

## 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

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

where  $u(x, t)$  is the temperature in  $^\circ\text{C}$ . 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  $\Delta t$  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

$$u = 100 \exp(-0.3738t) \sin(\pi x/2). \quad (2)$$

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,

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

## 2 METHODOLOGY

### 2.1 PROBLEM 1

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

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

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  $\alpha \equiv k/c\rho = 0.1515 \text{ m/s}^2$ . Solving for the unknown quantity, we obtain

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

where the diffusion number is defined as

$$d = \frac{\alpha \Delta t}{\Delta x^2} \quad \Rightarrow \quad \Delta t = \frac{d \Delta x^2}{\alpha} . \quad (6)$$

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

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

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:

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

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

where

$$\beta = \frac{\alpha \Delta t}{2\Delta x^2} \quad (10)$$

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

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

and make the necessary modifications to the diagonal and the RHS vector in our Matlab code based on evaluating (9) with  $i = N - 1$ , and substituting in the relationship (11). Recall that the solution vector for our matrix system is  $[u_2^{n+1}, \dots, u_{N-1}^{n+1}]^T$ . Performing the aforementioned steps, we see that the Neumann boundary condition modifies the equation for the last grid point *within* the domain as

$$\underbrace{(-\beta)}_{b_{N-1}} u_{N-2}^{n+1} + \underbrace{(1 + \beta)}_{a_{N-1}} u_{N-1}^{n+1} = \underbrace{\beta (u_{N-2}^n - 2u_{N-1}^n + u_N^n)}_{d_{N-1}} + u_{N-1}^n . \quad (12)$$

Note 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  $u_N^{n+1} = u_{N-1}^{n+1}$  to preserve (11).

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  $u_N^{n+1} = u_{N-1}^{n+1} + A \Delta x$ .

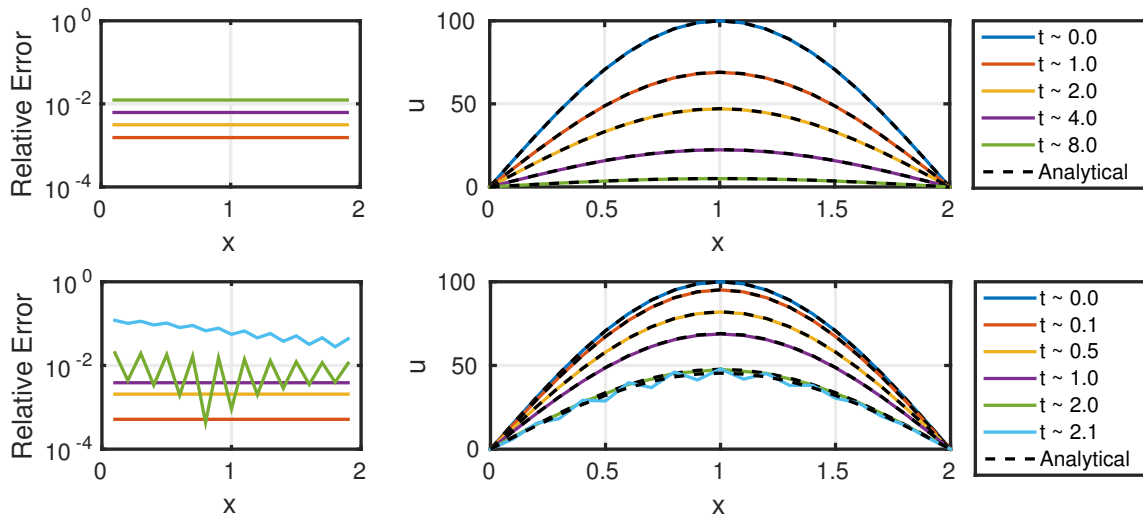
It should be noted that using these particular one-sided spatial differences, as we have, reduces the accuracy of our solution to only  $\mathcal{O}(\Delta t^2, \Delta x)$ .

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.



