Numerical Methods for Complex Systems
Auralius Manurung
ME - Universitas Pertamina

Table of Contents

1. Case 1: One-Dimensional Linear Advection[5]

Given a hyperbolic PDE as follows.

$$u_t + cu_x = f(x,t) \tag{1}$$

with the following initial condition:

$$u(x,0) = u_0(x) \tag{2}$$

where c is a positive real number. Here, we want to 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 application. Therefore, it is important to investigate its non-homogenous version to further allow an implementation of a PDE system. A PDE system is composed of more than one interacting PDEs. For example, in a heat exchanger, there are at least two PDEs: 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 with central difference for the distance to increase the approximation. To approximate u_t, we use the following approximation.

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

While to approach u_x, we use the following approximation.

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

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

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

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.

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

While to approach u_x, we use the following approximation.

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

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

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

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:

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

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

```
function next_u_array = upwind(u_array, f_array, c, dt, dx)

N = length(u_array);
next_u_array = u_array;

for i = 2:N
    next_u_array(i) = u_array(i)-c*dt/dx*(u_array(i)-u_array(i-1))+f_array(i)*dt;;
end

end
```

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

```
function next_u_array = upwind(u_array, f_array, c, dt, dx)

N = length(u_array);
next_u_array = u_array;

next_u_array(2:N) = u_array(2:N) - (c*dt/dx).* ...
                    (u_array(2:N)- u_array(1:N-1)) + f_array(2:N).*dt;

end
```

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

$$
\begin{gathered}
u_t + 0.5u_x = 0 \\
u(x,0) = e^{-(x-2)^2} \\
0 \leqslant x \leqslant 10
\end{gathered} \tag{8}
$$

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 ($dt = 0.05$).

```
L = 10;
dx = 0.1;
x = 0:dx:L;

dt = 0.05;
T = 20;
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.

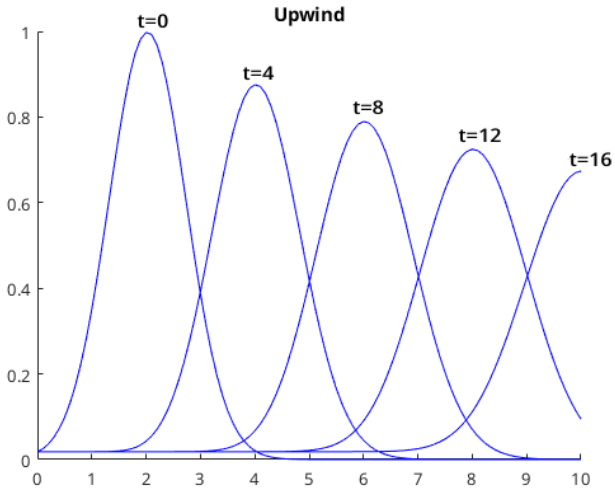


Figure 1: Upwind method for $u_t + 0.5u_{xx} = 0$

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.

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

While to approach u_x, we use the following approximation.

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

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

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

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.

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

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

```
function next_u_array = downwind(u_array, f_array, c, dt, dx)

N = length(u_array);
next_u_array = u_array;

for i = 1:N-1
    next_u_array(i) = u_array(i)-c*dt/dx*(u_array(i+1)-u_array(i))+f_array(i)*dt;
end

end
```

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

```
function next_u_array = downwind(u_array, f_array, c, dt, dx)

N = length(u_array);
next_u_array = u_array;

next_u_array(1:N-1) = u_array(1:N-1) - (c*dt/dx).* ...
                      (u_array(2:N) - u_array(1:N-1)) + f_array(1:N-1).*dt;

end
```

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

$$\begin{gathered} u_t - 0.5u_x = 0 \\ u(x,0) = e^{-(x-8)^2} \\ 0 \leqslant x \leqslant 10 \end{gathered} \tag{12}$$

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 ($dt = 0.05$).

```
L = 10;
dx = 0.1;
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.

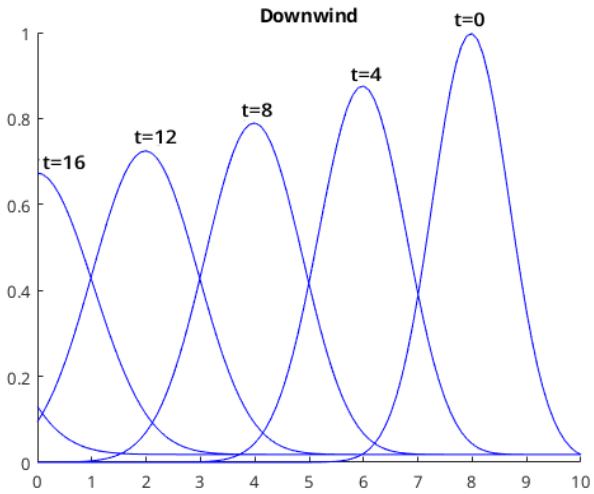


Figure 2: Downwind method for $u_t - 0.5u_{xx} = 0$

1.4. The Lax Method

The Lax method is based on the FTCS method.

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

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

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

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

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

On the other hand, u_x can be approximated as

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

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

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

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.

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

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.

```
function next_u_array = lax(u_array, f_array, c, dt, dx)

N = length(u_array);
next_u_array = u_array;

r = c*dt/dx;

for i = 2:N-1 % i = 1 and i = N are untouched !!!
    next_u_array(i) = 0.5*(u_array(i+1)+u_array(i-1)) - ...
                      0.5*r*(u_array(i+1)-u_array(i-1)) + ...
                      f_array(i)*dt;
end

% fill up the two ends by using upwind or downwind method
if c > 0
    next_u_array(N) = u_array(N) - r* ...
                     (u_array(N)- u_array(N-1)) + f_array(N).*dt;

elseif c < 0
    next_u_array(1) = u_array(1) - r* ...
                      (u_array(2) - u_array(1)) + f_array(1).*dt;

