1 Introduction

1.1 Problem 1

The partial differential equation (PDE) that governs one-dimensional heat transfer, with constants and boundary conditions given for a 2 cm-thick steel pipe, is

$$\frac{\partial u}{\partial t} = \frac{k}{c\rho}\frac{\partial^2 u}{\partial x^2}\,, \qquad \begin{aligned} &u(0,t) = 0\,, && k = 0.13 \quad \text{cal / sec cm } ^\circ\text{C}\,, \\ &u(2,t) = 0\,, && c = 0.11 \quad \text{cal / g } ^\circ\text{C}\,, \\ &u(x,0) = 100sin(\pi x/2)\,, && \rho = 7.8 \quad \text{g / cm}^3\,, \end{aligned} \tag{1}$$

where $u(x,t)$ is the temperature in $^\circ$C. Numerically integrate this PDE using the explicit forward-time, central-space (FTCS, or Euler) method with $\Delta x = 0.1$ cm. This corresponds to $N = 21$ spatial grid points. Choose Δt such that the diffusion criterion is (a) $d = \frac{1}{2}$ and (b) $d = 1$. Compare your solution for (a) at $t = \{1, 2, 4, 8\}$ sec with the analytical solution

$$u = 100\exp(-0.3738t)\sin(\pi x/2)\,. \tag{2}$$

Note that for (b), the diffusion criterion is higher than the one allowed by the von Neumann method. For this case, plot your solutions at approximately $t = \{\frac{1}{10}, \frac{1}{2}, 1, 2\}$ sec, and comment on stability.

1.2 Problem 2

Solve the system from Problem 1 using the implicit Crank-Nicolson method with forward time differencing. The Thomas algorithm can be utilized for this purpose. Try values for the diffusion criterion of $d = \{\frac{1}{2}, 1, 10\}$, and comment on the solution's stability.

1.3 Problem 3

Solve the system from Problem 1, but with an adiabatic boundary condition at the upper boundary,

$$u(0,t) = 0\,, \qquad \frac{\partial u}{\partial x}(2,t) = 0\,. \tag{3}$$

2 Methodology

2.1 Problem 1

Discretizing (1) using the forward-time, central-space (FTCS) method with explicit time advancement yields

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}\,, \tag{4}$$

where the subscript denotes the spatial grid point, and the superscript indexes the time step. Further note it is convenient to define the coefficient of thermal diffusivity as $\alpha \equiv k/c\rho = 0.1515$ m/s^2. Solving for the unknown quantity, we obtain

$$u_i^{n+1} = du_{i-1}^n + (1-2d)u_i^n + du_{i+1}^n\,, \tag{5}$$

where the diffusion number is defined as

$$d = \frac{\alpha \Delta t}{\Delta x^2} \qquad \Rightarrow \qquad \Delta t = \frac{d \Delta x^2}{\alpha} \,. \tag{6}$$

Using (5), it is trivial to step forward in time with Dirichlet boundary conditions on u, by simply solving for each u_i^{n+1} based on the solution for u at the previous time step. We use (6) to obtain the requested diffusion numbers by setting (a) $\Delta t = 0.033$ and (b) $\Delta t = 0.066$.

2.2 Problem 2

The implicit Crank-Nicolson method with forward time-differencing is most-commonly used to solve the diffusion equation, which can be written as

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{\alpha}{2\Delta x^2} \left(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1} + u_{i+1}^n - 2u_i^n + u_{i-1}^n \right) . \tag{7}$$

It is a simple matter to re-write this as a linear equation relating the value of u_i at the next time step to its spatial neighbors at the current and next time steps:

$$u_i^{n+1} = \beta \left(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1} + u_{i+1}^n - 2u_i^n + u_{i-1}^n \right) + u_i^n \tag{8}$$

$$\underbrace{(-\beta)}_{b_i} u_{i-1}^{n+1} + \underbrace{(1+2\beta)}_{a_i} u_i^{n+1} + \underbrace{(-\beta)}_{c_i} u_{i+1}^{n+1} = \underbrace{\beta \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right) + u_i^n}_{d_i} , \tag{9}$$

where

$$\beta = \frac{\alpha \Delta t}{2\Delta x^2} \tag{10}$$

is known, along with all point-wise values of u^n.

