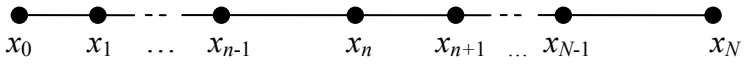
Numerical Solution of 2nd Order, Linear, ODEs.

We're still looking for solutions of the general 2nd order linear ODE

$$y''+p(x)\,y'+q(x)\,y = r(x)$$

with *p,q* and *r* depending on the independent variable. Numerical solutions can handle almost all varieties of these functions. Numerical solutions to second-order Initial Value (IV) problems can be solved by a variety of means, including Euler and Runge-Kutta methods, as discussed in §21.3 of your text. We won't discuss these applications here as we don't have many 2nd order IV problems in hydrology.

However, we have lots of 2nd order Boundary Value Problems (BVPs). In BVPs *x* usually represents space. For these situations we use *finite difference methods*, which employ Taylor Series approximations again, just like Euler methods for 1st order ODEs. Other methods, like the *finite element* (see Celia and Gray, 1992), *finite volume*, and *boundary integral element* methods are also used. The finite element method is the most common of these other methods in hydrology. You may also encounter the so-called "shooting method," discussed in Chap 9 of Gilat and Subramaniam's 2008 textbook (which you can safely ignore this semester). As most hydrological BVPs are solved with the finite difference method, that is where we'll focus our attention. For example, the popular groundwater flow code MODFLOW uses finite differences.



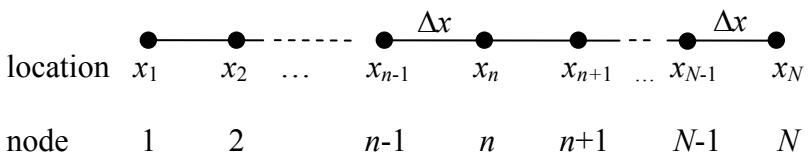
Recall the forward and backward (exact) Taylor expansions for *y(x)*. The *forward expansion* is

$$y(x_{n+1}) = y(x_n) + \Delta x_n\; y'(x_n) + \frac{\Delta x_n^{\,2}}{2!}\, y''(x_n) + \frac{\Delta x_n^{\,3}}{3!}\, y'''(x_n) + \ldots \qquad (1)$$

where step size, $\Delta x_n = x_{n+1} - x_n$. It projects forward in x, from x_n to x_{n+1}. The *backward expansion* is (this is shifted by one step, compared to the way we wrote it before)

$$y(x_{n-1}) = y(x_n) - \Delta x_{n-1}\; y'(x_n) + \frac{\Delta x_{n-1}^{\,2}}{2!}\, y''(x_n) - \frac{\Delta x_{n-1}^{\,3}}{3!}\, y'''(x_n) \pm \ldots \qquad (2)$$

We can manipulate these to get models for the first and second derivatives in a 2nd order BVP. Your textbook discusses this on pp. 910-911, but in the context of PDEs. The idea is just a generalization of the one we present here. It is easier to do this if the step size is uniform $\Delta x_n = \Delta x = h$ is a constant. We and your book make that assumption.



Notice that I've also relabled the first node point to be x_1, while your text insists on labeling the first point x_0. The sub-zero is a pain when indexing vectors and matrices in matlab and other programming languages. For IV problems, as in Text §21.1 this doesn't cause any problem, since these values or the gradient are known, and the solution propagates in one direction. But our 2nd order problems are (almost) all BV problems and, as we'll soon see, we have to solve simultaneous solutions using matrix algebra. Starting the index with the number 1 is very helpful, as the first row of the column vectors and the matrices that we derive will then correspond to the first node point, located at x_1, the nth row corresponds to the nth node point, and so on. (Aside: there is another way to handle this. We could have still started node 1 at location x_0, but labeled the node there as "node 1". Then the last node on the right side of the domain would still be "node N", located at, say, x_L. In other words we can be sophisticated and define an indexing map between node numbers and node locations. At the moment I prefer to take advantage of being able to equate location and node numbers.)

Besides forward and backward differences how else can we approximate the first derivative at a point x_n in a BVP? Instead of using either (1) or (2) to build a model for the first derivative we can use an average of both. That is we subtract (2) from (1) and solve for the first derivative. This eventually leads to the so-called *center difference* numerical approximation of $y'(x_n)$. To get the second derivative expansion of $y''(x_n)$ we can add (1) and (2). We'll do this below, first by looking at exact Taylor expansions, and then their finite difference approximations. From these results you can investigate the truncation error of the numerical approximations to each expansion.