```

```
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;
```

```
h1 = plot(0,0);
title('Lax, moving to the right c = 0.5')
ylim([0 1]);

for k = 1:length(t)
    u = lax(u,f, c, dt, dx);
    set(h1,'XData',x,'Ydata',u);
    drawnow;
end

%% ------------------------------------------------------------
c = -0.5; % moving to left

for k = 1:length(x)
    u(k) = exp(-10*(x(k)-8).^2);
end

figure;
h2 = plot(0,0);
title('Lax, moving to the left, c = -0.5')
ylim([0 1]);

for k = 1:length(t)
    u = lax(u,f, c, dt, dx);
    set(h2,'XData',x,'Ydata',u);
    drawnow;
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.

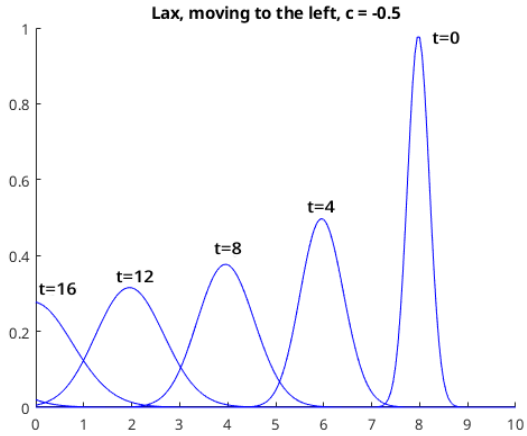


Figure 3: Lax method for $u_t - 0.5u_{xx} = 0$

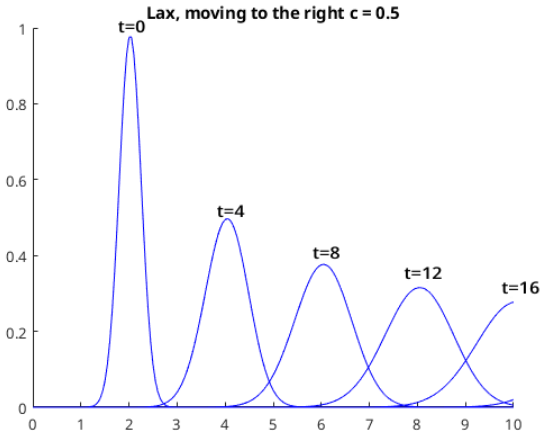


Figure 4: Lax method for $u_t + 0.5u_{xx} = 0$

1.5. The Lax-Wendroff Method[6]

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

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

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

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

Remember that

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

and

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

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

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

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.

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

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

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

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.

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

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

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

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

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.

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

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
```

```
if c > 0
    next_u_array(N) = u_array(N) - r* ...
                      (u_array(N)- u_array(N-1)) + f_array(N)*dt;

elseif c < 0
    next_u_array(1) = u_array(1) - r* ...
                        (u_array(2) - u_array(1)) + f_array(1)*dt;

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.

```
L = 10;
dx = 0.05;
x = 0:dx:L;

dt = 0.05;
T = 20;
t = 0:dt:T;

u = zeros(length(x), 1);
f = zeros(length(x), 1);

%% ------------------------------------------------------------
c = 0.5; % moving to right

for k = 1:length(x)
    u(k) = exp(-10*(x(k)-2).^2);
end

h = figure;
h1 = plot(0,0);
title('Lax-Wendroff, moving to the right c = 0.5')
ylim([0 1]);

for k = 1:length(t)
    u = lax_wendroff(u,f, c, dt, dx);
    set(h1,'XData',x,'Ydata',u);
    drawnow
end

%% ------------------------------------------------------------
c = -0.5; % moving to left

for k = 1:length(x)
    u(k) = exp(-10*(x(k)-8).^2);
end

h = figure;
h1 = plot(0,0);
title('Lax-Wendroff, moving to the left, c = -0.5')
ylim([0 1]);

for k = 1:length(t)
    u = lax_wendroff(u,f, c, dt, dx);
    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 tough the amplitudes are still decreasing, the decrease in the amplitude is much smaller than other methods that have been previously discussed.

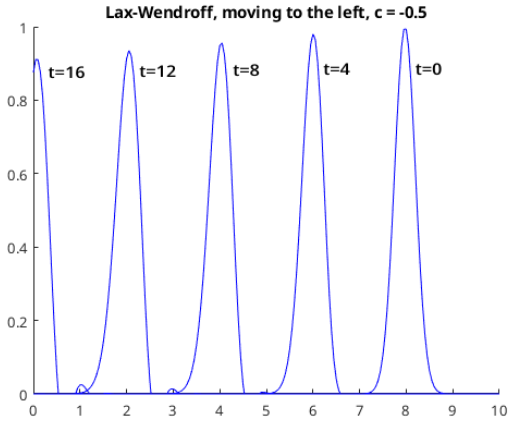


Figure 5: Lax method for $u_t - 0.5u_{xx} = 0$

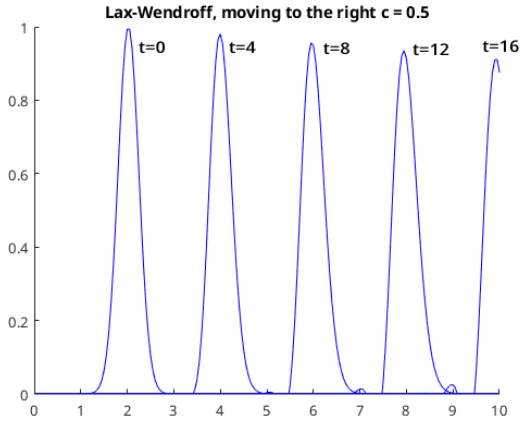


Figure 6: Lax method for $u_t + 0.5u_{xx} = 0$

1.6. Cross Current Heat Exchanger as An Example[7]

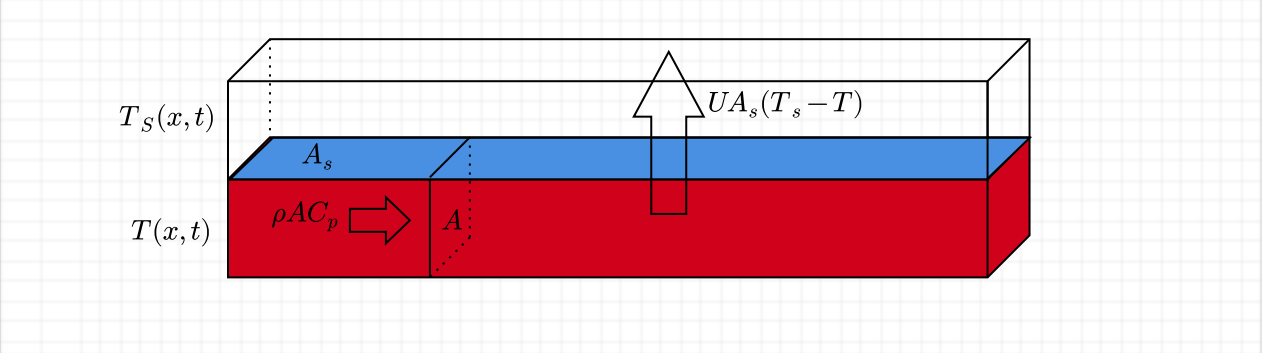
A heat exchanger is modeled with two or more interacting non-homogenous PDEs. The simplest model involves two PDEs only: the hot flow and the cold flow. Yet another simple method, three PDEs are involved: the hot flow, the cold flow, and the wall between them.