We use the formulation (9) to construct a matrix system of equations using Dirichlet boundary conditions with the same technique that was explained in Homework 4 with regard to central differences. The reader is directed to Homework 4 for an explanation of the Thomas algorithm subsequently used to solve this matrix system.

2.3 Problem 3

To incorporate the adiabatic boundary condition (3), we use one-sided differences to write it as

$$\left. \frac{\partial u}{\partial x} \right|_N^{n+1} \approx \frac{u_N^{n+1} - u_{N-1}^{n+1}}{\Delta x} = 0 \qquad \Rightarrow \qquad u_N^{n+1} = u_{N-1}^{n+1} \,, \tag{11}$$

and make the necessary modifications to the diagonal and the RHS vector in our Matlab code based on evaluating (9) with (11) substituted in for the terms associated with $i = N-1$. Recall that the solution vector for our matrix system is $[u_2^{n+1}, \ldots, u_{N-1}^{n+1}]^T$. In other words, the Neumann boundary condition modifies the last grid point *within* the domain as

$$\underbrace{(-\beta)}_{b_{N-1}} u_{N-2}^{n+1} + \underbrace{(1+\beta)}_{a_{N-1}} u_{N-1}^{n+1} = \underbrace{\beta \left(u_{N-2}^n - 2u_{N-1}^n + u_N^n \right) + u_{N-1}^n}_{d_{N-1}} , \tag{12}$$

noting that c_{N-1} simply does not exist in our matrix system, as it represents the dependence of u_{N-1}^{n+1} on u_N^{n+1}, which is not included in our solution vector. After the matrix has been solved for the internal points, we then assign $u_N^{n+1} = u_{N-1}^{n+1}$.

This approach can be extended to arbitrary Neumann boundary conditions by modifying d_N to account for a non-zero RHS of (11), if desired. In this case, the LHS of (12) remains the same, and the RHS becomes $d_N \rightarrow d_N + \beta A \Delta x$, where A is the Neumann boundary condition's value. In this case, we then update the final grid point to reflect the prescribed Neumann boundary condition using $u_N^{n+1} = u_{N-1}^{n+1} + A\Delta x$.

It should be noted that using these particular one-sided spatial differences, as we have, reduces the accuracy of our solution to only $\mathcal{O}(\Delta t^2, \Delta x)$.

Since only the RHS, d_i, changes as we progress forward in time, LU decomposition is well-suited for this problem. The **L** and **U** matrices need only be calculated once, and then passed to our `LU_Solve` function along with the time step-dependent RHS. As with the Thomas algorithm, the reader is directed to Homework 4 for an explanation of LU decomposition and solution.

3 Results

3.1 Problem 1

Results for the FTCS method are presented in Figure 1. Relative error at $t = 0$ sec is identically zero, since the initial guess is the analytical solution. This holds true for Problem 2 as well.

