

Numerical Methods with MATLAB for Complex Systems

Auralius Manurung
Mechanical Engineering, Universitas Pertamina

Table of Contents

1. Case 1: One-Dimensional Linear Advection¹	4
1.1. <u>Forward in Time Centered in Space (FTCS)</u>	4
1.2. <u>Upwind Method</u> ($c > 0$)	4
Figure 1: Upwind method for $u_t + 0.5u_{xx} = 0$	6
1.3. <u>Downwind Method</u> ($c < 0$)	6
Figure 2: Downwind method for $u_t - 0.5u_{xx} = 0$	8
1.4. <u>The Lax Method</u>	8
Figure 3: Lax method for $u_t - 0.5u_{xx} = 0$	11
Figure 4: Lax method for $u_t + 0.5u_{xx} = 0$	11
1.5. <u>The Lax-Wendroff Method</u> ²	11
Figure 5: Lax method for $u_t - 0.5u_{xx} = 0$	15
Figure 6: Lax method for $u_t + 0.5u_{xx} = 0$	15
1.6. <u>Cross Current Heat Exchanger as An Example</u> ³	15
1.6.1. <u>Two-Equation Model</u>	15
Drawing 1: Flowing liquid in red container while transferring heat to its environment	16
Drawing 2: Cross-current heat exchanger modeled with two PDEs	16
Figure 7: After 20 s, with upwind-downwind method	19
Figure 8: After 20 s, with Lax-Wendroff method	19
1.6.2. <u>Three-Equation Model</u>	19
Drawing 3: Cross-current heat exchanger modeled with three PDEs	19
Drawing 4: Temperature distributions ofter 5 seconds	21
2. Case 2: One-Dimensional Heat Equation⁴	22
Drawing 5: Temperature distribution (u) in a very thin rod	22
2.1. <u>Explicit Method</u>	22
2.1.1. <u>Dirichlet Boundary Conditions</u>	23
2.1.2. <u>Neumann Boundary Condition</u>	23
Drawing 6: Extra phantom nodes for Neumann boundary condition	23
2.1.3. <u>MATLAB Implementation</u>	23
2.1.4. Example 1	24
Drawing 7: Simulation result: Dirichlet-Dirichlet	26
2.1.5. Example 2	26

Drawing 8: Simulation result: Dirichlet-Neumann	27
2.1.6. Example 3	27
Drawing 9: Simulation result: Neumann-Dirichlet	28
2.2. <u>Implicit Method</u>	29
3. Case 3: Rectangular Two-Dimensional Heat Equation⁵	31
3.1. <u>The Five-Point Difference Method</u>	31
Drawing 10: Heat distribution on a two-dimensional plane	31
4. Case 4: Transient Heat Equation	32
4.1. <u>Using the UPWIND Method</u>	32
4.2. <u>Using the DOWNWIND Method</u>	33
4.3. <u>MATLAB Implementation</u>	33
Drawing 11: A discretized two-dimensional plane	33
4.4. <u>Neumann Boundary Condition</u>	34
Drawing 12: A discretized two-dimensional plane	35
4.4.1. Left Edge	35
4.4.2. Right Edge	35
4.4.3. Top Edge	35
4.4.4. Bottom Edge	36
4.5. <u>Energy Balance Model: Out-of-Plane Heat Transfer for a Thin Plate - Comparison with COMSOL</u>	36
5. Navier-Stokes with Heat Transfer	39
5.1. <u>Channel Flow with Heat Equation</u>	39
5.2. <u>Two Counter-Direction Channel Flows with Heat Equations</u>	40

1. Case 1: One-Dimensional Linear Advection⁶

Given a hyperbolic PDE as follows.

$$u_t + cu_x = f(x, t) \quad (1)$$

with the following initial condition:

$$u(x, 0) = u_0(x) \quad (2)$$

where c is a positive real number. Here, we want to find the solution to (1). When $f(x, t) = 0$, its analytical solution is given by: $u(x, t) = u_0(x - ct)$. Simply put, u_0 propagates along x with a constant speed of c .

This kind of PDE is widely used for real-time modeling and control applications. Therefore, it is essential to investigate its non-homogenous version to allow the implementation of a PDE system further. A PDE system is composed of more than one interacting PDE. For example, there are at least two PDEs in a heat exchanger: one for the hot system and another one for the cold system. The two PDEs interact by exchanging heat between them.

1.1. Forward in Time Centered in Space (FTCS)

This method is a finite difference method but with central difference for the distance to increase the approximation. To approximate u_t , we use the following approximation.

$$u_t \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \quad (3)$$

While to approach u_x , we use the following approximation.

$$u_x \approx \frac{u(x + \Delta x, t) - u(x - \Delta x, t)}{2\Delta x} \quad (4)$$

Next, substituting (3) and (4) back to (1) gives us

$$\begin{aligned} u_t &\approx -cu_x + f(x, t) \\ \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} &\approx -c \frac{u(x + \Delta x, t) - u(x - \Delta x, t)}{2\Delta x} + f(x, t) \\ u(x, t + \Delta t) - u(x, t) &\approx -\frac{c\Delta t}{2\Delta x} [u(x + \Delta x, t) - u(x - \Delta x, t)] + f(x, t)\Delta t \\ u(x, t + \Delta t) &\approx u(x, t) - \frac{c\Delta t}{2\Delta x} [u(x + \Delta x, t) - u(x - \Delta x, t)] + f(x, t)\Delta t \end{aligned}$$

FTCS method is unconditionally unstable. It should never be used. Nevertheless, FTCS is the fundamental building block for another methods, such as the Lax method.

1.2. Upwind Method ($c > 0$)

By using finite difference method, we can approximate both u_t and u_x . To approximate u_t , we use the following approximation.

$$u_t \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \quad (5)$$

While to approach u_x , we use the following approximation.

$$u_x \approx \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x} \quad (6)$$

By substituting (5) and (6) to (1), we then have:

$$\begin{aligned} u_t &\approx -cu_x + f(x, t) \\ \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} &\approx -c \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x} + f(x, t) \\ u(x, t + \Delta t) + u(x, t) &\approx -\frac{c\Delta t}{\Delta x} [u(x, t) - u(x - \Delta x, t)] + f(x, t) \\ u(x, t + \Delta t) &\approx u(x, t) - \frac{c\Delta t}{\Delta x} [u(x, t) - u(x - \Delta x, t)] + f(x, t)\Delta t \end{aligned}$$

Let us consider $\frac{c\Delta t}{\Delta x} = r$, $U_{j,k} = u(x_j, t_k)$, $F_{j,k} = f(x_j, t_k)$, $x_j = j\Delta x$ and $t_k = k\Delta t$, we then have the following shorter terms:

$$U_{j,k+1} = U_{j,k} - r(U_{j,k} - U_{j-1,k}) + F_{j,k}\Delta t \quad (7)$$

MATLAB implementation of the upwind method (7) is as follows.

```

1 function next_u_array = upwind(u_array, f_array, c, dt, dx)
2
3 N = length(u_array);
4 next_u_array = u_array;
5
6 for i = 2:N
7     next_u_array(i) = u_array(i) - c*dt/dx*(u_array(i) - u_array(i-1)) + f_array(i)*dt;;
8 end
9
10 end

```

To increase the efficiency, the listing above can be vectorized as follows.

```

1 function next_u_array = upwind(u_array, f_array, c, dt, dx)
2
3 N = length(u_array);
4 next_u_array = u_array;
5
6 next_u_array(2:N) = u_array(2:N) - (c*dt/dx).* ...
7                 (u_array(2:N) - u_array(1:N-1)) + f_array(2:N).*dt;
8
9 end

```

Next, we will implement the PDE in (8) into the MATLAB upwind function above.

$$\begin{aligned} u_t + 0.5u_x &= 0 \\ u(x, 0) &= e^{-(x-2)^2} \\ 0 \leq x \leq 10 \end{aligned} \quad (8)$$

The implementation is provided in the listing below. The length of the rod is divided into 100 segments ($dx = 0.1$) and the time is divided into 400 segments ($dt = 0.05$).

```

1 L = 10;
2 dx = 0.1;
3 x = 0:dx:L;
4
5 dt = 0.05;
6 T = 20;

```

```

7 t = 0:dt:T;
8
9 c = 0.5; % Upwind
10
11 u = zeros(length(x),1);
12 f = zeros(length(x),1);
13
14 for k = 1:length(x)
15     u(k) = exp(-(x(k)-2).^2);
16 end
17
18 figure;
19 h = plot(0,0);
20 title('Upwind')
21 ylim([0 1]);
22
23 for k = 1:length(t)
24     u = upwind(u,f, c, dt, dx);
25     set(h, 'XData',x, 'Ydata',u);
26     drawnow;
27 end

```

The analytical solution to the PDE on (8) is $u(x,t) = e^{-(x-0.5t)-2)^2}$. However, when executed, the code above will generate a numerical solution as in Figure 1. As we can see, the numerical solution is not exactly the same as its analytical solution. Over time, its amplitude is decreasing.

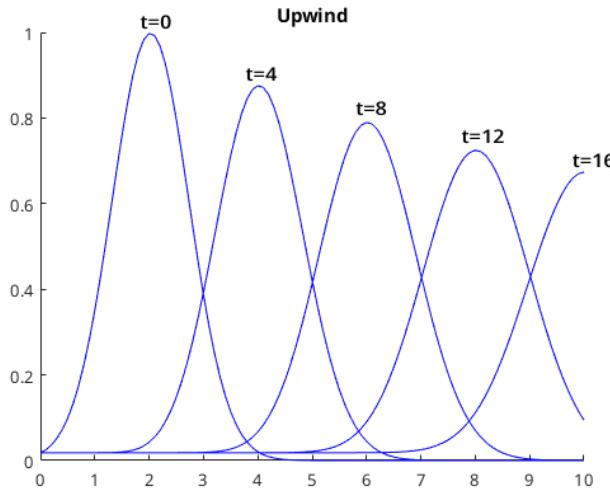


Figure 1: Upwind method for $u_t + 0.5u_{xx} = 0$

1.3. Downwind Method ($c < 0$)

By using finite difference method, we can approximate both u_t and u_x . To approximate u_t , we use the following approximation.

$$u_t \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \quad (9)$$

While to approach u_x , we use the following approximation.

$$u_x \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} \quad (10)$$

Substituting (9) and (10) back to (1) yields the following.

$$\begin{aligned} u_t &= -cu_x + f(x, t) \\ \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} &\approx -c \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} + f(x, t) \\ u(x, t + \Delta t) + u(x, t) &\approx -\frac{c\Delta t}{\Delta x} [u(x + \Delta x, t) - u(x, t)] + f(x, t)\Delta t \\ u(x, t + \Delta t) &\approx u(x, t) - \frac{c\Delta t}{\Delta x} [u(x + \Delta x, t) - u(x, t)] + f(x, t)\Delta t \end{aligned}$$

Let us consider $\frac{c\Delta t}{\Delta x} = r$, $U_{j,k} = u(x_j, t_k)$, $F_{j,k} = f(x_j, t_k)$, $x_j = j\Delta x$ and $t_k = k\Delta t$, we can then have the following shorter terms.

$$U_{j,k+1} = U_{j,k} - r(U_{j+1,k} - U_{j,k}) + F_{j,k}\Delta t \quad (11)$$

MATLAB implementation of an upwind method expressed in (11) is as follows.

```

1 function next_u_array = downwind(u_array, f_array, c, dt, dx)
2
3 N = length(u_array);
4 next_u_array = u_array;
5
6 for i = 1:N-1
7     next_u_array(i) = u_array(i) - c*dt/dx*(u_array(i+1) - u_array(i)) + f_array(i)*dt;
8 end
9
10 end

```

Further, to increase its efficiency, we can turn the MATLAB code above in to vectorized operations as follows.

```

1 function next_u_array = downwind(u_array, f_array, c, dt, dx)
2
3 N = length(u_array);
4 next_u_array = u_array;
5
6 next_u_array(1:N-1) = u_array(1:N-1) - (c*dt/dx).* ...
7                 (u_array(2:N) - u_array(1:N-1)) + f_array(1:N-1).*dt;
8
9 end

```

To test the listing above, let us solve the following first order linear PDE.

$$\begin{aligned} u_t - 0.5u_x &= 0 \\ u(x, 0) &= e^{-(x-8)^2} \\ 0 \leq x \leq 10 \end{aligned} \quad (12)$$

The implementation is provided in the listing below. The length of the rod is divided into 100 segments ($dx = 0.1$) and the time is divided into 400 segments ($dt = 0.05$).

```

1 L = 10;
2 dx = 0.1;
3 x = 0:dx:L;

```

```

4
5 dt = 0.05;
6 T = 20;
7 t = 0:dt:T;
8
9 c = -0.5; % Downwind
10
11 u = zeros(length(x), 1);
12 f = zeros(length(x), 1);
13
14 for k = 1:length(x)
15     u(k) = exp(-(x(k)-8).^2);
16 end
17
18 figure;
19 h = plot(0,0);
20 title('Downwind')
21 ylim([0 1]);
22
23 for k = 1:length(t)
24     u = downwind(u,f, c, dt, dx);
25     set(h,'XData',x, 'Ydata',u);
26     drawnow;
27 end

```

The analytical solution of (12) is $u(x,t) = e^{-(x+0.5t)-8)^2}$. However, when executed, the provided listing above generates the a solutoinas in Figure 2. As we can see, the numerical solution is not exactly the same as its analytical solution. Over time, its amplitude is actually decreasing, similar to the upwind cases.