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.

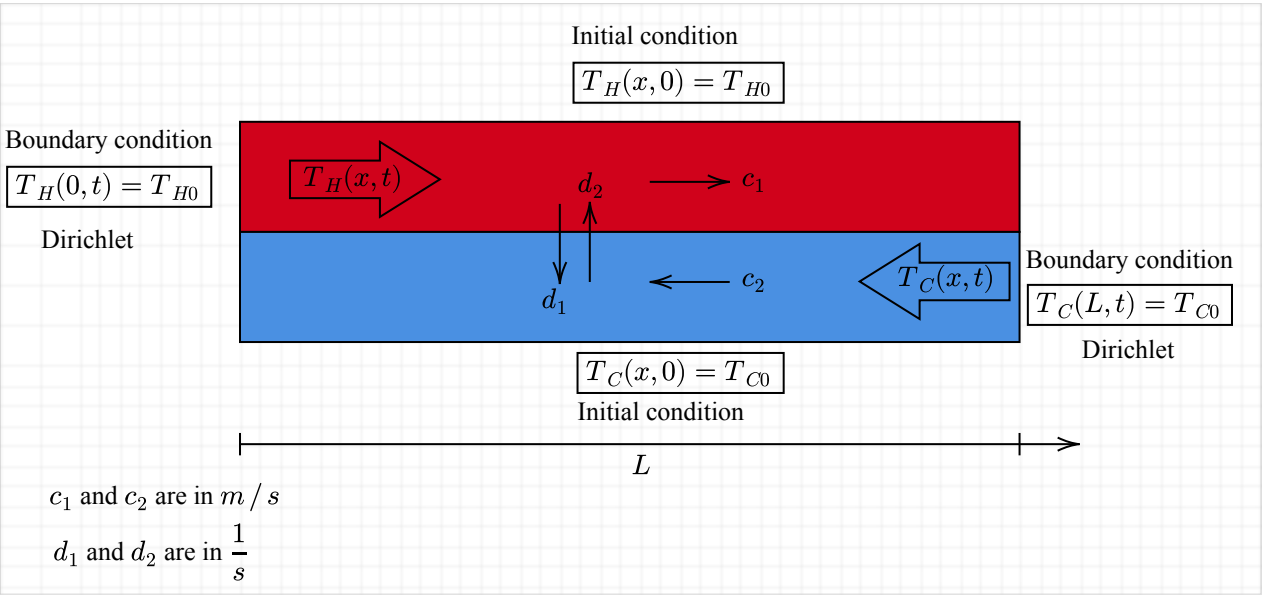
$$\underbrace{\frac{\partial T(x,t)}{\partial t} + v\frac{\partial T(x,t)}{\partial z}}_{\text{Advection}} = \underbrace{\frac{UA_S}{\rho A C_p}(T_S(x,t) - T(x,t))}_{\text{Fourier's law}} \tag{27}$$

where
C_p is the liquid specific heat capacity
A is cross sectional are 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.

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

where:

- d_1 and d_2 describe **convective** heat transfer **between** the 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 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.[8] 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

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

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.

```
L = 1;
dx = 0.2;
x = 0:dx:L;

dt = 0.01;
T = 20;
t = 0:dt:T;

% Define the parameters
c1 = 0.001711593407;
c2 = -0.01785371429;
d1 = -0.03802197802;
d2 = 0.3954285714;

% ========================================================================
% Upwind-downwind method
% ========================================================================
TH = zeros(length(x), 1);
TC = zeros(length(x), 1);
f = zeros(length(x), 1);

% Intial condition
TH(1:length(x)) = 273+30;
TC(1:length(x)) = 273+10;

figure;
hold on
h1 = plot(0,0,'r');
h2 = plot(0,0,'b');
title('Cross-current heat exchanger');
ylim([270 320]);
xlabel('x');
ylabel('Temperature');
legend('Hot flow','Cold flow');

for k = 1 : length(t)
    set(h1,'XData',x,'Ydata',TH);
    set(h2,'XData',x,'Ydata',TC);
    drawnow;

    f = d1.*(TH-TC);
    TH = upwind(TH, f, c1, dt, dx);
    f = d2.*(TH-TC);
    TC = downwind(TC, f, c2, dt, dx);
end

% ========================================================================
% Lax-Wnderoff
% ========================================================================
TH = zeros(length(x), 1);
TC = zeros(length(x), 1);
f = zeros(length(x), 1);

% Intial conditoin
TH(1:length(x)) = 273+30;
TC(1:length(x)) = 273+10;

