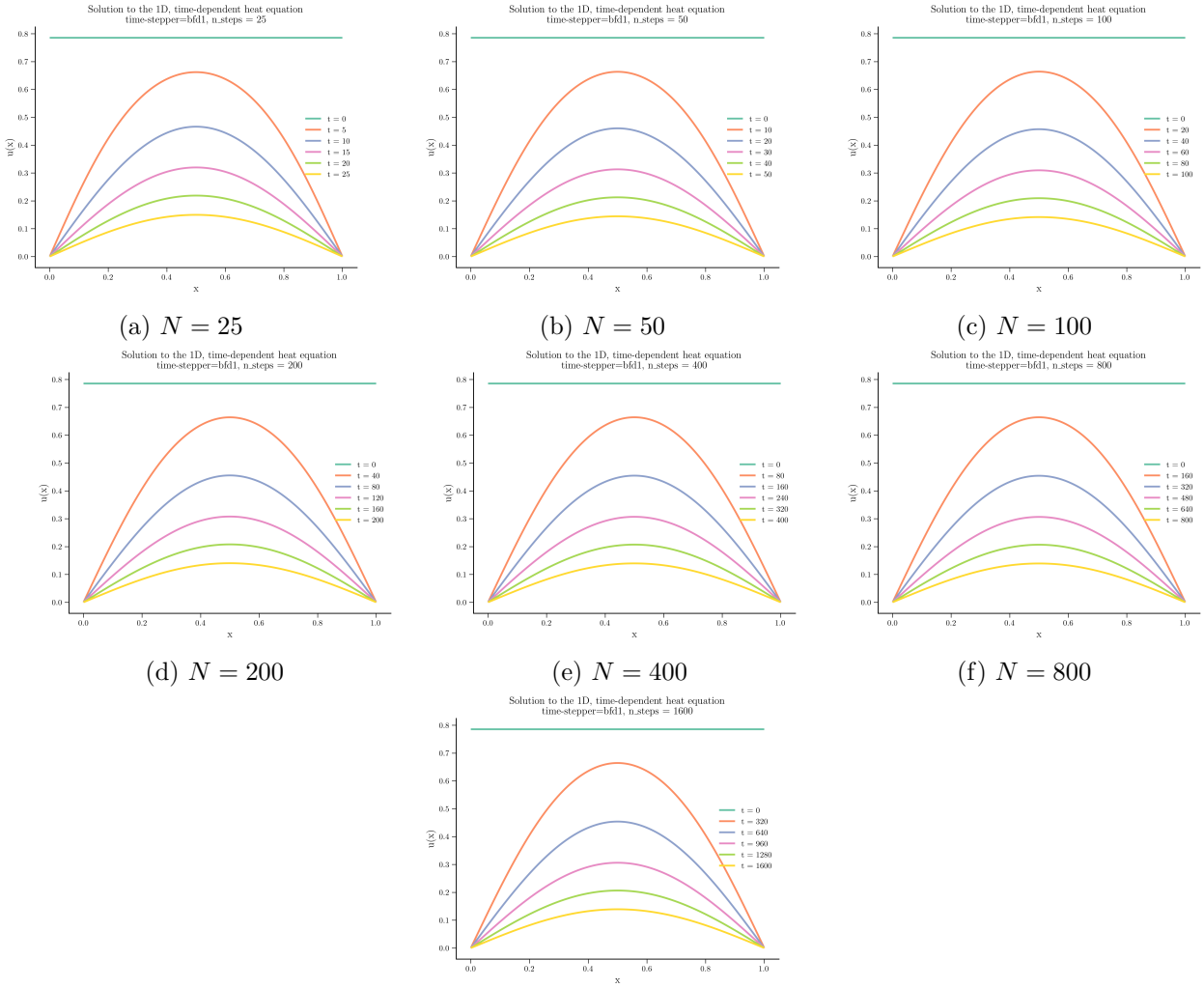
(d) $N = 200$ (e) $N = 400$ (f) $N = 800$