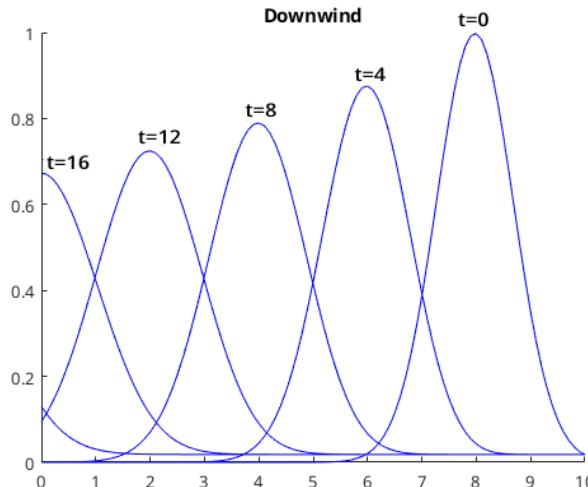


Figure 2: Downwind method for $u_t - 0.5u_{xx} = 0$

1.4. The Lax Method

The Lax method is based on the FTCS method.

$$u_t \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \quad (13)$$

However, $u(x, t)$ is replaced by an average of its two neighbors.

$$u(x, t) = \frac{u(x + \Delta x, t) + u(x - \Delta x, t)}{2} \quad (14)$$

Thus, by substituting (14) to (13), the approximation of u_t can be rewritten as

$$u_t \approx \frac{u(x, t + \Delta t) - \left[\frac{u(x + \Delta x, t) + u(x - \Delta x, t)}{2} \right]}{\Delta t} \quad (15)$$

On the other hand, u_x can be approximated as

$$u_x \approx \frac{u(x + \Delta x, t) - u(x - \Delta x, t)}{2\Delta x} \quad (16)$$

Substituting (15) and (16) back to (1) yields

$$\begin{aligned} u_t &\approx -cu_x + f(x, t) \\ \frac{u(x, t + \Delta t) - \left[\frac{u(x + \Delta x, t) + u(x - \Delta x, t)}{2} \right]}{\Delta t} &\approx -c \frac{u(x + \Delta x, t) - u(x - \Delta x, t)}{2\Delta x} + f(x, t) \\ u(x, t + \Delta t) - \left[\frac{u(x + \Delta x, t) + u(x - \Delta x, t)}{2} \right] &\approx -\frac{c\Delta t}{2\Delta x} [u(x + \Delta x, t) - u(x - \Delta x, t)] + f(x, t)\Delta t \\ u(x, t + \Delta t) &\approx \left[\frac{u(x + \Delta x, t) + u(x - \Delta x, t)}{2} \right] - \\ &\quad \frac{c\Delta t}{2\Delta x} [u(x + \Delta x, t) - u(x - \Delta x, t)] + f(x, t)\Delta t \end{aligned}$$

Now, let us consider $r = \frac{c\Delta t}{\Delta x}$, $U_{j,k} = u(x_j, t_k)$, $F_{j,k} = f(x_j, t_k)$, $x_j = j\Delta x$, $t_k = k\Delta t$, $U_{j,k+1} = u(x_j, t_k + \Delta t)$ and $U_{j-1,k} = u(x_j - \Delta x, t_k)$, we can then have the following shorter terms.

$$U_{j,k+1} = \frac{U_{j+1,k} + U_{j-1,k}}{2} - \frac{r}{2}(U_{j+1,k} - U_{j,k}) + F_{j,k}\Delta t \quad (17)$$

The Lax method can handle both upwind and downwind direction. There is no need for us to check the sign of c as in the previous methods. The listing below shows MATLAB implementation of the Lax method.

```

1  function next_u_array = lax(u_array, f_array, c, dt, dx)
2
3  N = length(u_array);
4  next_u_array = u_array;
5
6  r = c*dt/dx;
7
8  for i = 2:N-1 % i = 1 and i = N are untouched !!!
9    next_u_array(i) = 0.5*(u_array(i+1)+u_array(i-1)) - ...
10      0.5*r*(u_array(i+1)-u_array(i-1)) + ...
11      f_array(i)*dt;
12 end
13
14 % fill up the two ends by using upwind or downwind method
15 if c > 0
16    next_u_array(N) = u_array(N) - r* ...
17      (u_array(N) - u_array(N-1)) + f_array(N).*dt;
18
19 elseif c < 0
20    next_u_array(1) = u_array(1) - r* ...
21      (u_array(2) - u_array(1)) + f_array(1).*dt;
22

```

```
23 end
```

There is another practical issue with the Lax method. The Lax method leaves the two end segments untouched. This can be seen from the for loop statement that starts from 2 to N-1 (see the above listing, line 8). To address this issue, we can apply the upwind and downwind method for the two untouched ends.

Further, to increase the efficiency of the Lax method, we can turn the MATLAB code above in to vectorized operations as follows.

```
1 function next_u_array = lax(u_array, f_array, c, dt, dx)
2
3 N = length(u_array);
4 next_u_array = u_array;
5
6 r = c*dt/dx;
7
8 next_u_array(2:N-1) = 0.5.* (u_array(3:N)+(u_array(1:N-2))) - ...
9             (0.5*r).* (u_array(3:N)- u_array(1:N-2)) + ...
10            f_array(2:N-1).*dt;
11
12
13 % fill up the two ends by using upwind or downwind method
14 if c > 0
15     next_u_array(N) = u_array(N) - r* ...
16                     (u_array(N)- u_array(N-1)) + f_array(N).*dt;
17
18 elseif c < 0
19     next_u_array(1) = u_array(1) - r* ...
20                     (u_array(2) - u_array(1)) + f_array(1).*dt;
21
22 end
```

Next, let us test the code above for both the upwind and the downwind cases. For the upwind case, we will use the PDE in (8). As for the downwind direction, we will use the PDE in (12). For both methods, the length of the rod is divided into 100 segments ($dx = 0.1$) and the time is divided into 400 segments ($dt = 0.05$). The listing below is the implementation of (8) and (12).

```
1 L = 10;
2 dx = 0.05;
3 x = 0:dx:L;
4
5 dt = 0.05;
6 T = 20;
7 t = 0:dt:T;
8
9 u = zeros(length(x), 1);
10 f = zeros(length(x), 1);
11
12 %% -----
13 c = 0.5; % moving to right
14
15 for k = 1:length(x)
16     u(k) = exp(-10*(x(k)-2).^2);
17 end
18
19 figure;
```

```

20 h1 = plot(0,0);
21 title('Lax, moving to the right c = 0.5')
22 ylim([0 1]);
23
24 for k = 1:length(t)
25     u = lax(u,f, c, dt, dx);
26     set(h1,'XData',x,'Ydata',u);
27     drawnow;
28 end
29
30 %% -----
31 c = -0.5; % moving to left
32
33 for k = 1:length(x)
34     u(k) = exp(-10*(x(k)-8).^2);
35 end
36
37 figure;
38 h2 = plot(0,0);
39 title('Lax, moving to the left, c = -0.5')
40 ylim([0 1]);
41
42 for k = 1:length(t)
43     u = lax(u,f, c, dt, dx);
44     set(h2,'XData',x,'Ydata',u);
45     drawnow;
46 end

```

When executed, the above listing will generate Figure 3 and Figure 4 where we can see the resulting amplitude is also decreasing over time.

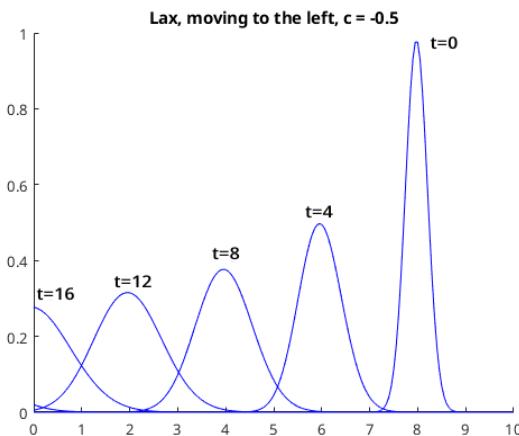


Figure 3: Lax method for $u_t - 0.5u_{xx} = 0$

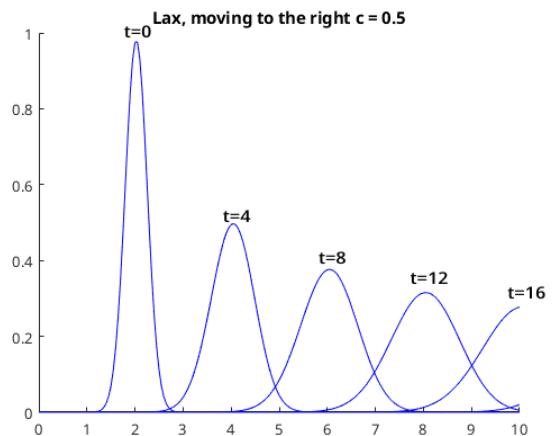


Figure 4: Lax method for $u_t + 0.5u_{xx} = 0$

1.5. The Lax-Wendroff Method⁷

We start from the Taylor series definition of $f(x + \Delta x)$

$$f(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{\Delta x^2}{2!} f''(x) + \frac{\Delta x^3}{3!} f'''(x) + \dots \quad (18)$$

Thus, taking only up to the second order, we can write

$$u(x + \Delta x, t) = u(x, t) + \Delta x u_x + \frac{\Delta x^2}{2} u_{xx} \quad (19)$$

Remember that

$$u_t = \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \quad (20)$$

and

$$u_x = \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} \quad (21)$$

Thus, $u_t = -cu_x + f(x, t)$ can be rewritten as

$$\frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} = -c \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} + f(x, t) \quad (22)$$

By substituting (19) to (22)

$$\begin{aligned} \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} &\approx -c \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} + f(x, t) \\ u(x, t + \Delta t) &\approx u(x, t) - c \frac{\Delta t}{\Delta x} [u(x + \Delta x, t) - u(x, t)] + f(x, t) \Delta t \\ u(x, t + \Delta t) &\approx u(x, t) - c \frac{\Delta t}{\Delta x} \left[\cancel{u(x, t)} + \Delta x u_x + \frac{\Delta x^2}{2} u_{xx} - \cancel{u(x, t)} \right] + f(x, t) \Delta t \\ u(x, t + \Delta t) &\approx u(x, t) - c \Delta t u_x - c \frac{\Delta t \Delta x}{2} u_{xx} + f(x, t) \Delta t \end{aligned}$$

Since c is the propagation speed of $u(x, t) = u_0(x - ct)$, thus, let us take $\Delta x = -c\Delta t$. As the result, now we have the following equation.

$$u(x, t + \Delta t) \approx u(x, t) - c \Delta t u_x + c^2 \frac{\Delta t^2}{2} u_{xx} + f(x, t) \Delta t \quad (23)$$

Next, we need to approximate u_x and u_{xx} . For u_x , we use central difference method as follows.

$$u_x \approx \frac{u(x + \Delta x, t) - u(x - \Delta x, t)}{2 \Delta x} \quad (24)$$

As for u_{xx} , we do not use the central difference method, instead, we take $u_x \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x}$. Therefore, we can express u_{xx} as follows.

$$\begin{aligned} u_{xx} &\approx \frac{u_x(x + \Delta x, t) - u_x(x, t)}{\Delta x} \\ &\approx \frac{\frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} - \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x}}{\Delta x} \\ &\approx \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2} \quad (25) \end{aligned}$$

Now that we have (24) and (25), we can substitute them back to (23):

$$\begin{aligned} u(x, t + \Delta t) &\approx u(x, t) - c \Delta t \cancel{u_x} + c^2 \frac{\Delta t^2}{2} \cancel{u_{xx}} + f(x, t) \Delta t \\ &\approx u(x, t) - c \Delta t \left[\frac{u(x + \Delta x, t) - u(x - \Delta x, t)}{2 \Delta x} \right] + \\ &\quad c^2 \frac{\Delta t^2}{2} \left[\frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2} \right] + f(x, t) \Delta t \end{aligned}$$

$$\approx u(x, t) - \frac{c\Delta t}{2\Delta x} [u(x + \Delta x, t) - u(x - \Delta x, t)] + \frac{c^2 \Delta t^2}{2\Delta x^2} [u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)] + f(x, t)\Delta t$$

Now let us consider $r = \frac{c\Delta t}{\Delta x}$, $U_{j,k} = u(x_j, t_k)$, $F_{j,k} = f(x_j, t_k)$, $x_j = j\Delta x$, $t_k = k\Delta t$, $U_{j,k+1} = u(x_j, t_k + \Delta t)$ and $U_{j-1,k} = u(x_j - \Delta x, t_k)$, we can then have the following shorter terms.

$$U_{j,k+1} = U_{j,k} - \frac{r}{2}(U_{j+1,k} - U_{j-1,k}) + \frac{r^2}{2}(U_{j+1,k} - 2U_{j,k} + U_{j-1,k}) + F_{j,k}\Delta t \quad (26)$$

MATLAB implementation of (26) is as follows (see line 6 to 11).

```

1 function next_u_array = lax_wendroff(u_array, f_array, c, dt, dx)
2
3 N = length(u_array);
4 next_u_array = u_array;
5
6 r = c*dt/dx;
7 for i = 2:N-1
8     next_u_array(i) = u_array(i) - 0.5*r*(u_array(i+1) - u_array(i-1)) ...
9                 + 0.5*r^2 * (u_array(i+1) - 2*u_array(i) + u_array(i-1)) ...
10                + f_array(i)*dt;
11 end
12
13 % fill up the two ends by using upwind or downwind method
14 if c > 0
15     next_u_array(N) = u_array(N) - r* ...
16                         (u_array(N) - u_array(N-1)) + f_array(N)*dt;
17
18 elseif c < 0
19     next_u_array(1) = u_array(1) - r* ...
20                         (u_array(2) - u_array(1)) + f_array(1)*dt;
21
22 end

```

Here, we encounter the same issue as in the previous Lax method. The Lax-Wendroff method also leaves the two end segments untouched. This can be seen clearly from the for-statement that starts from 2 to N-1 (see line 7 of the listing above). To address this issue, we apply upwind and downwind method for the two ends only (see line 14 to 20 of the listing above).