```

```
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 lisiting 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.

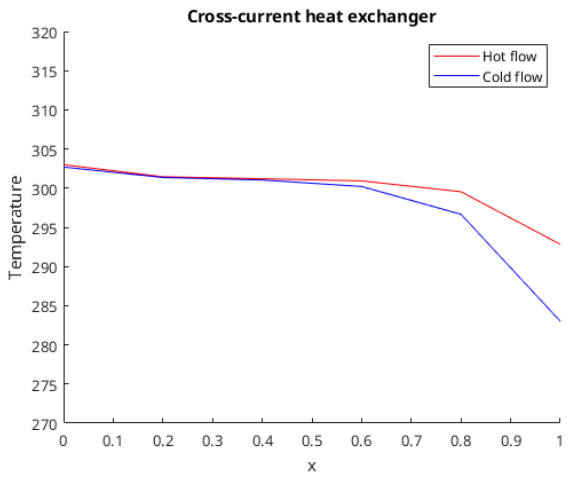


Figure 7: After 20 s, with upwind-downwind method

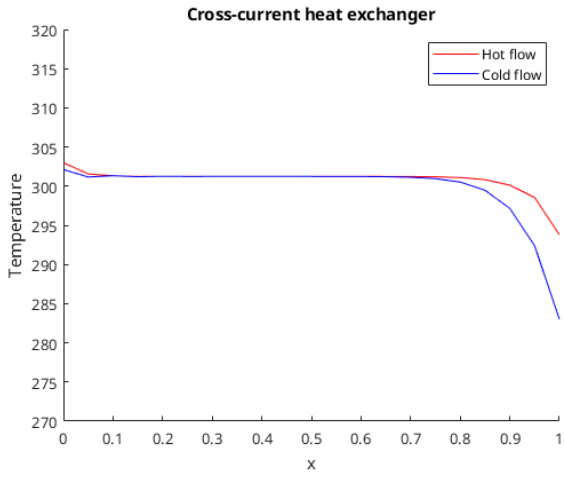
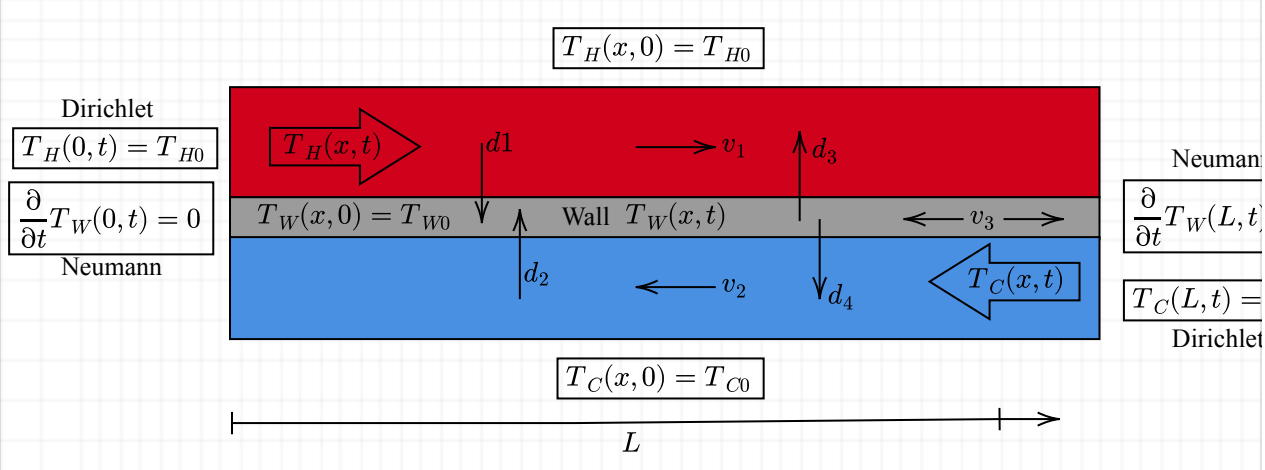


Figure 8: After 20 s, with Lax-Wendroff method

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 the 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 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 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.

$$\begin{aligned}
\frac{\partial}{\partial t}T_H(x,t) &= -v_1\frac{\partial}{\partial x}T_H(x,t) - d_1(T_H(x,t) - T_W(x,t)) \\
\frac{\partial}{\partial t}T_C(x,t) &= v_2\frac{\partial}{\partial x}T_C(x,t) + d_2(T_W(x,t) - T_C(x,t)) \\
\frac{\partial}{\partial t}T_W(x,t) &= v_3\frac{\partial^2}{\partial x^2}T_W(x,t) + d_3(T_H(x,t) - T_W(x,t)) - d_4(T_W(x,t) - T_C(x,t))
\end{aligned} \tag{29}$$

Where[9]:

$$\begin{aligned}
v_1 &= 0.6366197724 \\
v_2 &= 0.1657863990 \\
v_3 &= 0.00003585526318
\end{aligned}$$

$$\begin{aligned}
d_1 &= 0.9792 \\
d_2 &= 0.2986 \\
d_3 &= 2.3923 \\
d_4 &= 2.8708
\end{aligned}$$

The initial and boundary conditions are:

$T_H(0,t) = 360$ and $T_H(x,0) = 360$

$T_C(L,t) = 300$ and $T_C(x,0) = 300$

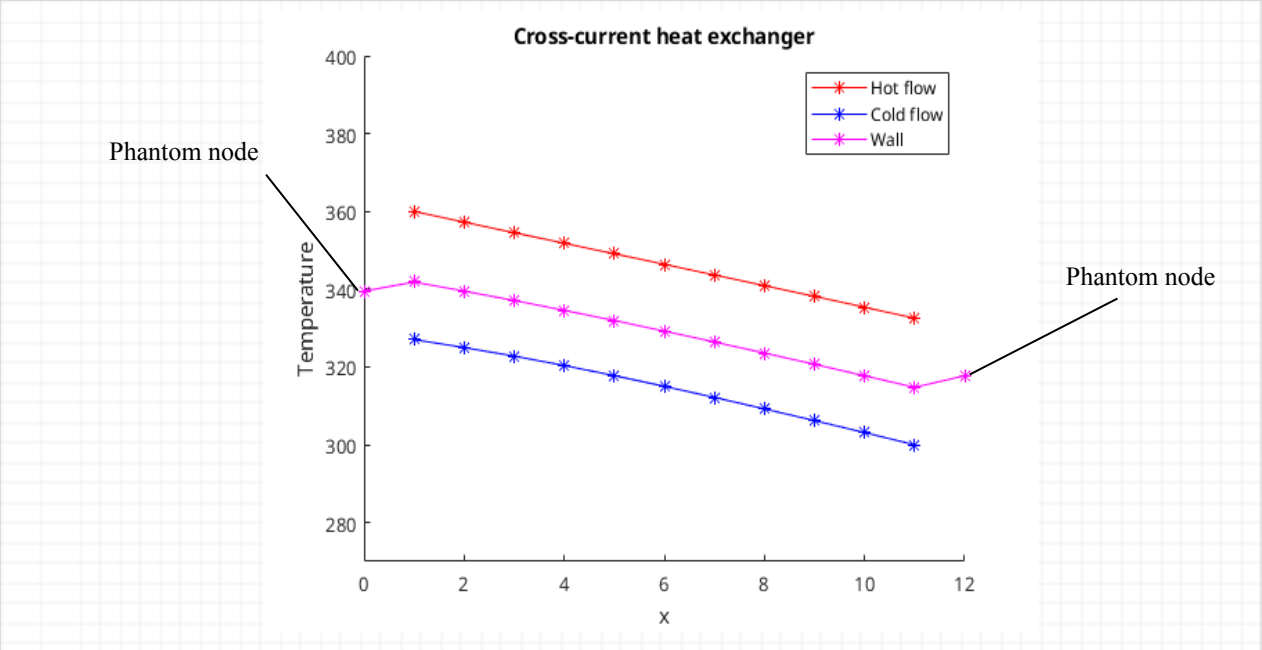
$T_W(x,0) = 330, \frac{\partial T_W(0,t)}{\partial x} = 0$, and $\frac{\partial T_W(L,t)}{\partial x} = 0$

However, 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, which we will discuss in the next section. MATLAB code for the system in (29) is presented in the following listing.

[9] https://www.maplesoft.com/applications/view.aspx?SID=119402&view=html

