## Mathematical Methods (10/24.539)

## V. General Boundary Value Problems (BVPs)

### Introduction

In the previous section we focused on various schemes (both analytical and numerical) for solving general IVPs. We now emphasize another important class of problems known as ***Boundary Value Problems*** (BVPs). The difference between these two problem classes is related to the specification of the n conditions needed to uniquely determine the n arbitrary coefficients in the general solution to an $n^{th}$ order ODE. The n conditions are specified at a ***single (initial) point*** for IVPs. However, for a BVP, the n boundary conditions are specified at ***two or more points*** in the domain of interest. This modification results in completely different algorithms for solving these problems - especially in the numerical solution schemes used for BVPs.

Analytical schemes for linear constant coefficient non-homogeneous BVPs have already been discussed in Section II - Linear Differential Equations. Exact solution methods for general nonlinear BVPs are simply not available (although a number of 'tricks' can be applied to specific problems). Variable coefficient linear problems do have a general solution procedure - referred to as the Power Series Solution Method - but it is often quite tedious and not overly useful for solving general engineering design problems. The Power Series technique is quite powerful, however, and it is very useful for deriving analytical solutions for several important problems. We will discuss this method in some detail later in this course (see Section VII - Power Series Solution Method and Section VIII - Special Functions). Note also that ***Eigenvalue Problems***, a special class of homogeneous BVPs, are discussed later in these notes (see Section IX - The Sturm-Liouville Problem). For now, we are interested in the solution of non-homogeneous BVPs that occur in real engineering applications, and the only general solution method for these problems rely on ***numerical schemes.***

In particular, in this section of notes we focus on two numerical techniques for treating general non-homogeneous BVPs with a single independent variable - the ***Shooting Method*** and ***the Finite Difference Method.***

The **Shooting Method** is essentially an iterative application of the numerical integration techniques used for IVPs. It is easy to apply in most cases, but it is not appropriate for solving BVPs with more that one independent variable (PDEs are discussed later in this course).

The **Finite Difference Method** takes a completely different approach to the problem. It essentially converts the ODE into a coupled set of algebraic equations, with one balance equation for each finite volume or node in the system. This technique is very powerful, and it is easily extended to multidimensional and space-time situations. Thus, this method will be used again when discussing the numerical solution techniques appropriate for general PDEs (later in the course).

The general notation associated with BVPs and an overview of the Shooting and Finite Difference methods are discussed within the following subsections:

The reader might also visit my website for my undergraduate ***Applied Problem Solving with Matlab*** course (see www.profjrwhite.com/courses.htm). There are some examples of both the Shooting method and the Finite Difference method. These are quite straightforward since the treatment in that course is quite introductory. However, these extra examples may be useful!!!

## General Notation for BVPs

Boundary Value Problems (BVPs) differ in one very important respect relative to Initial Value Problems (IVPs). This difference is related to the specification of the n conditions needed to uniquely determine the n arbitrary coefficients in the general solution to an $n^{th}$ order ODE. For a BVP, the n boundary conditions are specified at ***two or more points*** in the domain of interest (whereas the n conditions are specified at a ***single point*** for IVPs). This difference gives rise to significant variations in the algorithms used to solve these problems - especially in the numerical solution schemes.

For BVPs one needs to satisfy conditions at up to n boundary points. For example, we might have

$$y(p_1) = k_1, \quad y(p_2) = k_2, \quad \cdots \quad y(p_n) = k_n$$

where $p_1$, $p_2$, up to $p_n$ are the boundary points for an $n^{th}$ order system.

More often, however, there are only two boundary points, $p_1$ and $p_2$, with multiple conditions at each point. If we define $n_1$ and $n_2$ as the number of conditions at points $p_1$ and $p_2$, respectively, then $n = n_1 + n_2$ for an $n^{th}$ order system. The boundary conditions (BCs) at the two points can be written in a very general form as

$$B_j(p_1, \underline{z}) = 0 \quad \text{for } j = 1, 2, \cdots n_1 \tag{5.1}$$

$$B_k(p_2, \underline{z}) = 0 \quad \text{for } k = 1, 2, \cdots n_2 \tag{5.2}$$

where $\underline{z} = \left[ y \quad y' \quad y'' \quad \cdots \quad y^{(n-1)} \right]^T$.

The most common situation that arises in practical applications, of course, is the $2^{nd}$ order BVP. In this case, there is a single boundary specification for each of the two boundary points. If the BCs are linear in y and y′, then they can be written as

$$B(p_1, y, y') = a_1 y(p_1) + b_1 y'(p_1) - w_1 = 0 \tag{5.3}$$

$$B(p_2, y, y') = a_2 y(p_2) + b_2 y'(p_2) - w_2 = 0 \tag{5.4}$$

If both $w_1$ and $w_2$ are zero, the boundary conditions (BCs) are said to be homogeneous. Also if both $a_1$ and $b_1$, for example, are nonzero, then this boundary condition is referred to as a mixed BC. Various combinations can occur and a variety of specific examples are treated in this section of notes and throughout the remainder of the course.

**Note:** In the technical literature, the various BCs are often referred to as

| | | |
|---|---|---|
| Dirichlet BC: | $B(y, y') \rightarrow B(y)$ | only a function of y |
| Neumann BC: | $B(y, y') \rightarrow B(y')$ | only related to the derivative of y |
| Robin BC: | $B(y, y') \rightarrow B(y, y')$ | function of both y and y' (mixed BC) |

## Shooting Method for 2-Point BVPs

### The Basic Algorithm

The basic idea behind the **Shooting Method** is illustrated in Fig. 5.1. Trial integrations that satisfy the boundary condition at one endpoint are launched. The discrepancies from the desired boundary condition at the other endpoint are used to adjust the starting conditions, until boundary conditions at both endpoints are ultimately satisfied.

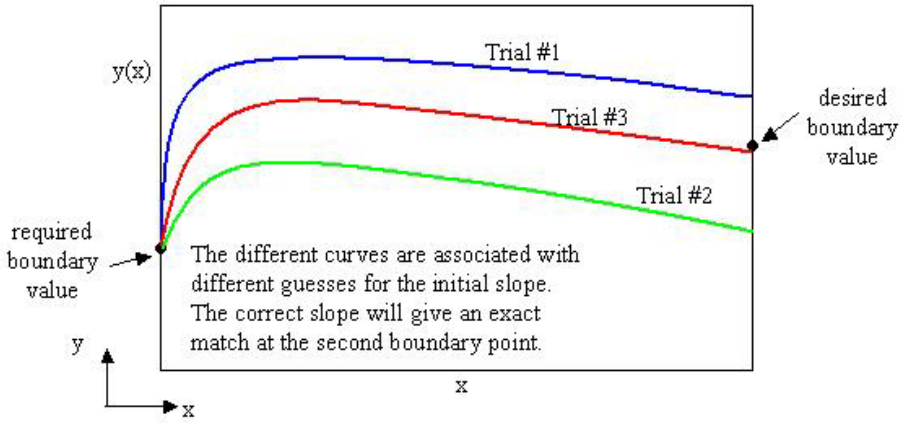



**Fig. 5.1 Schematic for the Shooting Method.**

This rather simple description can be formalized in the form of an algorithm as follows:

1. Given an $n^{th}$ order ODE, convert it to n $1^{st}$ order equations with n BCs by letting

$$\underline{z} = \left[ y \quad y' \quad y'' \quad \cdots \quad y^{(n-1)} \right]^T \tag{5.5}$$

2. Set up the problem as an IVP within a standard ODE solver. In addressing the initial conditions for the IVP, let $n_1$ conditions be specified at $x_1 = p_1$ (considered as the initial point for independent variable x). Therefore, we have $n_2$ unknown or free parameters at $x_1$,

$$n_2 = n - n_1 \tag{5.6}$$

Let $\underline{\alpha}$ be a vector of free parameters ($n_2$ in length). Therefore, at $x_1 = p_1$, $\underline{z}(p_1) = \underline{z}(p_1, \underline{\alpha})$, which simply says that the state at $p_1$ is really not known but it can be written as some function of the vector of free parameters. Our goal is to find $\underline{\alpha}$ in an iterative fashion such that the $n_2$ boundary conditions at point $x_2 = p_2$ are satisfied.

3. Guess $\underline{\alpha}^0$, which gives $\underline{z}(p_1) = \underline{z}\left(p_1, \underline{\alpha}^0\right)$. Now we have a full set of initial conditions so that the IVP can be solved in the usual way.

4. Now solve the IVP using your favorite ODE solver to get $\underline{z}^m(p_2) = \underline{z}\left(p_2, \underline{\alpha}^m\right)$ where m is an iteration counter (i.e. m = 0 for the first guess). Because we only have a guess for some of the initial conditions, this IVP problem will not be equivalent to the desired BVP. In particular, the $n_2$ boundary conditions at point $p_2$ will not be satisfied exactly.

5. To quantify the error, define a discrepancy vector, $\underline{\varepsilon}^m$, whose elements, $\varepsilon_k^m$, satisfy

$$\varepsilon_k^m = B_k\left(p_2, \underline{z}^m\right) \quad \text{for } k = 1, 2, \cdots n_2 \tag{5.7}$$

where, for the exact solution, $\tilde{\underline{z}}$, we have $B_k\left(p_2, \tilde{\underline{z}}\right) = 0$. That is, it satisfies the exact BCs.

6. Finally, check convergence against a user-specified convergence criterion, tol. In particular, if $\left|\varepsilon_k^m\right| < \text{tol}$ for all k, then one terminates the iterative process and edits the results. If, however, $\left|\varepsilon_k^m\right| > \text{tol}$ for any k, then we guess a new $\underline{\alpha}^{m+1}$ (recall that m is simply an iteration counter) and go back to Step 4 to perform another iteration.

The above scheme is actually very easy to implement and it represents a very powerful method for solving 2-point BVPs with only one independent variable.