(g) $N = 1600$

Figure 3: Crank-Nicholson solution for Problem 1 with sinusoidal initial condition and an increasing number of timesteps.

nt	Relative L2 Error	Ratio
25	1.0259E-03	None
50	2.5639E-04	4.00132896073532
100	6.4090E-05	4.00049420088793
200	1.6019E-05	4.0008429926872
400	4.0023E-06	4.00243566210429
800	9.9742E-07	4.01269685473481
1600	2.4586E-07	4.05692780622427

Table 3: Crank-Nicholson Sinusoidal Initial Condition Output Table



(a) $N = 25$ (b) $N = 50$ (c) $N = 100$

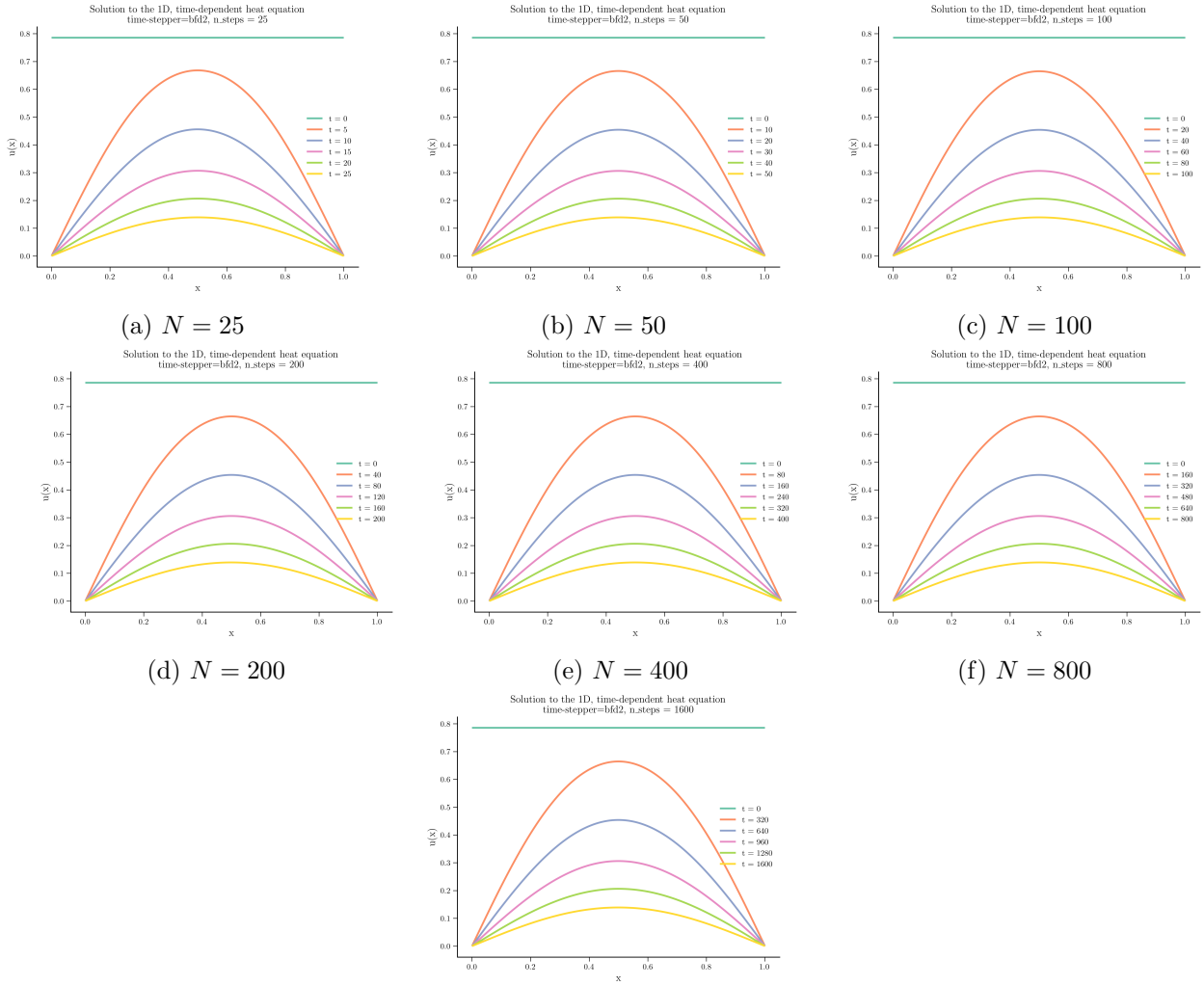
(d) $N = 200$ (e) $N = 400$ (f) $N = 800$