```
L = 1;
tF = 5;
Nx = 10;
Nt = 100;
dt = tF/Nt;
dx = L/Nx;

v1 = 0.6366197724;
v2 = 0.1657863990;
v3 = 0.00003585526318;
d1 = 0.9792;
d2 = 0.2986;
d3 = 2.3923;
d4 = 2.8708;

TH = zeros(1, Nx+1);
TC = zeros(1, Nx+1);
TW = zeros(1, Nx+1+2);  % +2 phantom nodes
fH = zeros(1, Nx+1);
fC = zeros(1, Nx+1);
fW = zeros(1, Nx+1+2);  % +2 phantom nodes

% Intial condition
TH(1:Nx+1) = 360;
TC(1:Nx+1) = 300;
TW(1:Nx+1+2) = 330;

h=figure;
hold on
h1 = plot(0,0,'r-*');
h2 = plot(0,0,'b-*');
h3 = plot(0,0,'m-*');
title('Cross-current heat exchanger')
ylim([270 400])
xlabel('x')
ylabel('Temperature')
legend('Hot flow', 'Cold flow', 'Wall')

xa = 1:Nx+1;
xb = 0:Nx+2;
for k = 1 : Nt
    set(h1,'XData',xa,'Ydata',TH);
    set(h2,'XData',xa,'Ydata',TC);
    set(h3,'XData',xb,'Ydata',TW);
    drawnow;

    TW = apply_bc(TW, L/Nx, ["Neumann", "Neumann"], [0, 0]);

    % TW(2:end-1) : it means we ignore pahnatom nodes

    fH = -d1.*(TH-TW(2:end-1));
    TH = upwind(TH, fH, v1, dt, dx);

    fC = d2.*(TH-TW(2:end-1));
    TC = downwind(TC, fC, -v2, dt, dx);

    fW(2:end-1) = d3.*(TH-TW(2:end-1)) - d4.*(TW(2:end-1)-TC);
    TW = diffusion_1d(TW, fW, sqrt(v3), dx, dt);
end
```

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



Drawing 4: Temperature distributions ofter 5 seconds

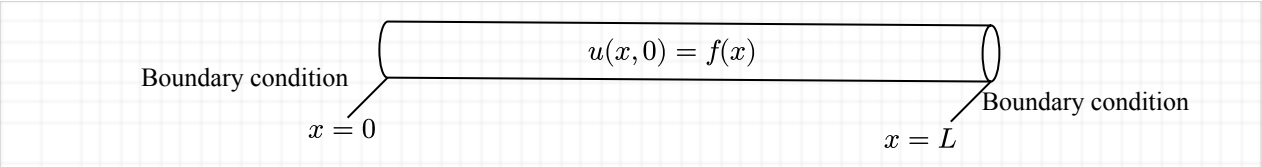
2. Case 2: One-Dimensional Heat Equation[10]

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 (30). α is the diffusivity coefficient. When $g(x,t)=0$, the system is homogenous.

$$\begin{gathered}\alpha^2 u_{xx} - u_t = g(x,t) \\ t \geqslant 0 \text{ and } 0 \leqslant x \leqslant L\end{gathered} \tag{30}$$

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.

$$u_t \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} \tag{31}$$

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

While to approach u_x, we use the following approximation.

$$\begin{aligned} u_{xx} &\approx \frac{u_x(x+\Delta x,t)-u_x(x,t)}{\Delta x} \\ &\approx \frac{\frac{u(x+\Delta x,t)-u(x,t)}{\Delta x} - \frac{u(x,t)-u(x-\Delta x,t)}{\Delta x}}{\Delta x} \\ &\approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{(\Delta x)^2} \end{aligned} \tag{33}$$

Substituting (31) and (32) back to (30) gives us the following equations.

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

where $r = \dfrac{\alpha^2 \Delta t}{(\Delta x)^2}$. 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.

$$\boxed{U_{j,k+1} = U_{j,k} + r\Big[U_{j+1,k} - 2U_{j,k} + U_{j-1,k}\Big] + G_{j,k}\Delta t} \tag{34}$$

2.1.1. Dirichlet Boundary Conditions

In (30), 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

$$u(0,t) = A$$
$$u(L,t) = B$$

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 $\left(\partial u(x,t)/\partial x \text{ or } \partial u(x,t)/\partial t\right)$ is kept constant. However, here we will only implement the derivative of the solution with respect to x.

$$\frac{\partial u(x,t)}{\partial x} = C \tag{35}$$

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

$$\frac{u(\Delta x,t) - u(-\Delta x,t)}{2\Delta x} = C \tag{36}$$

therefore

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

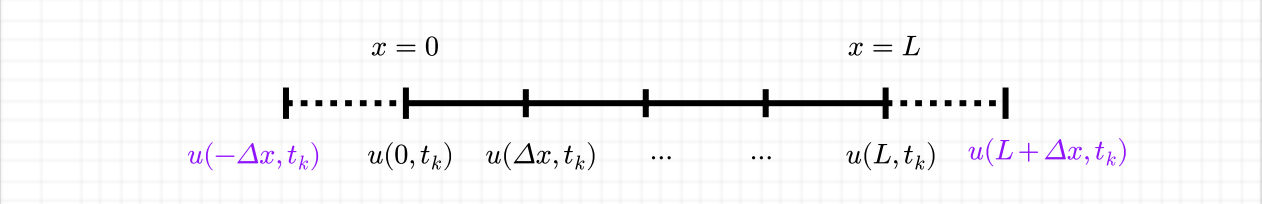
At another boundary when $x = L$

$$\frac{u(L+\Delta x,t) - u(L-\Delta x,t)}{2\Delta x} = D \tag{38}$$

where D is a real number. Therefore

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

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 (34) is as follows.