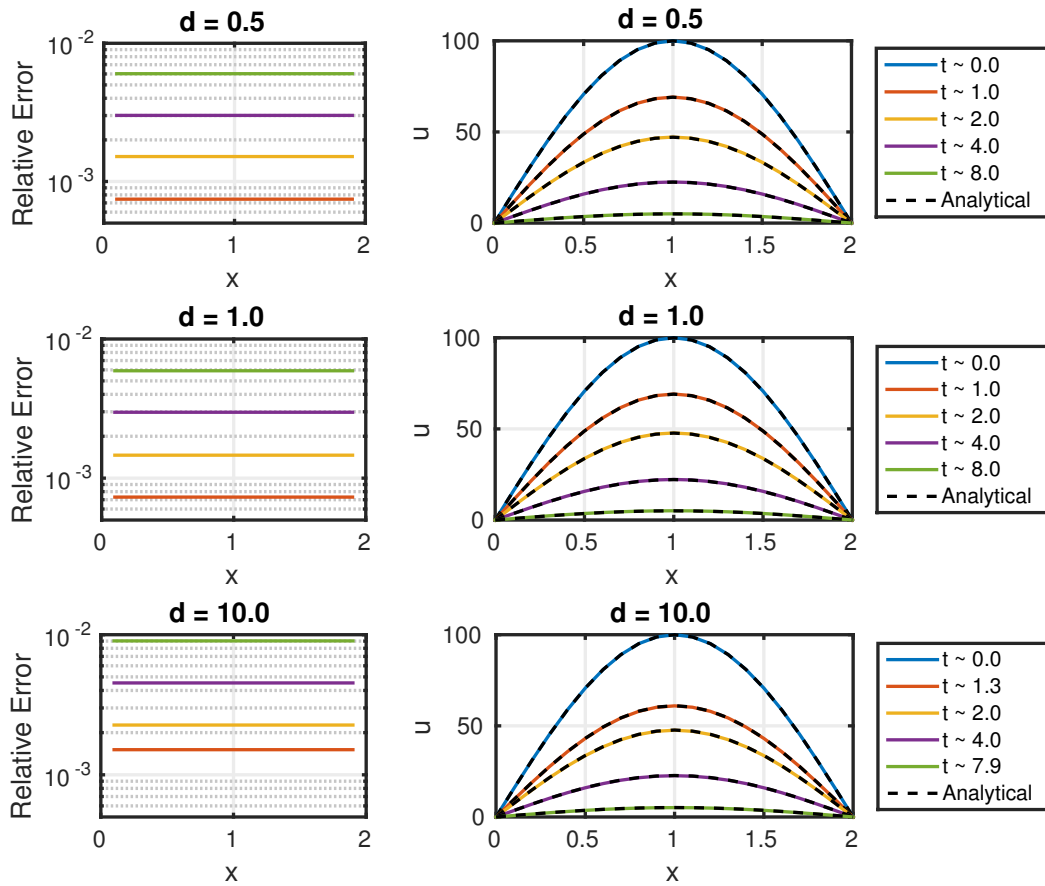
**Figure 1:** FTCS solutions to (1) compared to analytical solutions (2), and the point-wise relative error over time. Upper plots are for  $d = \frac{1}{2}$  ( $\Delta t = 0.033$ ), and lower plots correspond to  $d = 1$  ( $\Delta t = 0.066$ ).

#### 3.2 PROBLEM 2

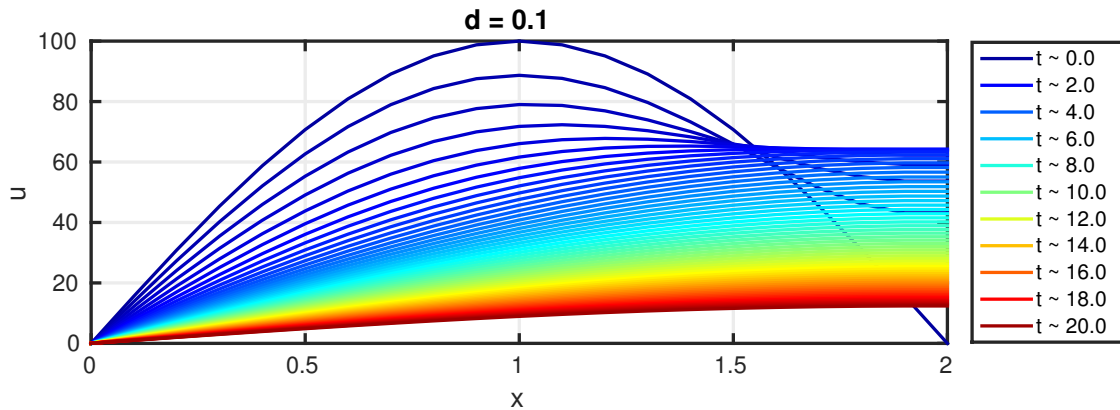
Results for the Crank-Nicolson method with Dirichlet boundary conditions are presented in Figure 2.