Exact Taylor Series Expansions for 1st and 2nd Derivatives, and the ODE

Rewriting (1) and (2) for constant step size we have

$$y(x_{n+1}) = y(x_n) + \Delta x\ y'(x_n) + \frac{\Delta x^2}{2!} y''(x_n) + \frac{\Delta x^3}{3!} y'''(x_n) + \ldots \tag{3}$$

$$y(x_{n-1}) = y(x_n) - \Delta x\ y'(x_n) + \frac{\Delta x^2}{2!} y''(x_n) - \frac{\Delta x^3}{3!} y'''(x_n) \pm \ldots \tag{4}$$

where (3) is the forward expansion and (4) is the backward expansion, and we've assumed a constant step size. Each of these can be solved for the derivative $y'(x_n)$, as we previously did on pages 97-99 of these notes. The forward (6) and backward (15) (equation numbers from pp. 97-99) expansions for the derivatives were

$$y'(x_n) = \frac{y(x_{n+1}) - y(x_n)}{\Delta x} + \frac{1}{\Delta x}\left[-\frac{\Delta x^2}{2!} y''(x_n) - \frac{\Delta x^3}{3!} y'''(x_n) - \ldots \right] \tag{5}$$

$$y'(x_n) = \frac{y(x_n) - y(x_{n-1})}{\Delta x} + \frac{1}{\Delta x}\left[+ \frac{\Delta x^2}{2!} y''(x_n) - \frac{\Delta x^3}{3!} y'''(x_n) \pm \ldots \right] \tag{6}$$

In the second equation we've shifted the backward difference by one step to get derivatives at x_n.

We can also get a central expansion for $y'(x_n)$. Subtracting (4) from (3) cancels out the even derivative terms and yields

$$y(x_{n+1}) - y(x_{n-1}) = 2\Delta x\ y'(x_n) + \frac{\Delta x^3}{3!} y'''(x_n) + \frac{\Delta x^5}{5!} y'''''(x_n) + \ldots \tag{7}$$

which we then inverse to solve for an exact value of $y'(x_n)$ in terms of these two Taylor Series, or

$$y'(x_n) = \frac{y(x_{n+1}) - y(x_{n-1})}{2\Delta x} - \frac{1}{2\Delta x}\left[+ \frac{\Delta x^3}{3!} y'''(x_n) + \frac{\Delta x^5}{5!} y'''''(x_n) + \ldots \right] \tag{8}$$

In (5), (6), and (8) we have three (exact) expansions for the first derivative at x_n. If we truncate these expressions we get numerical approximations to the first derivatives. Note the relative truncation error that will result, by comparing (8) to (5) and (6). The truncated central expansion is higher order than truncated forward or backward expansions.

For 2nd order ODEs we need a similar expansion for the 2nd derivative. We get that by adding equations (3) and (4), which cancels out the odd derivative terms, and yields

$$y(x_{n+1}) + y(x_{n-1}) = 2y(x_n) + 2\frac{\Delta x^2}{2!} y''(x_n) + 2\frac{\Delta x^4}{4!} y''''(x_n) + \ldots \tag{9}$$

We solve for the second derivative to get

$$y''(x_n) = \frac{y(x_{n+1}) - 2y(x_n) + y(x_{n-1})}{\Delta x^2} - \frac{1}{\Delta x^2}\left[\frac{\Delta x^4}{4!} y''''(x_n) + \ldots \right] \tag{9}$$

We now have expansions for both first and second derivatives and can substitute them into the ODE.

We're looking at 2nd order ODEs of the form

$$y'' + p(x)\, y' + q(x)\, y = r(x) \tag{10}$$

If we examine this equation at the point x_n in our domain, we rewrite (10) as

$$y''(x_n) + p(x_n)\, y'(x_n) + q(x_n)\, y(x_n) = r(x_n) \tag{11}$$

Then when we substitute our Taylor Series expansions, we still have an exact expression.

The only decision at this point is which of the three expansions for the first derivative to substitute. In practice all three are used, depending on the problem, but most often it is either the backward 6) or central (8) expansion that is used. The choice has to do with numerical truncation and stability issues that we encounter when truncating the expansions and solving the resulting finite difference equations numerically. In commercial codes it is often a user decision, so you need to be aware of this option.

Let's use the central expansion. Then (11) becomes

$$\frac{y(x_{n+1})-2y(x_n)+y(x_{n-1})}{\Delta x^2}+p(x_n)\frac{y(x_{n+1})-y(x_{n-1})}{2\Delta x}+q(x_n)y(x_n)=r(x_n)+\left[\frac{\Delta x^2}{4!}y''''(x_n)+\ldots\right]+\left[\frac{\Delta x^2}{2\cdot 3!}y'''(x_n)+\ldots\right] \quad (12)$$

We could also gather like terms together, and rewrite this as

$$\left[\frac{1}{\Delta x^2}-\frac{p(x_n)}{2\Delta x}\right]y(x_{n-1})+\left[\frac{-2}{\Delta x^2}+q(x_n)\right]y(x_n)+\left[\frac{1}{\Delta x^2}+\frac{p(x_n)}{2\Delta x}\right]y(x_{n+1})=r(x_n)+\left[\frac{\Delta x^2}{4!}y''''(x_n)+\ldots\right]+\left[\frac{\Delta x^2}{2\cdot 3!}y'''(x_n)+\ldots\right] \quad (13)$$

Note that besides the forcing term *r,* the right hand side contains only higher order terms in the expansions. These are the terms we'll neglect in our finite difference approximation and they define the local truncation error. As you can see, using centered expansions for both the first and second derivatives results in higher order terms of order Δx^2. (If the step size were to vary in x, then the truncation error is not as good. It's of order Δx.)