(g) $N = 1600$

Figure 4: BDF1 solution for Problem 1 with constant-value initial condition and an increasing number of timesteps.

nt	Relative L2 Error	Ratio
25	7.6865E-02	None
50	3.8698E-02	1.98629527605244
100	1.9415E-02	1.99315272312291
200	9.7243E-03	1.9965790456925
400	4.8663E-03	1.99829052801913
800	2.4342E-03	1.99914733965774
1600	1.2174E-03	1.99957398870672

Table 4: BFD1 Constant-Value Initial Condition Output Table



(a) $N = 25$ (b) $N = 50$ (c) $N = 100$

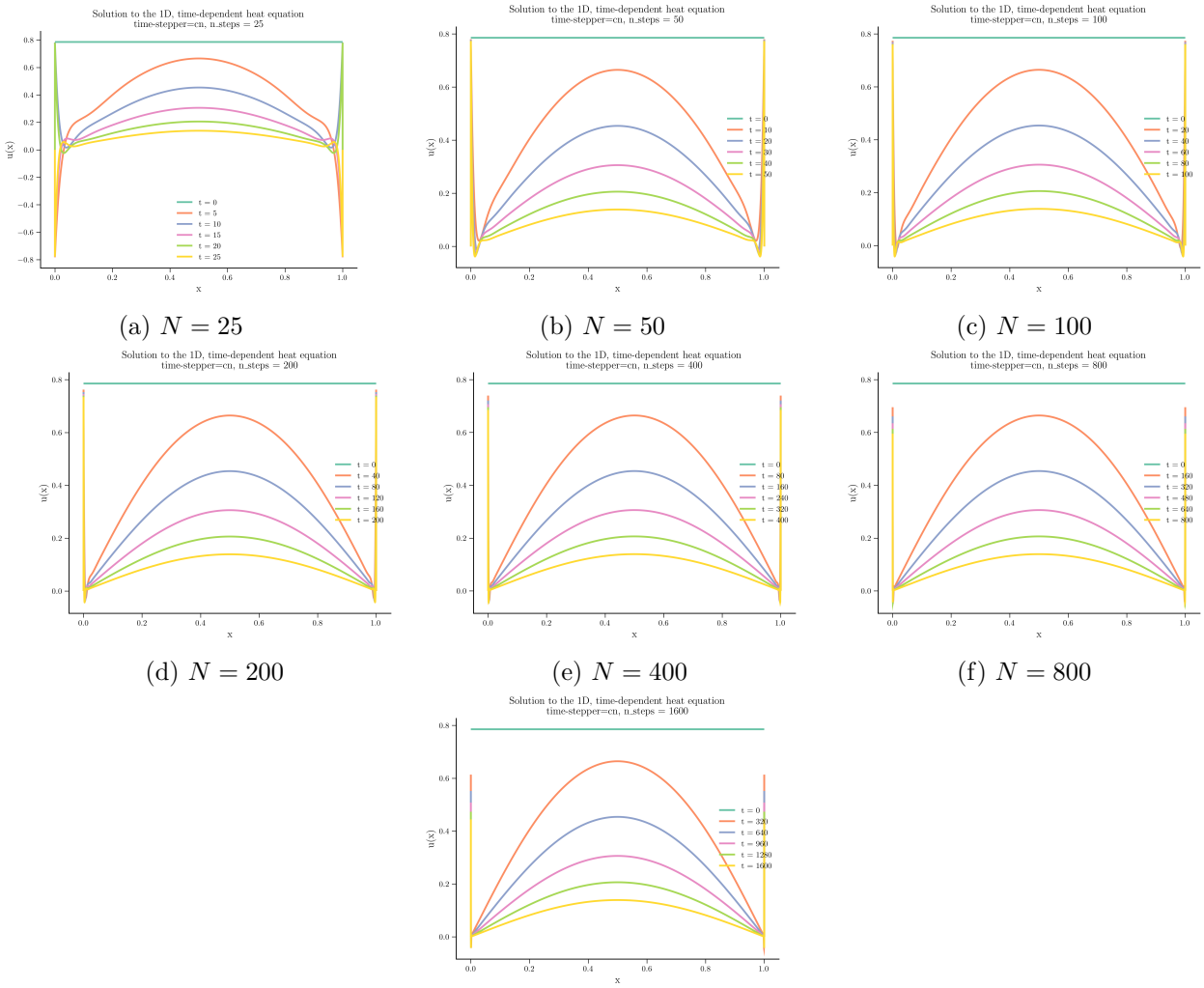
(d) $N = 200$ (e) $N = 400$ (f) $N = 800$