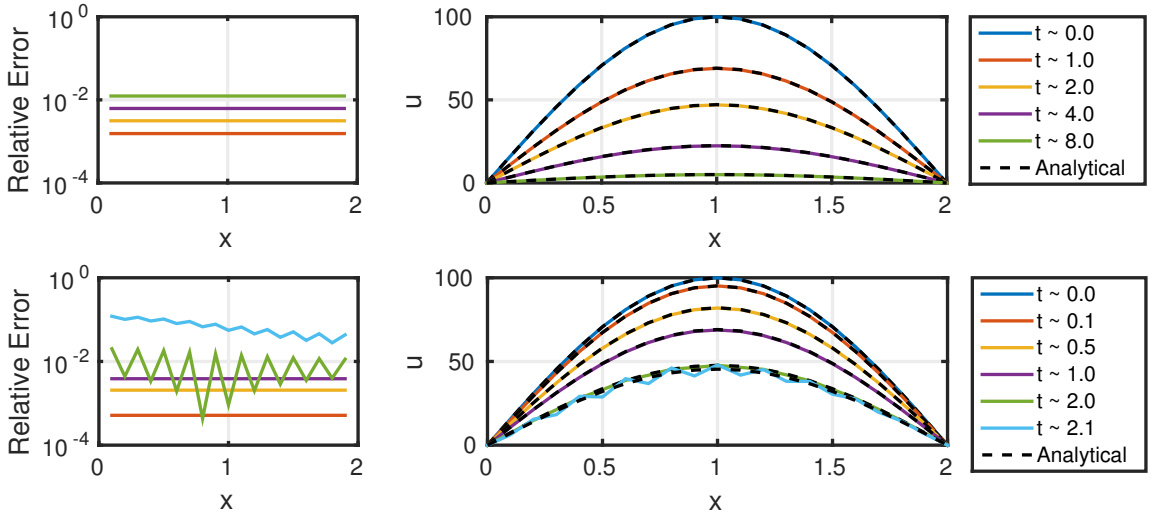

Figure 1: FTCS solutions to (1) compared to analytical solutions (2), and the point-wise relative error over time. Upper plots are for $d = \frac{1}{2}$ ($\Delta t = 0.033$), and lower plots correspond to $d = 1$ ($\Delta t = 0.066$).

3.2 Problem 2

Results for the Crank-Nicolson method with Dirichlet boundary conditions are presented in Figure 2.

3.3 Problem 3

Results for the Crank-Nicolson method with one Dirichlet and one Neumann boundary condition are presented in Figure 3.