Finite Difference Approximations

We can truncate all of our expansions and write finite difference approximations, indicating the order of the approximation (truncation error). Then the exact value $y(x_n)$ is replaced by its numerical approximation y_n, and the derivatives are replaced by their truncated expansions. We did this once before, for the forward and backward difference approximations to the first derivative when discussing the Euler method. In any case, here are the results for the present case, taken from truncating (5), (6), (8), and replacing the exact values of y with their numerical equivalents.

For the first derivative approximations:

Forward difference $$y'(x_n)\cong\frac{y_{n+1}-y_n}{\Delta x}+\mathrm{O}(\Delta x) \quad (14a)$$

Backward difference $$y'(x_n)\cong\frac{y_n-y_{n-1}}{\Delta x}+\mathrm{O}(\Delta x) \quad (14b)$$

Centered difference $$y'(x_n) \cong \frac{y_{n+1} - y_{n-1}}{2\Delta x} + \mathrm{O}(\Delta x^2) \qquad (14c)$$

For the second derivative approximaton:

$$y''(x_n) \cong \frac{y_{n+1} - 2y_n + y_{n-1}}{\Delta x^2} + \mathrm{O}(\Delta x^2) \qquad (15)$$

For the 2nd order ODE, $y''+p(x)\,y'+q(x)\,y = r(x)$, using the centered difference (14c) for the y' term we get (from truncating (12) and (13), respectively):

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{\Delta x^2} + p(x_n)\frac{y_{n+1} - y_{n-1}}{2\Delta x} + q(x_n)y_n = r(x_n) + \mathrm{O}(\Delta x^2) \qquad (16)$$

or

$$\left[\frac{1}{\Delta x^2} - \frac{p(x_n)}{2\Delta x}\right]y_{n-1} + \left[\frac{-2}{\Delta x^2} + q(x_n)\right]y_n + \left[\frac{1}{\Delta x^2} + \frac{p(x_n)}{2\Delta x}\right]y_{n+1} = r(x_n) + \mathrm{O}(\Delta x^2) \qquad (17)$$

Some people prefer to write the difference version of the ODE as in (16) and others as in (17). There is also a difference in programming styles that probably leads to these biases, as we'll see later when writing pseudocode. Also, many people prefer to multiply (16) and (17) by Δx^2. It leads to a cleaner product. We won't do that here.

There are N equations (16) or (17), one for each node. When solving these equations we drop the truncation error terms [terms of $\mathrm{O}(\Delta x^2)$], which are only used theoretically to give us some idea of the error. Also observe that each equation is an algebraic equation. By invoking the finite difference approximations we've reduced a single ODE to a system of N algebraic equations, which are presumably easier to solve.

Since each node in the interior of the domain has two neighbors, one to the right, and one to the left, each algebraic equation must be solved together with its neighbors. This chains all of the nodes together and the N equations must be solved simultaneously. It is important to grasp that this connection is omnidirectional, unlike the Euler method for IV problems which propagates in one direction. Because we have simultaneous algebraic equations we can use *matrix algebra* to solve them.

The terms in brackets on the left side of (17) become the entries along the diagonal (middle term), next lower diagonal (left term), and next upper diagonal (right term) of a *tridiagonal coefficient matrix*, **A** (see below). The forcing or load, $r(x_n)$, on the right goes into a column *load vector*, **b**. The unknowns, y_n, go into a column *vector* **y** *of unknowns*.

The N simultaneous equations (17), one equation for each node (n=1, 2, …, N-1, N), are then written as

$$\mathbf{A}\mathbf{y} = \mathbf{b} \tag{18}$$

and solved for **y** (and thus the nodal y's) using matrix *equation solvers* like Gaussian Elimination, LU factorization, or some other direct matrix solution technique, or by iteration (we'll talk about this later). Since there are N node points the load vector **b** is Nx1,

$$\mathbf{b} = \{r_1, r_2, \ldots r_{n-1}, r_n, r_{n+1}, \ldots, r_{N-1}, r_N\}^{\mathrm{T}}. \tag{19}$$

where $r_n = r(x_n)$. The column vector of unknowns (often called the *state vector*) is also Nx1,

$$\mathbf{y} = \{y_1, y_2, \ldots y_{n-1}, y_n, y_{n+1}, \ldots, y_{N-1}, y_N\}^{\mathrm{T}}. \tag{20}$$

The square coefficient matrix **A** is NxN. If $a_{i,j}$ is the ith row and jth column entry of the matrix **A**, then the three non-zero diagonals along a row are

$$a_{n,n-1} = \left[\frac{1}{\Delta x^2} - \frac{p(x_n)}{2\Delta x}\right] \tag{21a}$$

$$a_{n,n} = \left[\frac{-2}{\Delta x^2} + q(x_n)\right] \tag{21b}$$

$$a_{n,n+1} = \left[\frac{1}{\Delta x^2} + \frac{p(x_n)}{2\Delta x}\right] \tag{21c}$$

while all other enties $a_{i,j}$ are zero. For an example 5-node domain (N=5), the general version of matrix **A** is

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix} \tag{22}$$

For this example, we can then write out (18) as

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} \tag{23}$$

In practice we alter non-zero values of $a_{i,j}$ (and b_j) in the first row or two rows, and the last row or two rows, to account for the boundary conditions, one at x_1 and the other at x_N. The nature of the alteration depends on the type of boundary at each end. We'll see more about boundary conditions below.