**Picking the Next $\underline{\alpha}$**

The Shooting Method is a numerical scheme for BVPs in which one iteratively improves upon $n_2$ free parameters within the initial condition vector for a standard IVP. The error condition is related to the difference between the desired BCs at the second boundary point, $p_2$, and the actual values computed using a guess for part of the initial condition vector. Given an error vector for iteration m, $\underline{\varepsilon}^m$, our goal is to choose a free parameter vector, $\underline{\alpha}^{m+1}$, such that the error tends towards zero.

With this goal, let's choose $\underline{\alpha}^{m+1}$ as

$$\underline{\alpha}^{m+1} = \underline{\alpha}^m + \Delta\underline{\alpha}^m \tag{5.8}$$

where $\Delta\underline{\alpha}^m$ is the solution to

$$\underline{\underline{A}}\Delta\underline{\alpha}^m = -\underline{\varepsilon}^m \quad \text{with} \quad \underline{\underline{A}} = \left[a_{ij}\right] = \left[\frac{\partial\varepsilon_i}{\partial\alpha_j}\right] \tag{5.9}$$

Note that the error vector on the right hand side of this equation is negative so that $\Delta\underline{\alpha}^m$ is computed such that it tends to reduce the error on iteration m, $\underline{\varepsilon}^m$, to zero. In component form, this expression can be written as

$$\Delta\varepsilon_i = -\varepsilon_i^m = \sum_j \frac{\partial\varepsilon_i}{\partial\alpha_j}\Delta\alpha_j^m \tag{5.10}$$

In practice, the partial derivatives of the error terms with respect to the free variables are computed by a finite difference approximation, or

$$\frac{\partial\varepsilon_i}{\partial\alpha_j} = \left[\frac{\varepsilon_i\left(\alpha_1, \alpha_2, \cdots, \alpha_j + \Delta\alpha_j, \cdots\right) - \varepsilon_i\left(\alpha_1, \alpha_2, \cdots, \alpha_{n_2}\right)}{\Delta\alpha_j}\right] \tag{5.11}$$

A simple algorithm for computing $\partial\varepsilon_i/\partial\alpha_j$ is as follows:

1. Compute the error vector associated with the current choice for the free parameters, or $\underline{\alpha}^m \rightarrow \underline{\varepsilon}^m$.

2. Perturb only one element, the $j^{th}$ element, $\alpha_j$, and this leads to a new error vector, $\underline{\varepsilon}_j^m$. Then, dropping the iteration index notation for convenience, we have

$$\frac{\partial \underline{\varepsilon}}{\partial \alpha_j} = \frac{\underline{\varepsilon}_j - \underline{\varepsilon}}{\Delta \alpha_j} \tag{5.12}$$

This gives $\partial \varepsilon_i / \partial \alpha_j$ for all i and a single j.

3. Repeat Step 2 for all the elements of $\underline{\alpha}$ (i.e. let j vary from 1 to $n_2$ elements).

This procedure can be repeated for each step in the iteration process.

Finally, it should be noted that, in many practical applications, the fairly general procedure outlined here is really much easier than it may seem. For example, for the very common $2^{nd}$ order BVP, there is only one free parameter and one error component. Thus the above development simplifies considerably. Because of its importance in application, this case is highlighted in the next subsection.

**Second Order BVPs**

Second order systems occur so frequently that it makes sense to illustrate the Shooting Method for this specific case. In discussing this particular problem, let's try to follow, at least roughly, the basic algorithm outlined previously for the Shooting Method.

Step 1. Recall that a $2^{nd}$ order system can always be written as a set of coupled $1^{st}$ order equations, or

$$\frac{d}{dx}\underline{z} = \underline{f}(x, \underline{z}) \quad \text{with} \quad \underline{z} = \left[ y \;\; y' \right]^T \tag{5.13}$$

with linear boundary conditions of the form

$$a_1 y(x_1) + b_1 y'(x_1) - w_1 = 0 \tag{5.14}$$

$$a_2 y(x_2) + b_2 y'(x_2) - w_2 = 0 \tag{5.15}$$

where $x_1$ and $x_2$ are the boundary point locations [if the BCs are nonlinear, one simply uses the more general form given in eqns. (5.1) and (5.2)]. For ease of discussion, let's specialize this to a particular situation where the function, y(x), is known at both endpoints. In this case, the boundary conditions become (***note that this is a specific example***)

$$y(x_1) = w_1 \quad \text{and} \quad y(x_2) = w_2$$

Step 2**.** Now we write the system as an IVP with an ***unknown*** initial value,

$$\frac{d}{dx}\underline{z} = \underline{f}(x, \underline{z}, \alpha) \quad \text{with} \quad \underline{z}(x_1) = \begin{bmatrix} y(x_1) \\ y'(x_1) \end{bmatrix} = \begin{bmatrix} w_1 \\ \alpha \end{bmatrix}$$

where $\alpha = y'(x_1)$ is to be determined so that the second boundary condition, $y(x_2) = w_2$, is satisfied.

Step 3. Now guess $\alpha_1$ and solve the IVP for $y(x_2, \alpha_1)$, and guess $\alpha_2$ and solve the IVP again for $y(x_2, \alpha_2)$. With two values of $\alpha$ and their corresponding endpoint results, we can compute the errors in the second BC, or

$$\varepsilon_1 = y(x_2, \alpha_1) - w_2 \quad \text{and} \quad \varepsilon_2 = y(x_2, \alpha_2) - w_2$$

And, with these error estimates, an approximation to $\partial\varepsilon/\partial\alpha$ can be constructed, or

$$\frac{\partial \varepsilon}{\partial \alpha} \approx \frac{\varepsilon_2 - \varepsilon_1}{\alpha_2 - \alpha_1}$$

Step 4. Note that the desired change in error is the negative of the error in the previous stage. Therefore,

$$\Delta\varepsilon = \frac{\partial \varepsilon}{\partial \alpha}\Delta\alpha = -\varepsilon_2 \quad \text{and} \quad \Delta\alpha = \frac{-\varepsilon_2}{\partial\varepsilon/\partial\alpha}$$

which gives a new increment in the free variable. Now we can compute $\alpha_3$ as $\alpha_3 = \alpha_2 + \Delta\alpha$ and solve the IVP again, giving $\varepsilon_3$, etc., etc.

This basic procedure is continued until the user-specified convergence criterion is met (i.e. until $|\varepsilon_m| < \text{tol}$). In practice only a few steps are usually required for good accuracy in linear or mildly nonlinear problems (3-5 steps or less in most cases).

Example 5.1 and Example 5.3A illustrate the shooting method for two specific 2nd order BVPs. The first example implements both a manual and automated iterative scheme within Matlab to show how the basic method works, and the second sample problem represents a practical application to the classical circular fin heat transfer problem.

## Example 5.1 - The Shooting Method for 2-Point BVPs

**Problem Description:**

Solve the following 2-point BVP using the Shooting Method:

$$y'' + 3xy' + 7y = \cos(2x) \qquad \text{with} \quad y(0) = 1 \quad \text{and} \quad y(\pi) = 0$$

**Problem Solution:**

### *Case 1 Manual Interactive Iteration*

1. To start this problem, let's convert to a set of coupled equations. Letting $\underline{z} = [y \;\; y']^T$ gives

$$z_1' = z_2$$

$$z_2' = -3xz_2 - 7z_1 + \cos(2x)$$

or, in general terms, this becomes $\underline{z}' = \underline{f}(x, \underline{z})$.

2. Next we set this up as an IVP and use an ODE solver in Matlab to numerically integrate this system, with the initial conditions given by

$$\underline{z}(0) = \begin{bmatrix} y(0) \\ y'(0) \end{bmatrix} = \begin{bmatrix} 1 \\ \alpha \end{bmatrix}$$

3. In this interactive example, the user is prompted for an input value for $\alpha$ and, after the call to Matlab's ODE routine, we compute the error at the second boundary point as

$$\varepsilon = z_1(\pi, \alpha) - y(\pi)$$

4. Step 3 is repeated until the correct value of $\alpha$ is determined. That is, until $\varepsilon$ is small (i.e. less than the desired tolerance).

This manual interactive solution procedure for Case 1 has been implemented within the main Matlab script file **shootm1.m** and function file **shootf.m**. These files are listed in Tables 5.1 and 5.2, respectively. The particular implementation given here is very simple and it assumes that the user will do some auxiliary computations to help decide the next value of $\alpha$ to choose. Following the first two arbitrary guesses, the next guess should be made using the guidelines in the previous section.

A typical interactive session with **shootm1.m** has been saved with Matlab's ***diary*** feature and later this file was annotated with the information used to compute the third (and last) value of $\alpha$. A listing of the annotated diary file is given in Table 5.3. Note here that very good convergence was achieved with only three values of $\alpha$.

In addition to the session diary, the computed solutions, $y_i(x)$, for the three different values of $\alpha$ used in this illustration are plotted in Fig 5.2. This set of curves was generated in a separate m-file called **shootm2.m**. This file is given in Table 5.4. The value of the initial $y'$ that gives the correct solution (i.e. properly satisfies the second BC) is $\alpha = y'(0) = -5.4715$. The profiles

for the first two arbitrary guesses ( $\alpha = \pm 1$ ) are substantially different from the correct profile and, in particular, they give the wrong endpoint value at $x = \pi$. Note that the endpoint values are all quite similar, so a relatively tight convergence is needed. Notice, however, that it is relatively easy to converge on the proper value of the single free variable.