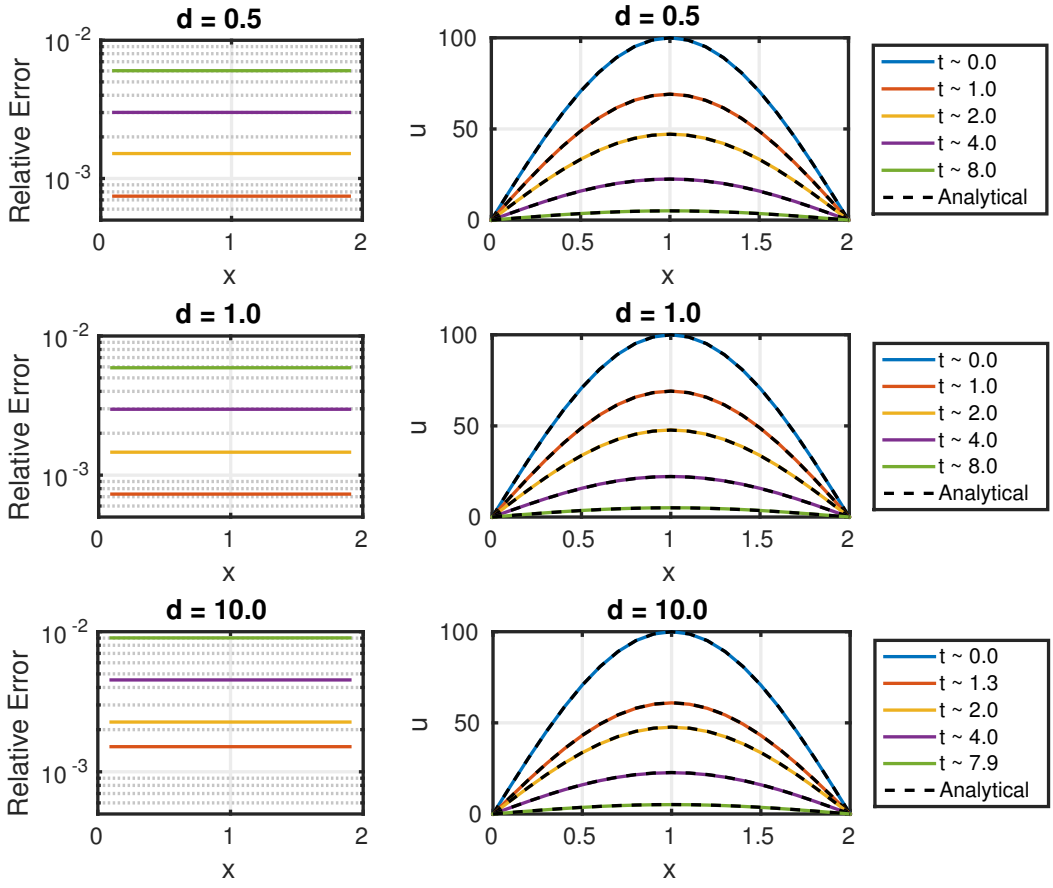

Figure 2: Crank-Nicolson solutions to (1) compared to analytical solutions (2), and the point-wise relative error over time. Diffusion numbers d are annotated. For $d = 10$, $\Delta t = 0.6601$.

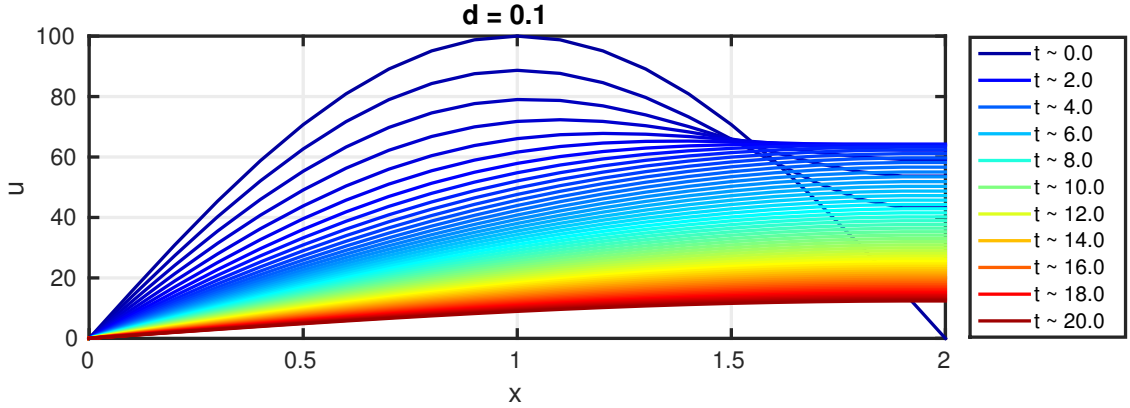

Figure 3: Crank-Nicolson solutions to the thermal diffusion proble with Dirichlet and Neumann boundary conditions of $u(0,t) = 0$ and $u_{,x}(2,t) = 0$. Diffusion number d is chosen to be small so we can show solution progression with high temporal resolution.

4 Discussion

4.1 Problem 1

As can be seen in Figure 1, FTCS results for a diffusion number of $d = \frac{1}{2}$ have fairly low relative error at all time-steps considered, and no numerical instability is present. For a condition number of $d = 1$, which

should be unstable according to the von Neumann method, we see high-frequency instabilities manifest at $t \sim 2.0$. Relative error grows rapidly, as can be seen at $t \sim 2.1$. Later times are not shown, because the solution quickly becomes non-physical.

4.2 Problem 2

As shown in Figure 2, the Crank-Nicolson method produces results with point-wise errors slightly better than the FTCS method, at diffusion numbers at least 10× greater than the maximum stable diffusion number of the FTCS method. We conclude that the Crank-Nicolson method is very stable. Though we do need to solve a matrix system at each time step, Δt can be much higher than the FTCS method, and thus the Crank-Nicolson method has the potential for much higher efficiency.