Notice that if the ODE (10) has a first derivative term (i.e., $p \neq 0$), then the matrix **A** is not symmetric ($a_{i,j} \neq a_{j,i}$ for the first two off-diagonals, $j=i$-1). If p=0 then the matrix is symmetric ($a_{i,j} = a_{j,i}$).. Equation solvers can (and should) take advantage of this symmetry (less computer storage and computational effort).

Equation solvers can also take advantage of sparseness. For example, the tridiagonal form of **A** (e.g., see (23)) leads to an especially efficient and oft-used Gaussian Elimination equation solver called the *Thomas Algorithm*.

If we approximate the first derivative with the backward difference (14b), then (17) would become, instead

$$\left[\frac{1}{\Delta x^2} - \frac{p(x_n)}{\Delta x}\right] y_{n-1} + \left[\frac{-2}{\Delta x^2} + \frac{p(x_n)}{\Delta x} + q(x_n)\right] y_n + \left[\frac{1}{\Delta x^2}\right] y_{n+1} = r(x_n) + \mathrm{O}(\Delta x^2) \quad (24)$$

This alters the entries in matrix **A** if p is non-zero and, while different, **A** remains non-symmetric for this case:

$$a_{n,n-1} = \left[\frac{1}{\Delta x^2} - \frac{p(x_n)}{\Delta x}\right] \quad (25a)$$

$$a_{n,n} = \left[\frac{-2}{\Delta x^2} + \frac{p(x_n)}{\Delta x} + q(x_n)\right] \quad (25b)$$

$$a_{n,n+1} = \left[\frac{1}{\Delta x^2}\right] \quad (25c)$$

The non-zero p first-derivative term is used to represent advection in heat and mass transport problems, while the second derivative is used to represent diffusion. The zero order q term is used to represent mass or heat transfer to another media (e.g., from water to sediments), or 1st order decay. Notice that in this application it is the presence of advection that leads to non-symmetry of matrix **A**.

Example 1:

For example, consider the steady-state solute transport model with advection, diffusion and decay,

$$v\frac{\partial C}{\partial x} = D\frac{\partial^2 C}{\partial x^2} - \alpha C \quad (a)$$

where $C(x)$ is concentration, v is advective velocity of a fluid, D is a diffusion (or dispersion) coefficient, and α is a first-order decay constant. I've used partial derivatives for a reason that will be clear below. Rewritten in standard form we have

$$\frac{\partial^2 C}{\partial x^2} - \frac{v}{D}\frac{\partial C}{\partial x} - \frac{\alpha}{D}C = 0 \qquad \text{(b)}$$

with $y=C$, $x=x$, $p= -v/D$, $= -\alpha/D$ and $r=0$, in the generic form $y''+p(x)\,y'+q(x)\,y = r(x)$.

If we ignore diffusion, then the problem reduces to first order. While we could write a code to solve second order equation (b) numerically, and then run it with D=0, this would not be a good idea. The 2nd order ODE finite difference method derived here embraces smoothness, but 1st-order ODE advection-only problems preserve sharp fronts. The finite difference method in (17) or (24) smoothes those fronts, whether diffusion is present or not. Use a 1st order method instead, like the Euler method. What about problems with diffusion, but that are still dominated by advection? One then lives with the resulting non-realistic numerical smoothing, or adopts a better scheme. This is one reason for choosing (24) over (17). If the flow is from left to right, using the backward difference on the advection term helps to address this problem. It's called "upwinding" or "upstream" weighting in more advanced methods. If the flow were from right-to-left one would use a forward difference instead. It's determined by how the grid is oriented to the flow. The term upwinding implies one goes toward the direction from which the "wind" is blowing. Another application is vadose zone flow. It's gravity that is causing the non-zero p term, while capillarity causes "diffusion". In this case "upwind" is upward. While I've raised this issue here it is mainly of importance in solving PDEs, such as (a) with $C(x,t)$, and an additional solute storage (accumulation) term, $\partial C/\partial t$ on the left hand side (see eqn. (b) on p. 120 of these notes). Then one chooses between both spatial and temporal forward, backward or centered difference schemes to manage the problem, with tradeoffs with each combination.

If we ignore advection (or there is none), then p=0 and the finite difference method in (17) or (24) works wonderfully. This is the case with heat conduction, solute diffusion, and groundwater flow, explaining the common use of the finite difference method to solve these problems.

Boundary Conditions:

On page 122 of these notes we talked about boundary conditions for 2nd order BVPs. Over a finite interval, $a \le x \le b$, there are two BCs, one at each end of the interval. In general, the boundary conditions can be written as

$$\begin{aligned} k_1\ y(a) + k_2 y'(a) &= K_a \\ l_1\ y(b) + l_2 y'(b) &= K_b \end{aligned} \qquad \text{(26a,b)}$$

where the k's and l's are parameters, weighing the relative contribution of y and y' at each boundary, and the K's are values. Recall the *first type* or *Dirichlet boundary*. For example, if you know the value of y at a, then k_1 =1, k_2 =0, and K_a is the value $y(a)$. Then there is the *second type* or *Neumann boundary.* Say, if you know the gradient of y at b, then l_1 =0, l_2 =1, and K_b is the value $y'(b)$. Both k_1 and k_2 are non-zero at a *third-type* or *Cauchy boundary*. We can implement all three boundary condition types in the finite difference method. We'll focus only on 1st and 2nd type boundary conditions. In practice, for big problems, it is preferred to apply 2nd type BCs before 1st type BCs. However, it is easier to learn by applying 1st type first.

