# CS555 Spring 2026 Homework Set 1.

Due Thursday, February 19, 8 PM.

- **Work in teams of 2. Each student on a team should submit the same solution under their name, with partner listed. This aids in getting all the grades entered.**
- *Please reach out to me if you have questions, even at the 11th hour. I'm happy to help clarify / debug, as needed.*

**1.** Consider the unsteady heat equation in 1D,

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2}, \qquad u(x,t=0) = u^0(t), \qquad u(0,t) = u(1,t) = 0. \tag{1}$$

For $\nu = 0.2$ and $t \in [0,1]$, solve this PDE using 2nd-order finite-difference in space and Crank-Nicolson (CN) in time and compare this to the results for Euler Backward (EB), which is coded up in `eb.m` with the driver code, `deb.m`. `eb.m` uses variable spacing for the spatial operator and you can use the same code for CN and BDF implementations. Simply modify this code pair to support CN. They should run either in matlab or octave.

For your CN implementation, plot the solution as a function of $x$ for several time points on the interval $t \in [0,1]$ for the two intial conditions prescribed below.

**1a.** [**15 points**] Using the initial condition $u^0 = \sin \pi x$, demonstrate that your CN implementation realizes 2nd-order accuracy in time by looking at the ratio of successive errors as $\Delta t$ is successively halved. The code will produce a table similar to that for EB. Submit both tables and discuss your observations.

**1b.** [**10 points**] Now consider the initial condition $u^0 \equiv \frac{\pi}{4}$ and run both the EB and CN cases. Submit both tables, along with the figures of $u(x,t)$ and discuss your observations.

**1c.** [**10 points**] Repeat this exercise using BDF2. (Bootstrap the process by using EB for the first timestep.) What do you observe in the plots and table? Submit the table and plots for both choices of IC.

**Extra Credit** [**10 points**] For the square wave, we assume that the solution at time $t = 1$ is approximately the same as that with the $k = 1$ eigenmode as the IC. Estimate the magnitude of the next leading-order term in the actual decaying square wave at time $t = 1$. How does it compare to the error in the finite difference results of the preceding questions?

**2.** [**10 points**]

**(a)** Assuming $\lambda \in \mathbb{R}$, what is the asymptotic value of the growth factor $G(\lambda\Delta t)$ for Crank-Nicolson as $\lambda\Delta t \longrightarrow -\infty$? (Show a derivation rather than just writing the answer.)

**(b)** On a single graph, plot the growth factors $G(\lambda\Delta t)$ for $-10 \leq \lambda\Delta t \leq 0$ for the following timesteppers: analytical, Euler forward, Euler backward, and Crank-Nicolson.

**2.** Consider the unsteady heat equation in two spatial dimensions,

$$\frac{\partial u}{\partial t} = \nu \nabla^2 u, \qquad u(\mathbf{x}, t=0) = u^0(t), \qquad u|_{\partial\Omega} = 0, \tag{2}$$

with $\Omega = [0, L_x] \times [0, L_y]$ and $L_x = L_y = 1$. We will take the initial condition to be an eigenmode of the differential equation,

$$u^0(x,y) = \sin(k\pi/L_x)\sin(l\pi/L_y), \tag{3}$$

for which the associated eigenvalue is

$$\lambda_{kl} = -\nu\left[\left(\frac{k\pi}{L_x}\right)^2 + \left(\frac{l\pi}{L_y}\right)^2\right], \tag{4}$$

with corresponding analytical solution,

$$u(x,y,t) = e^{\lambda_{kl}} u^0(x,y) \tag{5}$$

The code `heat2d.m` implements CN timestepping for this problem and presents the error for a battery of tests as well as the *runtime* for the full battery. In higher space dimensions (particularly 3D), runtime costs often limit the resolution in space ($N$) and time ($\Delta t$). These costs consequently constrain the overall accuracy, so it is important to reduce the costs to the extent possible.

You are to extend the code in three ways to improve the overall performance:

*i.* Permute $H_L$ from its banded structure to a permutation generated by the matlab/octave `symamd` routine. This change should give the same solution and, hence, error as the unpermuted case, but should run faster.

The idea is as follows. For a given SPD matrix $H$, compute a permutation vector `p` with the statement `p=symamd(H)`, which generates an approximate solution to the symmetric minimal degree algorithm. We then compute the Cholesky factorization, $LL^T = H_p$, where $H_p := H(p,p)$ is the permuted matrix. The idea behind `symamd` is to permute the matrix such that the Cholesky factors for $H_p$ are sparser (i.e., have fewer nonzeros) than the Cholesky factors of $H$. If $\underline{u}$ is the solution to $H\underline{u} = \underline{f}$, then the original solution would be

```
u = H \ f;                                  (6)
```

However, if we wanted to solve this system many times (e.g., $H\underline{u}^{(l)} = \underline{u}^{(l-1)}$, $l = 1, 2, \dots$), it will generally be more efficient to first factor $H$ into lower and upper triangular matrices $L$ and $L^T$, such that $LL^T = H$, which is always possible if $H$ is symmetric positive definite (SPD). Such an algorithm would read, in part,

```
L = chol(H) % expensive part                (7)
for l=1:nsteps;                             (8)
    u = L' \ ( L \ u );                     (9)
end;                                        (10)
```