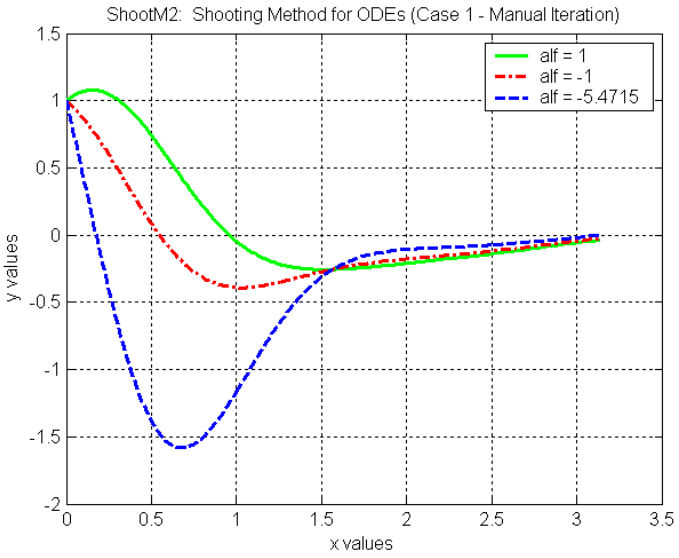



**Fig. 5.2 Some trial solutions for Case 1 from Example 5.1.**

Tables 5.1 - 5.4 summarize the basic procedures associated with the Shooting Method, and the student should study these files as needed to get a good understanding of this method. The interactive approach is given here as a good mechanism for understanding the overall iterative scheme. In practical applications, however, this procedure should be automated to eliminate the manual computation of $\alpha$ (and the possibility of error introduced via the hand calculations). An automated procedure, as illustrated below for Case 2, is simply easier and more efficient for routine application.

**Table 5.1 Listing of the Matlab main program shootm1.m.**

```
%
%   SHOOTM1.M    Example of Shooting Method for Solving 2-Point BVPs
%                      (Manual Iterative Procedure)
%
%   This demo solves a particular 2nd order ODE using an iterative form of
%   the shooting method.  The variable ALF is chosen interactively.  This is
%   discussed as Example 5.1 Case 1 Manual Interactive Iteration in the course
%   notes for Math Methods (10/24.539).
%
%   The goal is to find the value of the free parameter, ALF = y'(0) which,
%   upon solution of the IVP, satisfies the desired boundary condition at x = pi.
%
%   also see SHOOTF.M - contains description of given differential equation
%       y'' + 3xy' + 7y  =  cos(2x)     with  y(0) = 1  and  y(pi) = 0
%
%   also see related files:
%       SHOOTM2.M   - shows multiple curves of y(x) for different ALF
%       SHOOTA.M    - solves same problem using an automated iterative scheme
```

```
%       BVP2SH.M    - implements Shooting Method for 2nd order BVP with linear BCs
%
%   Files prepared by J. R. White, UMass-Lowell (Aug. 2003)
%


%
%   getting started
      clear all, close all
%
%   set domain limits and tolerance for ODE23
      xo = 0;    xf = pi;    tol = 1e-6;
%
%   specify desired BCs
      yxo = 1.0;   yxf = 0;
%
%   set format for edit to long words (more significant digits)
      format long
%
%   guess at initial conditions (note that ALF is defined manually)
      ALF = input('Input initial guess for free parameter (ALF) = ')
%
%   start looping (for various values of ALF)
      options = odeset('RelTol',tol);
      while ALF ~= 0
        zo = [yxo  ALF]';
%
%   call ODE23
        [x,z] = ode23('shootf',[xo xf],zo,options);
%
%   plot results for visual inspection of solution at each iteration
        plot(x,z(:,1),'LineWidth',2),title('ShootM1:  Shooting Method for ODEs')
        xlabel('x values'),ylabel('y values'),grid
%
%   edit error in BC at second boundary point (x = xf)
        [nr,nc] = size(z);
        eps =  z(nr,1)-yxf;
        disp('  ')
        disp(' Error in 2nd BC'),     eps
        ALF = input('Enter another guess for ALF (zero to quit)? ')
      end
      disp('Manual iterative procedure terminated by user...')
%
%   end demo
%
```

## Table 5.2 Listing of the ode23 function program shootf.m.

```
%
%   SHOOTF.M   Function file for SHOOTM1.M, SHOOTM2.M, and SHOOTA.M (used by ODE23)
%
%   Given differential eqn.:   y'' + 3xy' + 7y =  cos(2x)
%
%   File prepared by J. R. White, UMass-Lowell (Aug. 2003)
%
     function zp = odefile(x,z)
     zp = zeros(length(z),1);
     zp(1) = z(2);
     zp(2) = -3*x*z(2) - 7*z(1) + cos(2*x);
%
%   end of function
%
```

### Table 5.3 Matlab diary file from shootm1.m.

This listing illustrates the iterative process for the Shooting Method. The Matlab output has been edited in MS Word and the computation of the third guess for $\alpha$ = ALF has been added.

» shootm1
Input initial guess for free parameter (ALF) = 1 (this is $\alpha_1$)
Error in 2nd BC
eps = -0.03479395271931 (this is $\varepsilon_1$)
Enter another guess for ALF (zero to quit)?
ALF = -1 (this is $\alpha_2$)
Error in 2nd BC
eps = -0.02404080244781 (this is $\varepsilon_2$)
---
Computation for new choice of ALF (done outside Matlab)

$$\frac{\partial \varepsilon}{\partial \alpha} = \frac{\varepsilon_2 - \varepsilon_1}{\alpha_2 - \alpha_1} = -5.3765\text{e}-3$$

Therefore,

$$\alpha_3 = \alpha_2 - \frac{\varepsilon_2}{\partial\varepsilon/\partial\alpha} = -1 - \frac{-2.4041\text{e}-2}{-5.3765\text{e}-3} = -5.4715$$

---
Enter another guess for ALF (zero to quit)?
ALF = -5.47150000000000 (this is $\alpha_3$)
Error in 2nd BC
eps = 4.088124867279892e-007 (this is $\varepsilon_3$)
Enter another guess for ALF (zero to quit)?
ALF = 0 (eps was small enough, so quit)
Manual iterative procedure terminated by user...

### Table 5.4 Listing of Matlab file shootm2.m.

```
%
%   SHOOTM2.M   Example of Shooting Method for Solving ODEs
%
%   This is a follow up to demo SHOOTM1.M.  This file simply generates the
%   plots obtained for each of the guesses for the ALF used in SHOOTM1.M
%   The goal was to find the ALF at x = 0 which gives y(pi) = 0.
%
%   also see SHOOTF.M - contains description of given differential equation
%      y'' + 3xy' + 7y  =  cos(2x)      with  y(0) = 1  and  y(pi) = 0
%
%   File prepared by J. R. White, UMass-Lowell  (Aug. 2003)
%


%
%   getting started
      clear all, close all, nfig = 0;
%
%   with manual procedure, ALFs of 1, -1, and -5.4713 were used in SHOOTM1.M
%   let's plot the function for these different guesses to see how this works
```

```
      gs = [1 -1 -5.4715];
%
%   set domain limits and solve IVP
      xo = 0;   xf = pi;    tol = 1.0e-6;   yxo = 1.0;
      options = odeset('RelTol',tol);
      [x1,z1] = ode23('shootf',[xo xf],[yxo gs(1)],options);
      [x2,z2] = ode23('shootf',[xo xf],[yxo gs(2)],options);
      [x3,z3] = ode23('shootf',[xo xf],[yxo gs(3)],options);
%
%   now plot results
      nfig = nfig+1;  figure(nfig)
      plot(x1,z1(:,1),'g-',x2,z2(:,1),'r-.',x3,z3(:,1),'b--','LineWidth',2)
      title('ShootM2:  Shooting Method for ODEs (Case 1 - Manual Iteration)')
      xlabel('x values'),ylabel('y values'),grid
      legend('alf = 1','alf = -1','alf = -5.4715')
%
%   end of demo
```

### *Case 2 Automated Iteration*

The Shooting Method for a 2nd order BVP has been automated within Matlab file **bvp2sh.m**. This file is listed in Table 5.5 and, as apparent from the embedded comments, it is currently limited to the case where linear BCs of the form given in eqns. (5.14) and (5.15) are specified. This restriction was made only because it was easy to generalize the coding for BCs in this form. Note that, although it would be more difficult to allow general nonlinear BCs, the existing file could easily be modified for a particular problem.

The comments at the beginning of **bvp2sh.m** show how to use this routine. In particular, the user needs to supply information about the linear BCs for the problem of interest within the **zbc** array, as follows:

$$\text{zbc} = \begin{bmatrix} a_1 & b_1 & w_1 \\ a_2 & b_2 & w_2 \end{bmatrix}$$

where the individual coefficients, $a_1$, $b_1$, etc. are defined in eqns. (5.14) and (5.15). This array replaces the usual IC array that is used with Matlab's built-in ***ode23*** or ***ode45*** IVP solvers.

At the initial boundary point, $x_1$, a check is made within **bvp2sh.m** to determine the kind of BC that is imposed and, from this, it can determine the form of the IC vector for the iterative IVP solution. In particular, the three standard BCs can be represented as follows:

Dirichlet BC: if $b_1 = 0$, $y(x_1) = w_1/a_1$ then $\underline{z}_o = \begin{bmatrix} w_1/a_1 \\ \alpha \end{bmatrix}$

Neumann BC: if $a_1 = 0$, $y'(x_1) = w_1/b_1$ then $\underline{z}_o = \begin{bmatrix} \alpha \\ w_1/b_1 \end{bmatrix}$

Robin (mixed) BC: if both $a_1$ and $b_1$ are nonzero, then $\underline{z}_o = \begin{bmatrix} \alpha \\ (w_1 - a_1\alpha)/b_1 \end{bmatrix}$

where $\alpha$ is the unknown parameter at the initial point, $x_1$, which will be computed as part of the iterative solution. Note that the Neumann BC is just a subset of the Robin BC with $a_1 = 0$, and it is implemented as such within **bvp2sh.m**.