KEY POINT: Finally, a point not made previously that applies to all solutions of 2nd order ODEs, *you need at least one 1st or 3rd type BC*. If you make both BVP BCs of 2nd type, you won't get a unique solution (try it with a problem you can solve by the method of constant coefficients). In numerical methods the computer will pick one for you, but it will be arbitrary. This can be embarrassing.

Dirichlet BCs. There are several ways to handle 1st type boundary conditions. The simplest and most elegant method (but not the cheapest method for complex spatial domains and PDEs) is to simply *partition matrix* **A**, so that the nodes with known heads are no longer in the vector of unknowns, and matrix A is reduced in size by the number of nodes with known values of y. We have to explain this by example. Suppose that at the first node, located at x_1, the value of y is known. Then we know exactly that $y_1 = y(x_1)$, and we don't have to compute it. We can ignore the first row of our 5-node example in (23). However, y_2 depends on y_1 and we have to respect that. The second row of the computation in the 5-node matrix equation (23) is

$$a_{21} y_1 + a_{22} y_1 + a_{23} y_3 = b_2 \tag{27}$$

However, we know y_1. We can move it to the right side, creating a new entry in the load term for this 2nd node point:

$$a_{22} y_1 + a_{23} y_3 = b_2 - a_{21} y_1 \tag{28}$$

We *partition* the matrix A in our example 5-node matrix, by removing the first row and first column, yielding a new, smaller, but still square matrix, $\mathbf{A_{22}}$, or

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix} = \left[\begin{array}{c|cccc} a_{11} & a_{12} & 0 & 0 & 0 \\ \hline a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{array}\right] = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix} \tag{29}$$

4x4 subpartition

where

$$\mathbf{A_{22}} = \begin{bmatrix} a_{22} & a_{23} & 0 & 0 \\ a_{32} & a_{33} & a_{34} & 0 \\ 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & a_{54} & a_{55} \end{bmatrix} \tag{30}$$

We make similar adjustments to $\mathbf{b} = \{\mathbf{b_1}, \mathbf{b_2}\}^T$, where $\mathbf{b_1}=\{b_1\}^T$ and $\mathbf{b_2}=\{b_2-a_{21}y_1, b_3, b_4, b_5\}^T$. Then the 5-node matrix problem becomes

$$\begin{bmatrix} a_{22} & a_{23} & 0 & 0 \\ a_{32} & a_{33} & a_{34} & 0 \\ 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & a_{54} & a_{55} \end{bmatrix} \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} b_2 - a_{21} y_1 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} \tag{31}$$

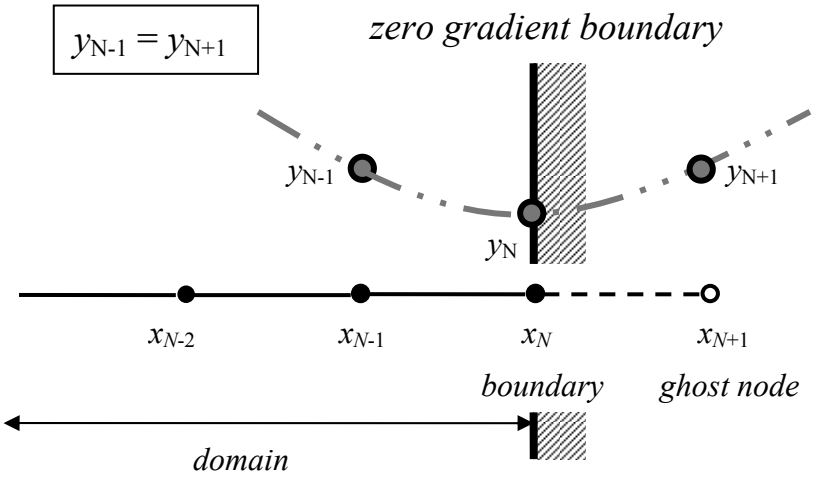
In general,

$$\mathbf{A_{22}}\ \mathbf{y_2} = \mathbf{b_2} \tag{32}$$

where, in our example, $\mathbf{y_2} = \{y_2, y_3, y_4, y_5\}^T$ is the vector of unknowns, having removed them from the vector **y** which contained both unknowns and the known (BC) values of y. Note that this approach works exactly the same way if the other (x_N) BC is also first type. We would partition again, ending up with a 3x3 partitioned matrix, which we would still call matrix $\mathbf{A_{22}}$, and a new 3x1 load vector, still called $\mathbf{b_2}$, but now $\mathbf{b_2} = \{b_2 - a_{21}y_1,\ b_3,\ b_4 - a_{45}y_5\}^T$. In fact, this way of handling 1st type BCs applies to the spatial portion of PDE numerical solutions, whether 1D, 2D, or 3D in space, to any number of 1st type nodes, and to many different numerical methods (e.g., finite difference, finite volume, finite element, boundary integral element methods).