```
% alpha^2 * u_xx = u_tt
% Divide 0<=x<=L with Delta_x increments
% Divide 0<=t<=tF with Delta_t5 increments

function u_array = diffusion_1d(u_array, ...
                                g_array, ...
                                alpha, ...
```

```
                                Delta_x, ...
                                Delta_t)

r = alpha^2 * Delta_t / Delta_x^2;

j = 2 : length(u_array)-1;
u_array(j) = r*u_array(j-1) + (1-2*r) * u_array(j) + r*u_array(j+1) + g_array(j)*Delta_t;

end
```

The implementation above is already vectorized. In line 13 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).

```
% Initial condition
function u_array = apply_ic(u_array, f, Delta_x)

% Apply the initial condition U(x,0)=f(x)
j = 1:length(u_array);
u_array(j) = f((j-1)*Delta_x);

end

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

% Boundary condition
function u_array = apply_bc(u_array, Delta_x, types, values)

% left-end
if lower(types(1)) == "dirichlet"
    u_array(1) = values(1);
elseif lower(types(1)) == "neumann"
    u_array(1) = u_array(3) - 2*Delta_x * values(1);
end

% right-end
if lower(types(2)) == "dirichlet"
    u_array(end) = values(2);
elseif lower(types(2)) == "neumann"
    u_array(end) = u_array(end-2) + 2 * Delta_x * values(2);
end
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

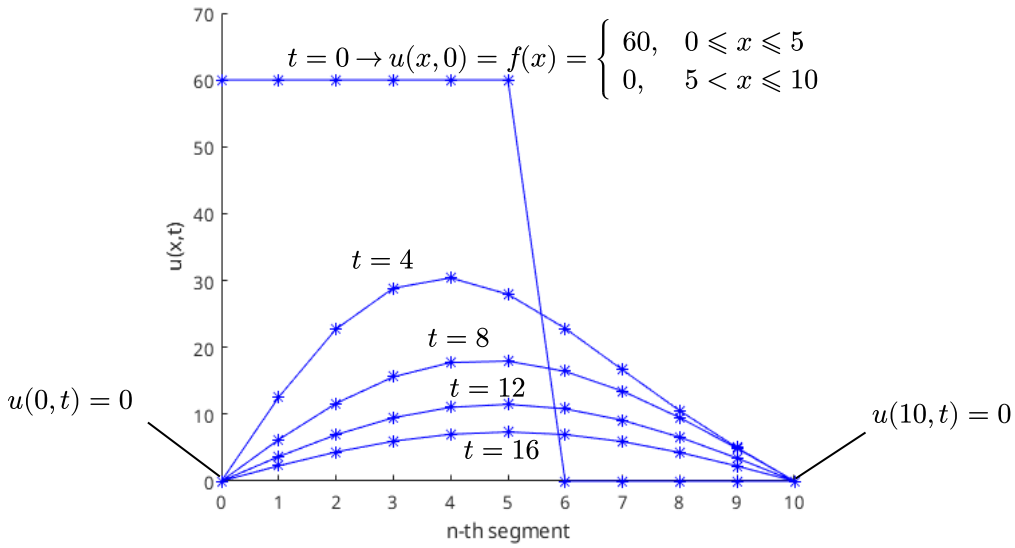
Given the following:

$$\begin{gathered} 1.14u_{xx} - u_t = 0,\ 0 \leqslant x \leqslant 10 \\ u(0,t) = u(10,t) = 0 \\ u(x,0) = f(x) = \begin{cases} 60, & 0 \leqslant x \leqslant 5 \\ 0, & 5 < x \leqslant 10 \end{cases} \end{gathered} \tag{40}$$

MATLAB implementation of (40) is as follows.

```
L = 10;
alpha = sqrt(1.14);

tF = 20;
Nx = 10;
Nt = 1000;

u_array = zeros(1,Nx+1); % +1 because we start from 0 to Nx
g_array = zeros(1,Nx+1); % homogenous
u_array = apply_ic(u_array,@f, L/Nx);

figure
hold on
h = plot(0,0,'-*');
ylim([0 70])

x = 0 : Nx;

for k = 1 : Nt
    set(h, 'XData', x, 'YData', u_array)
    drawnow;

    u_array = apply_bc(u_array, L/Nx, ["Dirichlet", "Dirichlet"], [0, 0]);
    u_array = diffusion_1d(u_array, g_array, alpha, L/Nx, tF/Nt);
end
xlabel('n-th segment')
ylabel('u(x,t)')

%% Intial Condition
function u=f(x)
L = 10;
u = zeros(1,length(x));

for k = 1 : length(x)
    if x(k) >= 0 && x(k) <=L/2
        u(k) = 60;
    elseif x(k) > L/2 && x(k) <= L
        u(k) = 0;
    end
end

end
```

The result is shown in Drawing 7 below.



Drawing 7: Simulation result: Dirichlet-Dirichlet

2.1.5. Example 2

Given the following:

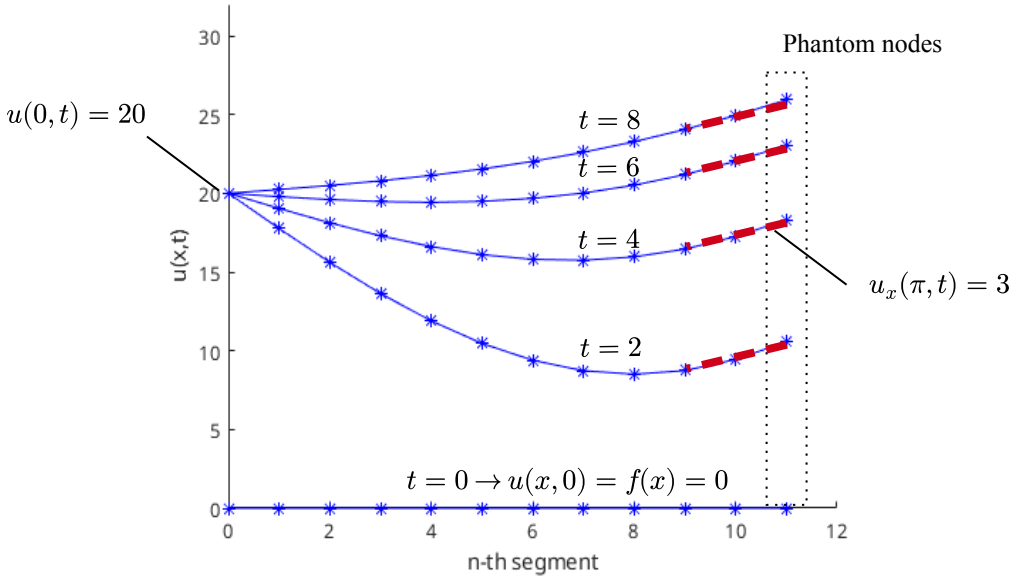
$$\begin{gathered} u_{xx} - u_t = 0,\ 0 \leqslant x \leqslant \pi \\ u(0,t) = 20 \\ u_x(\pi,t) = 3 \\ u(x,0) = 0 \end{gathered} \tag{41}$$