The value of $\alpha$ is determined as part of the overall solution such that the BC at $x_2$ is satisfied to within some small tolerance. An arbitrary guess for $\alpha$ initiates the iterative procedure, giving a first estimate of the solution with particular focus on the relative error in the second boundary condition. If the relative error at the second boundary point is greater than a user-specified tolerance, a 1% change is made in the current value of $\alpha$, a new IVP solution is obtained, and a finite difference estimate for $\partial\varepsilon/\partial\alpha$ is computed. Using this derivative estimate, a new value of $\alpha$ is chosen with the goal of reducing the relative error to zero. With the new $\alpha$, a new iteration is initiated, with the process continuing until the preset convergence criterion is met or the maximum number of iterations is reached.

This automated procedure was used within Matlab file **shoota.m** (see Table 5.6) to solve the same problem as for Case 1. This file calls **bvp2sh.m** for general implementation of the Shooting Method for problems with **linear BCs**. The starting point here is exactly the same as for Case 1. In particular, we set the problem up as an IVP using standard state space notation (as in Steps 1 and 2 from Case 1) and the same ODE function file, **shootf.m**, is used. In **shoota.m**, however, the **bvp2sh.m** routine handles all the details of the method. In this case, the required **zbc** matrix is simply **zbc = [1 0 1; 1 0 0]**, and the rest of the problem is then no more difficult to solve than a standard IVP that uses Matlab's built-in ***ode23*** routine.

The final converged solution from **shoota.m** is given in Fig. 5.3 and the solution is identical to the converged solution from Case 1 using the manual approach to the Shooting Method. Obviously, the automated methodology built into the **bvp2sh.m** routine is the easier of the two approaches, and the automated technique illustrated here should prove to be useful for solving a variety of BVPs of this type.

Finally, we note that another example, Example 5.3A, has been provided at the end of this section of notes to further illustrate the use of the **bvp2sh.m** function. This problem deals with conduction heat transfer in a circular fin arrangement and it represents a more practical application of the basic methodology.

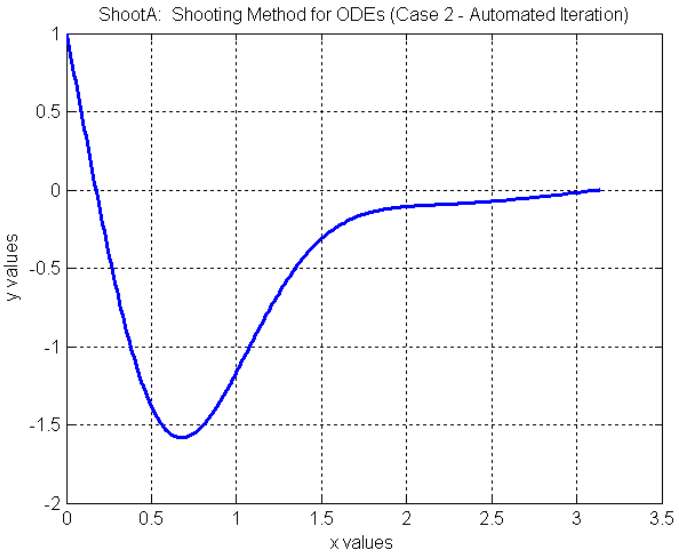



**Fig. 5.3 The converged solution for Case 2 from Example 5.1.**

## Table 5.5 Listing of Matlab file bvp2sh.m.

```
%
%   BVP2SH.M  This function file solves a 2nd order BVP with linear BCs
%                       via the Shooting Method
%
%   This method solves the BVP using an IVP solver with an initial guess
%   for the unknown initial conditions.  An iterative scheme automatically
%   modifies this guess until the boundary conditions at the second boundary
%   point are satisfied.  This iterative technique is called the Shooting Method.
%
%   A general 2nd order ODE can be written in the form
%        y'' + a*y' + b*y = f(t)
%   where, in general, a and b can be functions of y and t.
%
%   We can also write this as a set of two 1st order equations in so-called
%   state space form as
%        dz1/dt = z2   and   dz2/dt = f - a*z2 - b*z1
%   where z1 = y  and  z2 = y'
%
%   Now, if we had a set of initial conditions, zo = [y(to); y'(to)], then
%   this state space system could be easily solved as an IVP.
%
%   However, since this is a BVP, we do not have a complete set of initial
%   conditions.  Thus, the basic idea of the shooting method is to guess at the
%   unknown variable at the initial state so that the solution of the resulting
%   IVP gives the desired final state for the BVP.  This is done iteratively by
%   converging on the free parameter, alf, that is included as part of the
%   initial condition vector such that the desired final state is achieved.
%
%   For the case of a 2nd order 2-point BVP with linear BCs, a general way to
%   write the boundary conditions is as follows:
%     at p1:  a1*y(p1) + b1*y'(p1) - w1 = 0
%     at p2:  a2*y(p2) + b2*y'(p2) - w2 = 0
%   The coefficients in these expressions are input to this function in the
%   zbc matrix as:  zbc = [a1 b1 w1; a2 b2 w2].  This format essentially
%   defines the BCs for the 2nd order BVP with linear BCs (note that the equation
%   can be nonlinear -- but not the BCs).
%
%   At the initial boundary point, p1, a check is made to determine the
%   kind of boundary condition and thus the form of the initial condition
%   vector for the iterative IVP solution:
%     Dirichlet BC:     if b1 = 0,  y(p1) = w1/a1
%                       then      zo = [w1/a1; alf]          (guess initial slope)
%     Neumann BC:       if a1 = 0,  y'(p1) = w1/b1
%                       then      zo = [alf; w1/b1]          (guess initial value)
%     Robin (mixed) BC: if both a1 and b1 are nonzero,
%                       then  zo = [alf; (w1 - a1*alf)/b1]   (guess initial value)
%   where alf is the unknown parameter which will be computed as part
%   of the iterative solution.  alf is determined such that the second
%   BC is satisfied to within some small tolerance. Note that the Neumann
%   BC is just a subset of the Robin BC with a1 = 0, and it is implemented as
%   such in this routine.
%
%   The inputs to this function are similar to those used with Matlab's ode23
%   IVP solver (in its simplest implementation) except that the zbc matrix
%   replaces the initial state vector.  With zbc known, the zo vector is computed
%   internally as described above.  Also, there is an option to input a guess
%   for alf, if desired
%     func - name of function file that evaluates the RHS of the state equations
%     tspan   - vector that defines the range for the independent variable
%     zbc  - matrix containing the BC coefficients (as described above)
%     options - standard options structure for use in ode23  (optional)
%     alfo - inital guess for alf  (optional)
%
%   The outputs are:
%      t - column vector containing the independent variable
%      z - solution matrix with two columns with column 1 containing z1(t)
%          and column 2 containing z2(t)
%
%   Note #1: Extra parameters that may be needed in the ode function file must be
%   passed to the function file via a global statement in the main program and
%   the function file.  The current, relatively simple implementation of the
%   Shooting Method does not use the variable input arguments (varargin) that are
```

```
%   often used in the ode23 routine.  This is not overly restrictive, however, since
%   it is just as easy to pass needed variables into the function via the global
%   statement.
%
%   Note #2: This routine assumes that you are integrating from boundary point
%   p1 to boundary point p2.  This normally implies that p2 > p1, but this is not
%   absolutely essential.  In some cases, it may be appropriate to integrate
%   'backwards'  -- just be careful with the ordering of the BCs if you try this...
%
%   Note #3:  Nonlinear problems are sometimes sensitive to the initial guess for
%   the free variable, alf, in the initial condition vector.  Here we have simply
%   set alfo = 1 as default.  The user can override this as needed in particularly
%   difficult problems.  A poor initial guess usually just means that a few more
%   iterations may be needed. If the system does not converge within 10 - 20
%   iterations, you may want to change alfo via the last entry in the input argument
%   list to see if this helps.  This is not usually necessary,  however...
%
%   File prepared by J. R. White, UMass-Lowell (March 2003)
%

      function [t,z]  = bvp2sh(func,tspan,zbc,options,alfo)
%
%   check to see if the options structure is available
      if nargin < 4,  options = [];   end
%
%   check to see if there is an initial guess for alfo
      if nargin < 5,  alfo = 1;   end
%
%   extract BC coefficients (for convenience)
      a1 = zbc(1,1);  b1 = zbc(1,2);  w1 = zbc(1,3);
      a2 = zbc(2,1);  b2 = zbc(2,2);  w2 = zbc(2,3);
%
%   set some other iterative parameters
      err = 1e10;   tol = 1e-6;   icnt = 1;   icntmax = 25;
%
%   start iteration
      while abs(err) > tol  &  icnt <= icntmax
%
%   determine initial condition vector based on structure of zbc matrix
          if b1 == 0
            zo = [w1/a1; alfo];
          else
            zo = [alfo; (w1 - a1*alfo)/b1];
          end
%
%   solve IVP with current guess for initial conditions
          [t,z] = ode23(func,tspan,zo,options);
%
%   evaluate error in second BC
          e1 =  a2*z(end,1) + b2*z(end,2) - w2;  err = e1;
          fprintf(1,'  \n')
          fprintf(1,'For iteration # %2i: \n',icnt)
          fprintf(1,'  Initial conditions are: %13.5e %13.5e \n',zo)
          fprintf(1,'  Error in 2nd BC is:     %13.5e \n',err)
          fprintf(1,'  \n')
%
%   estimate new values for alf (if necessary)
          if abs(err) > tol
            alfp = 1.01*alfo;
            if b1 == 0
              zo = [w1/a1; alfp];
            else
              zo = [alfp; (w1 - a1*alfp)/b1];
            end
            [t,z] = ode23(func,tspan,zo,options);
            e2 =  a2*z(end,1) + b2*z(end,2) - w2;
            deda = (e2-e1)/(0.01*alfo);    alfn = alfo-e1/deda;
            icnt = icnt+1;   alfo = alfn;
          end
        end   % end of iteration loop
        if icnt >= icntmax
          fprintf(1,'  WARNING -- Hit maximum iteration limit!!! \n');
        end
%
%   end of function
```