#### 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 problem 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  $10\times$  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,  $\Delta t$  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

```
1 function [] = Problem_1()
2
3     %%%%%%
4     % Solves the heat conduction equation ( $u_t = \alpha u_{xx}$ ) using the FTCS method.
5     %
6     % Ryan Skinner, October 2015
7     %%%
8
9     Set_Default_Plot_Properties();
10
11     %%%
12     % Define variables specific to the boundary-value problem.
13     %%%
14
15     % Material properties.
16     alpha = 0.1515;
17
18     % Solution domain: the closed interval  $[0,2]$ .
19     N = 21;
20     x = linspace(0,2,N);
21     dx = x(2) - x(1);
22
23     % Boundary conditions ( $\theta$  = initial,  $f$  = final).
```

```

24 BC.u0 = 0;
25 BC.uf = 0;
26
27 % Initial conditions.
28 ux0 = 100 * sin(pi * x / 2);
29
30 % Condition numbers, and maximum time to reach in each case.
31 d = [0.5, 1.0];
32 tmax = [8,4];
33
34 % Time steps.
35 dt = dx^2 * d / alpha;
36
37 % Maximum time steps to reach.
38 tsmx = ceil(tmax ./ dt);
39
40 % Store solutions indexed by time step, and spatial index.
41 u = {nan(tsmx(1),length(x)), nan(tsmx(2),length(x))};
42
43 % Initialize solution and set boundary conditions.
44 u{1}(1,:) = ux0;
45 u{2}(1,:) = ux0;
46 u{1}(:,1) = BC.u0;
47 u{1}(:,N) = BC.uf;
48 u{2}(:,1) = BC.u0;
49 u{2}(:,N) = BC.uf;
50
51 %%%
52 % Solve problem numerically.
53 %%%
54
55 % Loop over condition numbers.
56 for di = 1:length(d)
57     fprintf('Working with d = %3.1f, dt = %4f\n', d(di), dt(di));
58     % Iterate over time steps.
59     for n = 1:tsmx(di)-1
60         % Loop over spatial points.
61         for j = 2:N-1
62             u{di}(n+1,j) = u{di}(n,j-1) * d(di) ...
63                 + u{di}(n,j) * (1-2*d(di)) ...
64                 + u{di}(n,j+1) * d(di);
65         end
66     end
67 end
68
69 %%%
70 % Process results: Part (a).
71 %%%
72
73 ts_plot = 1 + round([0,1,2,4,8] / dt(1));
74 t_plot = (ts_plot-1) * dt(1);
75
76 % Solution.
77
78 hl = cell(length(ts_plot)+1,1);
79 dn = cell(length(ts_plot)+1,1);
80
81 figure();
82 hold on;
83 for i = 1:length(ts_plot)
84     hl{i} = plot(x, u{1}(ts_plot(i),:));
85     dn{i} = sprintf('t ~ %.1f',t_plot(i));
86     if i ~= length(ts_plot)
87         plot(x, analytical(t_plot(i),x), '--k');
88     else
89         hl{i+1} = plot(x, analytical(t_plot(i),x), '--k');
90         dn{i+1} = 'Analytical';
91     end
92 end

