University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

NE 217

Laboratory 3

Prepared by

Sharda, Vid

UW Student ID Number: 20833192

UW User ID: v2sharda @uwaterloo.ca

Prepared by

Chockalingam Shanmugam, Anish

UW Student ID Number: 20814528

UW User ID: a2chocka @uwaterloo.ca

2B Electrical/Nanotechnolgy Engineering

27 June 2021

**3.1-7** **Implementing the Solution**

You are required to follow the same instructions from the previous Laboratory. You must include all the comments you were required to include in your previous submissions. You will be graded on your comments, your explanations, your coding style as well as functionality. Lack of proper indentation, documentation, and comments will result in a loss of marks.

function [x\_out, t\_out, U\_out] = crank\_nicolson1d( kappa, x\_rng, nx, t\_rng, nt, u\_init, u\_bndry )

% Using the heat-conduction/diffusion equation, we are able to determine the temperature at a particular time for a specific point in

%space. This will allow us to know the temperature and how it changes over time until an equilibrium is obtained.

%Parameters

% ==========

% kappa = The diffusivity coefficient

% x\_rng = The bounds of the space range that is being considered

% t\_rng = The range of time that is being considered

%

% u\_init = The function handle that is giving the initial state

% u\_bndry = The function handle giving the boundary conditions

%

% nx = The number of intervals the x-range should be divided by

% nt = The number of intervals the time should be divided into

%

% Return Values

% =============

% x\_out = The vector x is the boundary of values for the U\_out

% t\_out = The vector t is the top and bottom boundaries for the matrix

% U\_out = The matrix U is the corresponding temperature values at a given time for each point in that space

function [x\_out, t\_out, U\_out] = crank\_nicolson1d( kappa, x\_rng, nx, t\_rng, nt, u\_init, u\_bndry )

%Argument checking

if ~isscalar( kappa )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument kappa is not a scalar' ) );

end

if ~all( size( x\_rng ) == [1, 2] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument x\_rng is not a 2-dimensional row vector' ) );

end

if ~isscalar( nx ) || ( nx ~= round( nx ) )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument nx is not an integer' ) );

end

if ~all( size( t\_rng ) == [1, 2] )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument t\_rng is not a 2-dimensional row vector' ) );

end

if ~isscalar( nt ) || ( nt ~= round( nt ) )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument tx is not an integer' ) );

end

if ~isa( u\_init, 'function\_handle' )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument u\_init is not a function handle' ) );

end

if ~isa( u\_bndry, 'function\_handle' )

throw( MException( 'MATLAB:invalid\_argument', ...

'the argument u\_bndry is not a function handle' ) );

end

%Determining the number of discrete points we want for a certain space

%interval

h = (x\_rng(2) - x\_rng(1))/(nx - 1);

%Determining the change in time for a specific interval

dt = (t\_rng(2)- t\_rng(1))/(nt-1);

%Determining the constant for the heat/diffusion equation

const = (kappa\*dt)/(h^2);

r = const;

% Check to make sure the kappa\*dt/h^2 value is less than 0.5 to

% ensure small error. If it is not less than 0.5, we must alert the user to change their nt value and stop the function.

if r >= 0.5

warning( 'MATLAB:questionable\_argument', 'The arguments %d and %d are sub-optimal',nx, nt);

end

%Determining the x-values depending on the boundaries

x\_out = linspace(x\_rng(1), x\_rng(2), nx)';

%Determining the t-values depending on the boundaries

t\_out = linspace(t\_rng(1), t\_rng(2), nt);

%Creating a matrix of zeros that is going to be used for the values of the

%resulting heat/diffusion equation

U\_out = zeros(nx, nt);

%Intializing the values for the first column

col\_1 = u\_init(x\_out);

%Assigning the values for the first column of the resulting matrix

U\_out(:,1) = col\_1;

%Initializing the top and bottom boundaries of the matrix

boundaries= u\_bndry(t\_out(1, 2:end));

a\_bndry = boundaries(1,:);

b\_bndry = boundaries(2,:);

% Putting the boundaries into the matrix

U\_out(1,2:end) = a\_bndry;

U\_out(end,2:end) = b\_bndry;