## Table 5.6 Listing of Matlab file shoota.m.

```
%
%
%   SHOOTA.M    Example of Shooting Method for Solving 2-Point BVPs
%                      (Automated Iterative Procedure)
%
%   This demo solves a particular 2nd order ODE using an iterative form of
%   the shooting method.  The variable ALF is chosen automatically as part of the
%   iterative procedure.  This is discussed as Example 5.1 Case 2 Automated
%   Iteration in the course notes for Math Methods (10/24.539).
%
%   The goal is to find the value of the free parameter, ALF = y'(0) which,
%   upon solution of the IVP, satisfies the desired boundary condition at x = pi.
%   The automated algortithm for the Shooting Method is imlemented within the
%   BVP2SH.M file.
%
%   also see SHOOTF.M - contains description of given differential equation
%       y'' + 3xy' + 7y  =  cos(2x)     with  y(0) = 1  and  y(pi) = 0
%
%   also see related files:
%       SHOOTM1.M   - solves same problem using a manual iterative scheme
%       SHOOTM2.M   - shows multiple curves of y(x) for different ALF (manual case)
%
%   Files prepared by J. R. White, UMass-Lowell (Aug. 2003)
%


%
%   getting started
      clear all, close all
%
%   set domain limits
      xo = 0;    xf = pi;
%
%   set coefficients for BCs
      zbc = [1 0 1.0;      % left BC  --> y(xo) = 1.0
             1 0 0.0];     % right BC --> y(xf) = 0.0
%
%   solution via Shooting Method (numerical approx)
      tol = 1e-6;  options = odeset('RelTol',tol);   % set tight tolerance for ODE soln
      [xs,zs] = bvp2sh('shootf',[xo xf],zbc,options);
%
%   plot final function values
      plot(xs,zs(:,1),'LineWidth',2),grid
      title('ShootA:  Shooting Method for ODEs (Case 2 - Automated Iteration)')
      xlabel('x values'),ylabel('y values')
%
%   end demo
```

## The Finite Difference Method

### Basic Concepts

The **Finite Difference** (FD) method essentially converts the ODE into a coupled set of algebraic equations, with one balance equation for each finite volume or node in the system. The general technique is to replace the continuous derivatives within the ODE with finite difference approximations on a grid of mesh points that spans the domain of interest. The boundary nodes are treated as special cases. The coupled set of equations that results can then be solved by a variety of techniques (see Section VI - Numerical Solution of Algebraic Equations).

A simple algorithm for the basic FD method (for linear systems) is outlined as follows:

1. Convert continuous variables to discrete variables. Note that the mesh grid or nodal scheme for the independent variable may be evenly spaced (usually gives a relatively simple formulation) or set on an uneven grid (resultant equations are usually slightly more complex), as necessary.

2. Approximate the derivatives at each point using formulae derived from a formal Taylor Series expansion using the most accurate approximation available that is consistent with the given problem.

3. Treat the boundary points separately, being careful to get the best approximation as possible to the desired BCs.

4. Solve the resultant set of coupled equations using either ***direct*** or ***iterative*** schemes as appropriate for the problem.

If the original ODE is nonlinear the above procedure is still valid, except an ***additional outer iteration*** is needed in the overall solution strategy. Highly nonlinear problems are often difficult to converge, but mildly nonlinear or variable coefficient problems are usually solvable using a simple outer iteration scheme.

This section of notes first reviews the basic Taylor Series and the development of several derivative approximations, and then illustrates the FD method via an example.

### Derivative Approximations Using Taylor Series

A key step in the Finite Difference method is to replace the continuous derivatives in the original ODE with appropriate approximations in terms of the dependent variable evaluated at different mesh points in the region of interest. The basis for doing this is related to the Taylor Series expansion for a function in the vicinity of some discrete point, $x_i$.

To formalize this discussion, let's start by defining a discrete notation for both the independent and dependent variables. For the independent variable we simply evaluate the continuous variable x at discrete points $x_i$. If the domain of interest is broken into evenly spaced nodes, then

$$x_{i+1} = x_i + h \quad \text{and} \quad x_{i-1} = x_i - h \qquad \text{where} \quad h = |\Delta x| \tag{5.16}$$

For the dependent variable, f(x), the discrete representation becomes

$$f(x_i) \to f_i\ , \qquad f(x_i + h) \to f_{i+1}\ , \qquad f(x_i - h) \to f_{i-1}\ , \qquad \text{etc.} \tag{5.17}$$

Then, from the Taylor Series we have

$$f_{i+1} = f_i + f_i'h + \frac{f_i''h^2}{2!} + \frac{f_i'''h^3}{3!} + \cdots \tag{5.18}$$

$$f_{i-1} = f_i - f_i'h + \frac{f_i''h^2}{2!} - \frac{f_i'''h^3}{3!} + \cdots \tag{5.19}$$

These two equations can be used to derive several common derivative approximations, as follows:

### *First Derivative Approximations*

Forward Difference Approximation - use eqn. (5.18) directly to give

$$f_i' = \frac{f_{i+1} - f_i}{h} + O(h) \tag{5.20}$$

Backward Difference Approximation - use eqn. (5.19) directly to give

$$f_i' = \frac{f_i - f_{i-1}}{h} + O(h) \tag{5.21}$$

Central Difference Approximation - subtract eqn. (5.19) from eqn. (5.18) to give

$$f_{i+1} - f_{i-1} = 2f_i'h + 2\frac{f_i'''h^3}{3!} + \cdots$$

or $$f_i' = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2) \tag{5.22}$$

where we notice that the central derivative formula is more accurate, with truncation errors ***on the order of $h^2$*** [which is denoted as $O(h^2)$].

### *Second Derivative Approximations*

Central Difference Approximation - add eqn. (5.19) to eqn. (5.18) giving

$$f_{i+1} + f_{i-1} = 2f_i + f_i''h^2 + 2\frac{f_i^{(4)}h^4}{4!} + \cdots$$

or $$f_i'' = \frac{f_{i-1} - 2f_i + f_{i+1}}{h^2} + O(h^2) \tag{5.23}$$

These first and second derivative approximations are the most common representations used when converting the continuous ODE into FD form. There are other formulae that are sometimes used (see the next subsection), especially when 2nd order accuracy at the end points is desired.

## Some Additional Derivative Approximations

### *Order Δx Approximations for First and Second Derivatives*

Forward Difference Approximations:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta x} \tag{5.24}$$

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i+1}) + f(x_{i+2})}{\Delta x^2} \tag{5.25}$$

Backward Difference Approximations:

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{\Delta x} \tag{5.26}$$

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{\Delta x^2} \tag{5.27}$$

### *Order $\Delta x^2$ Approximations for First and Second Derivatives*

Forward Difference Approximations:

$$f'(x_i) = \frac{-3f(x_i) + 4f(x_{i+1}) - f(x_{i+2})}{2\Delta x} \tag{5.28}$$

$$f''(x_i) = \frac{2f(x_i) - 5f(x_{i+1}) + 4f(x_{i+2}) - f(x_{i+3})}{\Delta x^2} \tag{5.29}$$

Backward Difference Approximations:

$$f'(x_i) = \frac{3f(x_i) - 4f(x_{i-1}) + f(x_{i-2})}{2\Delta x} \tag{5.30}$$

$$f''(x_i) = \frac{2f(x_i) - 5f(x_{i-1}) + 4f(x_{i-2}) - f(x_{i-3})}{\Delta x^2} \tag{5.31}$$

Central Difference Approximations:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x} \tag{5.32}$$

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{\Delta x^2} \tag{5.33}$$

The above derivative approximations [eqns. (5.20) - (5.33)] represent a good inventory of formulae for use in the FD method. Many other formulae are also available, since there are a variety of schemes and assumptions that can be used in the development of such formulae.

As an example, let's derive the forward approximation for $f_i'$ with error $O(h^2)$ as given above in eqn. (5.28). We start with Taylor Series expansions for $f_{i+1}$ and $f_{i+2}$, which can be written as

$$f_{i+1} = f_i + f_i'h + f_i''\frac{h^2}{2} + f_i'''\frac{h^3}{6} + \cdots \tag{5.34}$$

$$f_{i+2} = f_i + f_i'(2h) + f_i''\left(\frac{4h^2}{2}\right) + f_i'''\left(\frac{8h^3}{6}\right) + \cdots \tag{5.35}$$

Now let's eliminate the second derivative term by multiplying eqn. (5.34) by 4 and subtracting eqn. (5.35) from the result, giving

$$4f_{i+1} - f_{i+2} = 3f_i + 2f_i'h - \frac{4}{6}f_i'''h^3 + \cdots \tag{5.36}$$

Solving this expression for $f_i'$ gives

$$f_i' = \frac{-3f_i + 4f_{i+1} - f_{i+2}}{2h} + O\left(h^2\right)$$

which is the same result as given above. This is only one example - however, many special purpose formulae can be derived as needed following the general scheme illustrated here.

Example 5.2 illustrates the use of some of these derivative formulae as applied to the same 2-point BVP as given in Example 5.1 (with solution based on the Shooting Method). This FD example also emphasizes the importance of using the best derivative approximation available -- for most applications one should use the derivative formulae that are accurate to second order (rather than a first order approximation).

A practical problem that involves heat transfer in a circular fin geometry is also given in Example 5.3B. This problem is simply another example of the use of the Finite Difference method for linear BVPs (see Example 5.3A for solution of the same problem via the Shooting Method).

