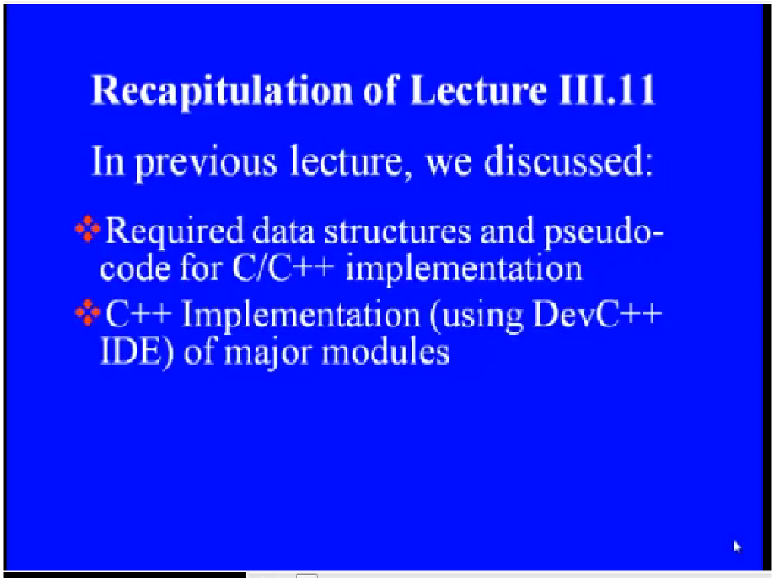
Computational Fluid Dynamics

Dr. Krishna M. Singh
Department of Mechanical and Industrial Engineering
Indian Institute of Technology - Roorkee

Lecture – 21
Computer Implementation of FDM for Steady State Heat Diffusion Problems - 3

Welcome to the last lecture in module 3 on finite difference. So concluding lecture on the computer implementation of finite difference method for 1-D heat conduction problems. Let us have a recap of what we finished in the previous lecture. We discussed a bit more about data structures and their implementation in a C++ code.

(Refer Slide Time: 00:47)



Recapitulation of Lecture III.11

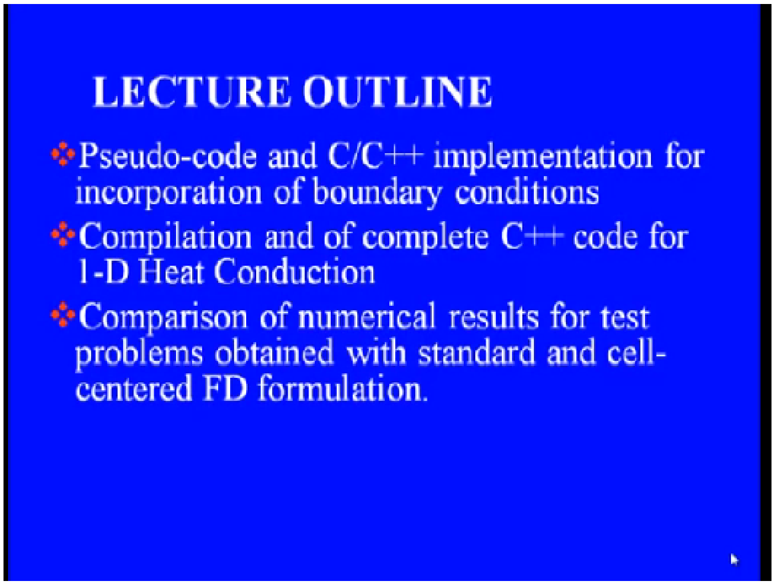
In previous lecture, we discussed:

- Required data structures and pseudo-code for C/C++ implementation
- C++ Implementation (using DevC++ IDE) of major modules

And we also finished implementing the major modules in a C++ program for solution of 1-dimensional problem using developer C++ IDE. Now in this lecture, we would try to finish the remaining modules basically submodules which require the implementation of boundary condition. So this is our last lecture in the series on computer implementation of finite difference method for steady state heat conduction problems.

So we will focus on the pseudo-code and C, C++ implementation for incorporation of boundary conditions for both standard as less cell-centred finite difference formulations and then we will complete our code that could be complete in all respects, we will compile it together and test for few, 1 or 2, sample problems and we would compare the numerical results for a test problem obtained with standard and cell-centred finite difference formulation.

(Refer Slide Time: 01:47)

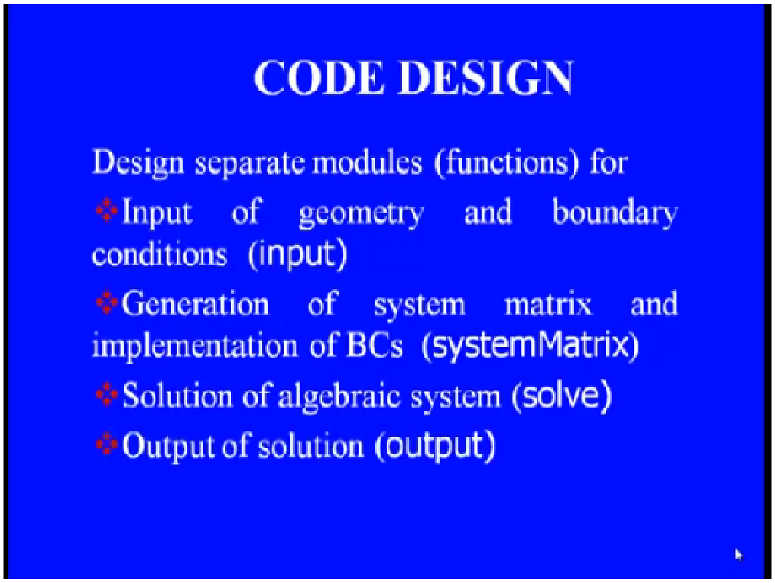


LECTURE OUTLINE

- Pseudo-code and C/C++ implementation for incorporation of boundary conditions
- Compilation and of complete C++ code for 1-D Heat Conduction
- Comparison of numerical results for test problems obtained with standard and cell-centered FD formulation.

So if all the dynamic code design, we had implemented our input module.

(Refer Slide Time: 01:57)



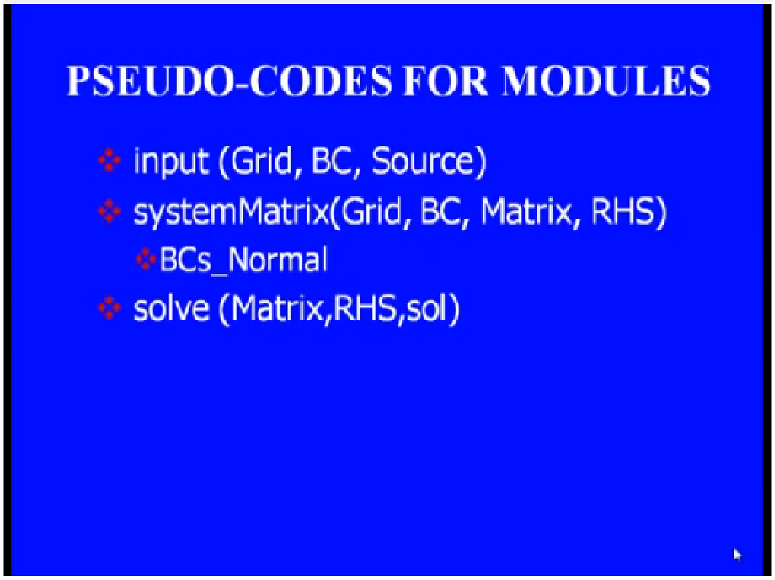
CODE DESIGN

Design separate modules (functions) for

- Input of geometry and boundary conditions (input)
- Generation of system matrix and implementation of BCs (systemMatrix)
- Solution of algebraic system (solve)
- Output of solution (output)

We had implemented the outline for the module system matrix for generation of finite difference discrete system and we need to complete the incorporation of boundary conditions in this module and then we said we are going to just call an already available module TDMA-based function to solve our system's equations. Like if you have already seen the previous lecture, the pseudo-code for input and systemMatrix module.