%Check if any boundary is insulated

a = isnan(a\_bndry(1));

b = isnan(b\_bndry(1));

if a == 1

%Boundary b is insulated

crank\_matrix\_A = zeros(nx-2, nx-2);

middle = 2.\*(1+r);

top\_bottom = -1.\*r;

crank\_matrix\_middle = ones(1, nx-2);

crank\_matrix\_middle = crank\_matrix\_middle.\*middle;

crank\_matrix\_middle(1) = 2+(2/3)\*r;

crank\_matrix\_middle = diag(crank\_matrix\_middle, 0);

crank\_matrix\_super = ones(1, nx-3);

crank\_matrix\_super = crank\_matrix\_super.\*top\_bottom;

crank\_matrix\_super(1) = (-2/3)\*r;

crank\_matrix\_super = diag(crank\_matrix\_super, 1);

crank\_matrix\_sub = ones(1, nx-3);

crank\_matrix\_sub = crank\_matrix\_sub.\*top\_bottom;

crank\_matrix\_sub = diag(crank\_matrix\_sub, -1);

crank\_matrix\_A = crank\_matrix\_A + crank\_matrix\_middle + crank\_matrix\_super + crank\_matrix\_sub;

for x = 2:nt

z = U\_out(2:end-1, x-1);

y = U\_out(1:end, x-1);

crank\_known\_1 = 2.\*z;

crank\_known\_2 = r.\*diff(y, 2);

crank\_known\_3 = zeros(nx-2, 1);

%crank\_known\_3(1) = r.\*U\_out(1, x);

crank\_known\_3(end) = r.\*U\_out(end, x);

crank\_known = crank\_known\_1 + crank\_known\_2 + crank\_known\_3;

crank\_matrix\_A;

new\_z = crank\_matrix\_A\crank\_known;

U\_out(2:end-1, x) = new\_z;

U\_out(1, x) = (4/3)\*U\_out(2, x) - (1/3)\*U\_out(3, x);

end

end

if b == 1

%Boundary b is insulated

crank\_matrix\_A = zeros(nx-2, nx-2);

middle = 2.\*(1+r);

top\_bottom = -1.\*r;

crank\_matrix\_middle = ones(1, nx-2);

crank\_matrix\_middle = crank\_matrix\_middle.\*middle;

crank\_matrix\_middle(end) = 2+(2/3)\*r;

crank\_matrix\_middle = diag(crank\_matrix\_middle, 0);

crank\_matrix\_super = ones(1, nx-3);

crank\_matrix\_super = crank\_matrix\_super.\*top\_bottom;

%crank\_matrix\_super(1) = (-2/3)\*r;

crank\_matrix\_super = diag(crank\_matrix\_super, 1);

crank\_matrix\_sub = ones(1, nx-3);

crank\_matrix\_sub = crank\_matrix\_sub.\*top\_bottom;

crank\_matrix\_sub(end) = (-2/3)\*r;

crank\_matrix\_sub = diag(crank\_matrix\_sub, -1);

crank\_matrix\_A = crank\_matrix\_A + crank\_matrix\_middle + crank\_matrix\_super + crank\_matrix\_sub;

for x = 2:nt

z = U\_out(2:end-1, x-1);

y = U\_out(1:end, x-1);

crank\_known\_1 = 2.\*z;

crank\_known\_2 = r.\*diff(y, 2);

crank\_known\_3 = zeros(nx-2, 1);

crank\_known\_3(1) = r.\*U\_out(1, x);

%crank\_known\_3(end) = r.\*U\_out(end, x);

crank\_known = crank\_known\_1 + crank\_known\_2 + crank\_known\_3;

new\_z = crank\_matrix\_A\crank\_known;

U\_out(2:end-1, x) = new\_z;

U\_out(end, x) = (4/3)\*U\_out(end-1, x) - (1/3)\*U\_out(end-2, x);

end

end

if (a == 0 && b == 0)