## Example 5.2 - The Finite Difference Method (Example 5.1 Revisited)

**Problem Description:**

Solve the following 2-point BVP using the FD method:

$$y'' + 3xy' + 7y = \cos(2x) \qquad \text{with} \qquad y(0) = 1 \quad \text{and} \quad y(\pi) = 0$$

**Problem Solution:**

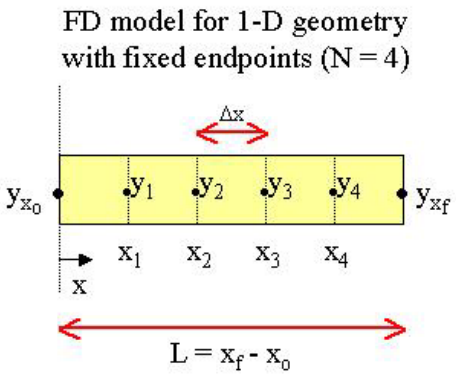
1. Let's start by defining the discrete geometry of interest and by discretizing the dependent and independent variables using a uniform mesh grid,

$$x \to x_i \quad \text{and} \quad x_{i+1} = x_i + \Delta x$$

$$y(x) \to y(x_i) = y_i\ , \quad y'(x) \to y'(x_i) = y'_i\ , \quad \text{etc.}$$

Here we want to be sure to start numbering the nodal points at the first unknown value, as illustrated in the sketch. Note that, in this case, since both endpoint values, $y(p_1)$ and $y(p_2)$, are known, the number of unknowns is one less than the number of mesh intervals. If we let N be the number of unknowns in the problem (N = 4 in the sketch), then the spatial increment can be determined by

$$h = \Delta x = \frac{x_f - x_o}{N+1}$$




Also, for this case, we can easily set the vector of independent variables using Matlab's colon operator as **x = xo+dx:dx:xf-dx** where, of course, **dx** would be the Matlab variable for Δx, **xo** means $x_o$, etc..

2. With a discrete geometry and discretized variables, we then replace the derivatives at discrete points with finite difference approximations (from the Taylor Series). Here one should always use the most accurate representation that is appropriate for the problem at hand. This point is illustrated below by comparing two different approximations for y′ (Case A uses a high order approximation and Case B uses a low order representation).

3. Finally, we put the equations for each node or mesh in matrix form and solve using some linear equation solver (direct or iterative method). In the examples below we use Matlab's standard equation solver (via the backslash operator), which is an implementation of an LU Decomposition method (a direct elimination scheme).

### *Case A -- High Order Approximation*

For this case let's use the central FD approximations in Step 2 for both y″ and y′, or

$$y_i'' = \frac{y_{i-1} - 2y_i + y_{i+1}}{\Delta x^2} \qquad \text{and} \qquad y_i' = \frac{y_{i+1} - y_{i-1}}{2\Delta x}$$

Therefore, substituting these into the discrete form of the original ODE, we have

$$y_i'' + 3x_i y_i' + 7y_i = \cos(2x_i)$$

$$\left[\frac{y_{i-1} - 2y_i + y_{i+1}}{\Delta x^2}\right] + 3x_i\left[\frac{y_{i+1} - y_{i-1}}{2\Delta x}\right] + 7y_i = \cos(2x_i)$$

Rearranging the expression, one has the following relationship for each interior node in the domain of interest:

$$\left(1 - \frac{3x_i\Delta x}{2}\right)y_{i-1} + \left(-2 + 7\Delta x^2\right)y_i + \left(1 + \frac{3x_i\Delta x}{2}\right)y_{i+1} = \Delta x^2\cos(2x_i)$$

Therefore, for the general internal node (row i corresponds to the balance equation for the $i^{th}$ node), we have

$$A(i, i-1) = 1 - \frac{3x_i\Delta x}{2}$$

$$A(i,i) = -2 + 7\Delta x^2 \quad \text{and} \quad b(i) = \Delta x^2\cos(2x_i)$$

$$A(i, i+1) = 1 + \frac{3x_i\Delta x}{2}$$

where the A(i, j) and b(i) notation is consistent with the matrix representation used within the Matlab code.

For the boundary nodes, we have to incorporate specific information about the boundary conditions for the problem. This example is particularly straightforward since ***the function values at both endpoints are specified***. This says that $y_0$ and $y_{N+1}$ are known quantities, where ***N represents the number of discrete unknowns in the problem***. Thus, the balance equation for the first and last nodal volumes can be written as follows:

Left Boundary:

For i = 1, $y_{i-1} = y_0 =$ yxo (in the code). Therefore, the balance equation is written as

$$\left(-2 + 7\Delta x^2\right)y_1 + \left(1 + \frac{3x_1\Delta x}{2}\right)y_2 = \Delta x^2\cos(2x_1) - \left(1 - \frac{3x_1\Delta x}{2}\right)\text{yxo}$$

Right Boundary:

For i = N, $y_{i+1} = y_{N+1} =$ yxf (in code). Therefore, the balance equation becomes

$$\left(1 - \frac{3x_N\Delta x}{2}\right)y_{N-1} + \left(-2 + 7\Delta x^2\right)y_N = \Delta x^2\cos(2x_N) - \left(1 + \frac{3x_N\Delta x}{2}\right)\text{yxf}$$

Now, for the specific case of N = 4, we can write out the resultant four explicit equations, as follows:

for i = 1: $\left(-2 + 7\Delta x^2\right)y_1 + \left(1 + \frac{3x_1\Delta x}{2}\right)y_2 = \Delta x^2\cos(2x_1) - \left(1 - \frac{3x_1\Delta x}{2}\right)\text{yxo}$

$$\text{for i = 2:} \quad \left(1-\frac{3x_2\Delta x}{2}\right)y_1+\left(-2+7\Delta x^2\right)y_2+\left(1+\frac{3x_2\Delta x}{2}\right)y_3=\Delta x^2\cos\left(2x_2\right)$$

$$\text{for i = 3:} \quad \left(1-\frac{3x_3\Delta x}{2}\right)y_2+\left(-2+7\Delta x^2\right)y_3+\left(1+\frac{3x_3\Delta x}{2}\right)y_4=\Delta x^2\cos\left(2x_3\right)$$

$$\text{for i = 4:} \quad \left(1-\frac{3x_4\Delta x}{2}\right)y_3+\left(-2+7\Delta x^2\right)y_4=\Delta x^2\cos\left(2x_4\right)-\left(1+\frac{3x_4\Delta x}{2}\right)yxf$$

Finally, these four coupled equations with four unknowns (or for the general case, N equations and N unknowns) can be put into matrix form and solved in Matlab. The final form of the equations is

$$\underline{\underline{A}}\,\underline{y}=\underline{b}$$

where $\underline{y}$ is the desired solution vector.

The Matlab file **fd.m** implements this example and it also looks at the sensitivity of the solution to the number of mesh used (i.e. mesh size, $\Delta\mathbf{x}$ ). It also has a comparison to Case B, which only uses a forward approximation to y′.

### *Case B -- Low Order Approximation*

Just to illustrate the strong dependence on the accuracy of the derivative approximations used, we redevelop the above formulation using the forward difference approximation for y′ which is accurate to only first order (instead of the central approximation which is accurate to second order). Thus, the discrete representation for y′ for this case is

$$y_i'=\frac{y_{i+1}-y_i}{\Delta x}$$

Using this relation, the discrete form of the original ODE becomes

$$\left[\frac{y_{i-1}-2y_i+y_{i+1}}{\Delta x^2}\right]+3x_i\left[\frac{y_{i+1}-y_i}{\Delta x}\right]+7y_i=\cos\left(2x_i\right)$$

or $$y_{i-1}-\left(2+3x_i\Delta x-7\Delta x^2\right)y_i+\left(1+3x_i\Delta x\right)y_{i+1}=\Delta x^2\cos\left(2x_i\right)$$

Also for the first node, i = 1, we have

$$-\left(2+3x_1\Delta x-7\Delta x^2\right)y_1+\left(1+3x_1\Delta x\right)y_2=\Delta x^2\cos\left(2x_1\right)-yxo$$

and for the last node, i = N, the balance equation becomes

$$y_{N-1}-\left(2+3x_N\Delta x-7\Delta x^2\right)y_N=\Delta x^2\cos\left(2x_N\right)-\left(1+3x_N\Delta x\right)yxf$$

These equations are also implemented into Matlab file **fd.m** and direct comparisons with Case A are made.

### *The Matlab Files*

The above two FD formulations for this problem are implemented into Matlab file **fd.m**. This file is set up to loop over four different values of N to evaluate the sensitivity to mesh size. For each N, both Cases A and B are solved, and the resultant solution profiles, y(x), are compared. The results for each value of N are plotted within one of four subplots. When complete, these four subplots represent a complete summary of the data for this problem and they are displayed in a single Matlab plot in Fig 5.4.

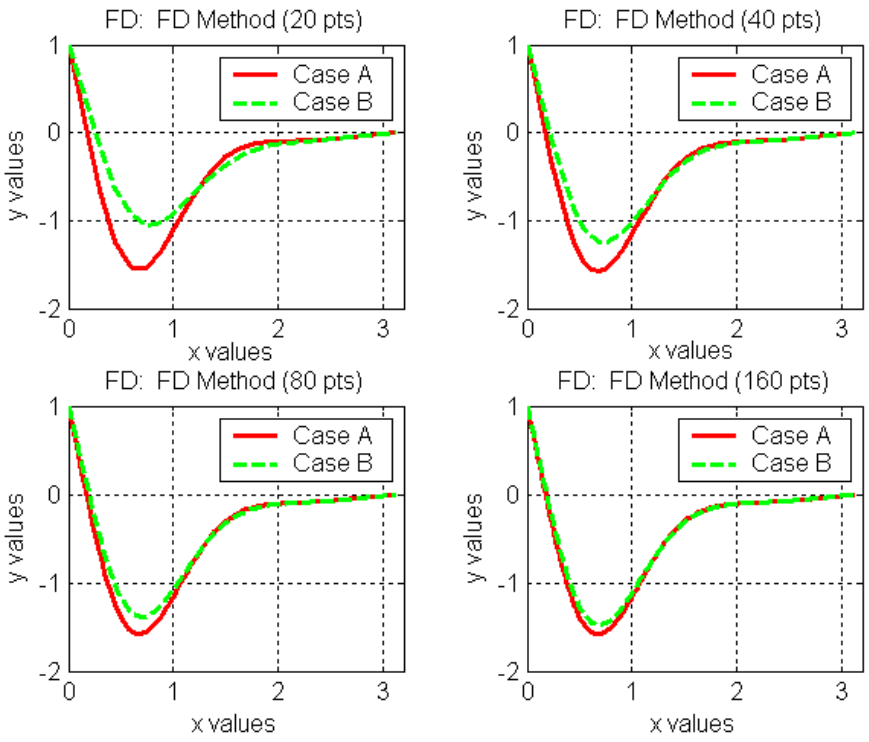



**Fig. 5.4 Solution profiles for Case A and Case B for various mesh structures.**

From the above figure it is easy to see that the Case A and Case B results differ, but they get progressively closer as the number of mesh intervals is increased ( $\Delta x$ is decreased). Also it should be clear that Case A has essentially found the correct solution with about 40 mesh points, and that Case B is still not exact with 160 points. This should illustrate nicely the fact that the best approximations should always be used. In this comparison, $y'$ has an approximation with error $O(h^2)$ for Case A and $O(h)$ for Case B, and clearly this has a significant effect on the accuracy in the FD calculation for a fixed mesh size.