(Refer Slide Time: 02:27)

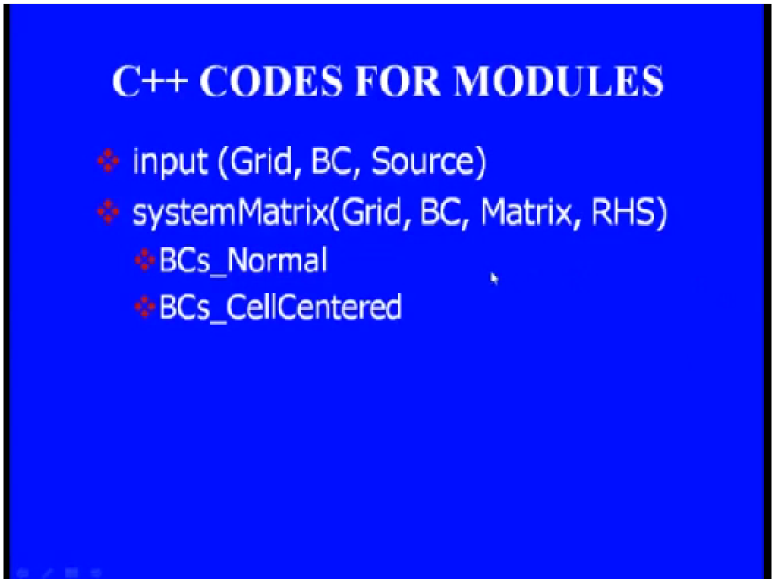


PSEUDO-CODES FOR MODULES

- input (Grid, BC, Source)
- systemMatrix(Grid, BC, Matrix, RHS)
  - BCs_Normal
- solve (Matrix,RHS,sol)

Now we are going to work on the implementation of boundary conditions. We will have a look at the pseudo-code for that.

(Refer Slide Time: 02:37)

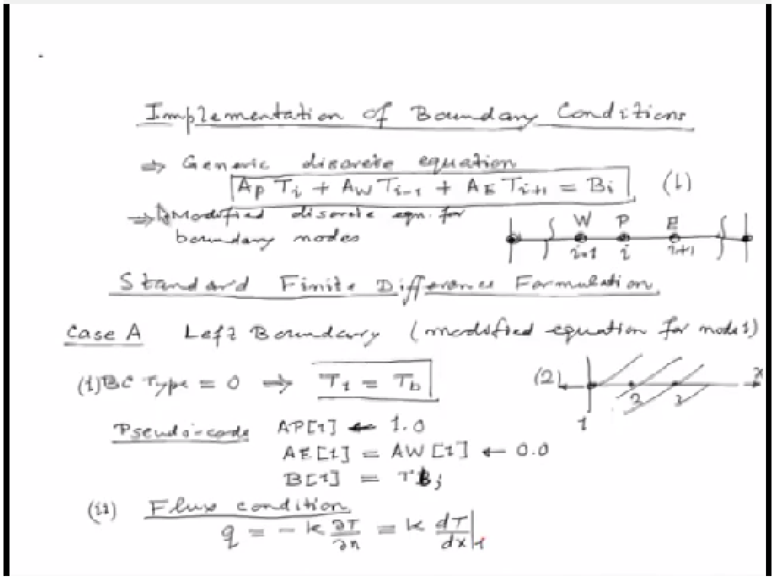


C++ CODES FOR MODULES

- input (Grid, BC, Source)
- systemMatrix(Grid, BC, Matrix, RHS)
  - BCs_Normal
  - BCs_CellCentered

And we would implement it in the code for both the cases, that is BCs normal, that would stand for standard formulation and the cell-centred formulation. Now before we go for implementation, let us first have a look at what modifications we need to incorporate in our code for implementation of boundary conditions.

(Refer Slide Time: 02:55)



Implementation of Boundary Conditions

⇒ Generic discrete equation

$$\boxed{A_P T_i + A_W T_{i-1} + A_E T_{i+1} = B_i} \quad (1)$$

⇒ Modified discrete eqn. for boundary nodes

W
P
E
i-1
i
i+1

Standard Finite Difference Formulation

Case A Left Boundary (modified equation for node 1)

(i) BC Type = 0 ⇒ $\boxed{T_1 = T_b}$

(2)
1
2
3
x

Pseudo-code
```
AP[1] ← 1.0
AE[1] = AW[1] ← 0.0
B[1] = Tb;
```

(ii) Flux condition

$$q = -k\frac{\partial T}{\partial n} = k\left.\frac{dT}{dx}\right|_1$$

Now the end result would be the modification in the discrete equation, we had our generic discrete equation for a node i, this was given as ApTi+AwTi-1+AETi+1=Bi for a generic node i. B stands for the current node, E stands for this stern neighbourhood i+1 and W stands for its left neighbour or a western neighbour.

Now what do you mean by boundary condition that we have to now modify, the boundary condition would be specified at the end of the domain. So we need to modify the nodes, the computational nodes which are in the vicinity, obtain modify discrete equation. So we will have a modified discrete equation for boundary nodes. So let us take the 2 cases separately. First let us deal with our standard finite difference grid wherein we have the nodes aligned with the boundary.

So for a finite difference formulation, I will put the pseudo-code only for 1 or 2 cases here. Let us say, the 2 possibilities and we have to deal both the possibilities separately, okay. So let us take the case when, let us call this case 1 or case A, left boundary. By left boundary, we mean we have to obtain a discrete equation or modified equation for node 1, node 2, node 3 and so on.

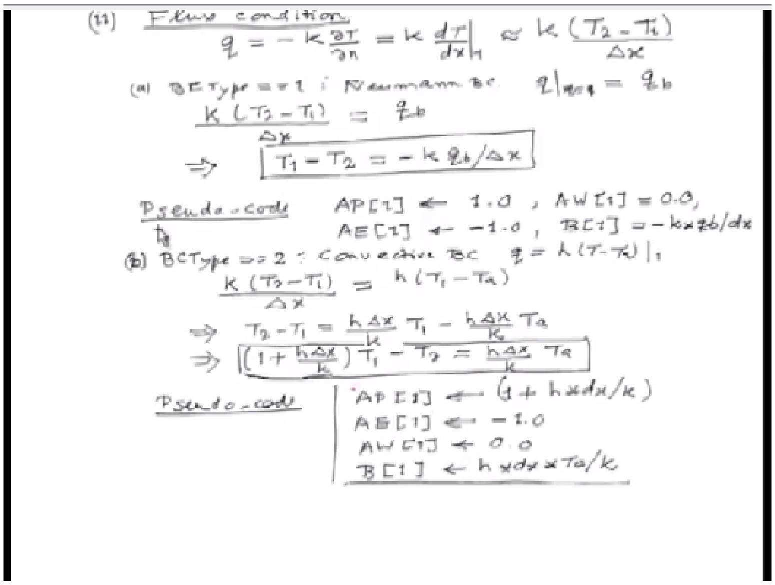
This is our interior of the domain. Now depending on the boundary conditions which are specified at node 1, we will have a different set of equations or different form equation for this particular node. So let us have our BC Type=0 that is we have T1-TB. Now if this is given, of course this becomes our discrete equation, okay and if we compare with standard or generic discrete equation which we have derived for all the nodes or would have computed in our code earlier for all the nodes, okay.

So on comparison what we say, that this coefficient node of right in a pseudo-code format, so that we can translate it easily into a code. So basically all that we need to do is set the coefficient AP of the node 1 as 1, AW and AE should be set to 0 and B 1 should be set to TB, that is all we need to do. So if AP1, this is set to 1.0, this left arrow, this is a standard notation in computer program that assign 1.0 to AP1, AE1=AW1 and these would be assigned 0 values.

Our B1 would be assigned the boundary conditions specified at this node T subscript B. So now this is one particular case or one condition. The second condition could be, we have got the flux specified. So flux conditions, we have to be careful the way we define the flux and take care of the directional normal. Normal is now directed in negative X direction. So we have got a flux definition q is given by -K del T/del n or dT/dn.

Now here n is aligned in negative x direction so we would get in terms of x, this becomes K*dT/dx at point 1. Now we have got this derivative here and it has to be approximated and we have got no choice other than using a forward difference schemes. So we are going to approximate this as K*T2-T1/delta x.