```

```

93     xlabel('x');
94     ylabel('u');
95     hleg = legend([hl{:}], dn);
96     set(hleg, 'Location', 'eastoutside');
97
98     % Relative error.
99
100    figure();
101    hax = axes();
102    hold on;
103    for i = 1:length(ts_plot)
104        relerr = abs(analytical(t_plot(i),x(2:end-1)) - u{1}(ts_plot(i),2:end-1)) ./ ...
105                analytical(t_plot(i),x(2:end-1));
106        plot(x(2:end-1), relerr, ...
107            'DisplayName', sprintf('t ~ %.1f',t_plot(i)));
108    end
109    set(hax,'YScale','log');
110    xlabel('x');
111    ylabel('Relative Error');
112    %hleg = legend('show');
113    %set(hleg, 'Location', 'eastoutside');
114
115    %%%
116    % Process results: Part (b).
117    %%%
118
119    ts_plot = 1 + round([0, 0.1, 0.5, 1, 2, 2.1] / dt(2));
120    t_plot = (ts_plot-1) * dt(2);
121
122    % Solution.
123
124    hl = cell(length(ts_plot)+1,1);
125    dn = cell(length(ts_plot)+1,1);
126
127    figure();
128    hold on;
129    for i = 1:length(ts_plot)
130        hl{i} = plot(x, u{2}(ts_plot(i),:));
131        dn{i} = sprintf('t ~ %.1f',t_plot(i));
132        if i ~= length(ts_plot)
133            plot(x, analytical(t_plot(i),x), '--k');
134        else
135            hl{i+1} = plot(x, analytical(t_plot(i),x), '--k');
136            dn{i+1} = 'Analytical';
137        end
138    end
139    xlabel('x');
140    ylabel('u');
141    hleg = legend([hl{:}], dn);
142    set(hleg, 'Location', 'eastoutside');
143
144    % Relative error.
145
146    figure();
147    hax = axes();
148    hold on;
149    for i = 1:length(ts_plot)
150        relerr = abs(analytical(t_plot(i),x(2:end-1)) - u{2}(ts_plot(i),2:end-1)) ./ ...
151                analytical(t_plot(i),x(2:end-1));
152        plot(x(2:end-1), relerr, ...
153            'DisplayName', sprintf('t ~ %.1f',t_plot(i)));
154    end
155    set(hax,'YScale','log');
156    xlabel('x');
157    ylabel('Relative Error');
158    %hleg = legend('show');
159    %set(hleg, 'Location', 'eastoutside');
160
161 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    tsmx = ceil(tmax ./ dt);
40
41    % Store solutions indexed by time step, and spatial index.
42    u = {nan(tsmx(1),length(x)), nan(tsmx(2),length(x)), nan(tsmx(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));

```



```

62     beta = alpha * dt(di) / (2 * dx^2);
63     % Iterate over time steps.
64     for n = 1:tsmax(di)-1
65         [diag, sub, sup, rhs] = Assemble_u(beta, u{di}(n,:), BC);
66         [sol] = Thomas(diag, sub, sup, rhs);
67         u{di}(n+1,:) = [BC.u0; sol; BC.uf];
68     end
69 end
70
71 %%%
72 % Process results.
73 %%%
74
75 for di = 1:length(d)
76
77     ts_plot = 1 + round([0,1,2,4,8] / dt(di));
78     t_plot = (ts_plot-1) * dt(di);
79
80     % Solution.
81
82     hl = cell(length(ts_plot)+1,1);
83     name = cell(length(ts_plot)+1,1);
84
85     figure();
86     hold on;
87     for i = 1:length(ts_plot)
88         hl{i} = plot(x, u{di}(ts_plot(i),:));
89         name{i} = sprintf('t ~ %.1f', t_plot(i));
90         if i ~= length(ts_plot)
91             plot(x, analytical(t_plot(i),x), '--k');
92         else
93             hl{i+1} = plot(x, analytical(t_plot(i),x), '--k');
94             name{i+1} = 'Analytical';
95         end
96     end
97     xlabel('x');
98     ylabel('u');
99     title(sprintf('d = %.1f', d(di)));
100    hleg = legend([hl{:}], name);
101    set(hleg, 'Location', 'eastoutside');
102
103    % Relative error.
104
105    figure();
106    hax = axes();
107    hold on;
108    for i = 1:length(ts_plot)
109        relerr = abs(analytical(t_plot(i),x(2:end-1)) - u{di}(ts_plot(i),2:end-1)) ./ ...
110                analytical(t_plot(i),x(2:end-1));
111        plot(x(2:end-1), relerr, ...
112            'DisplayName', sprintf('t ~ %.1f', t_plot(i)));
113    end
114    set(hax, 'YScale', 'log');
115    xlabel('x');
116    ylabel('Relative Error');
117    title(sprintf('d = %.1f', d(di)));
118    ylim([5*10^-4, 10^-2]);
119
120 end
121
122 end
123
124 function [ u_exact ] = analytical( t, x )
125     % Compute analytical solution.
126     u_exact = 100 * exp(-0.3738*t) * sin(pi * x / 2);
127 end