MATLAB implementation of (41) is as follows.

```
L = pi;
alpha = 1;

tF = 10;
Nx = 10;
Nt = 1000;

u_array = zeros(1,Nx+1+1); % +1 -> start from 0 to Nx
                           % another +1 -> add one phantom nodes
g_array = zeros(1,length(u_array));
u_array = apply_ic(u_array,@f, L/Nx);

figure
hold on
h = plot(0,0,'-*');
ylim([0 32])

x = 0 : Nx+1;

for k = 1 : Nt
    set(h, 'XData', x, 'YData', u_array)
    drawnow;

    u_array = apply_bc(u_array, L/Nx, ["Dirichlet", "Neumann"], [20, 3]);
    u_array = diffusion_1d(u_array, g_array, alpha, L/Nx, tF/Nt);
end
```

```
xlabel('n-th segment')
ylabel('u(x,t)')

%% Intial Condition
function u=f(x)
    u = zeros(1,length(x));
end
```

The result is shown in Drawing 8 below.



Drawing 8: Simulation result: Dirichlet-Neumann

2.1.6. Example 3

Given the following:

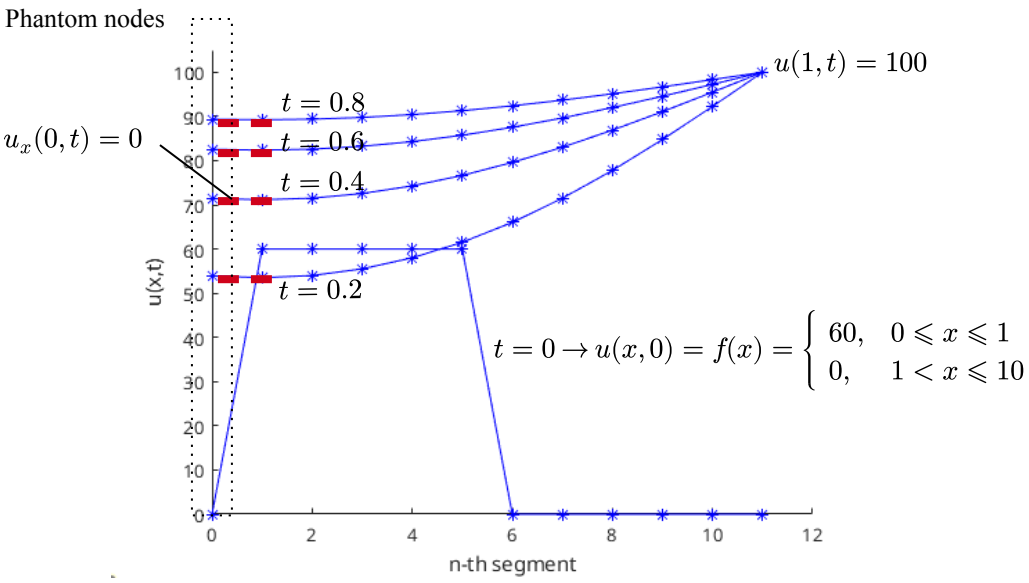
$$\begin{gathered} u_{xx} - u_t = 0,\ 0 \leqslant x \leqslant 1 \\ u_x(0,t) = 0 \\ u(1,t) = 100 \\ u(x,0) = f(x) = \begin{cases} 60, & 0 \leqslant x \leqslant 1/2 \\ 0, & 1/2 < x \leqslant 1 \end{cases} \end{gathered} \tag{42}$$