(Refer Slide Time: 09:57)



(ii) Flux condition

$$q = -k\frac{\partial T}{\partial n} = k\left.\frac{dT}{dx}\right|_1 \approx k\frac{(T_2 - T_1)}{\Delta x}$$

(a) BCType == 1 : Neumann BC $\quad q|_{x=1} = q_b$

$$\frac{K(T_2 - T_1)}{\Delta x} = q_b$$

$$\Rightarrow \boxed{T_1 - T_2 = -k\, q_b/\Delta x}$$

Pseudo-code

```
AP[1] ← 1.0 , AW[1] = 0.0,
AE[1] ← -1.0 , B[1] = -k*qb/dx
```

(b) BCType == 2 : Convective BC $\quad q = h(T - T_a)|_1$

$$\frac{K(T_2 - T_1)}{\Delta x} = h(T_1 - T_a)$$

$$\Rightarrow T_2 - T_1 = \frac{h\Delta x}{k}T_1 - \frac{h\Delta x}{k}T_a$$

$$\Rightarrow \boxed{\left(1 + \frac{h\Delta x}{k}\right)T_1 - T_2 = \frac{h\Delta x}{k}T_a}$$

Pseudo-code

```
AP[1] ← (1 + h*dx/k)
AE[1] ← -1.0
AW[1] ← 0.0
B[1] ← h*dx*Ta/k
```

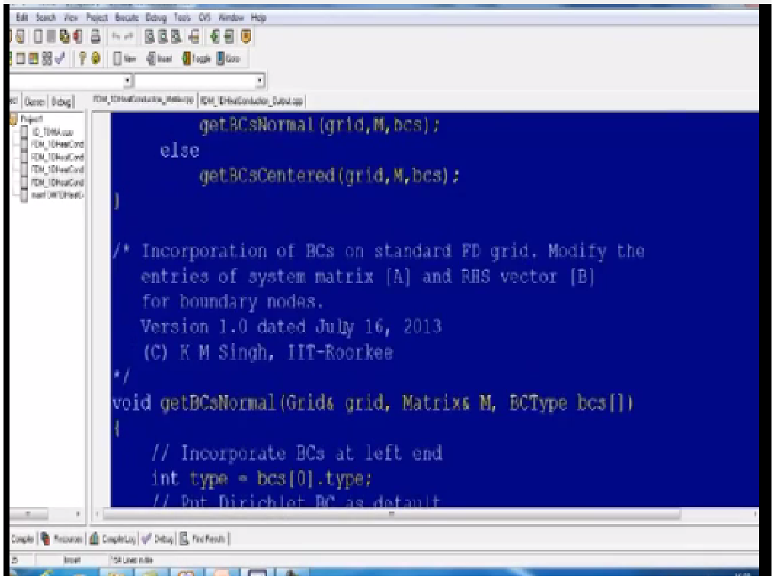
So now let us take 2 sub cases, the first sub case BC Type 1, BC Type=1 which stands for flux specifications that is normal boundary condition, q at i=1. This q is given as qb. So if you set this value to our finite difference approximation, we get K*T2-T1/delta x=qb and this gives us our standard equation T1-T2=-K*qb/delta x.

So pseudo-code for modification, is AP of 1, we set it as 1.0, AW at node 1 at 0.0 and AE at node 1, this will be set as -1.0. B for this node 1, that is -K*qb/dx. Now the next possibility, the last one is our convective boundary condition which stands for BCType in our code, we have set that as 2, is our (()) (12:35) flux, so this stands for convective BC q=h*T-Ta at node 1, so that simply means our K*T2-T1/delta x=h*T1-Ta.

If you rearrange it, we get T2-T1=h delta x/k*T1-h delta x/k*Ta that is 1+h delta x/k*T1-T2=h delta x/k*Ta. So a pseudo-code that is AP of 1, this has to be set a value which is a coefficient of T1 in this equation, 1+h*dx/k, this is the value which you assigned to AP of 1. AE of 1 would be assigned a value -1.0, AW of 1, we set at 0 and B of 1 which includes now the effect of convective boundary condition, this would be set as h*dx*Ta/k.

So these are the lines of code which we need to incorporate in our program. So now let us get back to our program and see the actual implementation.

(Refer Slide Time: 15:18)

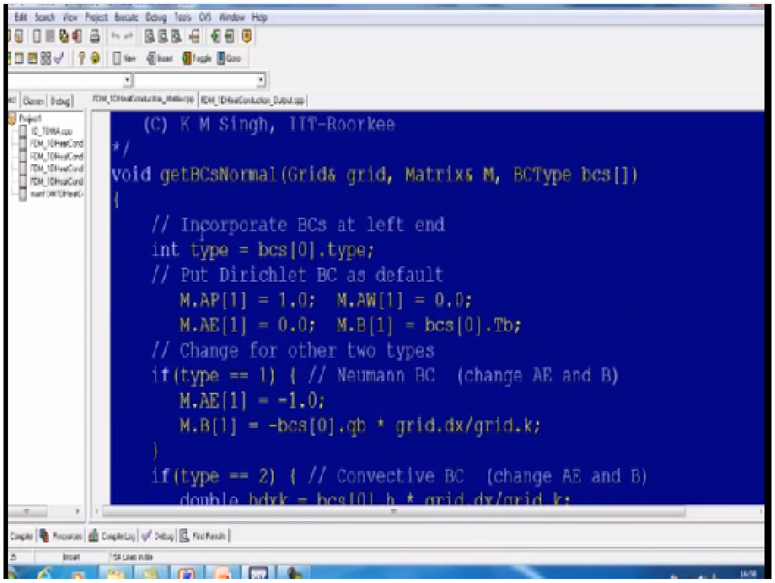


```
        getBCsNormal(grid,M,bcs);
    else
        getBCsCentered(grid,M,bcs);
}

/* Incorporation of BCs on standard FD grid. Modify the
   entries of system matrix [A] and RHS vector [B]
   for boundary nodes.
   Version 1.0 dated July 16, 2013
   (C) K M Singh, IIT-Roorkee
*/
void getBCsNormal(Grid& grid, Matrix& M, BCType bcs[])
{
    // Incorporate BCs at left end
    int type = bcs[0].type;
    // Put Dirichlet BC as default
```

That is incorporation of boundary condition on standard finite difference grid modified entries of systemMatrix A and RHS vector B for boundary nodes.

(Refer Slide Time: 15:29)



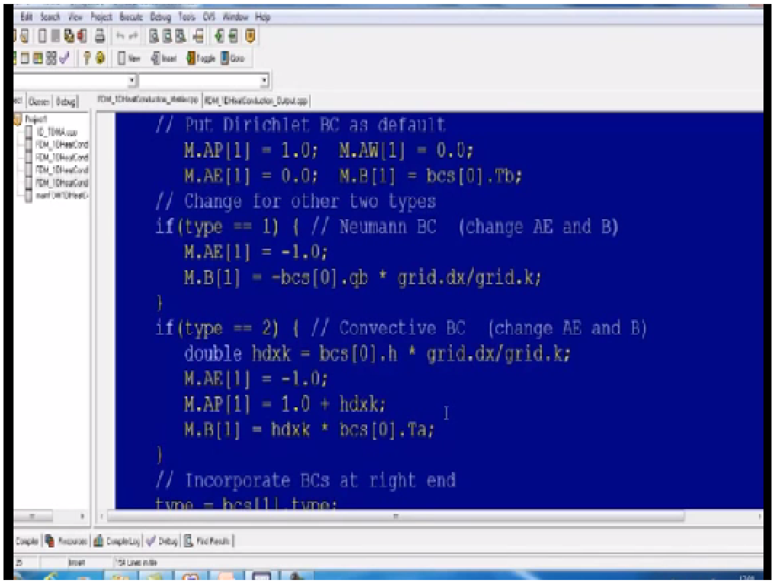
```
   (C) K M Singh, IIT-Roorkee
*/
void getBCsNormal(Grid& grid, Matrix& M, BCType bcs[])
{
    // Incorporate BCs at left end
    int type = bcs[0].type;
    // Put Dirichlet BC as default
       M.AP[1] = 1.0;  M.AW[1] = 0.0;
       M.AE[1] = 0.0;  M.B[1] = bcs[0].Tb;
    // Change for other two types
    if(type == 1) { // Neumann BC  (change AE and B)
       M.AE[1] = -1.0;
       M.B[1] = -bcs[0].qb * grid.dx/grid.k;
    }
    if(type == 2) { // Convective BC  (change AE and B)
       double hdxk = bcs[0].h * grid.dx/grid.k;
```

So we have called this routine as getBCNormal, normal stands for standard finite difference formulation of inputs grid matrix and boundary condition type and what we have discussed so far is our left incorporation of boundary conditions at left end. So let us increase a local variable type=bcs0.type and Dirichlet BC, so we can set this if we assume that Dirichlet BC to be present anywhere, we have seen that AP1 that is set to 1.0.