```

**Listing 3: Assemble\_u.m**

```

1 function [diag, sub, sup, rhs] = Assemble_u( beta, u_prev, BC )
2
3     %%%%%%
4     % Assembles the LHS matrix and the RHS vector for the g-system.
5     %   diag -- diagonal
6     %   sub -- sub-diagonal
7     %   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.
26    rhs(1) = rhs(1) - (-beta) * BC.u0;
27    rhs(end) = rhs(end) - (-beta) * BC.uf;
28
29 end

```

#### Listing 4: Problem\_3.m

```

1 function [] = Problem_3()
2
3     %%%%%%
4     % Solves the heat conduction equation ( $u_t = \alpha u_{xx}$ ) using Crank-Nicolson and an
5     % adiabatic far boundary condition (Neumann).
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 ( $\theta$  = initial,  $f$  = final).
25    BC.u0 = 0;
26    BC.upf = 0; % Where up = u prime = du/dx.
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 = 0.1;
33    tmax = 20;
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));
43
44 % Initialize solution and set boundary conditions.
45 u(1,:) = ux0;
46
47 %%%
48 % Solve problem numerically.
49 %%%
50
51 fprintf('Working with d = %4.1f, dt = %.4f\n', d, dt);
52
53 beta = alpha * dt / (2 * dx^2);
54
55 % Iterate over time steps, taking advantage of LU-decomposition efficiency.
56 for n = 1:tsmax-1
57     [diag, sub, sup, rhs] = Assemble_u_Prob3(beta, u(n,:), BC, dx);
58     if n == 1
59         [l_vec, u_vec] = LU-Decompose(diag, sub, sup);
60     end
61     [sol] = LU-Solve(sub, l_vec, u_vec, rhs);
62     u(n+1,:) = [BC.u0; sol; sol(end) + dx * BC.upf];
63 end
64
65 %%%
66 % Process results.
67 %%%
68
69 n_plot = 61;
70 cmap = jet(n_plot);
71
72 ts_plot = round(linspace(0,tmax,n_plot) / dt);
73 ts_plot(1) = ts_plot(1) + 1;
74 t_plot = (ts_plot) * dt;
75 t_plot(1) = t_plot(1) - dt;
76
77 % Solution.
78
79 hl = cell(length(ts_plot),1);
80 name = cell(length(ts_plot),1);
81
82 figure();
83 hold on;
84 for i = 1:length(ts_plot)
85     hl{i} = plot(x, u(ts_plot(i,:),:),'Color',cmap(i,:));
86     name{i} = sprintf('t ~ %.1f',t_plot(i));
87 end
88 xlabel('x');
89 ylabel('u');
90 title(sprintf('d = %.1f',d));
91 legend_indices = ceil(linspace(1,n_plot,11));
92 hleg = legend([hl{legend_indices}], name{legend_indices});
93 set(hleg, 'Location', 'eastoutside');
94
95 end

```

**Listing 5: Assemble\_u\_Prob3.m**

```

1 function [diag, sub, sup, rhs] = Assemble_u_Prob3( beta, u_prev, BC, dx )
2
3 %%%
4 % Assembles the LHS matrix and the RHS vector for the g-system.
5 %   diag -- diagonal
6 %   sub -- sub-diagonal
7 %   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

```