Newmann BCs. There are also several ways to handle 2nd type boundary conditions, and their application depends on the method of derivation of the finite difference approach. Our finite difference approach is called the *point-centered finite difference method*. The typical way to handle 2nd type boundary conditions in this approach is to use *ghost nodes*. These are pseudo nodes that lie outside the domain, along 2nd type BCs, and are used to control the gradient, y', at the boundary. In effect, we assume that the same ODE applies outside of the domain, across the second type boundary, for a distance of Δx.

For example, consider a 2nd type zero-gradient boundary at the node located at x_N . A sketch of this boundary and the solution might look like:



To maintain a zero gradient for uniform grid spacing, the value of y at the ghost node at x_{N+1}. is set equal to the value of y at the first interior node, located at x_{N-1}, or $y_{N+1} = y_{N-1}$. If the 2nd type

BC involves a known non-zero gradient $y'(x_N)$ we then apply the central difference (14c) to preserve it:

$$y_{N+1} = y_{N-1} + 2\,\Delta x\, y'(x_N) \qquad (33)$$

As you can see it is just a generalization of the zero gradient boundary. We now need to add a ghost node to our original problem. Let's proceed with our 5-node example, where node 1 is a 1st type node that we've already addressed. The resulting matrix equation is (31). Now add a 2nd type BC at x_N of the type shown in (33). We do that by conceptually adding (padding) an additional row and column to the matrix and vectors, leading to

$$\begin{bmatrix} a_{22} & a_{23} & 0 & 0 & 0 \\ a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} b_2 - a_{12} y_1 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix} \qquad (34)$$

where the subscript 6=N+1 refers to the ghost node and, according to (33), $y_6 = y_4 + 2\Delta x\, y'(x_N)$. The values of $a_{5,6}$, $a_{6,5}$, and $a_{6,6}$ come from (21) or (25), depending on the choice of first order difference. Ignore the last line. Instead, consider the second to last line, representing computations at node N=5.

$$a_{54} y_4 + a_{55} y_5 + a_{56} y_6 = b_5 \qquad (35)$$

Substitute the boundary expression (33) for y_6.

$$a_{54} y_4 + a_{55} y_5 + a_{56} y_4 + 2\, a_{56}\, \Delta x\, y'(x_N) = b_5 \qquad (36a)$$

or

$$[a_{54} + a_{56}]\, y_4 + a_{55} y_5 = b_5 - 2\, a_{56}\, \Delta x\, y'(x_N) \qquad (36b)$$

The problem reduces back to that of four unknowns = $\mathbf{y_2} = \{y_2, y_3, y_4, y_5\}^T$, with a further revised load vector and a revised matrix.

$$\begin{bmatrix} a_{22} & a_{23} & 0 & 0 \\ a_{32} & a_{33} & a_{34} & 0 \\ 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & a_{54} + a_{56} & a_{55} \end{bmatrix} \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} b_2 - a_{21} y_1 \\ b_3 \\ b_4 \\ b_5 - 2 a_{56}\, \Delta x\, y'(x_N) \end{bmatrix} \qquad (37)$$

Note that if the matrix **A** was symmetric before, it is not now. Also, you may wonder where to get the value of a_{56} since it appears to describe conditions just outside the domain. But if you inspect (21c) or (25c) you'll see that it depends only on the step size and, for the p-term central difference, on the value of p at node 5, which is on the boundary and is defined.

Aside: There is another major approach to finite differences, involving describing blocks first, and locating node points at the center of blocks. This *block-centered finite difference* approach is use by MODFLOW and most oil-reservoir simulators. This method treats 2nd type BCs differently. They don't need ghost nodes. In that approach zero-gradient 2nd type BCs are *natural boundary conditions.* That is, without telling the code anything else it assumes that the default boundary is zero gradient. This occurs at the edge of a gridblock. The nearest node to the boundary is half a block away. If you want to have a finite gradient across such a natural boundary you apply a source/sink term at that node. Such natural BCs are part of the algorithm, and not something you have to program in. The nearest thing to a natural boundary in a point-centered scheme, like that discussed here, is a 1st type BC.

Pseudo Code:

The basic structure a finite difference code for solving 2nd order ODEs by the finite difference method is

INPUT:	The step size, number of steps, BC1 (type and value), BC2, and parameters p, q, and r. Note 1: the length of the domain is often used instead of the step size or number of steps: length =step size x number of steps = Δx (N-1). Input two of these and calculate the third. Note 2: since the presence of $p \neq 0$ can affect storage and solution, a flag indicating the absence of that term, or simply a flag set by noting that it is zero, is usually included. Some codes simply require that p=0.
BUILD **A**:	Fill or build the coefficient matrix **A**.
BUILD **b**:	Fill or build the load vector **b**.
SOLVE:	Solve **Ay**=**b** for the vector **y**.
ANCILLARY:	Perform *backsubstitution* to find fluxes at 1st type BCs, check mass balance, and other ancillary calculations
OUTPUT:	Output **y** and fluxes
POSTPROCESS:	Plot y_n, n = 1 to N, and as needed, fluxes.