And rest of the coefficient AW and AE at node 1, they are set to 0.0 and the source term that becomes our specified temperature value. So that is what we saw in our pseudo-code. Now if we have the type=1 that is we have got Neumann BC, then we need to change, AW we have already set to 0, so we need to change AE and B. AP we have seen in this case remains as 1.0. So we do not need to write a separate line in this block.

So we just change AE, so M.AE1=-1.0 and M.B1 that is given by -qb dx/k, so we have put in the access specifier set bcs0.qb that will give us the specified heat flux at the left node, grid.dx that gives grid spacing and grid.k that has told the thermal conductivity.

(Refer Slide Time: 17:14)

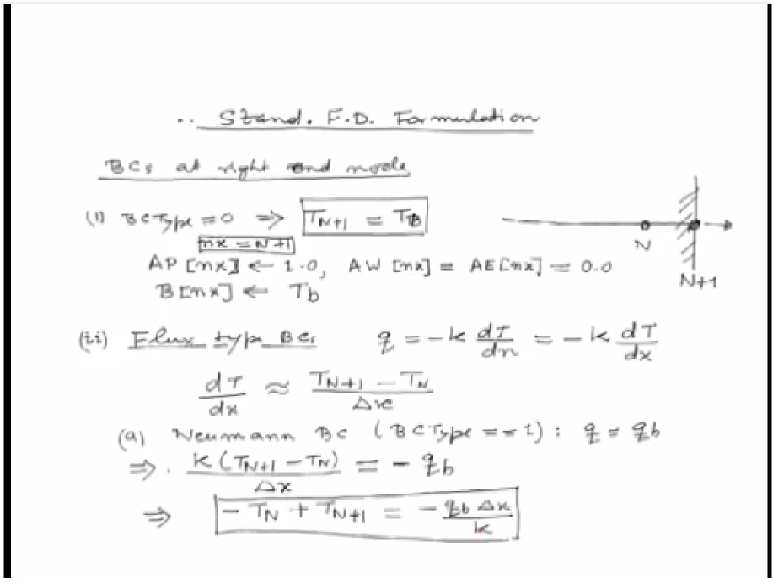


```
   // Put Dirichlet BC as default
      M.AP[1] = 1.0;  M.AW[1] = 0.0;
      M.AE[1] = 0.0;  M.B[1] = bcs[0].Tb;
   // Change for other two types
   if(type == 1) { // Neumann BC  (change AE and B)
      M.AE[1] = -1.0;
      M.B[1] = -bcs[0].qb * grid.dx/grid.k;
   }
   if(type == 2) { // Convective BC  (change AE and B)
      double hdxk = bcs[0].h * grid.dx/grid.k;
      M.AE[1] = -1.0;
      M.AP[1] = 1.0 + hdxk;
      M.B[1] = hdxk * bcs[0].Ta;
   }
   // Incorporate BCs at right end
   type = bcs[1].type;
```

Next, when we come to convective boundary conditions, we need to change AE, AP and B, these 3. AQ, we have already set to 0.0, so we do not need to worry in this block. Let us define a local variable hdxk which includes h*dx/k, so hdxk stands for BCS0.h. Remember this 0 index is being used for boundary condition at the left node, into grid.dx/grid.k and AE, M.AE1 that is the coefficient corresponding to the stern neighbour that becomes -1.0 coefficient corresponding to the current node that is 1.0+hdxk.

And our boundary condition which now influence our source term that becomes M.B1=hdxk*bcs0.Ta. So whatever pseudo-code which we have known, incorporated that in our code boundary conditions for the left node. We need to do the similar exercise and we ought to obtain the pseudo-code expression for boundary conditions at the right node.

(Refer Slide Time: 18:38)



.. Stand. F.D. Formulation

BCs at right end node

(i) BCType == 0 $\Rightarrow$ $\boxed{T_{N+1} = T_B}$

$\boxed{nx = N+1}$

AP [nx] $\leftarrow$ 1.0, AW [nx] = AE[nx] = 0.0

B[nx] $\leftarrow$ T_b

(ii) Flux type BCs $q = -k\frac{dT}{dn} = -k\frac{dT}{dx}$

$$\frac{dT}{dx} \approx \frac{T_{N+1} - T_N}{\Delta x}$$

(a) Neumann BC (BCType == 1) : $q = q_b$

$$\Rightarrow \frac{k(T_{N+1} - T_N)}{\Delta x} = -q_b$$

$$\Rightarrow \boxed{-T_N + T_{N+1} = -\frac{q_b \Delta x}{k}}$$

So we are still with standard finite difference formulation. So BCs at right end node, so let us draw sample grid, this is our grid point and its index would be an n+1 if the number of division is N, this left node will have an index of N and so on. Boundary conditions would be specified here. So let us have 3 cases. The first one is our BC Type=0, we simply say that TN+1 is specified as given boundary temperature.

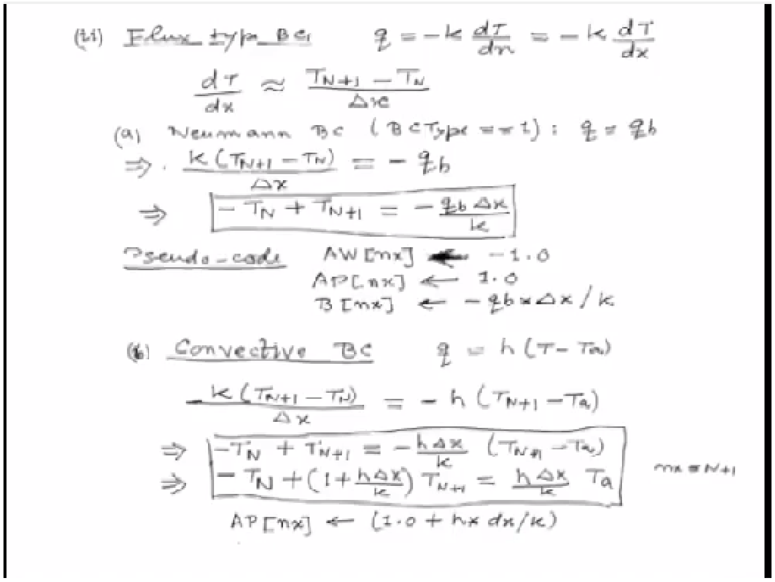
So this where the case, all that we need to do is set the corresponding entry. So let us use a simpler term, let us call it, let us define that nx=n+1, so AP of nx set its value to be 1.0. AWnx and AEnx value both provide the 0 value and our right-hand side vector is Bnx, we need to store in it the specified temperature value.

Now the next one, flux type boundary condition, flux type BCs, so now in this case, we need to come up with an approximation for the flux term itself which involved derivative. Now this q=-kdT/dn, now in this case, our vector n is aligned with positive x direction. So dT/dn and dT/dx, they would have the same meaning or they would-be identical. So this -KdT/dx and this now dT/dx, we will have to approximate it using a one-sided difference.

So we will use backward difference approximation using the values at node n+1 and node n. So this will be approximated by Tn+1-Tn/delta x. So now let us take 2 sub cases. The first one is

Neumann BC which corresponds to an integer flag BC Type=1, we have q=qb. So this simply tells us that k*Tn+1-Tn/delta x, this is equal to -qb or we can write this as -Tn+T of N+1= -qb delta x/k. So if you compare with what we had obtained, the expression for the left and is its presence fairly similar, on the indices have changed.