Further, to increase its efficiency, we can turn the MATLAB code above in to vectorized operations as follows.

```

1 function next_u_array = lax_wendroff(u_array, f_array, c, dt, dx)
2
3 N = length(u_array);
4 next_u_array = u_array;
5
6 r = c*dt/dx;
7
8 next_u_array(2:N-1) = u_array(2:N-1) - 0.5*r*(u_array(3:N) - u_array(1:N-2)) ...
9                 + 0.5*r^2 * (u_array(3:N) - 2*u_array(2:N-1) + u_array(1:N-2)) ...
10                + f_array(2:N-1)*dt;
11
12 % fill up the two ends by using upwind or downwind method

```

```

13 if c > 0
14     next_u_array(N) = u_array(N) - r* ...
15                 (u_array(N)- u_array(N-1)) + f_array(N)*dt;
16
17 elseif c < 0
18     next_u_array(1) = u_array(1) - r* ...
19                 (u_array(2) - u_array(1)) + f_array(1)*dt;
20
21 end

```

Next, let us test the code listings above for both the upwind and the downwind cases. For the upwind case, we will use the same system as in (8). As for the downwind direction, we will also use the same system as in (12). We will divide the length of the rod into 200 segments ($dx = 0.05$) and the time into 200 segments as well ($dt = 0.05$). See the listing below.

```

1 L = 10;
2 dx = 0.05;
3 x = 0:dx:L;
4
5 dt = 0.05;
6 T = 20;
7 t = 0:dt:T;
8
9 u = zeros(length(x), 1);
10 f = zeros(length(x), 1);
11
12 %% -----
13 c = 0.5; % moving to right
14
15 for k = 1:length(x)
16     u(k) = exp(-10*(x(k)-2).^2);
17 end
18
19 h = figure;
20 h1 = plot(0,0);
21 title('Lax-Wendroff, moving to the right c = 0.5')
22 ylim([0 1]);
23
24 for k = 1:length(t)
25     u = lax_wendroff(u,f, c, dt, dx);
26     set(h1,'XData',x,'Ydata',u);
27     drawnow
28 end
29
30 %% -----
31 c = -0.5; % moving to left
32
33 for k = 1:length(x)
34     u(k) = exp(-10*(x(k)-8).^2);
35 end
36
37 h = figure;
38 h1 = plot(0,0);
39 title('Lax-Wendroff, moving to the left, c = -0.5')
40 ylim([0 1]);
41
42 for k = 1:length(t)
43     u = lax_wendroff(u,f, c, dt, dx);
44     set(h1,'XData',x,'Ydata',u);

```

```

45 drawnow
46 end

```

When executed, the listing above will generate the Figure 5 and Figure 6. As shown by those two figures, the Lax-Wendroff method does provide more stable solutions. Even though the amplitudes are still decreasing, the decrease in the amplitude is much smaller than other methods that have been previously discussed.

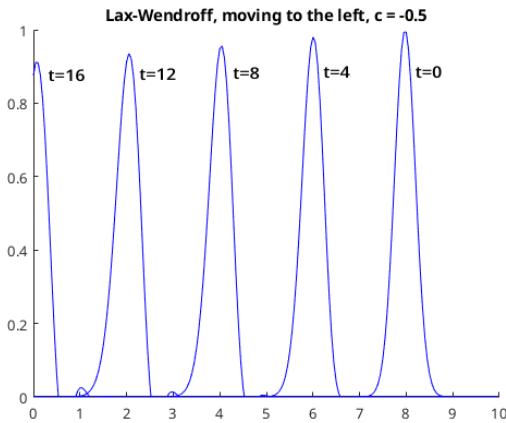


Figure 5: Lax method for $u_t - 0.5u_{xx} = 0$

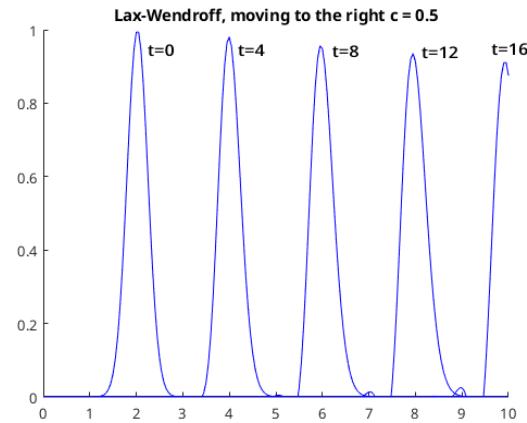


Figure 6: Lax method for $u_t + 0.5u_{xx} = 0$

1.6. Cross Current Heat Exchanger as An Example⁸

A heat exchanger is modeled with two or more interacting non-homogenous PDEs. The simplest model involves two PDEs only: the hot flow and the cold flow. Yet another simple method, three PDEs are involved: the hot flow, the cold flow, and the wall between them.

1.6.1. Two-Equation Model

Flowing liquid inside a container can be modeled as an advective PDE. When external environment is involved, the PDE becomes non-homogenous (see Drawing 1). Interaction with the environment is modeled with Fourier's Law. The overall process can be modeled as follows.

$$\underbrace{\frac{\partial T(x,t)}{\partial t} + v \frac{\partial T(x,t)}{\partial z}}_{\text{Advection}} = - \underbrace{\frac{UA_S}{\rho AC_p}(T(x,t) - T_S(x,t))}_{\text{Fourier's law}} \quad (27)$$

where

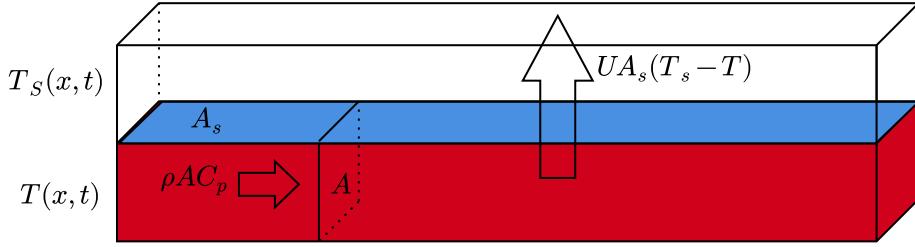
C_p is the liquid specific heat capacity

A is cross sectional area of the tube where the liquid is flowing

A_S is the heat transfer area (blue surface)

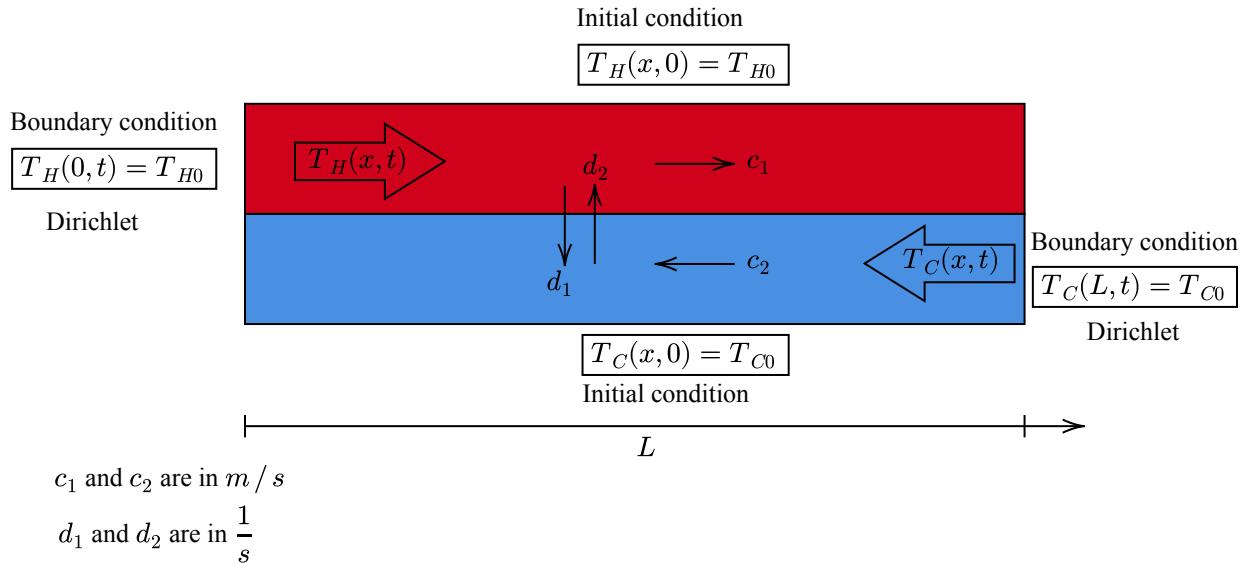
U is the overall heat transfer coefficient between the tube to its environment

ρ is the density of the liquid



Drawing 1: Flowing liquid in red container while transferring heat to its environment

A diagram of a simplest heat exchanger is presented in Drawing 2. From Drawing 2, we can see that hot liquid flows from left to right and cold liquid flows from right to left. Source temperature of the hot flow is kept at T_{H0} . While source temperature of the cold flow is kept at T_{C0} . Those two temperatures never change. This condition is called Dirichlet condition.



Drawing 2: Cross-current heat exchanger modeled with two PDEs

The system model of a heat exchanger based on Drawing 2 can be described as follows.

$$\begin{aligned}\frac{\partial}{\partial t}T_H(x, t) + c_1 \frac{\partial}{\partial x}T_H(x, t) &= -d_1(T_H(x, t) - T_C(x, t)) \\ \frac{\partial}{\partial t}T_C(x, t) - c_2 \frac{\partial}{\partial x}T_C(x, t) &= -d_2(T_C(x, t) - T_H(x, t))\end{aligned}\tag{28}$$

or

$$\begin{aligned}\frac{\partial}{\partial t}T_H(x, t) &= -c_1 \frac{\partial}{\partial x}T_H(x, t) - d_1(T_H(x, t) - T_C(x, t)) \\ \frac{\partial}{\partial t}T_C(x, t) &= c_2 \frac{\partial}{\partial x}T_C(x, t) + d_2(T_H(x, t) - T_C(x, t))\end{aligned}\tag{29}$$

where:

- d_1 and d_2 describe **convective** heat transfer **between** the two flowing liquids: **from** hot liquid **to** cold liquid and **from** cold liquid **to** hot liquid, respectively.
- c_1 and c_2 describes **advective** heat transfer **inside** the hot and cold liquid, respectively. This is analogous to

the actual movement of the liquids since the heat moves together with the liquids.

For the simulation, we will set the parameters according to the paper by Zobiri et.al.⁹ In their paper, it is mentioned that they use upwind-downwind method. However, we will solve the system above by using both the upwind-downwind method and the Lax-Wendroff method so that we can compare their results. The listing below shows the implementation of a cross-current heat exchanger as in (28) which runs for 20s with 0.01s step-time.

```

1  L = 1;
2  dx = 0.2;
3  x = 0:dx:L;
4
5  dt = 0.01;
6  T = 20;
7  t = 0:dt:T;
8
9  % Define the parameters
10 c1 = 0.001711593407;
11 c2 = -0.01785371429;
12 d1 = -0.03802197802;
13 d2 = 0.3954285714;
14
15 % =====
16 % Upwind-downwind method
17 % =====
18 TH = zeros(length(x), 1);
19 TC = zeros(length(x), 1);
20 f = zeros(length(x), 1);
21
22 % Intial condition
23 TH(1:length(x)) = 273+30;
24 TC(1:length(x)) = 273+10;
25
26 figure;
27 hold on
28 h1 = plot(0,0,'r');
29 h2 = plot(0,0,'b');
30 title('Cross-current heat exchanger');
31 ylim([270 320]);
32 xlabel('x');
33 ylabel('Temperature');
34 legend('Hot flow','Cold flow');
35
36 for k = 1 : length(t)
37     set(h1,'XData',x,'Ydata',TH);
38     set(h2,'XData',x,'Ydata',TC);
39     drawnow;
40
41     f = d1.* (TH-TC);
42     TH = upwind(TH, f, c1, dt, dx);
43     f = d2.* (TH-TC);
44     TC = downwind(TC, f, c2, dt, dx);
45 end
46
47 % =====
48 % Lax-Wnderoff
49 % =====
50 TH = zeros(length(x), 1);

```

⁹PDE Observer Design for Counter-Current Heat Flows in a Heat-Exchange by F. Zobiri, et. al., published at IFAC-PapersOnLine

```

51 TC = zeros(length(x), 1);
52 f = zeros(length(x), 1);
53
54 % Intial conditoin
55 TH(1:length(x)) = 273+30;
56 TC(1:length(x)) = 273+10;
57
58 figure;
59 hold on
60 h3 = plot(0,0,'r');
61 h4 = plot(0,0,'b');
62 title('Cross-current heat exchanger');
63 ylim([270 320]);
64 xlabel('x');
65 ylabel('Temperature');
66 legend('Hot flow','Cold flow');
67
68 for k = 1:length(t)
69     set(h3,'XData',x,'Ydata',TH);
70     set(h4,'XData',x,'Ydata',TC);
71     drawnow;
72
73     f = d1.* (TH-TC);
74     TH = lax_wendroff(TH, f, c1, dt, dx);
75     f = d2.* (TH-TC);
76     TC = lax_wendroff(TC, f, c2, dt, dx);
77 end

```

From line 41 to 44 of the listing above, we use the upwind method for the hot flow since the flow moves from left to right. As for the cold flow, we use the downwind method since the cold flow moves from right to left. As for the Lax-Wendroff method, there is only one function that we call (line 73 to 76). However, we still need to call that function twice, one for the cold flow, another one is for the hot flow. When executed, the provided listing above will generate animations of the temperature evolutions for both methods.