%Looping through all the points in the matrix with a zero value and

%determining the value using the diffusion equation

crank\_matrix\_A = zeros(nx-2, nx-2);

middle = 2.\*(1+r);

top\_bottom = -1.\*r;

crank\_matrix\_middle = ones(1, nx-2);

crank\_matrix\_middle = crank\_matrix\_middle.\*middle;

crank\_matrix\_middle = diag(crank\_matrix\_middle, 0);

crank\_matrix\_super = ones(1, nx-3);

crank\_matrix\_super = crank\_matrix\_super.\*top\_bottom;

crank\_matrix\_super = diag(crank\_matrix\_super, 1);

crank\_matrix\_sub = ones(1, nx-3);

crank\_matrix\_sub = crank\_matrix\_sub.\*top\_bottom;

crank\_matrix\_sub = diag(crank\_matrix\_sub, -1);

crank\_matrix\_A = crank\_matrix\_A + crank\_matrix\_middle + crank\_matrix\_super + crank\_matrix\_sub;

for x = 2:nt

z = U\_out(2:end-1, x-1);

y = U\_out(1:end, x-1);

crank\_known\_1 = 2.\*z;

crank\_known\_2 = r.\*diff(y, 2);

crank\_known\_3 = zeros(nx-2, 1);

crank\_known\_3(1) = r.\*U\_out(1, x);

crank\_known\_3(end) = r.\*U\_out(end, x);

crank\_known = crank\_known\_1 + crank\_known\_2 + crank\_known\_3;

crank\_matrix\_A;

new\_z = crank\_matrix\_A\crank\_known;

U\_out(2:end-1, x) = new\_z;

end

end

end

**3.8** **Testing your Implementation**

We are now ready to test your code.

**3.8*a*** In the laboratory slides (pg 35), an example is shown where the initial condition is  
*u*init(*x*) = 1, *a*bndry(*t*) = –1 and *b*bndry(*t*) = 2 with *x* on [0, 1] with *nx* = 41and *t* on [0, 1] with *nt* = 11. The implementation of the two functions are provided here:

function [u] = u3a\_bndry(t)

u = [0\*t + -1;

0\*t + 2];

end

function [u] = u3a\_init(x)

u = 0\*x + 1;

end

Execute the plot command shown in the slides and copy the resulting plot into Figure 1 including the appropriate title that includes the UW User IDs of the laboratory members.

[x3a, t3a, U3a] = crank\_nicolson1d( 1.5, [0 1], 6, [0 1], 21, @u3a\_init, @u3a\_bndry );

mesh( t3a, x3a, U3a )

% title( 'uwuserid' );

% title( 'uwuserid and uwuserid' );

Repeat this exercise with the example with *nt* = 41 and copy the resulting plot into Figure 2.

Chart, surface chart

Description automatically generated

Figure 1. The first approximation.

Nx=41 and Nt=41

Chart, surface chart

Description automatically generated

Figure 2. The second approximation.

**3.8*b*** On the Laboratory web site in the source directory (see Extra Files) is an initial function combining a number of sine functions:



The actual solution to the heat-conduction/diffusion equation when this is the initial state with ** = 1 on the *x*-interval [0, **] with zero boundary conditions (*a*(*t*) = *b*(*t*) = 0) is the function



When *t* = 0, the function *u*(*x*, 0) equals *u*init(*x*) and if you differentiate *u*(*x*,*t*) once with respect to time, you will see that it equals *u*(*x*,*t*) when differentiated twice with respect to space. Plot the output of the command

[x3b, t3b, U3b] = crank\_nicolson1d( 1, [0,pi], 10, [0, 2], 20, @u3b\_init, @u3b\_bndry );

in Figure 3. The boundary-condition function is not provided.

Chart

Description automatically generated

Figure 3. The plot of the approximation.

Now, U3a(:,end) is the approximation of the solution *u*(*x*, *t*final ). Consider the following that calculates the error between the approximation of the solution at the final time and the actual value of the function at the final time.

plot( x3b, u3b(x3b, t3b(end)), 'or' );

hold on

plot( x3b, U3b(:,end), 'xb' );

norm( U3b(:,end) - u3b(x3b, t3b(end)) )/length( x3b )

What happens to this error if you:

1. Double the number of points in space (double *nx*)?
2. Double the number of points in time (double *nt*)?

Provide the output of the calls to the norm function.