(Refer Slide Time: 23:20)



(ii) Flux type BC $\quad q = -k\frac{dT}{dn} = -k\frac{dT}{dx}$

$$\frac{dT}{dx} \approx \frac{T_{N+1} - T_N}{\Delta x}$$

(a) Neumann BC (BCType == 1) : $q = q_b$

$$\Rightarrow \frac{k(T_{N+1} - T_N)}{\Delta x} = -q_b$$

$$\Rightarrow \boxed{-T_N + T_{N+1} = -\frac{q_b \Delta x}{k}}$$

Pseudo-code

```
AW[nx] ← -1.0
AP[nx] ← 1.0
B[nx] ← -qb*Δx/k
```

(b) Convective BC $\quad q = h(T - T_a)$

$$\frac{-k(T_{N+1} - T_N)}{\Delta x} = -h(T_{N+1} - T_a)$$

$$\Rightarrow \boxed{-T_N + T_{N+1} = -\frac{h\Delta x}{k}(T_{N+1} - T_a)}$$

$$\Rightarrow \boxed{-T_N + \left(1 + \frac{h\Delta x}{k}\right)T_{N+1} = \frac{h\Delta x}{k} T_a}$$

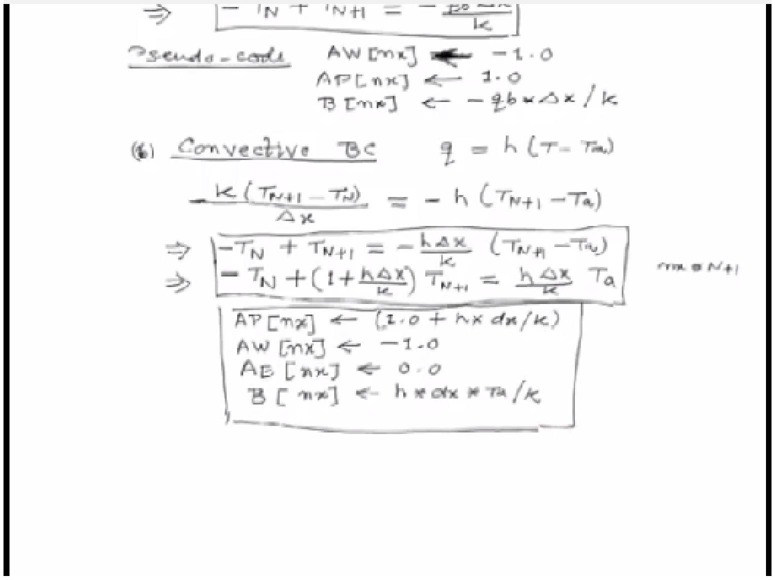
nx = N+1

AP[nx] ← (1.0 + h*dx/k)

So the pseudo-code now which we need to incorporate in our function, our AWnx, this should be set with the value of -1.0. APnx, this should be set a value of 1.0 and Bnx, this should be assigned a value -qb*delta x/k. Now the next case is convective boundary condition, it says q=h*T-Ta, so this has K*T of N +1-TN/delta x=-h*TN+1-Ta. Remember in our, I think we had just transposed the signs.

Alternatively, we can rearrange it as -TN-T of N+1=-h delta x/k*T of N+1-Ta. Rearrange it further to put in standard form. So -TN +1+ h delta x/k*T of N+1=h delta x/k*Ta. So compare this expression with our standard discrete equation for this node and then we can say that from what we need to modify that is our AP of nx, remember this nx, we have introduced as a alter notation for an index N+1. So remember that, nx stands for N+1. So APnx should be assigned a value 1.0+h*dx/k.

(Refer Slide Time: 26:24)



$$\Rightarrow \boxed{-T_N + T_{N+1} = -\frac{q_b \Delta x}{k}}$$

Pseudo-code

```
AW[nx] ← -1.0
AP[nx] ← 1.0
B[nx] ← -qb*Δx/k
```

(c) Convective BC $q = h(T - T_a)$

$$-\frac{k(T_{N+1} - T_N)}{\Delta x} = -h(T_{N+1} - T_a)$$

$$\Rightarrow -T_N + T_{N+1} = -\frac{h\Delta x}{k}(T_{N+1} - T_a)$$

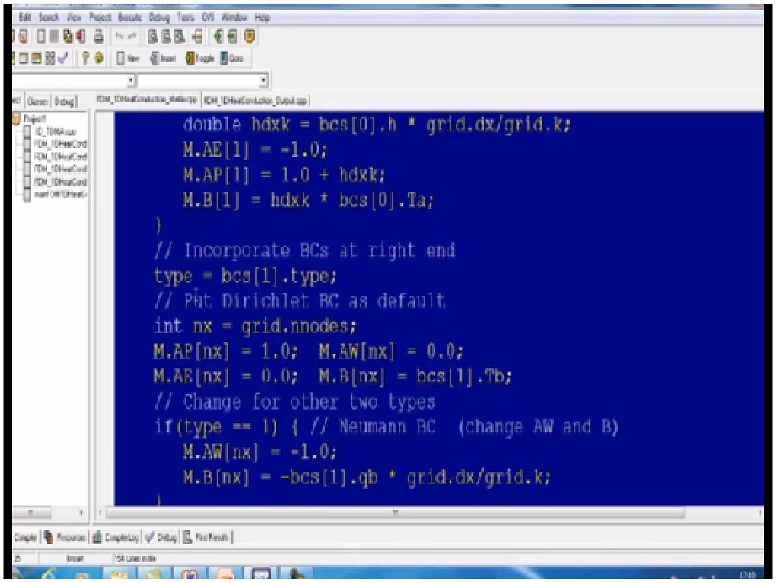
$$\Rightarrow -T_N + \left(1 + \frac{h\Delta x}{k}\right)T_{N+1} = \frac{h\Delta x}{k} T_a$$

nx = N+1

```
AP[nx] ← (1.0 + h×dx/k)
AW[nx] ← -1.0
AE[nx] ← 0.0
B[nx] ← h*dx*Ta/k
```

AWnx should be assigned a value of -1.0 and AEnx, this will be set to 0.0 and our load vector element of B at this point nx, this should be assigned a value h*dx*Ta/k. So now this completes our pseudo-code, these 4 lines are the ones which we need to directly put in our code to take care of the incorporation of the effect of convective boundary condition.

(Refer Slide Time: 27:13)



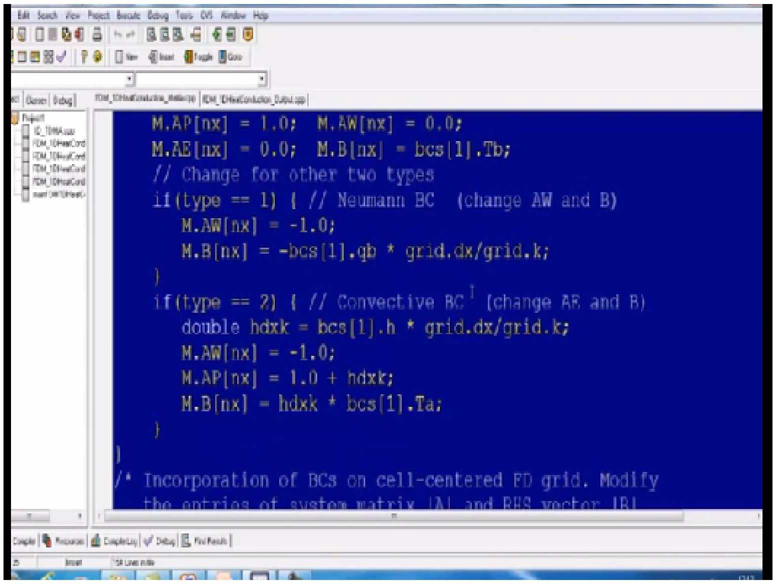
```
      double hdxk = bcs[0].h * grid.dx/grid.k;
      M.AE[1] = -1.0;
      M.AP[1] = 1.0 + hdxk;
      M.B[1] = hdxk * bcs[0].Ta;
   }
   // Incorporate BCs at right end
   type = bcs[1].type;
   // Put Dirichlet BC as default
   int nx = grid.nnodes;
   M.AP[nx] = 1.0;  M.AW[nx] = 0.0;
   M.AE[nx] = 0.0;  M.B[nx] = bcs[1].Tb;
   // Change for other two types
   if(type == 1) { // Neumann BC  (change AW and B)
      M.AW[nx] = -1.0;
      M.B[nx] = -bcs[1].qb * grid.dx/grid.k;
   }
```

So let us get back to our code. So incorporation of BCs at right n. So type=bcs1, this argument of 1 stands for the boundary condition at right end. So get type of boundary condition and as we said we are going to now introduce a local integer nx=gride.nnodes. So nx is actually, this stands for the value Capital N+1 where n is a number of divisions.