Figure 7 and Figure 8 below show the temperature distribution of the modeled heat exchanger after 20 seconds for two different methods. From Figure 8, we can see that Lax-Wendroff method generates flat curve in its middle area. Theoretically, this makes more sense when compared to the results acquired from the upwind-downwind method. However, if we think with a practical perspective, the upwind-and downwind method might be more desirable since a theoretically perfect advection might be impossible in a real world.

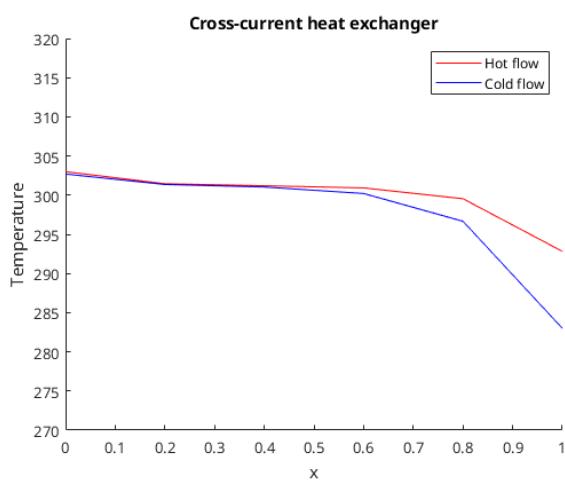


Figure 7: After 20 s, with upwind-downwind method

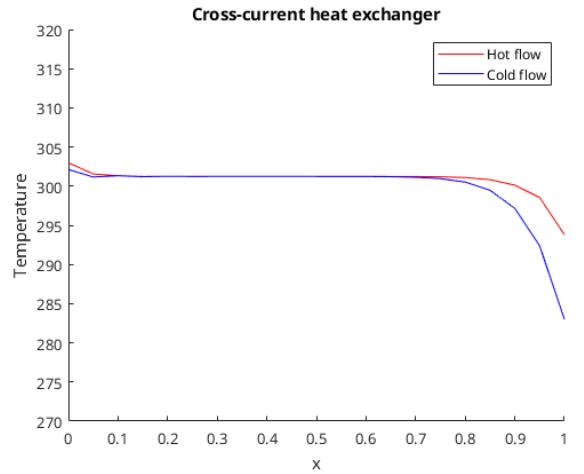
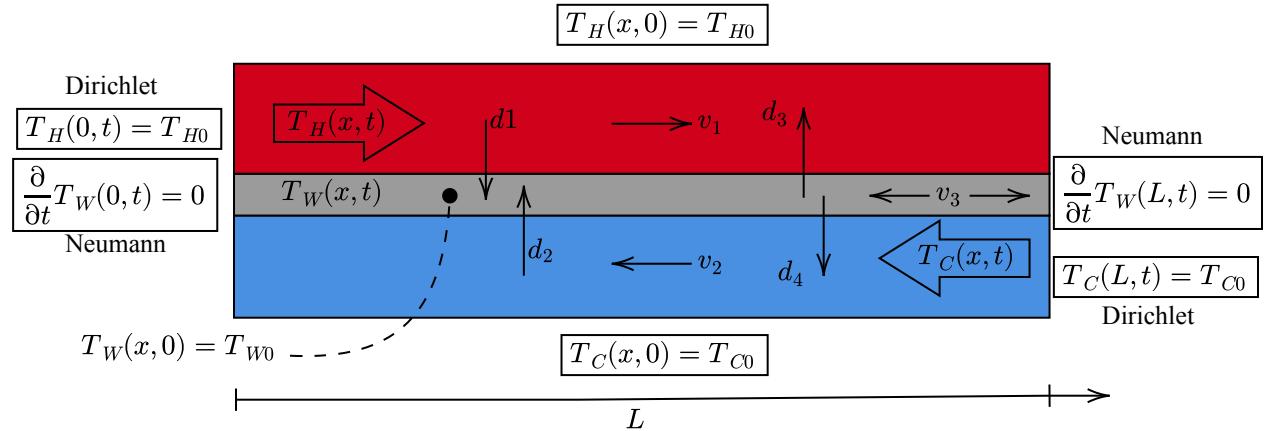


Figure 8: After 20 s, with Lax-Wendroff method

1.6.2. Three-Equation Model



Drawing 3: Cross-current heat exchanger modeled with three PDEs

In three-equation model (see Drawing 3), we consider the wall between the hot flow and the cold flow. In general, we can summarize how the heat are being exchanged as follows.

- d_1 and d_2 describe **convective** heat transfer **from** the hot liquid **to** the wall: from the hot liquid to the wall and from the cold liquid to the wall, respectively.
- d_3 and d_4 describe **convective** heat transfer **from** the wall **to** the liquid: from the wall to the hot liquid and from the wall to the cold liquid, respectively.
- v_1 and v_2 describes **advective** heat transfer **inside** the hot and cold liquid, respectively. This is analogous to the actual movement of the liquids since the heat moves together with the liquids.
- v_3 describes diffusive/conductive heat transfer **inside** the tube wall.

Therefore, the three-equation model of a cross-current heat exchanger can be expressed as follows.

$$\begin{aligned}
\frac{\partial}{\partial t} T_H(x, t) &= -v_1 \frac{\partial}{\partial x} T_H(x, t) - d_1(T_H(x, t) - T_W(x, t)) \\
\frac{\partial}{\partial t} T_C(x, t) &= v_2 \frac{\partial}{\partial x} T_C(x, t) + d_2(T_W(x, t) - T_C(x, t)) \\
\frac{\partial}{\partial t} T_W(x, t) &= v_3 \frac{\partial^2}{\partial x^2} T_W(x, t) + d_3(T_H(x, t) - T_W(x, t)) - d_4(T_W(x, t) - T_C(x, t))
\end{aligned} \tag{30}$$

Where¹⁰:

$v_1 = 0.6366197724$	$d_1 = 0.9792$
$v_2 = 0.1657863990$	$d_2 = 0.2986$
$v_3 = 0.00003585526318$	$d_3 = 2.3923$
	$d_4 = 2.8708$

The initial and boundary conditions are:

$$\begin{aligned}
T_H(0, t) &= 360 \text{ and } T_H(x, 0) = 360 \\
T_C(L, t) &= 300 \text{ and } T_C(x, 0) = 300 \\
T_W(x, 0) &= 330, \frac{\partial T_W(0, t)}{\partial x} = 0, \text{ and } \frac{\partial T_W(L, t)}{\partial x} = 0
\end{aligned}$$

In (30), the third equation is different than the first two equations. The third equation is a non-homogenous one-dimensional heat equation (diffusion) with Neumann boundary conditions. We will discuss this in the next section. Nevertheless, we will directly show the implementation of the three-equation model of a heat exchanger in MATLAB. For the first and second equations, we will use upwind and downwind method, respectively (see line 51 to 55). The rest of the code can be understood after reading the next section.

```

1 L = 1;
2 tF = 5;
3 Nx = 10;
4 Nt = 100;
5 dt = tF/Nt;
6 dx = L/Nx;
7
8 v1 = 0.6366197724;
9 v2 = 0.1657863990;
10 v3 = 0.00003585526318;
11 d1 = 0.9792;
12 d2 = 0.2986;
13 d3 = 2.3923;
14 d4 = 2.8708;
15
16 TH = zeros(1, Nx+1);
17 TC = zeros(1, Nx+1);
18 TW = zeros(1, Nx+1+2); % +2 phantom nodes
19 fH = zeros(1, Nx+1);
20 fC = zeros(1, Nx+1);
21 fW = zeros(1, Nx+1+2); % +2 phantom nodes
22
23 % Intial condition
24 TH(1:Nx+1) = 360;
25 TC(1:Nx+1) = 300;
26 TW(1:Nx+1+2) = 330;
27
28 h=figure;

```

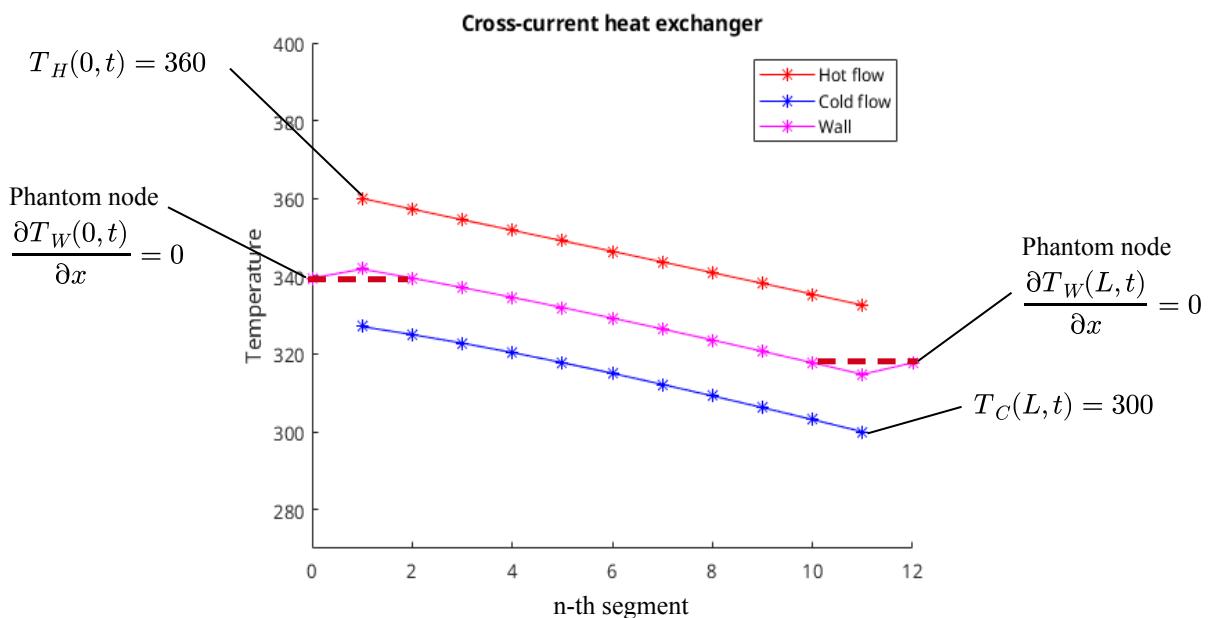
¹⁰ <https://www.maplesoft.com/applications/view.aspx?SID=119402&view=html>

```

29 hold on
30 h1 = plot(0,0,'r-*');
31 h2 = plot(0,0,'b-*');
32 h3 = plot(0,0,'m-*');
33 title('Cross-current heat exchanger')
34 ylim([270 400])
35 xlabel('x')
36 ylabel('Temperature')
37 legend('Hot flow', 'Cold flow', 'Wall')
38
39 xa = 1:Nx+1;
40 xb = 0:Nx+2;
41 for k = 1 : Nt
42     set(h1,'XData',xa,'Ydata',TH);
43     set(h2,'XData',xa,'Ydata',TC);
44     set(h3,'XData',xb,'Ydata',TW);
45     drawnow;
46
47 TW = apply_bc(TW, L/Nx, ["Neumann", "Neumann"], [0, 0]);
48
49 % TW(2:end-1) : it means we ignore phantom nodes
50
51 fH = -d1.* (TH-TW(2:end-1));
52 TH = upwind(TH, fH, v1, dt, dx);
53
54 fC = d2.* (TH-TW(2:end-1));
55 TC = downwind(TC, fC, -v2, dt, dx);
56
57 fW(2:end-1) = d3.* (TH-TW(2:end-1)) - d4.* (TW(2:end-1)-TC);
58 TW = diffusion_1d(TW, fW, sqrt(v3), dx, dt);
59 end

```

When executed for 20 seconds, temperature distributions of different parts of the heat exchanger are shown in Drawing 4.