The improved algorithm would read

```
p = symamd(H)                               (11)
L = chol(H(p,p)) % expensive part           (12)
for l=1:nsteps;                             (13)
    rhs = u(p);                             (14)
    u = L' \ ( L \ rhs );                   (15)
    u(p) = u ;                              (16)
end;                                        (17)
```

Note that if no other operations are applied to `u` in the inner loop, we do not need to permute it back to its original ordering (16) on each loop iteration, nor do we need to permute the RHS, (14). Consequently, a faster loop would read:

```
p = symamd(H)                           (18)
L = chol(H(p,p)) % expensive part       (19)
u = u(p); % permute                     (20)
for l=1:nsteps;                         (21)
    u = L' \ ( L \ u );                 (22)
end;                                    (23)
u(p) = u; % unpermute                   (24)
```

In the CN implementation, however, we *do* need to multiply by another operator, so the permutation within each loop iteration is possibly warranted.

**2a. [10 points]** Modify the heat2d code to support the permuted factorization and compare the timings to the original code. How do the timings change as you increase $N$?

*ii.* Here, we consider *alternating direction implicit* (ADI) methods, which are ideally suited to this class of problems but are also excellent candidates for fast iterative methods for the Poisson equation on structured grids.

To introduce the idea, consider the Helmholtz matrices $H_L$ and $H_R$ associated with the 2D CN update step,

$$\begin{aligned} H_L &= I_y \otimes I_x + \left(\frac{\nu \Delta t}{2}\right)(I_y \otimes A_x + A_y \otimes I_x) \qquad (25)\\ H_R &= I_y \otimes I_x - \left(\frac{\nu \Delta t}{2}\right)(I_y \otimes A_x + A_y \otimes I_x). \end{aligned}$$

Defining the following matrices,

$$\begin{aligned} I &:= I_y \otimes I_x \qquad (26)\\ H_{lx} &:= I + (\nu\Delta t/2)(I_y \otimes A_x)\\ H_{ly} &:= I + (\nu\Delta t/2)(A_y \otimes I_x)\\ H_{rx} &:= I - (\nu\Delta t/2)(I_y \otimes A_x)\\ H_{ry} &:= I - (\nu\Delta t/2)(A_y \otimes I_x), \end{aligned}$$

we observe that

$$\begin{aligned} H_L &= H_{ly}\,H_{lx} - (\nu\Delta t/2)^2(A_y \otimes A_x) \qquad (27)\\ H_R &= H_{ry}\,H_{rx} - (\nu\Delta t/2)^2(A_y \otimes A_x). \end{aligned}$$

Consequently, we can recast the CN update step,

$$H_L \underline{u}^l = H_R \underline{u}^{l-1} + \frac{\Delta t}{2}(\underline{q}^l + \underline{q}^{l-1}), \qquad (28)$$

as

$$\begin{aligned} H_{ly}\,H_{lx}\underline{u}^l &= H_{ry}\,H_{rx}\underline{u}^{l-1} + \frac{\Delta t}{2}(\underline{q}^l + \underline{q}^{l-1}) + (\nu\Delta t/2)^2(A_y \otimes A_x)(\underline{u}^{l-1} - \underline{u}^l) \qquad (29)\\ &= H_{ry}\,H_{rx}\underline{u}^{l-1} + \frac{\Delta t}{2}(\underline{q}^l + \underline{q}^{l-1}) + O(\Delta t^3). \qquad (30) \end{aligned}$$

To implement the ADI algorithm, we drop the $O(\Delta t^3)$ term in (30), which is of the same order as the LTE for CN. Why is this advantageous? The key point is that the $H_l$ matrices are now of the form $H_{lx} = I_y \otimes T_x$ and $H_{ly} = T_y \otimes I_x$, where the $T$ matrices are *tridiagonal.* Consequently, if we view the (lexicographically ordered) vector $\underline{b}$ as a 2D array, $B = \{b_{ij}\}$, we can compute the solution to $H_{lx}\underline{u} = \underline{b}$ as $U = T_x^{-1}B$, which involves $n_y$ tridiagonal solves of length $n_x$ (roughly $8n$ operations total, where $n = n_x n_y$). Similarly, we can solve $H_{ly}\underline{u} = \underline{b}$ as $U = (T_y^{-1}B^T)^T$, for the same cost.

**2b. [10 points]** Update the `heat2d.m` code to support ADI and rerun the comparison test to the CN baseline and to the results of **2a.** Comment on what you observe, particularly regarding accuracy, timing, and scalability as $N$ is increased.

*iii. Extra extra credit.*

**2c. [10 points]** Extend your ADI scheme to work for BDF3. (Use two steps of ADI-CN to bootstrap the first two steps.) The principal goal here is two-fold: (1) to have an L-stable scheme and (2) to have 3rd-order accuracy in time. Note that the LTE is now $O(\Delta t^4)$, which implies you need to do something extra (but *simple*) to account for the ADI splitting error, which was $O(\Delta t^3)$ in the preceding case.