4.3 Problem 3

Solutions for the adiabatic far boundary condition appear stable and in agreement with physical intuition, as can be seen in Figure 3. From $t = 0$ sec to about $t \sim 2.0$ sec, the right side of the domain diffuses temperature rapidly to satisfy the Neumann boundary condition. After $t \sim 2.0$ sec, temperature is distributed fairly evenly across the domain. At this point, progress in the solution is dictated mostly by the Dirichlet boundary condition on the left side of the domain, which slowly removes heat from the domain.

It is anecdotally observed that LU decomposition substantially reduces computation time relative to the Thomas algorithm when solving successive iterations.

5 References

No external references were used other than the course notes for this assignment.

Appendix: MATLAB Code

The following code listings generate all figures presented in this homework assignment. Functions such as `Thomas.m`, `LU_Decompose.m`, and `LU_Solve.m` are omitted, since they were previously presented in Homework 4.

Listing 1: `Problem_1.m`

```
function [] = Problem_1()

    %%%%%%
    % Solves the heat conduction equation (u_,t = a u_,xx) using the FTCS method.
    %
    % Ryan Skinner, October 2015
    %%%

    Set_Default_Plot_Properties();

    %%%
    % Define variables specific to the boundary-value problem.
    %%%

    % Material properties.
    alpha = 0.1515;

    % Solution domain: the closed interval [0,2].
    N = 21;
    x = linspace(0,2,N);
    dx = x(2) - x(1);

    % Boundary conditions (0 = initial, f = final).
```

```
    BC.u0 = 0;
    BC.uf = 0;

    % Initial conditions.
    ux0 = 100 * sin(pi * x / 2);

    % Condition numbers, and maximum time to reach in each case.
    d = [0.5, 1.0];
    tmax = [8,4];

    % Time steps.
    dt = dx^2 * d / alpha;

    % Maximum time steps to reach.
    tsmax = ceil(tmax ./ dt);

    % Store solutions indexed by time step, and spatial index.
    u = {nan(tsmax(1),length(x)), nan(tsmax(2),length(x))};

    % Initialize solution and set boundary conditions.
    u{1}(1,:) = ux0;
    u{2}(1,:) = ux0;
    u{1}(:,1) = BC.u0;
    u{1}(:,N) = BC.uf;
    u{2}(:,1) = BC.u0;
    u{2}(:,N) = BC.uf;

    %%%
    % Solve problem numerically.
    %%%

    % Loop over condition numbers.
    for di = 1:length(d)
        fprintf('Working with d = %3.1f, dt = %.4f\n', d(di), dt(di));
        % Iterate over time steps.
        for n = 1:tsmax(di)-1
            % Loop over spatial points.
            for j = 2:N-1
                u{di}(n+1,j) = u{di}(n,j-1) * d(di) ...
                             + u{di}(n,j  ) * (1-2*d(di)) ...