Now let us put the Dirichlet BC, that is the simplest one and let us assume that for a default BC, so in that case our M.APnx that is provided with a value 1, remaining coefficients was set to 0 and M.Bnx is set to bcs1.TB. Now change for other 2 types of boundary conditions, if type=1 that is our Neumann BC.

Now in this case for the Neumann BC, we have already seen a pseudo-code, that APnx that should be set to 1 that we have already done. So we need to just change the values of AWnx, so M.AWnx=-1.0 and next M.Bnx which includes the effect of qb, so -bcs subscript 1.qb*grid.dx/grid.k.

(Refer Slide Time: 28:57)

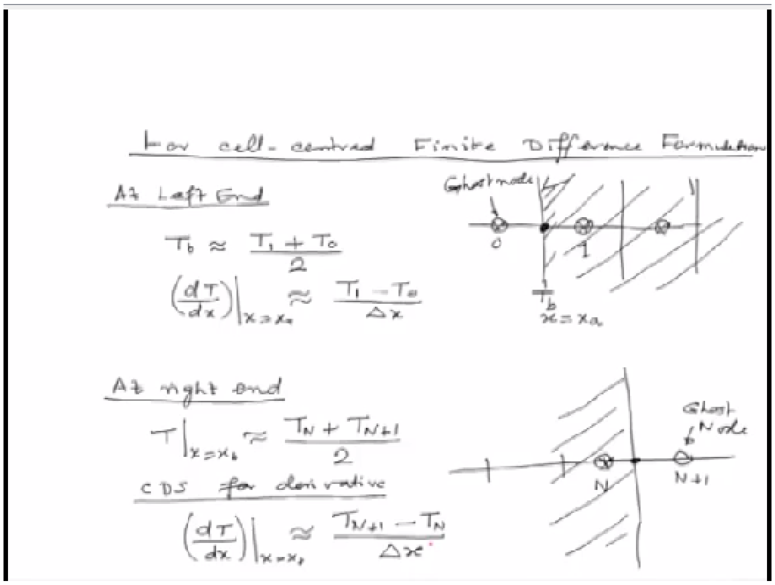

Edit Search View Project Execute Debug Tools CVS Window Help
New Insert Toggle Goto
Classes Debug
Project
Compiler Resources Compile Log Debug Find Results
Insert

```
    M.AP[nx] = 1.0;  M.AW[nx] = 0.0;
    M.AE[nx] = 0.0;  M.B[nx] = bcs[1].Tb;
    // Change for other two types
    if(type == 1) { // Neumann BC  (change AW and B)
      M.AW[nx] = -1.0;
      M.B[nx] = -bcs[1].qb * grid.dx/grid.k;
    }
    if(type == 2) { // Convective BC  (change AE and B)
      double hdxk = bcs[1].h * grid.dx/grid.k;
      M.AW[nx] = -1.0;
      M.AP[nx] = 1.0 + hdxk;
      M.B[nx] = hdxk * bcs[1].Ta;
    }
}
/* Incorporation of BCs on cell-centered FD grid. Modify
   the entries of system matrix [A] and RHS vector [B]
```

Now in case of, we have got convective boundary conditions. So we need to change that again AW and AP and B. So let us define the short hand notation variable hdxk which stands for h*dx/k our x is specifiers, so h would be bcs subscript 1.h*grid.dx/grid.k. So M.AW for the node nx is -1.0. M.APnx that becomes 1.0+hdxk and M.Bnx=hdxk*bcs1.Ta. So now we have finished our coding part for the incorporation of boundary conditions with standard grid that is our vertex centred grid.

Similar exercise we can repeat for the cell-centred finite difference grid. I will not do the complete derivation for the pseudo-code, that I would leave as an exercise but few salient points I would just like to mention which you should remember for cell-centred grid.

(Refer Slide Time: 30:33)



For cell-centred Finite Difference Formulation

At Left End

$$T_b \approx \frac{T_1 + T_0}{2}$$

$$\left(\frac{dT}{dx}\right)\bigg|_{x=x_a} \approx \frac{T_1 - T_0}{\Delta x}$$

At right end

$$T\big|_{x=x_b} \approx \frac{T_N + T_{N+1}}{2}$$

CDS for derivative

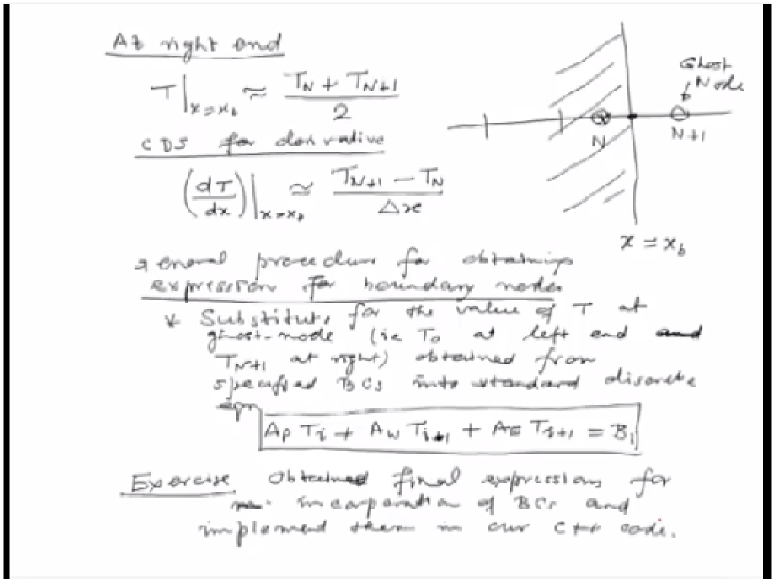
$$\left(\frac{dT}{dx}\right)\bigg|_{x=x_b} \approx \frac{T_{N+1} - T_N}{\Delta x}$$

For cell-centred finite difference formulation, please remember at 2 ends, at left end, at both the ends, we are going to introduce what we call ghost points. Remember the computational nodes are in the centre of the cell, this is our problem domain, this is our boundary, left boundary. The boundary conditions are specified here. So we will introduce a ghost node of zero and temperature at boundary, let us call it as T of b. Now Tb would be approximated as an average of the temperature at point grid node 1 or ghost node 0.

So Tb would be given as T1+T0/2 and we can easily use central difference approximation for derivative. So dT/dx at, let us call as x=xa, our left end, this would be approximated as T1-T0/delta x. So you need to just take care of these 2 definitions in deriving the expressions and the corresponding pseudo-code for incorporation of either Dirichlet boundary condition or Neumann or convective boundary conditions.

Similarly, a right end, let us redraw our domain. So we are at, now x=xb, this is our computational node for the index N, N+1 nth node becomes now our ghost node. Now what will be the temperature values at the boundary T at x=xb, this has to be approximated as TN+TN+1/2. Use CDS for derivative and all that it means is dT/dx at x=xb, this can be approximated as TN+1-TN/grid spacing delta x.

(Refer Slide Time: 34:17)



Ghost Node
N
N+1
$x = x_b$

At right end

$$T\Big|_{x=x_b} \simeq \frac{T_N + T_{N+1}}{2}$$

CDS for derivative

$$\left(\frac{dT}{dx}\right)\Big|_{x=x_b} \simeq \frac{T_{N+1} - T_N}{\Delta x}$$

general procedure for obtaining expression for boundary nodes

- Substitute for the value of T at ghost-node (ie T_0 at left end ~~and~~ and T_{N+1} at right) obtained from specified BCs into standard discrete eqn

$$\boxed{A_P T_i + A_W T_{i-1} + A_E T_{i+1} = B_i}$$

Exercise Obtained final expression for incorporation of BCs and implement them in our C++ code.