(g) $N = 1600$

Figure 5: BDF2 solution for Problem 1 with constant-value initial condition and an increasing number of timesteps.

nt	Relative L2 Error	Ratio
25	5.9968E-04	None
50	1.4517E-04	4.13097377163643
100	3.5987E-05	4.03384761961328
200	8.9736E-06	4.01033601077211
400	2.2382E-06	4.00931897266113
800	5.5911E-07	4.00312492433322
1600	1.4345E-07	3.89762898733195

Table 5: BFD2 Constant-Value Initial Condition Output Table



(a) $N = 25$ (b) $N = 50$ (c) $N = 100$

(d) $N = 200$ (e) $N = 400$ (f) $N = 800$

(g) $N = 1600$

Figure 6: Crank-Nicholson solution for Problem 1 with constant-value initial condition and an increasing number of timesteps.

nt	Relative L2 Error	Ratio
25	6.7230E-01	None
50	4.7317E-01	1.42083681543755
100	3.3144E-01	1.42761547073308
200	2.2994E-01	1.44143618794027
400	1.5639E-01	1.47032932084702
800	1.0198E-01	1.53353211016495
1600	6.0521E-02	1.68499598728917

Table 6: Crank-Nicholson Constant-Value Initial Condition Output Table

```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
from scipy.sparse import csr_matrix, identity
from scipy.sparse.linalg import spsolve

plt.style.use("matplotlib.rc")


def create_space_fd_mat(n, dx):
    diag = np.ones(n) * 2
    bands = -np.ones(n - 1)
    A = np.diag(diag) + np.diag(bands, k=-1) + np.diag(bands, k=1)
    dense_mat = A * (1 / dx**2)
    return csr_matrix(dense_mat)


def timeStep(u_prev1, u_prev2, dt, A, nu, time_stepper):
    match time_stepper:
        case "bfd1":
            return bfd1(u_prev1, dt, A, nu)
        case "bfd2":
            return bfd2(u_prev1, u_prev2, dt, A, nu)
        case "cn":
            return cn(u_prev1, dt, A, nu)
        case _:
            raise ValueError(
                f"time_stepper value not valid! time_stepper = {time_stepper}"
            )


def bfd1(u_prev, dt, A, nu):
    n = len(u_prev)
    mat = identity(n) + dt * nu * A
    u_next = spsolve(mat, u_prev)
    return u_next


def bfd2(u_prev1, u_prev2, dt, A, nu):
    n = len(u_prev1)
    mat = 3 * identity(n) + 2 * nu * dt * A
    vec = 4 * u_prev1 - u_prev2
    u_next = spsolve(mat, vec)
    return u_next