                             + u{di}(n,j+1) * d(di);
            end
        end
    end

    %%%
    % Process results: Part (a).
    %%%

    ts_plot = 1 + round([0,1,2,4,8] / dt(1));
    t_plot = (ts_plot-1) * dt(1);

    % Solution.

    hl = cell(length(ts_plot)+1,1);
    dn = cell(length(ts_plot)+1,1);

    figure();
    hold on;
    for i = 1:length(ts_plot)
        hl{i} = plot(x, u{1}(ts_plot(i),:));
        dn{i} = sprintf('t ~ %.1f',t_plot(i));
        if i ~= length(ts_plot)
            plot(x, analytical(t_plot(i),x), '--k');
        else
            hl{i+1} = plot(x, analytical(t_plot(i),x), '--k');
            dn{i+1} = 'Analytical';
        end
    end
```

```
    xlabel('x');
    ylabel('u');
    hleg = legend([hl{:}], dn);
    set(hleg, 'Location', 'eastoutside');

    % Relative error.

    figure();
    hax = axes();
    hold on;
    for i = 1:length(ts_plot)
        relerr = abs(analytical(t_plot(i),x(2:end-1)) - u{1}(ts_plot(i),2:end-1)) ./ ...
                    analytical(t_plot(i),x(2:end-1));
        plot(x(2:end-1), relerr, ...
            'DisplayName', sprintf('t ~ %.1f',t_plot(i)));
    end
    set(hax,'YScale','log');
    xlabel('x');
    ylabel('Relative Error');
    %hleg = legend('show');
    %set(hleg, 'Location', 'eastoutside');

    %%%
    % Process results: Part (b).
    %%%

    ts_plot = 1 + round([0, 0.1, 0.5, 1, 2, 2.1] / dt(2));
    t_plot = (ts_plot-1) * dt(2);

    % Solution.

    hl = cell(length(ts_plot)+1,1);
    dn = cell(length(ts_plot)+1,1);

    figure();
    hold on;
    for i = 1:length(ts_plot)
        hl{i} = plot(x, u{2}(ts_plot(i),:));
        dn{i} = sprintf('t ~ %.1f',t_plot(i));
        if i ~= length(ts_plot)
            plot(x, analytical(t_plot(i),x), '--k');
        else
            hl{i+1} = plot(x, analytical(t_plot(i),x), '--k');
            dn{i+1} = 'Analytical';
        end
    end
    xlabel('x');
    ylabel('u');
    hleg = legend([hl{:}], dn);
    set(hleg, 'Location', 'eastoutside');

    % Relative error.

    figure();
    hax = axes();
    hold on;
    for i = 1:length(ts_plot)
        relerr = abs(analytical(t_plot(i),x(2:end-1)) - u{2}(ts_plot(i),2:end-1)) ./ ...
                    analytical(t_plot(i),x(2:end-1));
        plot(x(2:end-1), relerr, ...
            'DisplayName', sprintf('t ~ %.1f',t_plot(i)));
    end
    set(hax,'YScale','log');
    xlabel('x');
    ylabel('Relative Error');
    %hleg = legend('show');
    %set(hleg, 'Location', 'eastoutside');

end
```

```
162
163 function [ u_exact ] = analytical( t, x )
164     % Compute analytical solution.
165     u_exact = 100 * exp(-0.3738*t) * sin(pi * x / 2);
166 end
```