Drawing 4: Temperature distributions after 5 seconds

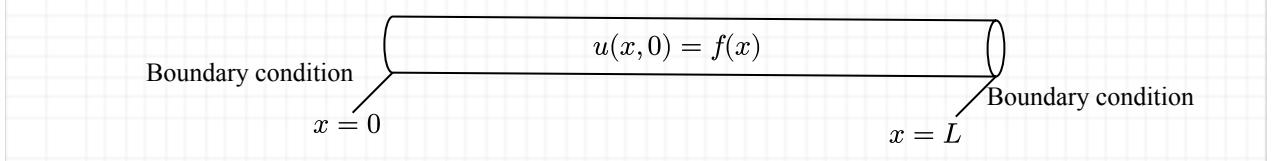
2. Case 2: One-Dimensional Heat Equation¹¹

The equation of temperature distribution (u) in a one-dimensional rod (see Drawing 5) over a certain distance (x) and time (t) is given by (31). α is the diffusivity coefficient. When $g(x, t) = 0$, the system is homogenous.

$$\begin{aligned} \alpha^2 u_{xx} - u_t &= g(x, t) \\ t \geq 0 \text{ and } 0 \leq x \leq L \end{aligned} \quad (31)$$

when at $t = 0$, the following initial condition is applied.

$$u(x, 0) = f(x)$$



Drawing 5: Temperature distribution (u) in a very thin rod

2.1. Explicit Method

With finite difference method, we can approach u_t and u_{xx} . To approximate u_t , we use the following approximation.

$$u_t \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \quad (32)$$

$$u_x \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} \quad (33)$$

While to approach u_{xx} , we use the following approximation.

$$\begin{aligned} u_{xx} &\approx \frac{u_x(x + \Delta x, t) - u_x(x, t)}{\Delta x} \\ &\approx \frac{\frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} - \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x}}{\Delta x} \\ &\approx \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{(\Delta x)^2} \end{aligned} \quad (34)$$

Substituting (32) and (33) back to (31) gives us the following equations.

$$\begin{aligned} u_t &\approx \alpha^2 u_{xx} + g(x, t) \\ \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} &\approx \alpha^2 \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2} + g(x, t) \\ u(x, t + \Delta t) - u(x, t) &\approx \frac{\alpha^2 \Delta t}{\Delta x^2} (u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)) + g(x, t) \Delta t \\ u(x, t + \Delta t) &\approx \frac{\alpha^2 \Delta t}{\Delta x^2} u(x + \Delta x, t) + \left(1 - 2 \frac{\alpha^2 \Delta t}{\Delta x^2}\right) u(x, t) + \frac{\alpha^2 \Delta t}{\Delta x^2} u(x - \Delta x, t) + g(x, t) \Delta t \\ &\approx r u(x + \Delta x, t) + (1 - 2r) u(x, t) + r u(x - \Delta x, t) + g(x, t) \Delta t \\ &\approx u(x, t) + r [u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)] \end{aligned}$$

where $r = \frac{\alpha^2 \Delta t}{(\Delta x)^2}$. Now, let us consider $U_{j,k} = u(x_j, t_k)$, $G_{j,k} = g(x_j, t_k)$, $x_j = j\Delta x$, $t_k = k\Delta t$, $U_{j,k+1} = u(x_j, t_k + \Delta t)$ and $U_{j-1,k} = u(x_j - \Delta x, t_k)$, we can then have the following shorter terms.

$$U_{j,k+1} = U_{j,k} + r [U_{j+1,k} - 2U_{j,k} + U_{j-1,k}] + G_{j,k} \Delta t \quad (35)$$

2.1.1. Dirichlet Boundary Conditions

In (31), certain conditions are applied and kept true at all time. These conditions are called boundary conditions. There are several types of boundary conditions. One of them is Dirichlet boundary condition. A typical Dirichlet boundary condition would be

$$\begin{aligned} u(0, t) &= A \\ u(L, t) &= B \end{aligned}$$

where A and B are real numbers. The implementation of the Dirichlet boundary condition is very straightforward by directly controlling values at the two end nodes.

2.1.2. Neumann Boundary Condition

Another type of boundary condition is the Neumann boundary condition. In Neumann boundary condition, the derivative of the solution function ($\partial u(x, t) / \partial x$ or $\partial u(x, t) / \partial t$) is kept constant. However, here we will only implement the derivative of the solution with respect to x .

$$\frac{\partial u(x, t)}{\partial x} = C \quad (36)$$

where C is a real number. At the boundary when $x = 0$, we can write

$$\frac{u(\Delta x, t) - u(-\Delta x, t)}{2\Delta x} = C \quad (37)$$

therefore

$$u(-\Delta x, t) = u(\Delta x, t) - 2\Delta x C \quad (38)$$

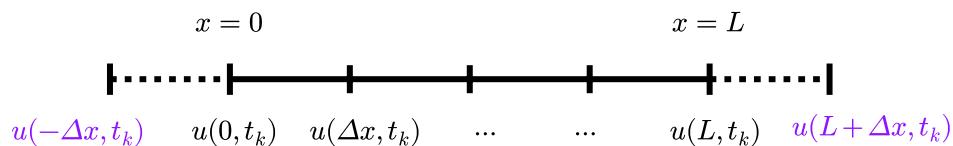
At another boundary when $x = L$

$$\frac{u(L + \Delta x, t) - u(L - \Delta x, t)}{2\Delta x} = D \quad (39)$$

where D is a real number. Therefore

$$u(L + \Delta x, t) = u(L - \Delta x, t) + 2\Delta x D \quad (40)$$

The implementation of Neumann boundary condition will be done by adding additional phantom nodes as shown in Drawing 6.



Drawing 6: Extra phantom nodes for Neumann boundary condition

2.1.3. MATLAB Implementation

MATLAB implementation of (35) is as follows.

```

1 % alpha^2 * u_xx = u_tt
2 % Divide 0<=x<=L with Delta_x increments
3 % Divide 0<=t<=tf with Delta_t increments
4
5 function u_array = diffusion_1d(u_array, g_array, alpha, Delta_x, Delta_t)
6
7 r = alpha^2 * Delta_t / Delta_x^2;

```

```

8
9  j = 2 : length(u_array)-1;
10 u_array(j) = ...
11     u_array(j) + r*(u_array(j+1) - 2 * u_array(j) + u_array(j-1)) + ...
12     g_array(j)*Delta_t;
13 end

```

The implementation above is already vectorized. In line 9 of the listing above, we can see that the diffusion is not applied to the first and last nodes. At those two end-nodes, we will apply the boundary conditions. Therefore, we also create two more functions for the initial and boundary conditions (see the listing below).

```

1 % Initial condition
2 function u_array = apply_ic(u_array, f, Delta_x)
3
4 % Apply the initial condition U(x,0)=f(x)
5 j = 1:length(u_array);
6 u_array(j) = f((j-1)*Delta_x);
7
8 end
9
10 %=====
11
12 % Boundary condition
13 function u_array = apply_bc(u_array, Delta_x, types, values)
14
15 % left-end
16 if lower(types(1)) == "dirichlet"
17     u_array(1) = values(1);
18 elseif lower(types(1)) == "neumann"
19     u_array(1) = u_array(3) - 2*Delta_x * values(1);
20 end
21
22 % right-end
23 if lower(types(2)) == "dirichlet"
24     u_array(end) = values(2);
25 elseif lower(types(2)) == "neumann"
26     u_array(end) = u_array(end-2) + 2 * Delta_x * values(2);
27 end
28 end

```

In the Neumann boundary condition, we need to add additional phantom nodes. There are some examples provided in the next section to understand how to use the aforementioned three functions.

2.1.4. Example 1

Given the following heat transfer equation:

$$1.14u_{xx} - u_t = 0, \quad 0 \leq x \leq 10 \quad (41)$$

with the following boundary and initial conditions:

$$\begin{aligned} u(0, t) &= u(10, t) = 0 \\ u(x, 0) &= f(x) = \begin{cases} 60, & 0 \leq x \leq 5 \\ 0, & 5 < x \leq 10 \end{cases} \end{aligned} \quad (42)$$

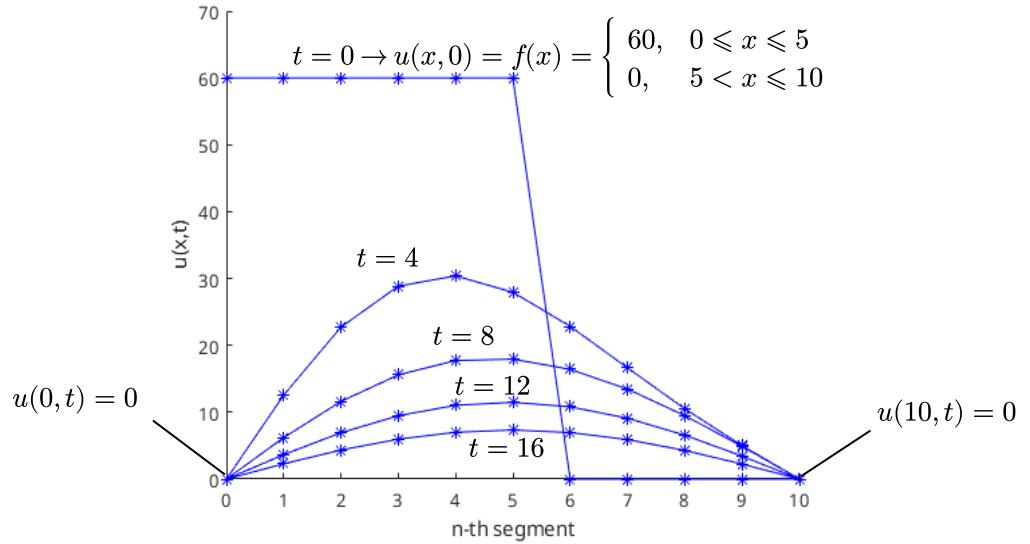
MATLAB implementation of (41) is as follows.

```

1 L = 10;
2 alpha = sqrt(1.14);
3
4 tF = 20;
5 Nx = 10;
6 Nt = 1000;
7
8 u_array = zeros(1,Nx+1); % +1 because we start from 0 to Nx
9 g_array = zeros(1,Nx+1); % homogenous
10 u_array = apply_ic(u_array,@f, L/Nx);
11
12 figure
13 hold on
14 h = plot(0,0,'-*');
15 ylim([0 70])
16
17 x = 0 : Nx;
18
19 for k = 1 : Nt
20     set(h, 'XData', x, 'YData', u_array)
21     drawnow;
22
23     u_array = apply_bc(u_array, L/Nx, ["Dirichlet", "Dirichlet"], [0, 0]);
24     u_array = diffusion_1d(u_array, g_array, alpha, L/Nx, tF/Nt);
25 end
26 xlabel('n-th segment')
27 ylabel('u(x,t)')
28
29 %% Intial Condition
30 function u=f(x)
31 L = 10;
32 u = zeros(1,length(x));
33
34 for k = 1 : length(x)
35     if x(k) >= 0 && x(k) <=L/2
36         u(k) = 60;
37     elseif x(k) > L/2 && x(k) <= L
38         u(k) = 0;
39     end
40 end
41
42 end

```

The result is shown in Drawing 7 below.



Drawing 7: Simulation result: Dirichlet-Dirichlet

2.1.5. Example 2

Given the following heat transfer equation:

$$u_{xx} - u_t = 0, \quad 0 \leq x \leq \pi \quad (43)$$

with the following boundary and initial conditions:

$$\begin{aligned} u(0,t) &= 20 \\ u_x(\pi,t) &= 3 \\ u(x,0) &= 0 \end{aligned} \quad (44)$$

MATLAB implementation of (43) is as follows.

```

1 L = pi;
2 alpha = 1;
3
4 tF = 10;
5 Nx = 10;
6 Nt = 1000;
7
8 u_array = zeros(1,Nx+1+1); % +1 -> start from 0 to Nx
9 % another +1 -> add one phantom nodes
10 g_array = zeros(1,length(u_array));
11 u_array = apply_ic(u_array,@f, L/Nx);
12
13 figure
14 hold on
15 h = plot(0,0,'-*');
16 ylim([0 32])
17
18 x = 0 : Nx+1;
19
20 for k = 1 : Nt
21     set(h, 'XData', x, 'YData', u_array)
22     drawnow;
23
24 u_array = apply_bc(u_array, L/Nx, ["Dirichlet", "Neumann"], [20, 3]);

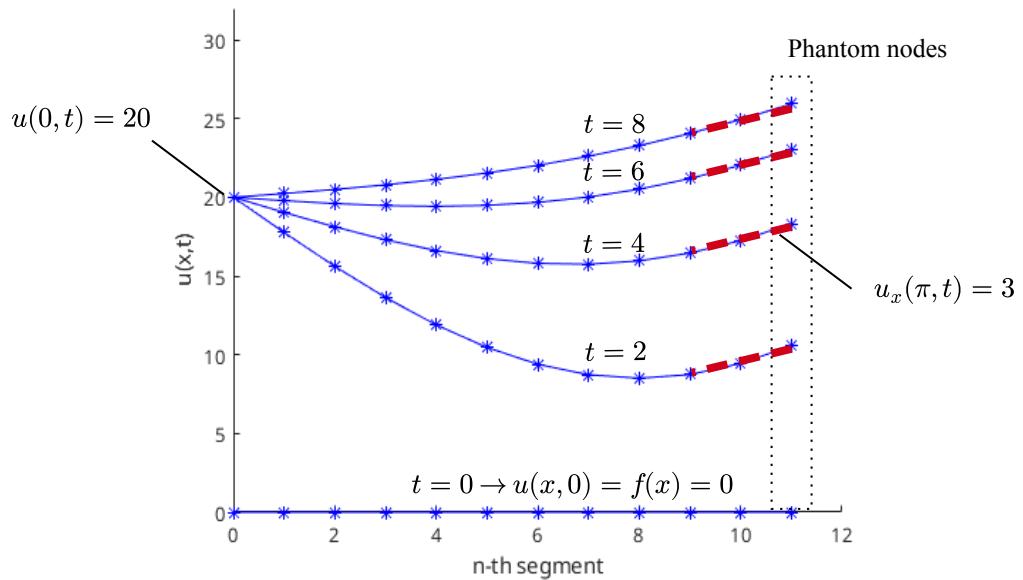
```

```

25     u_array = diffusion_1d(u_array, g_array, alpha, L/Nx, tF/Nt);
26 end
27 xlabel('n-th segment')
28 ylabel('u(x,t)')
29
30 % Intial Condition
31 function u=f(x)
32     u = zeros(1,length(x));
33 end

```

The result is shown in Drawing 8 below.