```

```python

def cn(u_prev, dt, A, nu):
    n = len(u_prev)
    coeff = dt / 2 * nu
    w = (identity(n) - coeff * A) @ u_prev
    mat = identity(n) + coeff * A
    u_next = spsolve(mat, w)
    return u_next


def ic(index, x):
    match index:
        case 1:
            return np.sin(np.pi * x)
        case 2:
            return np.pi / 4
        case _:
            raise ValueError(f"Index number passed not valid! index = {index}")


# ------------------------------------------------------------
# user-set parameters
# ------------------------------------------------------------
length = 1
nu = 0.2
t_max = 1
nx = 15000
n_timesteps = [25, 50, 100, 200, 400, 800, 1600]

time_stepper_index = 3  # 1=bfd1, 2=bfd2, 3=Crank-Nicholson
ic_index = 2  # value of 1 or 2
n_curves = 5  # number of curves in the plot


# ------------------------------------------------------------
# numerical solution
# ------------------------------------------------------------

xs, dx = np.linspace(0, length, nx, retstep=True)

time_steppers = ["bfd1", "bfd2", "cn"]
time_stepper = time_steppers[time_stepper_index - 1]

n = nx - 2  # homogeneous dirichlet conditions, so we don't need to solve for them
A = create_space_fd_mat(n, dx)

```

```
lam_ex = -nu * np.pi**2 / length

# initialize output table
df = pd.DataFrame(
    {"Time Differencing": [], "nx": [], "nt": [], "Relative L2 Error": [], "Ratio": []}
)

prev_error = None
for time_steps in n_timesteps:
    plt.figure()
    plt.xlabel("x")
    plt.ylabel("u(x)")
    plt.title(
        f"Solution to the 1D, time-dependent heat equation\ntime-stepper={time_stepper},
    )
    n_between_plots = round(time_steps / (n_curves))
    dt = t_max / time_steps
    u_prev = np.zeros(n)
    u_current = np.array([ic(ic_index, x) for x in xs])
    plt.plot(xs, u_current, label="t = 0")
    u_current = u_current[1:-1]

    for t in range(1, time_steps + 1):
        if time_stepper == "bfd2" and t == 1:
            # bootstrap if necessary
            u_next = timeStep(u_current, u_prev, dt, A, nu, "bfd1")
        else:
            u_next = timeStep(u_current, u_prev, dt, A, nu, time_stepper)
        if t == time_steps or t % n_between_plots == 0:
            prepend = np.insert(u_next, 0, 0)
            u_plot = np.append(prepend, 0)
            plt.plot(xs, u_plot, label=f"t = {t}")

        t_current = t * dt
        u_exact = np.array(
            [np.exp(t_current * lam_ex) * np.sin(np.pi * x) for x in xs[1:-1]]
        )
        error_vec = u_exact - u_next

        el2 = np.sqrt(np.dot(error_vec, error_vec) / np.dot(u_exact, u_exact))

        u_prev = u_current
        u_current = u_next

    if prev_error is not None:
        ratio = prev_error / el2
```

```
    else:
        ratio = None  # first run has no convergence ratio

    prev_error = el2

    new_table_row = [
        time_stepper,
        nx,
        time_steps,
        el2,
        ratio if ratio is not None else "None",
    ]
    df.loc[len(df)] = new_table_row

    plt.legend()
    plt.savefig(f"p1_{time_stepper}_{time_steps}.png")

df.to_csv(f"p1_table_{time_stepper}.csv")
```

Problem 2
Growth Factors

Solution: In part (a) we are asked to derive the asymptotic growth factor $G(\lambda\Delta t)$ for CN given that $\lambda \in \mathbb{R}$. The formula for CN is:

$$u^{n+1} = u^n + \frac{\Delta t}{2}\lambda(u^n + u^{n+1})$$

We look for the growth factor G such that $u^{n+1} = Gu^n$.

$$u^{n+1} = u^n + \frac{\Delta t}{2}\lambda(u^n + u^{n+1}) \tag{1}$$

$$u^{n+1} - \frac{\Delta t}{2}\lambda u^{n+1} = u^n + \frac{\Delta t}{2}\lambda u^n \tag{2}$$

$$u^{n+1}\left(1 - \frac{\Delta t}{2}\lambda\right) = u^n\left(1 + \frac{\Delta t}{2}\lambda\right) \tag{3}$$

$$G = \frac{1 + \frac{\Delta t}{2}\lambda}{1 - \frac{\Delta t}{2}\lambda} \tag{4}$$