Listing 2: `Problem_2.m`

```
1  function [] = Problem_2()
2
3      %%%%%%
4      % Solves the heat conduction equation (u_,t = a u_,xx) using Crank-Nicolson and
5      % Dirichlet boundary conditions at both ends of the domain.
6      %
7      % Ryan Skinner, October 2015
8      %%%
9
10     Set_Default_Plot_Properties();
11
12     %%%
13     % Define variables specific to the boundary-value problem.
14     %%%
15
16     % Material properties.
17     alpha = 0.1515;
18
19     % Solution domain: the closed interval [0,2].
20     N = 21;
21     x = linspace(0,2,N);
22     dx = x(2) - x(1);
23
24     % Boundary conditions (0 = initial, f = final).
25     BC.u0 = 0;
26     BC.uf = 0;
27
28     % Initial conditions.
29     ux0 = 100 * sin(pi * x / 2);
30
31     % Condition numbers, and maximum time to reach in each case.
32     d = [0.5, 1.0, 10.0];
33     tmax = [8,8,8];
34
35     % Time steps.
36     dt = dx^2 * d / alpha;
37
38     % Maximum time steps to reach.
39     tsmax = ceil(tmax ./ dt);
40
41     % Store solutions indexed by time step, and spatial index.
42     u = {nan(tsmax(1),length(x)), nan(tsmax(2),length(x)), nan(tsmax(3),length(x))};
43
44     % Initialize solution and set boundary conditions.
45     u{1}(1,:) = ux0;
46     u{2}(1,:) = ux0;
47     u{3}(1,:) = ux0;
48     u{1}(:,1) = BC.u0;
49     u{1}(:,N) = BC.uf;
50     u{2}(:,1) = BC.u0;
51     u{2}(:,N) = BC.uf;
52     u{3}(:,1) = BC.u0;
53     u{3}(:,N) = BC.uf;
54
55     %%%
56     % Solve problem numerically.
57     %%%
58
59     % Loop over condition numbers.
60     for di = 1:length(d)
61         fprintf('Working with d = %4.1f, dt = %.4f\n', d(di), dt(di));
```

```
        beta = alpha * dt(di) / (2 * dx^2);
        % Iterate over time steps.
        for n = 1:tsmax(di)-1
            [diag, sub, sup, rhs] = Assemble_u(beta, u{di}(n,:), BC);
            [sol] = Thomas(diag, sub, sup, rhs);
            u{di}(n+1,:) = [BC.u0; sol; BC.uf];
        end
    end

    %%%
    % Process results.
    %%%

    for di = 1:length(d)

        ts_plot = 1 + round([0,1,2,4,8] / dt(di));
        t_plot = (ts_plot-1) * dt(di);

        % Solution.

        hl = cell(length(ts_plot)+1,1);
        name = cell(length(ts_plot)+1,1);

        figure();
        hold on;
        for i = 1:length(ts_plot)
            hl{i} = plot(x, u{di}(ts_plot(i),:));
            name{i} = sprintf('t ~ %.1f',t_plot(i));
            if i ~= length(ts_plot)
                plot(x, analytical(t_plot(i),x), '--k');
            else
                hl{i+1} = plot(x, analytical(t_plot(i),x), '--k');
                name{i+1} = 'Analytical';
            end
        end
        xlabel('x');
        ylabel('u');
        title(sprintf('d = %.1f',d(di)));
        hleg = legend([hl{:}], name);
        set(hleg, 'Location', 'eastoutside');

        % Relative error.

        figure();
        hax = axes();
        hold on;
        for i = 1:length(ts_plot)
            relerr = abs(analytical(t_plot(i),x(2:end-1)) - u{di}(ts_plot(i),2:end-1)) ./ ...
                         analytical(t_plot(i),x(2:end-1));
            plot(x(2:end-1), relerr, ...
                'DisplayName', sprintf('t ~ %.1f',t_plot(i)));
        end
        set(hax,'YScale','log');
        xlabel('x');
        ylabel('Relative Error');
        title(sprintf('d = %.1f',d(di)));
        ylim([5*10^-4,10^-2]);

    end

end

function [ u_exact ] = analytical( t, x )
    % Compute analytical solution.
    u_exact = 100 * exp(-0.3738*t) * sin(pi * x / 2);
end
```