Drawing 8: Simulation result: Dirichlet-Neumann

2.1.6. Example 3

Given the following heat equation:

$$u_{xx} - u_t = 0, \quad 0 \leq x \leq 1 \quad (45)$$

with the following initial and boundary conditions:

$$\begin{aligned} u_x(0, t) &= 0 \\ u(1, t) &= 100 \\ u(x, 0) = f(x) &= \begin{cases} 60, & 0 \leq x \leq 1/2 \\ 0, & 1/2 < x \leq 1 \end{cases} \end{aligned} \quad (46)$$

MATLAB implementation of (45) is as follows.

```

1 L = 1;
2 alpha = 1;
3
4 tF = 1;
5 Nx = 10;
6 Nt = 1000;
7
8 u_array = zeros(1,Nx+1+1); % +1 because we start from 0 to Nx
9 % another +1 -> add one phantom nodes
10 g_array = zeros(1,length(u_array));

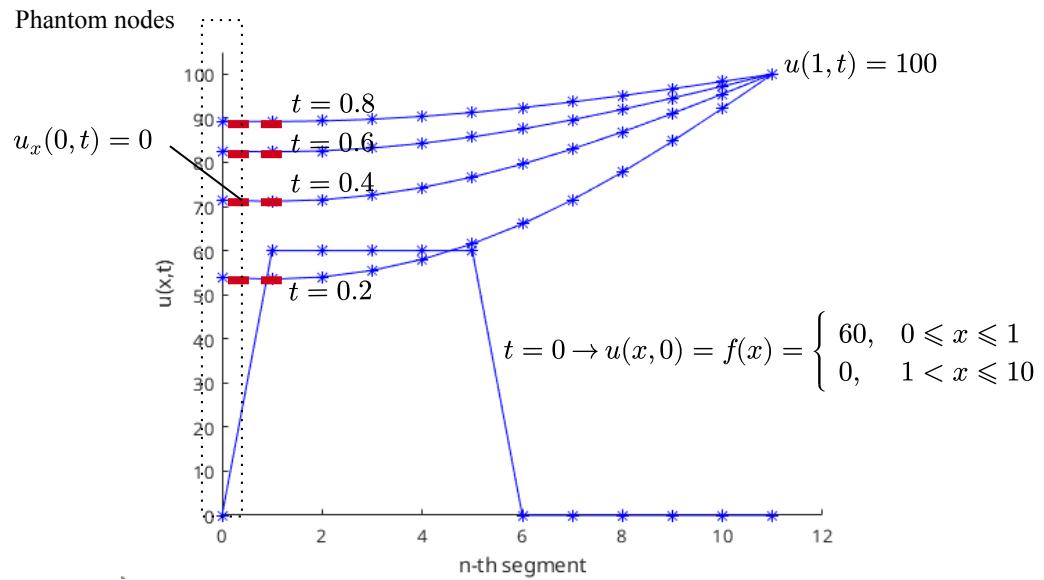
```

```

11 u_array = apply_ic(u_array,@f, L/Nx);
12
13 hf = figure;
14 hold on
15 h = plot(0,0,'-*');
16 ylim([0 105])
17 xlabel('n-th segment')
18 ylabel('u(x,t)')
19
20 x = 0 : Nx+1;
21
22 for k = 1 : Nt
23     set(h, 'XData', x, 'YData', u_array)
24     drawnow;
25
26     u_array = apply_bc(u_array, L/Nx, ["Neumann", "Dirichlet"], [0, 100]);
27     u_array = diffusion_1d(u_array, g_array, alpha, L/Nx, tF/Nt);
28 end
29
30 %% Initial condition
31 function u=f(x)
32 L = 1;
33 u = zeros(1,length(x));
34
35 for k = 1 : length(x)
36     if x(k) >= 0 && x(k) <=L/2
37         u(k) = 60;
38     elseif x(k) > L/2 && x(k) <= L
39         u(k) = 0;
40     end
41 end
42
43 end

```

The result is shown in Drawing 9 below.



Drawing 9: Simulation result: Neumann-Dirichlet

2.2. Implicit Method

The main idea of an implicit method is to calculate u_{xx} not only at the current time, but also at the time one step ahead. Let us consider:

$$u_{xx}(x, t) = (1 - \theta)u_{xx}(x, t) + \theta u_{xx}(x, t + \Delta t), \quad 0 \leq \theta \leq 1 \quad (47)$$

where θ acts as a weighting factor. When $\theta = 0$, (47) becomes similar to (35). What we currently have in this case is an explicit method. On the other hand, when $\theta = 1$, in this case, what we have is an implicit method. However, when $\theta = 0.5$, the method is called Crank-Nicolson method. In this section, let us focus on mathematical derivations of the Crank-Nicolson method.

Expanding (47) by first setting $\theta = \frac{1}{2}$ gives us:

$$\begin{aligned} u_{xx} &= \frac{1}{2} \frac{u_x(x + \Delta x, t) - u_x(x, t)}{\Delta x} + \frac{1}{2} \frac{u_x(x + \Delta x, t + \Delta t) - u_x(x, t + \Delta t)}{\Delta x} \\ &= \frac{\frac{u(x + \Delta x, t) - u(x, t)}{\Delta x} - \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x}}{2\Delta x} + \frac{\frac{u(x + \Delta x, t + \Delta t) - u(x, t + \Delta t)}{\Delta x} - \frac{u(x, t + \Delta t) - u(x - \Delta x, t + \Delta t)}{\Delta x}}{2\Delta x} \\ &= \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{2\Delta x^2} \\ &\quad + \frac{u(x + \Delta x, t + \Delta t) - 2u(x, t + \Delta t) + u(x - \Delta x, t + \Delta t)}{2\Delta x^2} \end{aligned}$$

which can be shortened as

$$u_{xx} = \frac{U_{j+1,k} - 2U_{j,k} + U_{j-1,k} + U_{j+1,k+1} - 2U_{j,k+1} + U_{j-1,k+1}}{2\Delta x^2} \quad (48)$$

Substituting (48) back to: $u_t = \alpha^2 u_{xx}$ gives us:

$$\begin{aligned} \underbrace{\frac{U_{j,k+1} - U_{j,k}}{\Delta t}}_{u_t} &= \alpha^2 \underbrace{\frac{U_{j+1,k} - 2U_{j,k} + U_{j-1,k} + U_{j+1,k+1} - 2U_{j,k+1} + U_{j-1,k+1}}{2\Delta x^2}}_{u_{xx}} \\ U_{j,k+1} - U_{j,k} &= \frac{1}{2} \frac{\alpha^2 \Delta t}{\Delta x^2} (U_{j+1,k} - 2U_{j,k} + U_{j-1,k} + U_{j+1,k+1} - 2U_{j,k+1} + U_{j-1,k+1}) \end{aligned} \quad (49)$$

Since $r = \frac{\alpha^2 \Delta t}{\Delta x^2}$, (49) can then be simplified as:

$$\begin{aligned} U_{j,k+1} - U_{j,k} &= \frac{r}{2} (U_{j+1,k} - 2U_{j,k} + U_{j-1,k} + U_{j+1,k+1} - 2U_{j,k+1} + U_{j-1,k+1}) \\ U_{j,k+1} + rU_{j,k+1} &= \frac{r}{2} U_{j+1,k} + (1-r)U_{j,k} + \frac{r}{2} U_{j-1,k} + \frac{r}{2} U_{j+1,k+1} + \frac{r}{2} U_{j-1,k+1} \\ 2(1+r)U_{j,k+1} &= rU_{j+1,k} + 2(1-r)U_{j,k} + rU_{j-1,k} + rU_{j+1,k+1} + rU_{j-1,k+1} \\ -rU_{j-1,k+1} + 2(1+r)U_{j,k+1} - rU_{j+1,k+1} &= rU_{j-1,k} + 2(1-r)U_{j,k} + rU_{j+1,k} \end{aligned} \quad (50)$$

Thus, the well-known Crank-Nicolson scheme can be expressed as follows.

$$\boxed{-rU_{j-1,k+1} + 2(1+r)U_{j,k+1} - rU_{j+1,k+1} = rU_{j-1,k} + 2(1-r)U_{j,k} + rU_{j+1,k}} \quad (51)$$

or in matrix form:

$$\underbrace{\begin{bmatrix} -r & 2(1+r) & -r & & \cdots & 0 \\ 0 & -r & 2(1+r) & -r & & \vdots \\ & & -r & 2(1+r) & -r & \\ & & & \ddots & & \\ \vdots & & & & -r & 2(1+r) \\ 0 & \cdots & & & 0 & -r \end{bmatrix}}_A \underbrace{\begin{bmatrix} p((k+1)\Delta t) \\ U_{1,k+1} \\ U_{2,k+1} \\ \vdots \\ U_{N-2,k+1} \\ q((k+1)\Delta t) \end{bmatrix}}_U = \\
\underbrace{\begin{bmatrix} r & 2(1-r) & r & & \cdots & 0 \\ 0 & r & 2(1-r) & r & & \vdots \\ & & r & 2(1-r) & r & \\ & & & \ddots & & \\ \vdots & & & & r & 2(1-r) \\ 0 & \cdots & & & 0 & r \end{bmatrix}}_B \underbrace{\begin{bmatrix} p(k\Delta t) \\ U_{1,k} \\ U_{2,k} \\ \vdots \\ U_{N-2,k} \\ q(k\Delta t) \end{bmatrix}}_U \quad (52)$$

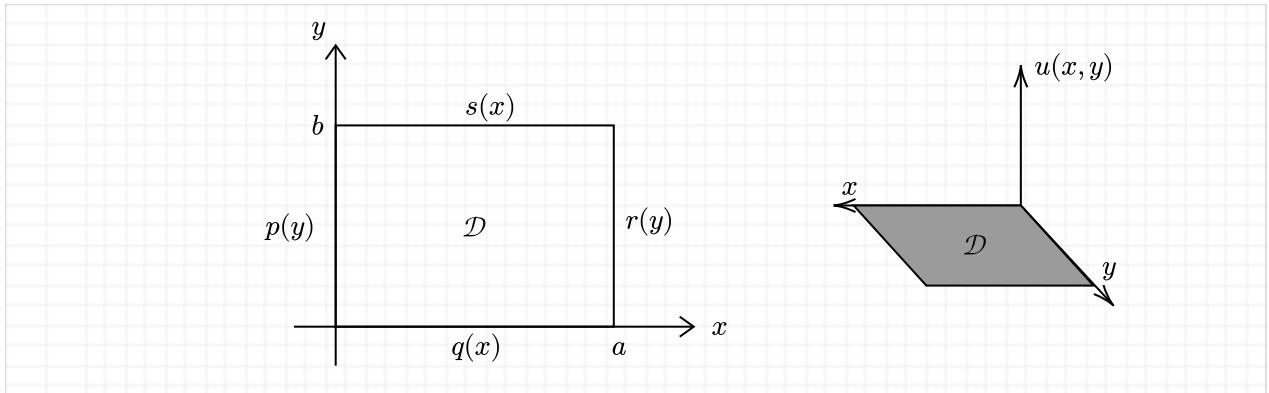
Red means the values are known. In (52), we can notice that some known values reside in U . We must remove these known values from U and put them to the right side. The goal is to have a perfect $AU = B$ system where all the unknowns are all in U .

$$\begin{aligned}
&\underbrace{\begin{bmatrix} 2(1+r) & -r & & \cdots & 0 \\ -r & 2(1+r) & -r & & \vdots \\ & & \ddots & & \\ & & -r & 2(1+r) & -r \\ & \cdots & & -r & 2(1+r) \end{bmatrix}}_A \underbrace{\begin{bmatrix} U_{1,k+1} \\ U_{2,k+1} \\ \vdots \\ U_{N-2,k+1} \\ U_{N-1,k+1} \end{bmatrix}}_{U_{new}} = \\
&\underbrace{\begin{bmatrix} 2(1-r) & r & & \cdots & \cdots \\ r & 2(1-r) & r & & \\ & & \ddots & & \\ & & r & 2(1-r) & r \\ & \cdots & & r & 2(1-r) \end{bmatrix}}_B \underbrace{\begin{bmatrix} U_{1,k} \\ U_{2,k} \\ \vdots \\ U_{N-2,k} \\ U_{N-1,k} \end{bmatrix}}_{U_{now}} + \underbrace{\begin{bmatrix} rp((k+1)\Delta t) + rp(k\Delta t) \\ 0 \\ \vdots \\ 0 \\ rq((k+1)\Delta t) + rq(k\Delta t) \end{bmatrix}}_C \quad (53)
\end{aligned}$$

Now, we can find $U_{new} = A^{-1}B$. In real implementation, this might not be plausible and we have to perform an iteration-based algorithm to find the optimal U_{new} . In Maple, we can use the $U_{new} = \text{LinearSolve}(A,B)$ command. In MATLAB, we can use the popular $U_{new} = A \setminus B$ command. Other alternatives are using the Jacobi method and the Gauss-Siedel method as explained the Greenberg's book.