As $\lambda\Delta t \to \infty$, we get:

$$\lim_{\lambda\Delta t\to\infty} G = \lim_{\lambda\Delta t\to\infty} \frac{1 + \frac{\Delta t}{2}\lambda}{1 - \frac{\Delta t}{2}\lambda} = -1$$

In part (b) we are asked to plot the growth factors of the analytic solution, forward Euler, backward Euler, and CN for $\lambda\Delta t \in [-10, 0]$.

Problem 3
`symamd` speedup.

Solution: (i) The original code gives the output seen in Table 7 with a runtime of 33.164 seconds.

N	Δt	$n_{\textbf{time steps}}$	**time**	$L2$ **Error**	**Ratio**
150	1.50E-01	8	1.20	2.5530E-01	3.9170E+00
150	8.00E-02	15	1.20	7.5673E-02	3.3737E+00
150	4.00E-02	30	1.20	1.9138E-02	3.9540E+00
150	2.00E-02	60	1.20	4.7981E-03	3.9887E+00
150	1.00E-02	120	1.20	1.2004E-03	3.9972E+00
150	5.00E-03	240	1.20	3.0015E-04	3.9993E+00
150	2.50E-03	480	1.20	7.5040E-05	3.9998E+00
150	1.25E-03	960	1.20	1.8760E-05	4.0000E+00

Table 7: Given code output.

Implemented code (only showing changed solver section):

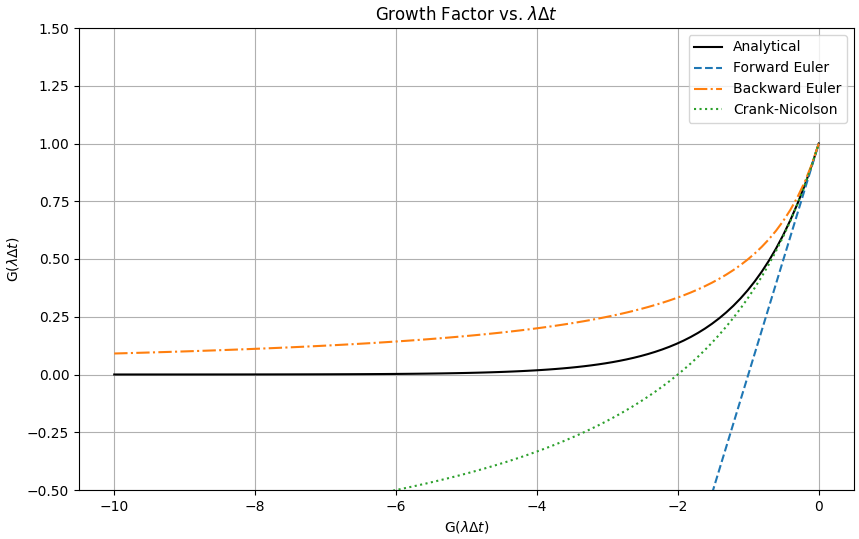


Figure 7: Growth factors for analytic solution, forward Euler, backward Euler, and Crank-Nicholson methods.

```
p = symamd(HL);
L = chol(HL(p,p),'lower');
HRp = HR(p,p);
for istep=1:nstep; time=istep*dt;
    rhs = HRp*u(p);
    u = L'\( L \ (rhs));
    u(p) = u;
end;
```

This creates the table seen in Table 8, which, as follows from the logic presented in the problem description, is the exact same. The runtime was estimated to be 5.2506 seconds. Compared to an initial runtime of 44.548, this represents about an 8.5x performance improvement.