This is the basic organization of most finite difference codes. A principle difference between codes is in building **A** and **b**, and in the solution. Many codes contain a library of solution methods (later). As for building **A** and **b** the two main approaches are illustrated by the two ways of writing the ODE difference approximation in (16) and in (17). Each would have its own, different pseudo code.

The code related to (16) looks at each term in the equation, and builds **A** and **b** in sequential steps. A crude pseudo code for building **A** and **b** is:

Matrix Size: First you might note the presence of 1st type BCs, and aim the final matrix size accordingly. For a one-dimensional ODE the matrix is of size NxN with two 3rd type BCs, (N-1xN-1) with one 1st type BC, and (N-2xN-2) with two 1st type BCs. You would note this and partition the matrix from the start.

Loop over rows:

Then row by row (or node by node) you would build in the contributions from the first term on the left side of (16). For example, on row n you would add $+1/\Delta x^2$ to $a_{n,n-1}$ and $-2/\Delta x^2$ to $a_{n,n}$, the diagonal.

Next you would cycle through the rows again and add in the contributions from the second term on the left of (16). Using $a_{n,n-1}$ as the example for row n, you would add $-p(x_n)/2\Delta x$ to $a_{n,n-1}$ and 0 to the diagonal.

Next would come the third term on the left of (16). You would add $q(x_n)$ to the diagonal.

Finally, you would add the loads $r(x_n)$ to the nth row of load vector **b**.

At this point you would have built the basic versions of **A** and **b**, but without accounting for the BCs.

Loop over BCs, types and locations

Next, you would adjust the appropriate (first or last) row of **A** and **b** to account for 2nd or 3rd type BCs.

Finally, again, you would insert the adjustment to the appropriate (first or last) row of **b** to account for 1st type BCs.

While the main pseudo code separates building A and b, in this version those operations merge when it comes to BCs, although they don't have to.

This approach to building **A** and **b** is typical of block-centered codes. It is very powerful when dealing with complicated, multidimensional, coupled processes problems with complex boundaries.

Point centered codes typically use a fill method instead, more along the lines of (17). These codes are typically restricted to simpler problems. A pseudo code for filling **A** and **b** might be:

Matrix Size: First note the presence of 1st type BCs, and aim the final matrix size accordingly. For a one-dimensional ODE the matrix is of size NxN with two 3rd type BCs, (N-1xN-1) with one 1st type BC, and (N-2xN-2) with

two 1st type BCs. You would note this and partition the matrix from the start. The indices used below are for this reduced matrix.

For all interior nodes loop over the rows, filling the diagonals of **A** and the entry of **b**:

Fill $a_{n,n-1}$ from (21a)
Fill $a_{n,n}$ from (21b)
Fill $a_{n,n+1}$ from (21c)
} This is often done by defining three vectors, each representing the entries in one of the diagonals.
Fill $b_n = r(x_n)$

At x_1 adjust b_1 and if needed $a_{1,2}$ for the BC.

At x_N adjust b_N and if needed $a_{N-1,N}$ for the BC.

Example 2:
Consider essentially horizontal, steady groundwater flow in a leaky, confined aquifer, modeled by:

ODE
$$\frac{d}{dx}\left[T\frac{d\phi}{dx}\right]-\frac{K'}{B'}\phi=-\frac{K'}{B'}\phi_0-Q_w' \qquad (38a)$$

BC1
$$\left.\frac{d\phi}{dx}\right|_{x=0}=0 \qquad \text{BC2} \quad \phi\big|_{x=L}=H \qquad (38b,c)$$

where $\phi(x)$ is hydraulic head, $T(x)$ is spatially distributed transmissivity, K'/B' is a spatially distributed leakage coefficient, $\phi_0(x)$ is the known, spatially varying head in the overlying phreatic aquifer, and $Q'_w(x)$ is spatially distributed pumping (sign: positive for injection, negative for pumping). BC1 represents a no-flow boundary, BC2 represents a fully-penetrating known head of value H (a lake). Rewrite the ODE in standard form[7].

$$\frac{d^2\phi}{dx^2}+\frac{1}{T}\frac{dT}{dx}\frac{d\phi}{dx}-\frac{K'}{TB'}\phi=-\frac{K'}{TB'}\phi_0-\frac{Q_w'}{T} \qquad (39)$$

In terms that we are using in this course, where the general ODE is written as $y''+p(x)\,y'+q(x)\,y=r(x)$, our unknown and parameters are

$y(x)=\phi$, (40a)
$p(x)=d\ln T/dx$, (40b)
$q(x)=-K'/TB'$, and (40c)
$r(x)=-(K'/TB')\phi_0 - Q'_w/T = q(x)\,\phi_0 - Q'_w/T$ (40d)

[7] In general, it is not a good idea in finite difference methods to break up the 2nd derivative into a first and second order terms, for spatially variable coefficient T, as I have done here. It is better, and common, to develop a 2nd order difference operator that explicitly handles this variation. Such issues occur in a wide variety of diffusion-like problems. Why? Recall that 2nd derivative approximation of finite differences (in space) smear solutions, and breaking it up this way leads to additional error. Also note the derivative of T, which itself represents a problem if T is not a continuous function of x.