The Matlab files associated with this example are given in Tables 5.7 and 5.8. The base **fd.m** program is given in Table 5.7 and the student is encouraged to study this as an example of how to implement the ***Finite Difference (FD) method*** within a Matlab script file. Also note that, in a typical application, only Case A would be implemented and the loop over several different values of N can be eliminated. Thus, with only a few modifications, **fd.m** can be streamlined into a very short and easy-to-follow example of the FD method in Matlab. This editing has been done and Table 5.8 contains the resultant code within file **fds.m**. The final solution from **fds.m** for Case A with 80 mesh points is plotted in Fig. 5.5, and this is clearly the same solution that

was obtained via the Shooting Method (see Fig. 5.3 in the previous subsection). Either Matlab file, **fd.m** or **fds.m**, can be used as a typical example of the Finite Difference Method applied to a second order non-homogeneous linear BVP.

The student is also referred to Example 5.3B which represents another example of the finite difference method to linear BVPs. This problem deals with conduction heat transfer in a circular fin arrangement.

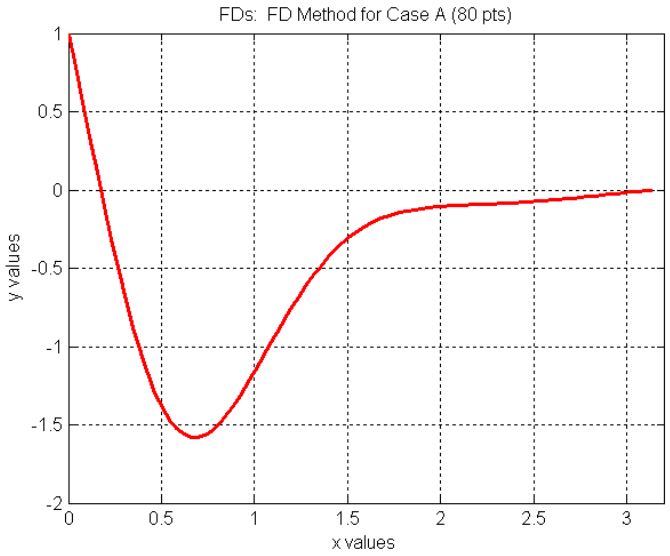



**Fig. 5.5 Final solution profile from fds.m for Case A with 80 mesh.**

**Table 5.7 Listing of Matlab program fd.m.**

```
%
%   FD.M   Example of Finite Difference Method for Solving ODEs
%
%   This demo uses the finite difference technique as the solution method
%   for a particular ODE.  The mesh interval is constant.  Two different
%   approximations for the first derivative are used.  This comparison
%   demonstrates that the best approximation for the derivative terms
%   leads to a more efficient numerical scheme.  One should always use
%   the best approximation available for the given problem!
%
%   The sensitivity to mesh size is also addressed by looping over four
%   different values of N (number of unknowns in the problem).
%
%   Description of given differential equation
%     y'' + 3xy' + 7y  =  cos(2x)      with  y(0) = 1  and  y(pi) = 0
%
%   A related Matlab file, FDS.M, is also available (this contains a simple
%   version of this example - Case A only - with no mesh sensitivity study).
%
%   File prepared by J. R. White, UMass-Lowell (Aug. 2003).
%


%
%   getting started
      clear all, close all, nfig = 0;
%
%   set domain limits and boundary conditions
```

```
      xo = 0;   xf = pi;  yxo = 1;   yxf = 0;
%
%   loop over four values of N to evaluate sensitivity to mesh size
      NN = [20 40 80 160];
%
      for n = 1:4
        N = NN(n);
%
%   compute interval size and discrete x vector
        dx = (xf-xo)/(N+1);   dx2 = dx*dx;    x = (xo+dx):dx:(xf-dx);
%
%
%   Approximations:  y'' = [y(i-1) - 2y(i) + y(i+1)]/(dx^2)  error => (dx^2)
%     (Case A)       y'  = [y(i+1) - y(i-1)]/(2dx)           error => (dx^2)
%
%
%   setup matrix eqns. (treat boundary terms as special cases)
        a = zeros(N,N);   b = zeros(N,1);
        for i = 2:N-1
          a(i,i-1) = 1-3*x(i)*dx/2;
          a(i,i) = -2+7*dx2;
          a(i,i+1) = 1+3*x(i)*dx/2;
          b(i) = dx2*cos(2*x(i));
        end
%   left boundary
        a(1,1) = -2+7*dx2;
        a(1,2) = 1+3*x(1)*dx/2;
        b(1) = dx2*cos(2*x(1))-(1-3*x(1)*dx/2)*yxo;
%   right boundary
        a(N,N-1) = 1-3*x(N)*dx/2;
        a(N,N) = -2+7*dx2;
        b(N) = dx2*cos(2*x(N))-(1+3*x(N)*dx/2)*yxf;
%   solve system of eqns
        y = a\b;
%   add boundary points to solution for plotting
        za = [yxo y' yxf];   xa = [xo x xf];
%
%
%   Approximations: y'' = [y(i-1) - 2y(i) + y(i+1)]/(dx^2)  error => (dx^2)
%     (Case B)      y'  = [y(i+1) - y(i)]/(dx)              error => (dx)
%
%
%   setup matrix eqns. (treat boundary terms as special cases)
        a = zeros(N,N);   b = zeros(N,1);
        for i = 2:N-1
          a(i,i-1) = 1;
          a(i,i) = -(2+3*x(i)*dx-7*dx2);
          a(i,i+1) = 1+3*x(i)*dx;
          b(i) = dx2*cos(2*x(i));
        end
%   left boundary
        a(1,1) = -(2+3*x(1)*dx-7*dx2);
        a(1,2) = 1+3*x(1)*dx;
        b(1) = dx2*cos(2*x(1))-yxo;
%   right boundary
        a(N,N-1) = 1;
        a(N,N) = -(2+3*x(N)*dx-7*dx2);
        b(N) = dx2*cos(2*x(N))-(1+3*x(N)*dx)*yxf;
%   solve system of eqns
        y = a\b;
%   add boundary points to solution for plotting
        zb = [yxo y' yxf];   xb = [xo x xf];
%
%   plot results for both cases
        t = '220+n';
        subplot(eval(t)),plot(xa,za,'r-',xb,zb,'g--','LineWidth',2)
        axis([0 3.2 -2 1]);
        title(['FD:  FD Method (',num2str(N),' pts)'])
        xlabel('x values'),ylabel('y values'),grid
        legend('Case A','Case B')
%
      end
%
%   end of demo
%
```

## Table 5.8 Listing of Matlab program fds.m.

```
%
%   FDS.M   Example of Finite Difference Method for Solving ODEs
%
%   This demo uses the finite difference technique as the solution method
%   for a particular ODE.  The mesh interval is constant.
%
%   Description of given differential equation
%     y'' + 3xy' + 7y  =  cos(2x)      with  y(0) = 1  and  y(pi) = 0
%
%   A related Matlab file, FD.M, is also available (this contains a more detailed
%   version of this example - compares two different approximations for y' and
%   performs a mesh sensitivity study).
%
%   File prepared by J. R. White, UMass-Lowell (Aug. 2003).
%


%
%   getting started
      clear all, close all, nfig = 0;
%
%   set domain limits and boundary conditions
      xo = 0;   xf = pi;  yxo = 1;   yxf = 0;
%
%   user defined number of unknowns in problem
      N = input('Input number of unknowns in problem (N): ');
%
%   compute interval size and discrete x vector
      dx = (xf-xo)/(N+1);   dx2 = dx*dx;    x = (xo+dx):dx:(xf-dx);
%
%
%   Approximations:  y'' = [y(i-1) - 2y(i) + y(i+1)]/(dx^2)  error => (dx^2)
%     (Case A)       y'  = [y(i+1) - y(i-1)]/(2dx)           error => (dx^2)
%
%
%   setup matrix eqns. (treat boundary terms as special cases)
      a = zeros(N,N);   b = zeros(N,1);
      for i = 2:N-1
        a(i,i-1) = 1-3*x(i)*dx/2;
        a(i,i) = -2+7*dx2;
        a(i,i+1) = 1+3*x(i)*dx/2;
        b(i) = dx2*cos(2*x(i));
      end
%   left boundary
      a(1,1) = -2+7*dx2;
      a(1,2) = 1+3*x(1)*dx/2;
      b(1) = dx2*cos(2*x(1))-(1-3*x(1)*dx/2)*yxo;
%   right boundary
      a(N,N-1) = 1-3*x(N)*dx/2;
      a(N,N) = -2+7*dx2;
      b(N) = dx2*cos(2*x(N))-(1+3*x(N)*dx/2)*yxf;
%   solve system of eqns
      y = a\b;
%   add boundary points to solution for plotting
      za = [yxo y' yxf];   xa = [xo x xf];
%
%   plot results
      nfig = nfig+1;  figure(nfig)
      plot(xa,za,'r-','LineWidth',2)
      axis([0 3.2 -2 1]);
      title(['FDs:  FD Method for Case A (',num2str(N),' pts)'])
      xlabel('x values'),ylabel('y values'),grid
%
%   end of demo
%
```

## A Classical BVP -- Heat Transfer in a Circular Fin

### Development of the Mathematical Model

Consider the steady state heat transfer process in a circular fin arrangement as shown in Fig. 5.6.