Your arguments, plots, Matlab code, *etc*., here.

function [u] = u3b\_bndry(t)

u = [0\*t ;

0\*t ];

end

[x3b, t3b, U3b] = crank\_nicolson1d ( 1, [0,pi], 10, [0, 2], 20, @u3b\_init, @u3b\_bndry );

Chart, scatter chart

Description automatically generated

ans =

6.7997e-04

[x3b, t3b, U3b] = crank\_nicolson1d( 1, [0,pi], 20, [0, 2], 20, @u3b\_init, @u3b\_bndry );

Chart, scatter chart

Description automatically generated

ans =

7.2203e-05

[x3b, t3b, U3b] = crank\_nicolson1d( 1, [0,pi], 10, [0, 2], 40, @u3b\_init, @u3b\_bndry );

Chart, scatter chart

Description automatically generated

ans =

7.3098e-04

When the number of points in space doubles the error almost is one-tenth

But when the number of points in time is doubles the error is approximate the same. (or in this case is slightly higher)

**3.8*c*** Up to this point, we have seen Dirichlet boundary conditions. Suppose we have the following initial and boundary conditions:



where the space interval is [0, 1]. Use a value of ** = 1 and estimate at which time *t*1\* the value at the second boundary point (*b* = 1) is approximately equal to 0.05. Indicate the function calls that you made and show how you determined the appropriate value of *t*1\*. You must be able to convince the reader that the value of *t*1\* is appropriate. You must include at least one plot and your arguments must reference any plots. Your plots must have the UW User IDs of the student in the title.

Consider the following to help you refine your search for *t*1\*:

U3c(end,:)

t3c

and, once you are closer,

U3c(end,end-10:end)

t3c(end-10:end)

Your arguments, plots, Matlab code, *etc*., here.

>> [x3c, t3c, U3c] = crank\_nicolson1d( 1, [0,1], 10, [0, 2], 40, @u3c\_init, @u3c\_bndry );

>> [U3c(end,:)' t3c']

ans =

…

0.0651 1.2308

0.0573 1.2821

0.0505 1.3333

0.0445 1.3846

0.0392 1.4359

0.0346 1.4872

…

To find the approximate area of U at b = 0.05. (some rows omitted in shown output)

>>[x3c, t3c, U3c] = crank\_nicolson1d ( 1, [0,1], 30, [0, 1.35], 100, @u3c\_init, @u3c\_bndry );

>> [U3c(end,end-10:end)' t3c(end-10:end)']

ans =

0.0648 1.2136

0.0627 1.2273

0.0606 1.2409

0.0586 1.2545

0.0567 1.2682

0.0548 1.2818

0.0530 1.2955

0.0512 1.3091

0.0495 1.3227

0.0479 1.3364

0.0463 1.3500

>> [x3c, t3c, U3c] = crank\_nicolson1d ( 1, [0,1], 300, [0, 1.35], 200, @u3c\_init, @u3c\_bndry );

>> [U3c(end,end-10:end)' t3c(end-10:end)']

ans =

0.0543 1.2822

0.0534 1.2889

0.0525 1.2957

0.0516 1.3025

0.0508 1.3093

0.0499 1.3161

0.0491 1.3229

0.0483 1.3296

0.0475 1.3364

0.0467 1.3432

0.0459 1.3500

Chart, surface chart

Description automatically generated

From the graph, it can be seen that at t = 1.3 (X), and b = 1 (Y) the value of U = 0.05 (Z)

So, at time t=1.3161, the boundary b = 0.05

**3.8*d*** In Question 3.8*c*, you used a value of ** = 1. What happens to the value of *t*\* if you use a value of ** = 0.5 or ** = 2? Can you come up with a general rule? Indicate any function calls in Matlab you made that support your general rule. Use *t*0.5\* and *t*2\* to refer to the critical times when ** = 0.5 or ** = 2, respectively.

At k = 2

>> [U3c(end,:)' t3c']

ans =

…

0.0520 0.6612

…

K = 0.5

Increased t range as u=0.05 was not found

>> [x3c, t3c, U3c] = crank\_nicolson1d( 0.5, [0,1], 100, [0, 3], 50, @u3c\_init, @u3c\_bndry );

>> [U3c(end,:)' t3c']

ans =

…

0.0513 2.6327

…

So, time to attain a certain temperature is inversely proportional to K.

**3.8*e*** Determine what happens if both boundaries are insulated. Do this by starting with various initial functions and watching what happens as time increases. Once you have determined the general rule, you must state that rule and support your arguments with at least two plots. In addition, you may consider the following function calls:

sum( U3e(:, ell) )

**3.9** Did you remember to copy your entire function into Question 3.7? That is, all comments and all code?