For this problem, which is basically diffusive, we should use the central difference approximations of (21). Thus, for interior node points,

$$a_{n,n\text{-}1} = \left[\frac{1}{\Delta x^2} - \frac{p(x_n)}{2\Delta x}\right] \tag{41a}$$

$$a_{n,n} = \left[\frac{-2}{\Delta x^2} + q(x_n)\right] \tag{41b}$$

$$a_{n,n+1} = \left[\frac{1}{\Delta x^2} + \frac{p(x_n)}{2\Delta x}\right] \tag{41c}$$

$$b_n = r(x_n) \tag{41d}$$

where *p, q,* and *r* are defined in (40). We can then use a ghost node at x_{-1} and partition the matrix at x_N , respectively, to represent BC1 and BC2.

Example 3:
For simplicity, lets assume *T*=constant, *K'/B'* = constant, and ϕ_0 = constant. We'll let the pumping vary with *x*. Then *p*=0, *q*= constant, and *r*=constant + fcn. of *x*, and the general equation is of type $y'' + qy = r(x)$ This simplifies the entries for the interior nodes, *n*=2, 3, …, *N*-2, *N*-1:

$$a_{n,n\text{-}1} = a_{n,n+1} = \left[\frac{1}{\Delta x^2}\right] \tag{42a}$$

$$a_{n,n} = \left[\frac{-2}{\Delta x^2} + q\right] \tag{42b}$$

$$b_n = q\,\phi_0 - Q'_w(x)/T \tag{42c}$$

At node 1 we need to account for the no-flow (zero-gradient) boundary condition.

$$a_{1,2} = \left[\frac{2}{\Delta x^2}\right] \tag{43a}$$

$$b_{1,\text{new}} = -\frac{2}{\Delta x}\phi'(x_1) + q\,\phi_0 - Q'_w(x_1)/T = q\,\phi_0 - Q'_w(x_1)/T \tag{43b}$$

The last equation simplifies since the gradient is zero at the boundary. At node *N-1*, the last node with an unknown head in this model, we need to revise the load vector.

$$b_{5,\text{new}} = -\frac{1}{\Delta x^2}H + q\,\phi_0 - Q'_w(x_N)/T \tag{44}$$

where *H* is the known head at x_N.

Example 4:
Let's simplify further, by assuming a six node model, including the known head, H. That leaves 6-1 =5 unknown heads. Let's also assume that the pumping only takes place from node 3, located at $x = 2\Delta x$. Finally, let's multiply through by Δx^2 to get rid of those pesky denominators. The interior node, n=2, 3, 4, 5, contributions to **A** are

$$a_{n,n-1} = a_{n,n+1} = 1 \quad (45a)$$
$$a_{n,n} = -2 + q\Delta x^2 \quad (45b)$$

The load vector at n=2, 4, 5 is
$$b_n = q\ \phi_0\ \Delta x^2 \quad (45c)$$

while at node n= 3
$$b_n = q\ \phi_0\ \Delta x^2 - Q'_w(x_3)\ \Delta x^2/T \quad (45d)$$

At node 1
$$a_{1,2} = 2 \quad (46a)$$
$$b_1 = q\ \phi_0 \Delta x^2 \quad (46b)$$

while at node 5,

$$b_5 = -H + q\ \phi_0\ \Delta x^2 \quad (46c)$$

where H is the known head at x_5. Written out the matrix equation becomes

$$\begin{bmatrix} -2+q\Delta x^2 & 2 & 0 & 0 & 0 \\ 1 & -2+q\Delta x^2 & 1 & 0 & 0 \\ 0 & 1 & -2+q\Delta x^2 & 1 & 0 \\ 0 & 0 & 1 & -2+q\Delta x^2 & 1 \\ 0 & 0 & 0 & 1 & -2+q\Delta x^2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} q\Delta x^2\phi_0 \\ q\Delta x^2\phi_0 \\ q\Delta x^2\phi_0 - Q\Delta x^2/T \\ q\Delta x^2\phi_0 \\ q\Delta x^2\phi_0 - H \end{bmatrix} \quad (47)$$

where $y_n = \phi_n$ and $Q = Q'_w$. When p=0 the resulting homogeneous equation $y'' + y = 0$ is sometimes called the *Poisson equation*. This is a solution for that equation, in 1D, for the given BCs, discretization, and *poin- sink* pumping.

Example 5:
Suppose that there is no leakage, so that q=0. Then the ODE becomes a 1D version, $y'' = 0$, of the *LaPlace equation* ($\nabla^2 y$=0), in this case with a point sink at $x = 2\Delta x$. Then (47) simplifies to

$$\begin{bmatrix} -2 & 2 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -Q\Delta x^2/T \\ 0 \\ -H \end{bmatrix} \tag{48}$$

Note that the finite difference operator for the LaPlacian has the form,

+1 -2 +1

This is called the LaPlacian *stencil* and is found in basic finite difference solutions of diffusive mass transfer, heat conduction, and groundwater flow in a homogeneous aquifer. Note how BC1, the zero-gradient boundary, alters the stencil.