MATLAB implementation of (42) is as follows.

```
L = 1;
alpha = 1;

tF = 1;
Nx = 10;
Nt = 1000;

u_array = zeros(1,Nx+1+1); % +1 because we start from 0 to Nx
                           % another +1 -> add one phantom nodes
g_array = zeros(1,length(u_array));
u_array = apply_ic(u_array,@f, L/Nx);

hf = figure;
hold on
```

```
h = plot(0,0,'-*');
ylim([0 105])
xlabel('n-th segment')
ylabel('u(x,t)')

x = 0 : Nx+1;

for k = 1 : Nt
    set(h, 'XData', x, 'YData', u_array)
    drawnow;

    u_array = apply_bc(u_array, L/Nx, ["Neumann", "Dirichlet"], [0, 100]);
    u_array = diffusion_1d(u_array, g_array, alpha, L/Nx, tF/Nt);
end

%% Initial condition
function u=f(x)
L = 1;
u = zeros(1,length(x));

for k = 1 : length(x)
    if x(k) >= 0 && x(k) <=L/2
        u(k) = 60;
    elseif x(k) > L/2 && x(k) <= L
        u(k) = 0;
    end
end

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:

$$u_{xx}(x,t) = (1-\theta)u_{xx}(x,t) + \theta u_{xx}(x,t+\Delta t),\ 0 \leqslant \theta \leqslant 1 \tag{43}$$

where θ acts as a weighting factor. When $\theta = 0$, (43) becomes similar to (34). 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 (43) bu first setting $\theta = \frac{1}{2}$ gives us:

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

which can be shortened as

$$u_{xx} = \frac{U_{j+1,k}-2U_{j,k}+U_{j-1,k}+U_{j+1,k+1}-2U_{j,k+1}+U_{j-1,k+1}}{2\Delta x^2} \tag{44}$$

Substituting (44) back to: $u_t = \alpha^2 u_{xx}$ gives us:

$$\begin{aligned} \underbrace{\frac{U_{j,k+1}-U_{j,k}}{\Delta t}}_{u_t} &= \alpha^2 \underbrace{\frac{U_{j+1,k}-2U_{j,k}+U_{j-1,k}+U_{j+1,k+1}-2U_{j,k+1}+U_{j-1,k+1}}{2\Delta x^2}}_{u_{xx}} \\ U_{j,k+1}-U_{j,k} &= \frac{1}{2}\frac{\alpha^2\Delta t}{\Delta x^2}(U_{j+1,k}-2U_{j,k}+U_{j-1,k}+U_{j+1,k+1}-2U_{j,k+1}+U_{j-1,k+1}) \end{aligned} \tag{45}$$

Since $r = \frac{\alpha^2\Delta t}{\Delta x^2}$, (45) can then be simplified as:

$$\begin{aligned} U_{j,k+1}-U_{j,k} &= \frac{r}{2}(U_{j+1,k}-2U_{j,k}+U_{j-1,k}+U_{j+1,k+1}-2U_{j,k+1}+U_{j-1,k+1}) \\ U_{j,k+1}+rU_{j,k+1} &= \frac{r}{2}U_{j+1,k}+(1-r)U_{j,k}+\frac{r}{2}U_{j-1,k}+\frac{r}{2}U_{j+1,k+1}+\frac{r}{2}U_{j-1,k+1} \\ 2(1+r)U_{j,k+1} &= rU_{j+1,k}+2(1-r)U_{j,k}+rU_{j-1,k}+rU_{j+1,k+1}+rU_{j-1,k+1} \\ -rU_{j-1,k+1}+2(1+r)U_{j,k+1}-rU_{j+1,k+1} &= rU_{j-1,k}+2(1-r)U_{j,k}+rU_{j+1,k} \end{aligned} \tag{46}$$

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

$$\boxed{-rU_{j-1,k+1}+2(1+r)U_{j,k+1}-rU_{j+1,k+1} = rU_{j-1,k}+2(1-r)U_{j,k}+rU_{j+1,k}} \tag{47}$$

or in matrix form:

$$\underbrace{\begin{bmatrix} -r & 2(1+r) & -r & & \cdots & 0 \\ 0 & -r & 2(1+r) & -r & & \vdots \\ & & -r & 2(1+r) & -r & \\ & & & \ddots & & \\ \vdots & & & & -r & 2(1+r) \\ 0 & \cdots & & & 0 & -r \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} p((k+1)\Delta t) \\ U_{1,k+1} \\ U_{2,k+1} \\ \vdots \\ U_{N-2,k+1} \\ q((k+1)\Delta t) \end{bmatrix}}_{U} =$$

$$\underbrace{\begin{bmatrix} r & 2(1-r) & r & & \cdots & 0 \\ 0 & r & 2(1-r) & r & & \vdots \\ & & r & 2(1-r) & r & \\ & & & \ddots & & \\ \vdots & & & & r & 2(1-r) \\ 0 & \cdots & & & 0 & r \end{bmatrix} \begin{bmatrix} p(k\Delta t) \\ U_{1,k} \\ U_{2,k} \\ \vdots \\ U_{N-2,k} \\ q(k\Delta t) \end{bmatrix}}_{B} \tag{48}$$

Red means the values are known. In (48), 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.

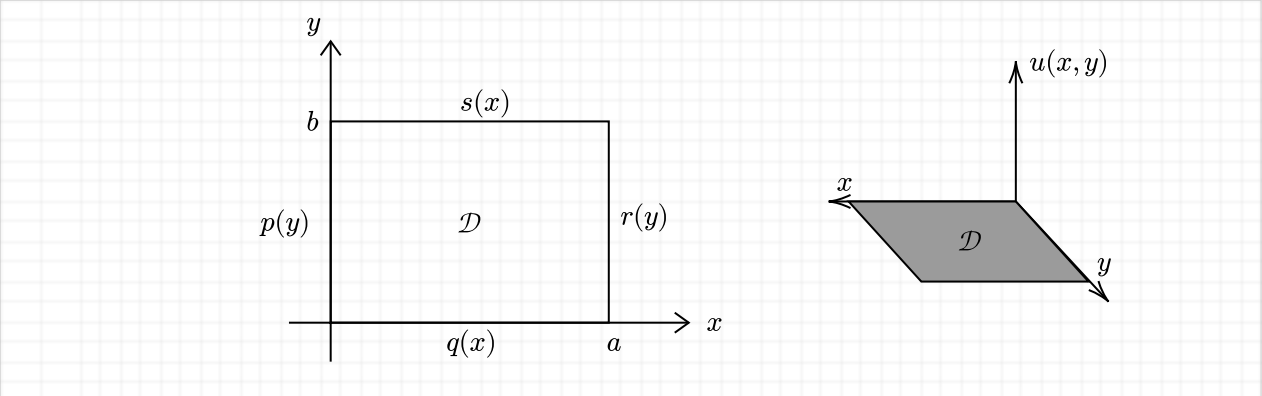
$$\underbrace{\begin{bmatrix} 2(1+r) & -r & & \cdots & 0 \\ -r & 2(1+r) & -r & & \vdots \\ & & \ddots & & \\ & & -r & 2(1+r) & -r \\ \cdots & & & -r & 2(1+r) \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} U_{1,k+1} \\ U_{2,k+1} \\ \vdots \\ U_{N-2,k+1} \\ U_{N-1,k+1} \end{bmatrix}}_{U_{new}} =$$

$$\underbrace{\underbrace{\begin{bmatrix} 2(1-r) & r & & \cdots & \cdots \\ r & 2(1-r) & r & & \\ & \ddots & & & \\ & & r & 2(1-r) & r \\ \cdots & & & r & 2(1-r) \end{bmatrix}}_{b} \underbrace{\begin{bmatrix} U_{1,k} \\ U_{2,k} \\ \vdots \\ U_{N-2,k} \\ U_{N-1,k} \end{bmatrix}}_{U_{now}} + \underbrace{\begin{bmatrix} rp((k+1)\Delta t) + rp(k\Delta t) \\ 0 \\ \vdots \\ 0 \\ rq((k+1)\Delta t) + rq(k\Delta t) \end{bmatrix}}_{c}}_{B=(b\cdot U_{now})+c} \tag{49}$$

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} = 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 Equation[11]

3.1. The Five-Point Difference Method



Drawing 10: Heat distribution on a two-dimensional plane

Finite difference method for the system above is as follows.

3.2. Energy Balance Model: Out-of-Plane Heat Transfer for a Thin Plate - Comparison with COMSOL