N	Δt	$n_{\textbf{time steps}}$	**time**	$L2$ **Error Ratio**	**Ratio**
150	1.50E-01	8	1.20	2.5530E-01	3.9170E+00
150	8.00E-02	15	1.20	7.5673E-02	3.3737E+00
150	4.00E-02	30	1.20	1.9138E-02	3.9540E+00
150	2.00E-02	60	1.20	4.7981E-03	3.9887E+00
150	1.00E-02	120	1.20	1.2004E-03	3.9972E+00
150	5.00E-03	240	1.20	3.0015E-04	3.9993E+00
150	2.50E-03	480	1.20	7.5040E-05	3.9998E+00
150	1.25E-03	960	1.20	1.8760E-05	4.0000E+00

Table 8: Part 1 Results

The number of spatial points was then varied and compared with the original code. This

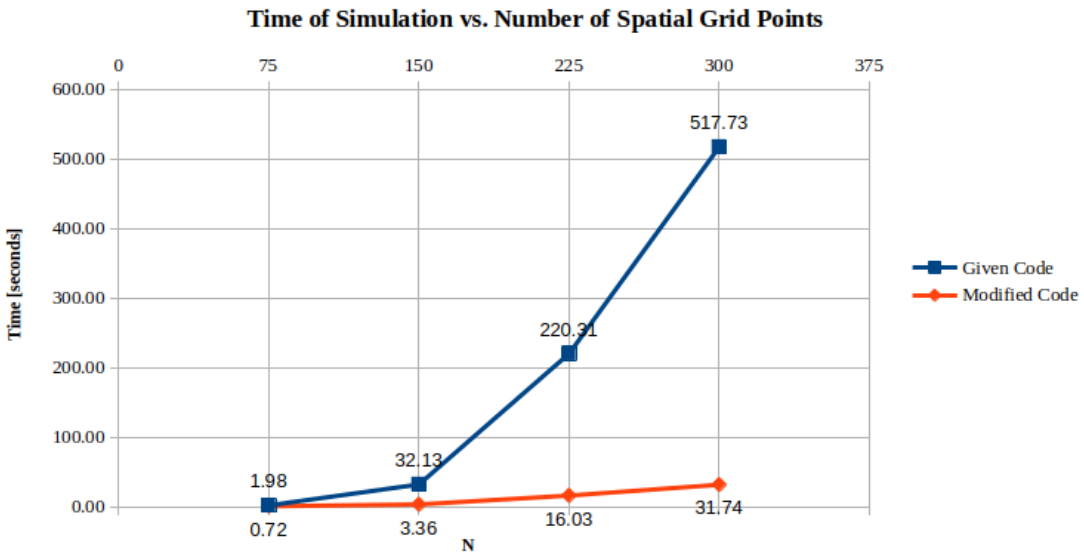


Figure 8: Comparison of improved code in problem 3(i) with given MatLab code. Number of spatial points, N, was varied from 75 to 300 in increments of 75. Runtime is report in seconds.

experiment yielded the chart seen in Figure 8. As can be seen, the computation time for the given code grows much more rapidly than that of the modified code.

(ii) Implementing the ADI algorithm saw an improvement in the runtime. We got a runtime of 0.4397 seconds as opposed the 5.2506 seconds from part (i). The matlab code provided fixes N, but varies Δt. If we vary N instead, we get the runtime scaling that we expect with ADI of $O(n_x n_y) = O(N^2)$. For the error scaling, both spatial and temporal errors contribute. Being careful to avoid saturation from one or the other, we can achieve the quadratic error scaling we expect.

N	Δt	$n_{\textbf{time steps}}$	**time**	$L2$ **Error Ratio**	**Ratio**
150	1.50E-01	8	1.20	1.9057E-01	5.2473E+00
150	8.00E-02	15	1.20	5.5571E-02	3.4294E+00
150	4.00E-02	30	1.20	1.3990E-02	3.9721E+00
150	2.00E-02	60	1.20	3.5036E-03	3.9931E+00
150	1.00E-02	120	1.20	8.7628E-04	3.9983E+00
150	5.00E-03	240	1.20	2.1909E-04	3.9996E+00
150	2.50E-03	480	1.20	5.4775E-05	3.9999E+00
150	1.25E-03	960	1.20	1.3694E-05	4.0000E+00

Table 9: Part 2 Results