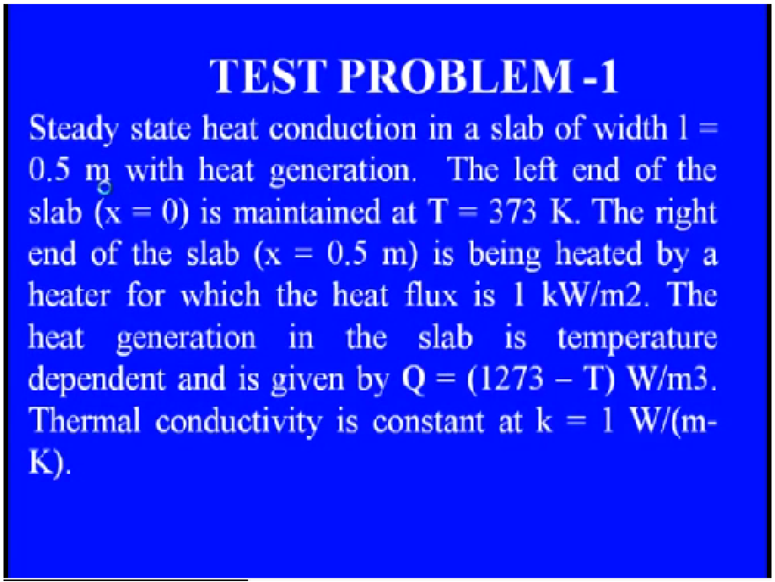
So what we will do, our general procedure would be, general procedure for obtaining expression or expressions for boundary nodes would be substitute for their value of T at ghost node. That is T0 at left end and TN+1 at right end, obtained from specified BCs, specified boundary conditions into standard discrete equation, APTi+AWTi-1+AETi+1=Bi, okay.

So at one end of Ti-1, left end, Ti-1 would become T0 eliminate or we have to get at the expression for T0 from either Tb=T1+T0/2, Dirichlet boundary conditions were specified or from the flux specifications, solve for T0, substitute that in this equation and get a modified discrete equation for that node. So I would leave the completion of this task as an exercise. So obtain final expressions for incorporation of BCs and put them and rather implement them in our C++ code.

Remember this C++ code would be available to you, what we are looking right now in the lecture on NPTEL site, so you can download it and do your modifications. I will just give you a brief look at the skeleton of what I have done, that BCCentred and this is exactly in the same way, this is a bit more elaborate compared to the standard format but the remaining code is exactly similar to what we had looked in detail for standard boundary condition.

Now this completes our code, okay and you can compile it and run it and test it for a certain set of problems. Now let us do a sample run and if you perform a sample run, our state will give you some results for 2 sample test problems.

(Refer Slide Time: 38:21)

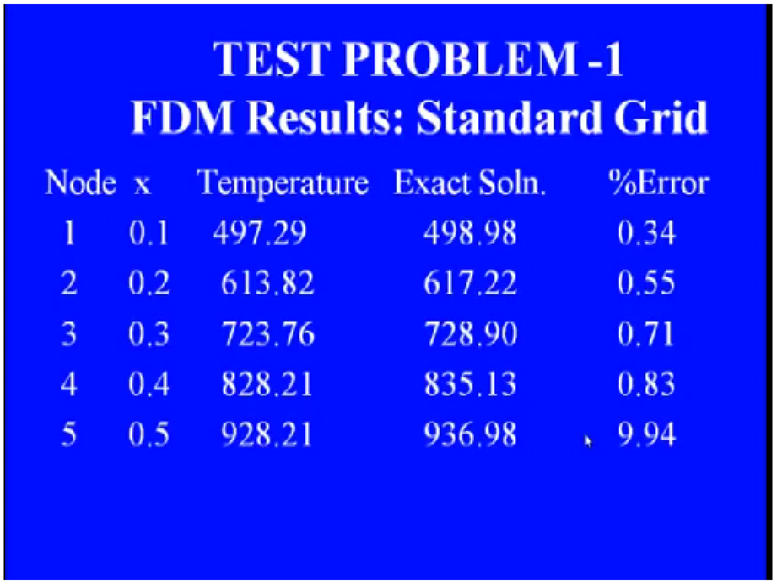


TEST PROBLEM -1

Steady state heat conduction in a slab of width l = 0.5 m with heat generation. The left end of the slab (x = 0) is maintained at T = 373 K. The right end of the slab (x = 0.5 m) is being heated by a heater for which the heat flux is 1 kW/m2. The heat generation in the slab is temperature dependent and is given by Q = (1273 – T) W/m3. Thermal conductivity is constant at k = 1 W/(m-K).

So the first test problem is the steady state heat conduction in slab of width 0.5 with temperature dependent heat generation that is q is 1273-T watt per meter cube and the left end of the slab is maintained at a T=373 Kelvin, right end of the slab is being heated by a heater for which the heat flux is 1 kilowatt per metre square and thermal conductivity is constant, it can be taken as K=1. So just input all these values in our code and this is what do we get for a standard grid by taking 5 grid divisions, okay.

(Refer Slide Time: 38:58)



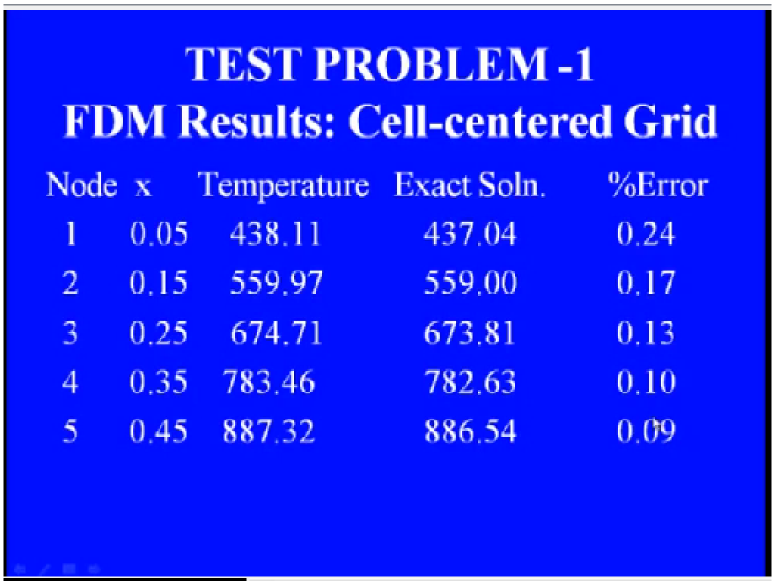
TEST PROBLEM -1
FDM Results: Standard Grid

Node	x	Temperature	Exact Soln.	%Error
1	0.1	497.29	498.98	0.34
2	0.2	613.82	617.22	0.55
3	0.3	723.76	728.90	0.71
4	0.4	828.21	835.13	0.83
5	0.5	928.21	936.98	9.94

The temperature values range from 497 to 928, exact solution for this problem can be easily obtained that is again left as an exercise to you, it is all decoded there is a part of the code, simple function is available which gives the exact solution and that our output routines also computes the percentage error based on the exact solution. So for this choice of the grid, percentage error ranges from 0.34-9.94.

Now you can observe one thing that error increases from node 1 to node 5 and the reason is close to this node, in fact the nodes 2 3 4 5 6, node 0 which is close to, x=0.1, we had temperature specified. So at this end, errors are less but at the right end where we had approximated our flux by the first-order backward difference scheme, we get fairly large value error.

(Refer Slide Time: 40:09)



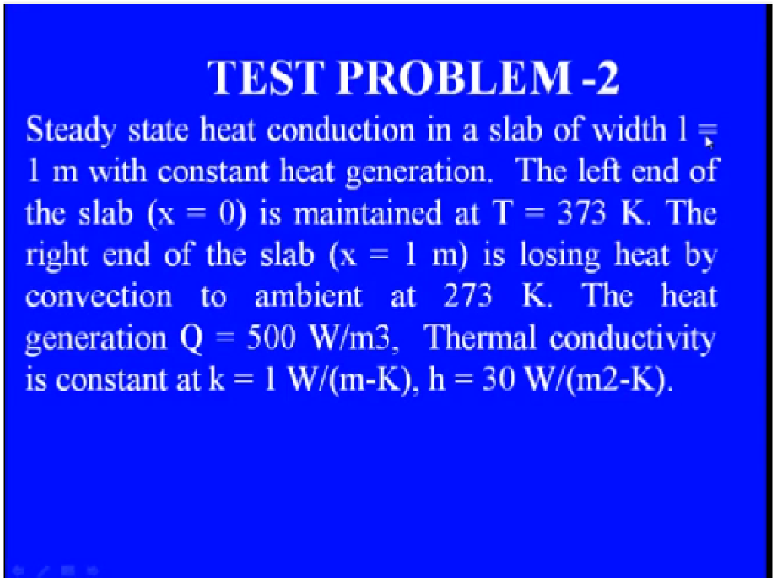
TEST PROBLEM -1
FDM Results: Cell-centered Grid

Node	x	Temperature	Exact Soln.	%Error
1	0.05	438.11	437.04	0.24
2	0.15	559.97	559.00	0.17
3	0.25	674.71	673.81	0.13
4	0.35	783.46	782.63	0.10
5	0.45	887.32	886.54	0.09

Now FDM results for the cell-centred grid, now the grid points are in the middle of the cell 0.05, 0.15, 0.25 and so on, okay. So you can easily say that errors are very small, they are much smaller compared to the solutions which we have obtained with standard finite difference grid and specifically prominent is this last figure, at close to the right boundary where the flux was specified.