Listing 3: `Assemble_u.m`

```
function [diag, sub, sup, rhs] = Assemble_u( beta, u_prev, BC )

    %%%%%%
    % Assembles the LHS matrix and the RHS vector for the g-system.
    %   diag -- diagonal
    %    sub -- sub-diagonal
    %    sup -- super-diagonal
    %    rhs -- right-hand side vector
    %
    % Ryan Skinner, October 2015
    %%%

    N = length(u_prev);

    diag_range = 2:N-1;
     sub_range = 3:N-1;
     sup_range = 2:N-2;

    diag = (1 + 2 * beta) * ones(length(diag_range),1);
     sub = (       -beta) * ones(length(sub_range),1);
     sup = (       -beta) * ones(length(sup_range),1);
     rhs = u_prev(diag_range) + ...
           beta * (u_prev(diag_range-1) - 2 * u_prev(diag_range) + u_prev(diag_range+1));

    % Account for boundary conditions.
    rhs(1)   = rhs(1)   - (-beta) * BC.u0;
    rhs(end) = rhs(end) - (-beta) * BC.uf;

end
```

Listing 4: `Problem_3.m`

```
function [] = Problem_3()

    %%%%%%
    % Solves the heat conduction equation (u_,t = a u_,xx) using Crank-Nicolson and an
    % adiabatic far boundary condition (Neumann).
    %
    % Ryan Skinner, October 2015
    %%%

    Set_Default_Plot_Properties();

    %%%
    % Define variables specific to the boundary-value problem.
    %%%

    % Material properties.
    alpha = 0.1515;

    % Solution domain: the closed interval [0,2].
    N = 21;
    x = linspace(0,2,N);
    dx = x(2) - x(1);

    % Boundary conditions (0 = initial, f = final).
    BC.u0 = 0;
    BC.upf = 0; % Where up = u prime = du/dx.

    % Initial conditions.
    ux0 = 100 * sin(pi * x / 2);

    % Condition numbers, and maximum time to reach in each case.
    d = 0.1;
    tmax = 20;

    % Time steps.
    dt = dx^2 * d / alpha;
```

```

    % Maximum time steps to reach.
    tsmax = ceil(tmax ./ dt);

    % Store solutions indexed by time step, and spatial index.
    u = nan(tsmax(1),length(x));

    % Initialize solution and set boundary conditions.
    u(1,:) = ux0;

    %%%
    % Solve problem numerically.
    %%%

    fprintf('Working with d = %4.1f, dt = %.4f\n', d, dt);

    beta = alpha * dt / (2 * dx^2);

    % Iterate over time steps, taking advantage of LU-decomposition efficiency.
    for n = 1:tsmax-1
        [diag, sub, sup, rhs] = Assemble_u_Prob3(beta, u(n,:), BC, dx);
        if n == 1
            [l_vec, u_vec] = LU_Decompose(diag, sub, sup);
        end
        [sol] = LU_Solve(sub, l_vec, u_vec, rhs);
        u(n+1,:) = [BC.u0; sol; sol(end) + dx * BC.upf];
    end

    %%%
    % Process results.
    %%%

    n_plot = 61;
    cmap = jet(n_plot);

    ts_plot = round(linspace(0,tmax,n_plot) / dt);
    ts_plot(1) = ts_plot(1) + 1;
    t_plot = (ts_plot) * dt;
    t_plot(1) = t_plot(1) - dt;

    % Solution.

    hl = cell(length(ts_plot),1);
    name = cell(length(ts_plot),1);

    figure();
    hold on;
    for i = 1:length(ts_plot)
        hl{i} = plot(x, u(ts_plot(i),:),'Color',cmap(i,:));
        name{i} = sprintf('t ~ %.1f',t_plot(i));
    end
    xlabel('x');
    ylabel('u');
    title(sprintf('d = %.1f',d));
    legend_indices = ceil(linspace(1,n_plot,11));
    hleg = legend([hl{legend_indices}], name{legend_indices});
    set(hleg, 'Location', 'eastoutside');

end
```

Listing 5: `Assemble_u_Prob3.m`

```
function [diag, sub, sup, rhs] = Assemble_u_Prob3( beta, u_prev, BC, dx )

    %%%%%%
    % Assembles the LHS matrix and the RHS vector for the g-system.
    %   diag -- diagonal
    %    sub -- sub-diagonal
    %    sup -- super-diagonal
```

```
8       %    rhs -- right-hand side vector
9       %
10      % Ryan Skinner, October 2015
11      %%%
12
13      N = length(u_prev);
14
15      diag_range = 2:N-1;
16       sub_range = 3:N-1;
17       sup_range = 2:N-2;
18
19      diag = (1 + 2 * beta) * ones(length(diag_range),1);
20       sub = (       -beta) * ones(length(sub_range),1);
21       sup = (       -beta) * ones(length(sup_range),1);
22       rhs = u_prev(diag_range) + ...
23             beta * (u_prev(diag_range-1) - 2 * u_prev(diag_range) + u_prev(diag_range+1));
24
25      % Account for boundary conditions: Dirichlet.
26      rhs(1)   = rhs(1) - (-beta) * BC.u0;
27
28      % Account for boundary conditions: Neumann du/dx = BC.upf at far boundary.
29       sub(end) =  -beta;
30      diag(end) = 1+beta;
31       rhs(end) = rhs(end) + beta * BC.upf * dx;
32
33  end
```