3. Case 3: Rectangular Two-Dimensional Heat Equation¹²

3.1. The Five-Point Difference Method



Drawing 10: Heat distribution on a two-dimensional plane

Heat transfer on a two-dimensional plane is described as follows:

Finite difference method for the system above is as follows.

4. Case 4: Transient Heat Equation

Let us consider the flowing PDE:

$$\frac{\partial T}{\partial t} + \underbrace{u \frac{\partial T}{\partial x} + v \frac{\partial T}{\partial y}}_{\text{2D Advection}} = \alpha \left(\underbrace{\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}}_{\text{2D Diffusion}} \right) + f \quad (54)$$

(54) is the general transient heat equation in 2D space. It is composed of advection process and diffusion process. α is defines as:

$$\alpha = \frac{\mathcal{K}}{\rho c_p} \quad (55)$$

where:

c_p is the specific heat at constant ($\text{J}(\text{kg K})$)

ρ is the fluid density (kg / m^3)

\mathcal{K} is the thermal conductivity ($\text{W}(\text{m K})$)

f is the additional forcing component

In order to discretize (54), we will use the simplest methods, which are the upwind/downwind method for the advection and the explicit FTCS method for the diffusion.

4.1. Using the UPWIND Method

The upwind method is used when both velocities are positive or zero. We start by discretizing (54) as follows:

$$v(x, y, t) \frac{T(x, y, t) - T(x, y - \Delta y, t)}{\Delta y} = \alpha \left(\frac{T(x + \Delta x, y, t) - 2T(x, y, z) + T(x - \Delta x, y, t)}{\Delta x^2} \right) + \dots \quad (56)$$

$$\alpha \left(\frac{T(x, y + \Delta y, t) - 2T(x, y, z) + T(x, y - \Delta y, t)}{\Delta y^2} \right) + f(x, y, t)$$

Rearranging (56) give us:

$$T(x, y, t + \Delta t) = \frac{\alpha \Delta t}{\Delta x^2} (T(x + \Delta x, y, t) - 2T(x, y, z) + T(x - \Delta x, y, t)) + \frac{\alpha \Delta t}{\Delta y^2} (T(x, y + \Delta y, t) - 2T(x, y, z) + T(x, y - \Delta y, t)) - \frac{u(x, y, t) \Delta t}{\Delta x} (T(x, y, t) - T(x - \Delta x, y, t)) - \frac{v(x, y, t) \Delta t}{\Delta y} (T(x, y, t) - T(x, y - \Delta y, t)) + T(x, y, t) + f(x, y, t) \Delta t \quad (57)$$

Taking: $T_{i,j,k} = T(x_i, y_j, t_k)$, $x_j = j \Delta x$, $y_k = k \Delta y$, and $t_k = k \Delta t$, we can rewrite (57) as follows:

$$T_{i,j,k+1} = \alpha \Delta t \left(\left(\frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{\Delta x^2} \right) + \left(\frac{T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}}{\Delta y^2} \right) \right) - \Delta t \left(\frac{u_{i,j,k}}{\Delta x} (T_{i,j,k} - T_{i-1,j,k}) + \frac{v_{i,j,k}}{\Delta y} (T_{i,j,k} - T_{i,j-1,k}) \right) + T_{i,j,k} + \Delta t f_{i,j,k} \quad (58)$$

4.2. Using the DOWNDOWN Method

The upwind method is used when both velocities are negative or zero. The discretization process is similar to the upwind method. The only difference is in the advection part. The discretization is done as follows:

$$\begin{aligned} \frac{T(x, y, t + \Delta t) - T(x, y, t)}{\Delta t} + u(x, y, t) \frac{T(x + \Delta x, y, t) - T(x, y, t)}{\Delta x} + \dots \\ v(x, y, t) \frac{T(x, y + \Delta y, t) - T(x, y, t)}{\Delta y} = \alpha \left(\frac{T(x + \Delta x, y, t) - 2T(x, y, z) + T(x - \Delta x, y, t)}{\Delta x^2} \right) + \dots \quad (59) \\ \alpha \left(\frac{T(x, y + \Delta y, t) - 2T(x, y, z) + T(x, y - \Delta y, t)}{\Delta y^2} \right) + f(x, y, t) \end{aligned}$$

Rearranging (59) give us:

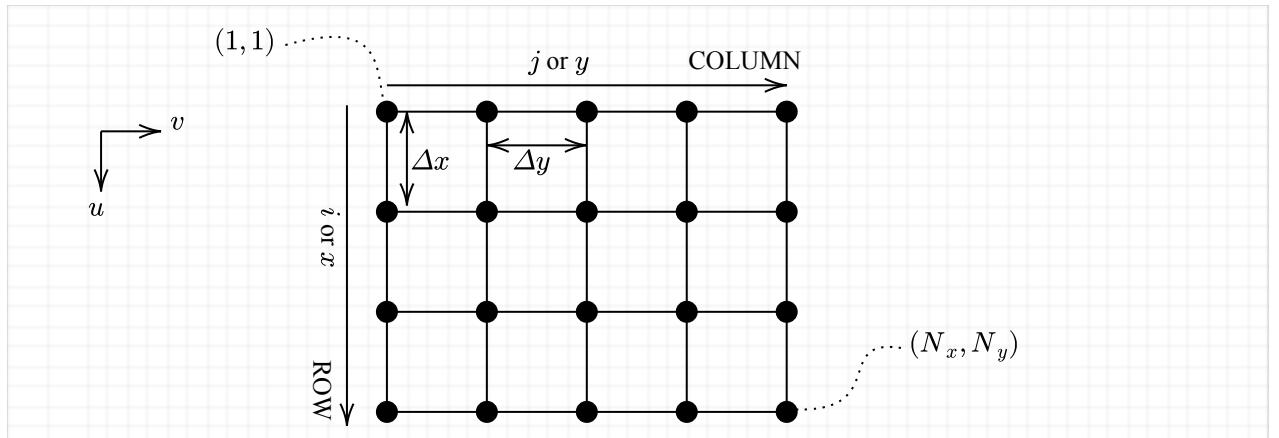
$$\begin{aligned} T(x, y, t + \Delta t) = \frac{\alpha \Delta t}{\Delta x^2} (T(x + \Delta x, y, t) - 2T(x, y, z) + T(x - \Delta x, y, t)) + \\ \frac{\alpha \Delta t}{\Delta y^2} (T(x, y + \Delta y, t) - 2T(x, y, z) + T(x, y - \Delta y, t)) - \quad (60) \\ \frac{u(x, y, t) \Delta t}{\Delta x} (T(x + \Delta x, y, t) - T(x, y, t)) - \\ \frac{v(x, y, t) \Delta t}{\Delta y} (T(x, y + \Delta y, t) - T(x, y, t)) + T(x, y, t) + f(x, y, t) \Delta t \end{aligned}$$

Taking: $T_{i,j,k} = T(x_i, y_j, t_k)$, $x_j = j\Delta x$, $y_k = k\Delta y$, and $t_k = k\Delta t$, we can rewrite (60) as follows:

$$T_{i,j,k+1} = \alpha \Delta t \left(\left(\frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{\Delta x^2} \right) + \left(\frac{T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}}{\Delta y^2} \right) \right) - \\ \Delta t \left(\frac{u_{i,j,k}}{\Delta x} (T_{i+1,j,k} - T_{i,j,k}) + \frac{v_{i,j,k}}{\Delta y} (T_{i,j+1,k} - T_{i,j,k}) \right) + T_{i,j,k} + \\ \Delta t f_{i,j,k} \quad (61)$$

4.3. MATLAB Implementation

In MATLAB, matrix element of M at row i and column j is defined as: $M_{ij} = M(i, j)$. Therefore, here, our x -axis is the vertical axis and our y -axis is the horizontal axis (see Drawing 11).



Drawing 11: A discretized two-dimensional plane

Implementations of (58) and (61) in MATLAB are as follows:

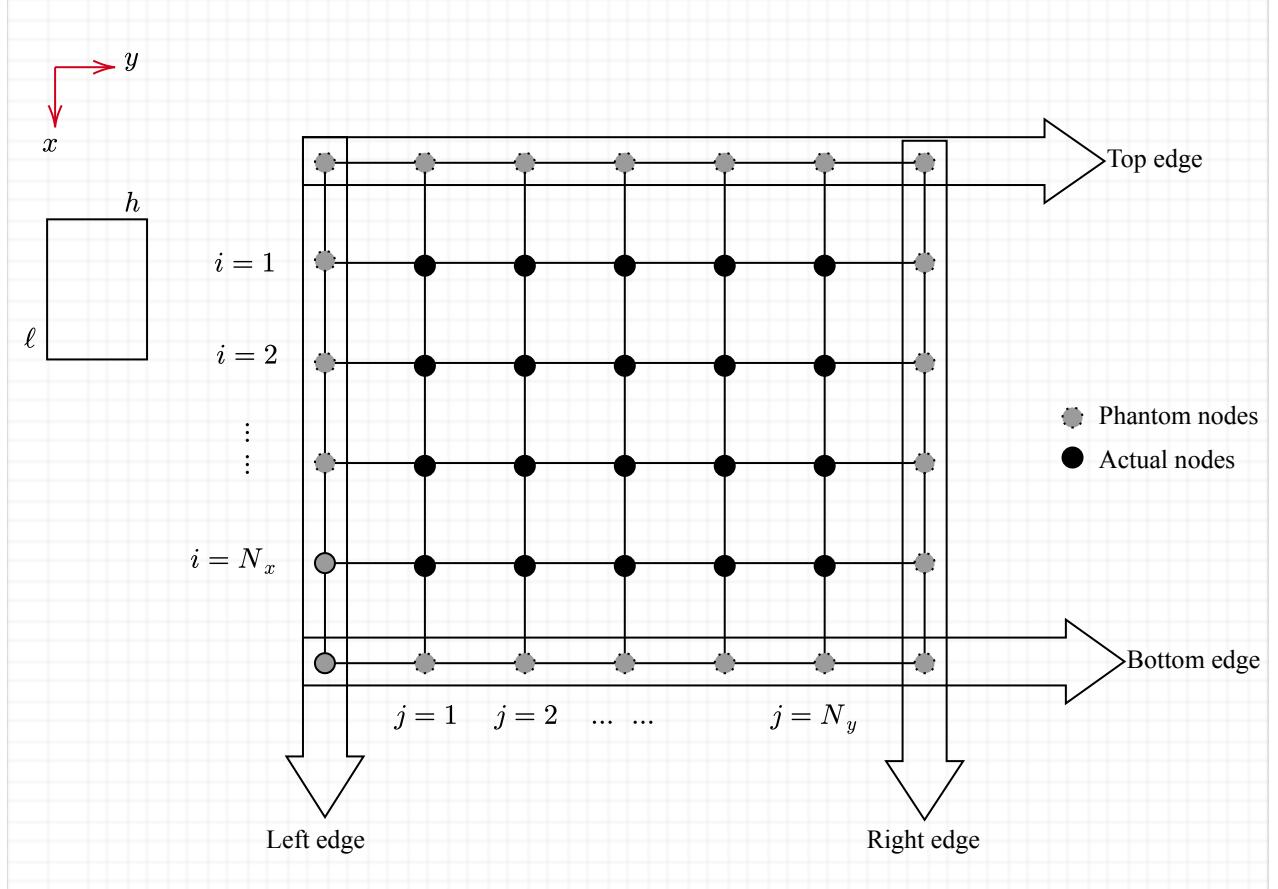
```

1  function T = advection_diffusion_downwind(T, ...
2                               u, ...
3                               v, ...
4                               dt, ...
5                               dx, ...
6                               dy, ...
7                               alpha)
8  r = size(T,1);
9  c = size(T,2);
10
11 for i = 2:r-1
12     for j = 2:c-1
13         T(i,j) = alpha*dt * ( (T(i+1,j)-2*T(i,j)+T(i-1,j))/dx^2 + ...
14                               (T(i,j+1)-2*T(i,j)+T(i,j-1))/dy^2 ) - ...
15                               dt * (u(i,j)/dx * (T(i+1,j)-T(i,j)) + ...
16                               v(i,j)/dy * (T(i,j+1)-T(i,j)) ) + ...
17                               T(i,j);
18     end
19 end
20
21 end
22 % =====
23 function T = advection_diffusion_upwind(T, ...
24                               u, ...
25                               v, ...
26                               dt, ...
27                               dx, ...
28                               dy, ...
29                               alpha)
30 r = size(T,1);
31 c = size(T,2);
32
33 for i = 2:r-1
34     for j = 2:c-1
35         T(i,j) = alpha*dt * ( (T(i+1,j)-2*T(i,j)+T(i-1,j))/dx^2 + ...
36                               (T(i,j+1)-2*T(i,j)+T(i,j-1))/dy^2 ) - ...
37                               dt * (u(i,j)/dx * (T(i,j)-T(i-1,j)) - ...
38                               v(i,j)/dy * (T(i,j)-T(i,j-1)) ) + ...
39                               T(i,j);
40     end
41 end
42 end

```

In the code listing above, we can see that the implemented functions never update the boundary. Thus, we can conveniently set the boundary condition once for the Dirichlet boundary condition. However, for the Neumann boundary condition, we need to create phantom nodes.