Now we get error less than 0.1% and the reason for the sake of accuracy is that at the right end, flux has been approximated by a second order accurate central difference scheme.

(Refer Slide Time: 40:56)

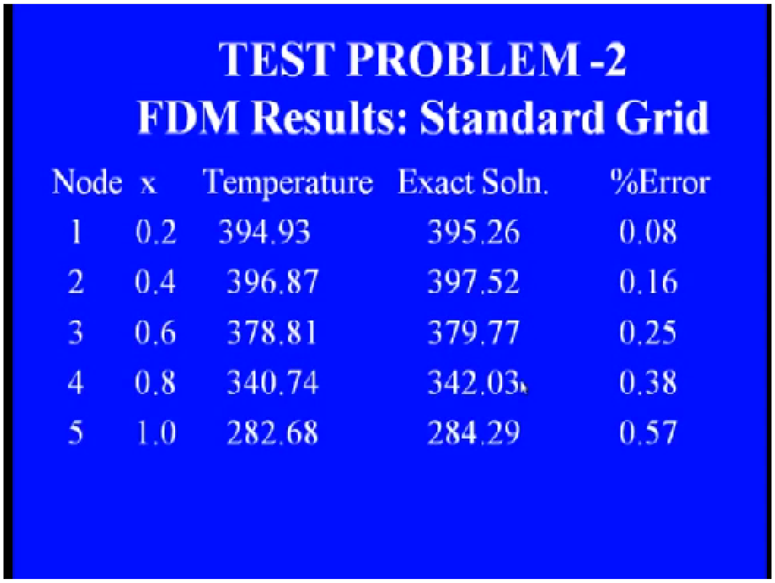


TEST PROBLEM -2

Steady state heat conduction in a slab of width l = 1 m with constant heat generation. The left end of the slab (x = 0) is maintained at T = 373 K. The right end of the slab (x = 1 m) is losing heat by convection to ambient at 273 K. The heat generation Q = 500 W/m3. Thermal conductivity is constant at k = 1 W/(m-K), h = 30 W/(m2-K).

Take another set of results which you can obtain, you can try run the code for this particular test case, a slab of width 1 m with constant heat generation. This is a test case for convective boundary condition. The left end of the slab is maintained at a T=373 degree and the right end is losing heat by convection to ambient at 273 Kelvin. The heat generation is a constant, 500 Watt per meter cube and the thermal conductivity can be taken as 1. The convective heat transfer coefficient, let us say we have taken 30.

(Refer Slide Time: 41:30)

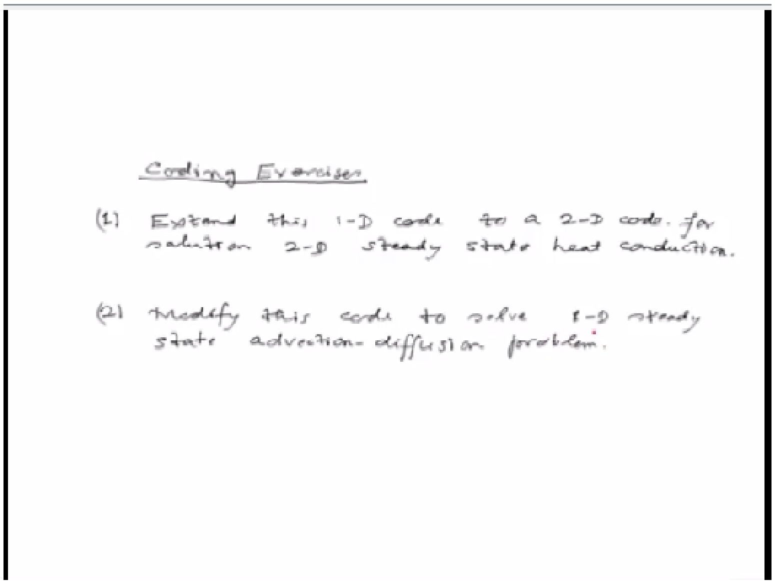


TEST PROBLEM -2
FDM Results: Standard Grid

Node	x	Temperature	Exact Soln.	%Error
1	0.2	394.93	395.26	0.08
2	0.4	396.87	397.52	0.16
3	0.6	378.81	379.77	0.25
4	0.8	340.74	342.03	0.38
5	1.0	282.68	284.29	0.57

And this is the result obtained with our standard grid. So now this is where we are now going to put our end to our discussions on finite difference but before we close, I would like to leave certain exercises for you.

(Refer Slide Time: 41:50)



Coding Exercises

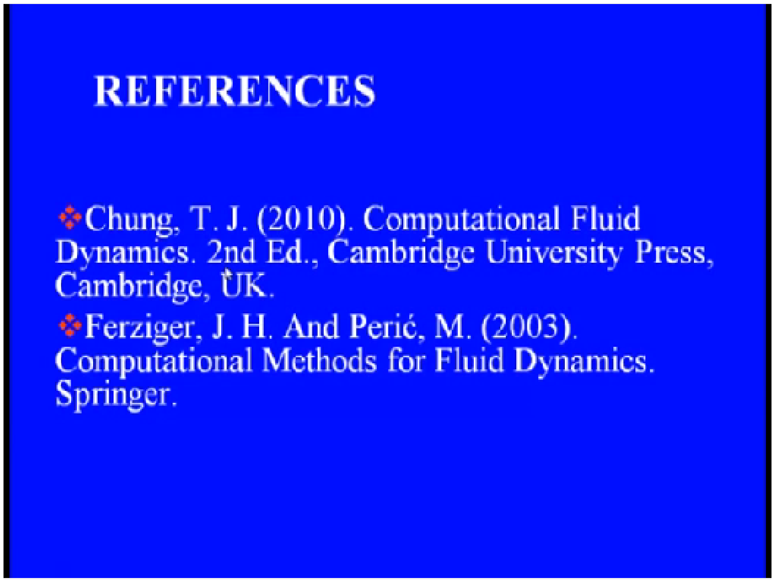
(1) Extend this 1-D code to a 2-D code. for solution 2-D steady state heat conduction.

(2) Modify this code to solve 1-D steady state advection-diffusion problem.

These exercises pertain to writing an actual implementation. So let me call them as coding exercises. Number first, extend 1-D code to a 2-D code. So design philosophy you can follow the same the way we have discussed but now the things will change, the data structures would change, the solvers would change and you can use or you can implement an iterative solver for solution of 2-D steady state heat conduction.

A simple exercise could be to modify this code to solve 1-D steady state advection diffusion problems. Now some of the solvers, we will discuss in the next module which you can make use of.

(Refer Slide Time: 43:41)



REFERENCES

- Chung, T. J. (2010). Computational Fluid Dynamics. 2nd Ed., Cambridge University Press, Cambridge, UK.
- Ferziger, J. H. And Perić, M. (2003). Computational Methods for Fluid Dynamics. Springer.

Now this broad references for what we have, you can get lot more material, many more finite difference algorithms and the approximation schemes in these 2 books, particularly Chung's book Computational Fluid Dynamics, it is a compendium of a huge collection of difference approximations together with finite volume and finite element methods for CFD applications and more readable accounts, much shorter account you can find in Ferziger's and Peric's book.

You can also try it another book on CFD, Introduction to Computational Fluid Dynamics by John D. Anderson which is a lot simpler to read than either of these 2 books. So for now this way we would put a full stop to our discussions on finite difference method and in next module, we will focus on the solution of discrete algebraic systems.