4.4. Neumann Boundary Condition



Drawing 12: A discretized two-dimensional plane

4.4.1. Left Edge

Assuming that boundary condition for the left edge is $\frac{\partial T(x, 0, t)}{\partial y} = A$. thus we have:

$$\frac{T(i, 2) - T(i, 0)}{2\Delta y} = A \quad (62)$$

Hence:

$$T(i, 0) = T(i, 2) - 2\Delta y A \quad (63)$$

4.4.2. Right Edge

Assuming that boundary condition for the left edge is $\frac{\partial T(x, h, t)}{\partial y} = B$. thus we have:

$$\frac{T(i, N_y + 1) - T(i, N_y - 1)}{2\Delta y} = B \quad (64)$$

Hence

$$T(i, N_y + 1) = T(i, N_y - 1) + 2\Delta y B \quad (65)$$

4.4.3. Top Edge

Assuming that boundary condition for the left edge is $\frac{\partial T(0, y, t)}{\partial y} = C$. thus we have:

$$\frac{T(2, j) - T(0, j)}{2\Delta x} = C \quad (66)$$

Hence

$$[T(0, j) = T(2, j) - 2\Delta x C] \quad (67)$$

4.4.4. Bottom Edge

Assuming that boundary condition for the left edge is $\frac{\partial T(\ell, y, t)}{\partial y} = D$, thus we have:

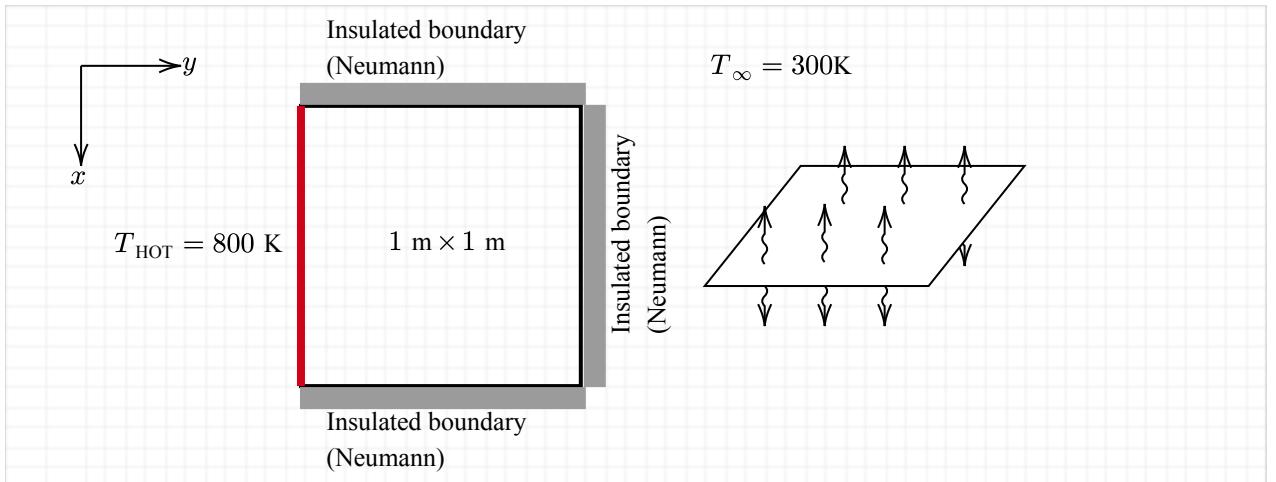
$$\frac{T(N_x + 1, j) - T(N_x - 1, j)}{2\Delta x} = D \quad (68)$$

Hence

$$[T(N_x + 1, j) = T(N_x - 1, j) + 2\Delta x D] \quad (69)$$

4.5. Energy Balance Model: Out-of-Plane Heat Transfer for a Thin Plate

Up to now, we only deal with homogenous equation. However, when the system interacts with its environment or with other system, the governing equations become non-homogenous. To demonstrate this, we will implement the following scenario in MATLAB and compare the result with COMSOL. This scenario is taken from the COMSOL Application Gallery¹³. A very thin plate ($1 \text{ m} \times 1 \text{ m}$, and thickness of 0.001 m) is heated only at the left edge. The temperature applied heat is 800 K . Other edges are thermally insulated. Heat transfer to the external environment only happens from the two surfaces of the plate.



The governing heat equation of the system above is as follows:

$$\begin{aligned} d_z \rho c_p \frac{\partial T}{\partial t} &= d_z \mathcal{K} \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) - h_u(T - T_{\infty}) - h_d(T - T_{\infty}) \\ &\quad - \varepsilon_u \sigma (T^4 - T_{\infty}^4) - \varepsilon_d \sigma (T^4 - T_{\infty}^4) \end{aligned} \quad (70)$$

where

$$d_z = 0.01 \text{ m}$$

$$h_u = h_d = h = 10 \text{ W / (m}^2\text{K)}$$

$$\varepsilon_u = \varepsilon_d = \varepsilon = 0.5$$

$$\sigma = 5.67 \times 10^{-8} \text{ W / (m}^2\text{K}^4)$$

The material is copper where:

¹³ <https://www.comsol.com/model/out-of-plane-heat-transfer-for-a-thin-plate-493>

$$c_p = 385 \text{ J / (kg K)}$$

$$\mathcal{K} = 400 \text{ W / (m K)}$$

$$\rho = 8960 \text{ kg / m}^3$$

We can simplify (70) as follows:

$$\frac{\partial T}{\partial t} = \frac{\mathcal{K}}{\rho c_p} \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) - \frac{2h}{d_z \rho c_p} (T - T_\infty) - \frac{2\varepsilon\sigma}{d_z \rho c_p} (T^4 - T_\infty^4) \quad (71)$$

The first thing we can do to see if the formula in (71) is correct is by checking the correctness of the units. To check the units, we do it as follows:

$$\begin{aligned} \frac{\text{K}}{\text{s}} &= \frac{\text{W / (m K)}}{\text{kg / m}^3 \text{ J / (kg K)}} \left(\frac{\text{K}}{\text{m}^2} + \frac{\text{K}}{\text{m}^2} \right) - \frac{\text{W / (m}^2\text{K)}}{\text{m kg / m}^3 \text{ J / (kg K)}} \text{K} - \frac{\text{W / (m}^2\text{K}^4)}{\text{m kg / m}^3 \text{ J / (kg K)}} \text{K}^4 \\ \frac{\text{K}}{\text{s}} &= \frac{\text{K}}{\text{s}} - \frac{\text{K}}{\text{s}} - \frac{\text{K}}{\text{s}} \end{aligned}$$

The units are indeed correct.

MATLAB implementation of (71) is as follows:

```

1  dz = 0.01;
2  cp = 385;
3  K = 400;
4  rho = 8960;
5  alpha = K/(rho*cp);
6  beta = dz*rho*cp;
7
8  dx = 0.01;
9  dy = 0.01;
10 dt = 0.001; % Small enough to not cause instability
11
12 ly = 1;      % 1x1 m^2
13 lx = 1;
14 Nx = lx/dx;
15 Ny = ly/dy;
16 Nt = tF/dt;
17
18 tF = 10000; % Simulation time, modify accordingly
19
20 h = 10;
21 sigma = 5.67e-8;
22 e = 0.5;
23 T_inf = 300;
24
25 clear T_array u_array v_array f_array;
26
27 % Some extra nodes for phantom nodes
28 u_array = zeros(Nx+2,Ny+1); % Disable advection
29 v_array = zeros(Nx+2,Ny+1); % Disable advection
30 T_array = 297.15*ones(Nx+2,Ny+1);
31 f_array = zeros(Nx+2,Ny+1);
32
33 % Preparation for plotting
34 hfig = figure;
35 axis off;
36 hold on;
```

```

37 title('Diffusion')
38 caxis([300 800])
39
40 for k = 1 : Nt
41     % Heat source at the center of the plate, keep it true at all time
42     T_array = apply_bc(T_array, dx, dy, ...
43                         ["Dirichlet", "Neumann", "Neumann", "Neumann"], ...
44                         [800, 0, 0, 0]);
45
46     % Calculate the forcing component
47     f_array(2:end-1,2:end-1) = -(2*h/beta*(T_array(2:end-1,2:end-1) - T_inf) ...
48                                + 2*e*sigma/beta*(T_array(2:end-1,2:end-1).^4 - T_inf.^4));
49
50     % Solve the PDE
51     T_array = advection_diffusion_upwind(T_array, ...
52                                         u_array, ...
53                                         v_array, ...
54                                         dt, ...
55                                         dx, ...
56                                         dy, ...
57                                         alpha, ...
58                                         f_array);
59 end
60
61 heatmap(T_array, false);

```

In the listing above, we disable the advection by setting $u = v = 0$. The main iteration starts from line 40 to line 59. There are three things we put in the iteration: boundary condition update, extremal forcing component calculation, and PDE solving. After 10000 second, we will acquire the following result:

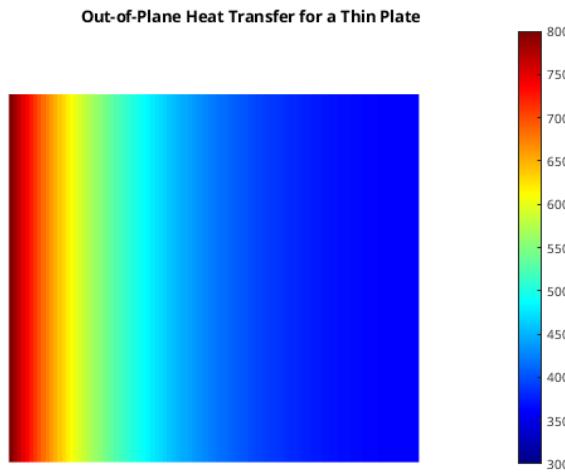


Figure 9: Temperature distribution of the plate after 10000 seconds.

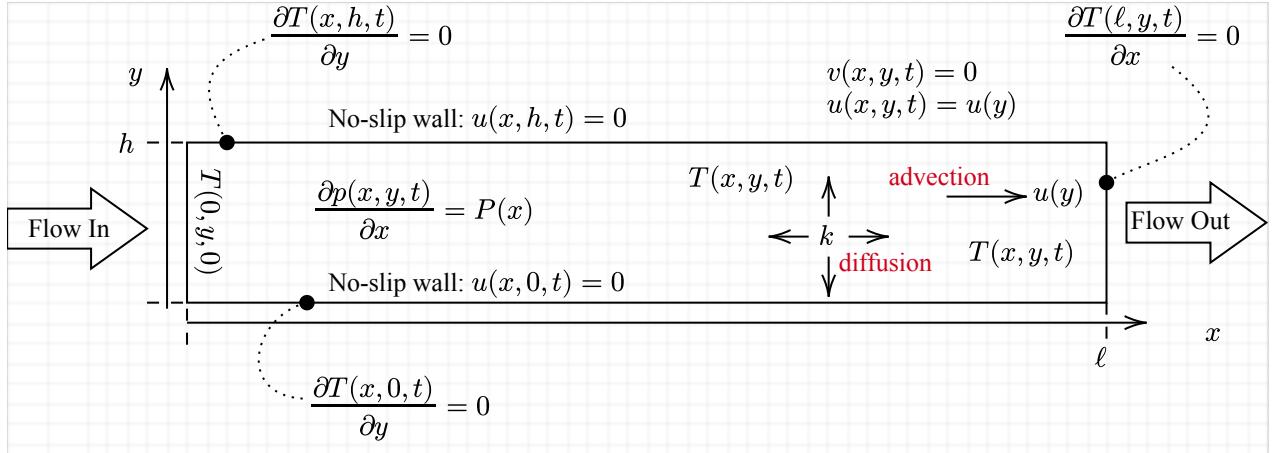
We can compare the result with COMSOL. However, the provided COMSOL simulation is static and it provides us only the result at steady-state condition.

5. Navier-Stokes with Heat Transfer

In this section, we will model a cross-current heat exchanger by using the famous Navier-Stokes equation for fluid flows with heat transfer. We will start with one flow first. After that, we will implement both flows and the occurring heat transfer between them.

5.1. Channel Flow with Heat Equation

We will model the flow of the fluid as a two-dimensional flow. The flow is driven by pressure gradient (planar Poiseuille flow).



Conservation of mass:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \quad (72)$$

Conservation of linear momentum:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} = -\frac{1}{\rho} \frac{\partial p}{\partial x} + \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \quad (73)$$

$$\frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} = -\frac{1}{\rho} \frac{\partial p}{\partial y} + \nu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \quad (74)$$

Conservation of energy:

$$\rho c_p \left(\frac{\partial T}{\partial t} + u \frac{\partial T}{\partial x} + v \frac{\partial T}{\partial y} \right) = k \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \quad (75)$$

where:

$u(x, y, t)$ is the x -velocity at location (x, y) at time t

$v(x, y, t)$ is the y -velocity at location (x, y) at time t

$p(x, y, t)$ is the pressure at location (x, y) at time t

$T(x, y, t)$ is the temperature at location (x, y) at time t

ν is the fluid viscosity coefficient.

c_p is the specific heat at constant

ρ is the fluid density

k is the thermal diffusivity rate

In planar Poiseuille flow, the flow is only in x -direction. u becomes a function of only y . Therefore: $\frac{\partial u}{\partial t} = 0, v = 0$,

$\frac{\partial p}{\partial y} = 0, \frac{\partial u}{\partial x} = 0$ and $\frac{\partial^2 u}{\partial x^2} = 0$. As a result, (72) to (74) can be simplified as follows:

$$\frac{1}{\rho} \frac{\partial p}{\partial x} = \nu \frac{\partial^2 u}{\partial y^2} \quad (76)$$

Next, we discretize (76) as follows:

5.2. Two Counter-Direction Channel Flows with Heat Equations