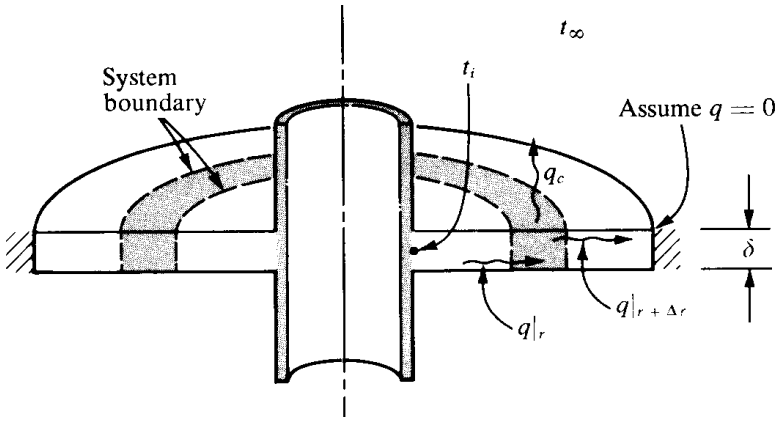



**Fig. 5.6 Overall configuration and finite control volume for circular fin.**

For steady state conditions with azimuthal symmetry and no axial temperature variation, an energy balance on the shaded element gives

$$q_r - q_{r+\Delta r} - q_c = 0 \tag{5.37}$$

In words, this says that the energy coming into the node by conduction must exactly balance the energy leaving the node by conduction and by convection. The conduction terms are given by Fourier's law and the convection component is given by Newton's law of cooling, or

$$-kA_r \left.\frac{dT}{dr}\right|_r + kA_r \left.\frac{dT}{dr}\right|_{r+\Delta r} - hA_c\left(T - T_\infty\right) = 0 \tag{5.38}$$

where the conduction heat transfer area, $A_r$, and convection heat transfer area, $A_c$, are given by

$$A_r = 2\pi r\delta \qquad \text{and} \qquad A_c = 2\pi\left[(r+\Delta r)^2 - r^2\right] = 2\pi\left[2r\Delta r + \Delta r^2\right] \tag{5.39}$$

Upon substitution and division by $\Delta r$, one has

$$\frac{k2\pi r\delta \left.\frac{dT}{dr}\right|_{r+\Delta r} - k2\pi r\delta \left.\frac{dT}{dr}\right|_r}{\Delta r} - 2\pi\left[2r + \Delta r\right]h\left(T - T_\infty\right) = 0$$

and taking the limit as $\Delta r \to 0$ gives

$$\frac{d}{dr}\left[k2\pi r\delta\frac{dT}{dr}\right] - 4\pi rh\left(T - T_\infty\right) = 0 \tag{5.40}$$

Performing the indicated operations with constant k and $\delta$, and rearranging slightly gives the final form as

$$r^2\frac{d^2T}{dr^2}+r\frac{dT}{dr}-\frac{2h}{k\delta}r^2\left(T-T_\infty\right)=0 \tag{5.41}$$

This equation represents the steady state energy balance for this system assuming radial conduction along the fin and convection from the surface of the fin. The boundary conditions for the configuration shown in the above diagram are

$$T(r_w)=T_w \qquad \text{and} \qquad dT/dr\big|_{r_s}=0 \tag{5.42}$$

with w and s (i and o in the figure) referring to the wall and outer surface, respectively, and where all properties are assumed constant.

Equation (5.41) can be written in non-dimensional form using the following substitutions:

$$u=\frac{T-T_\infty}{T_w-T_\infty} \qquad \text{and} \qquad x=\frac{r}{r_s} \tag{5.43}$$

With these relationships the derivative expressions become

$$\frac{du}{dx}=\frac{du}{dr}\frac{dr}{dx}=\frac{r_s}{T_w-T_\infty}\frac{dT}{dr} \qquad \text{and} \qquad \frac{d^2u}{dx^2}=\frac{d}{dr}\left(\frac{du}{dx}\right)\frac{dr}{dx}=\frac{r_s^2}{T_w-T_\infty}\frac{d^2T}{dr^2}$$

Substitution of these relationships into eqn. (5.41) gives

$$\left(r_sx\right)^2\left(\frac{T_w-T_\infty}{r_s^2}\right)\frac{d^2u}{dx^2}+\left(r_sx\right)\left(\frac{T_w-T_\infty}{r_s}\right)\frac{du}{dx}-\frac{2h}{k\delta}\left(r_sx\right)^2\left(T_w-T_\infty\right)u=0$$

and simplification gives

$$x^2\frac{d^2u}{dx^2}+x\frac{du}{dx}-\alpha^2x^2u=0$$

or

$$x^2u''+xu'-\alpha^2x^2u=0 \qquad \text{with} \qquad \alpha^2=\frac{2hr_s^2}{k\delta} \tag{5.44}$$

Also, with the dimensionless variables, the boundary conditions become

$$\text{at } x=r_w/r_s=a,\quad u(a)=1 \qquad \text{and} \qquad \text{at } x=r_s/r_s=b=1,\quad u'(b)=0 \tag{5.45}$$

The solution of the dimensional or non-dimensional equations [eqns. (5.41) - (5.42) or (5.44) - (5.45)] gives the radial temperature profile within this circular fin arrangement, including the tip temperature, $T(r_s)$ or $u(1)$. Also, since this problem is clearly a classical boundary value problem (BVP), one can use both the ***Shooting Method*** (see Example 5.3A) and the ***Finite Difference Method*** (see Example 5.3B) to solve this problem. In addition, this particular problem also can be treated analytically using ***Bessel Functions*** (in Section VIII of these notes -- Example 8.3).

**Reference**: The background for this development and the fin diagram are from ***Analytical Methods in Conduction Heat Transfer*** by G. E. Myers (McGraw-Hill, 1971).

## Example 5.3A -- Shooting Method Solution to the Circular Fin Problem

**Problem Description:**

With the figure, general notation, and the model development given previously, use the ***Shooting Method*** to determine the temperature and temperature gradient profiles for the circular fin problem given the following numerical data:

| | | |
|---|---|---|
| $r_w = 1$ in. | $r_s = 1.5$ in. | $\delta = 0.0625$ in. |
| $T_w = 200$ °F | $T_\infty = 70$ °F | |
| $h = 20$ BTU/hr-ft$^2$-°F | $k = 75$ BTU/hr-ft-°F | |

Evaluate and plot the normalized temperature and gradient profiles and determine the absolute fin edge temperature. Also determine the total heat loss from the fin and compute the fin efficiency, $\eta$, where

$$\eta = \frac{\text{actual heat transfer}}{\text{heat transfer if entire fin is at } T_w}$$

**Problem Solution:**

With the Shooting Method we start the solution process by converting the defining 2nd order ODE into a set of coupled 1st order equations. Using the dimensionless form of the defining equation,

$$x^2u'' + xu' - \alpha^2x^2u = 0 \qquad \text{with} \qquad \alpha^2 = \frac{2hr_s^2}{k\delta}$$

and boundary conditions,

$$\text{at } x = r_w/r_s = a, \quad u(a) = 1 \qquad \text{and} \qquad \text{at } x = r_s/r_s = b = 1, \quad u'(b) = 0$$

letting $\underline{z} = \left[u \;\; u'\right]^T$ gives

$$z_1' = z_2$$

$$z_2' = -\frac{1}{x}z_2 + \alpha^2 z_1$$

or, in general terms, this becomes $\underline{z}' = \underline{f}\left(x, \underline{z}\right)$.

We next set this up as an IVP and use an ODE solver in Matlab to numerically integrate this system, with the initial conditions given by

$$\underline{z}(a) = \begin{bmatrix} u(a) \\ u'(a) \end{bmatrix} = \begin{bmatrix} 1 \\ \alpha \end{bmatrix}$$

An arbitrary guess for $\alpha$, which represents the temperature gradient at the inner wall surface, initiates an automated iterative procedure (as described in Example 5.1 Case 2). In this case the

convergence check is made on the temperature gradient at the fin's outer surface, with the error computed as

$$\varepsilon = z_2(b,\alpha) - u'(b)$$

This quantity should approach zero (an absolute tolerance of 1.0e-6 is set inside **bvp2sh.m**). If the error at the second boundary point is greater than the user-specified tolerance, a 1% change is made in the current value of $\alpha$, a new IVP solution is obtained, and a finite difference estimate for $\partial\varepsilon/\partial\alpha$ is computed. Using this derivative estimate, a new value of $\alpha$ is chosen and a new iteration is initiated, with the process continuing until the preset convergence criterion is met or the maximum number of iterations is reached.

This basic scheme, with the numerical values from the above problem description, was implemented into the main Matlab file, **cylfinsh.m**, and the ODE function file, **cylfinshf.m**. The solution algorithm uses **bvp2sh.m** as described earlier -- with **zbc = [1 0 1.0; 0 1 0.0]**, which specifies both linear endpoint conditions, u(a) = 1.0 and u′(b) = 0.0, appropriately. The Matlab files for this problem are listed in Table 5.9 (a listing of **bvp2sh.m** was given previously in Table 5.5). These programs solve the given problem for the normalized temperature and gradient profiles, u(x) and u′(x), and then evaluate some additional auxiliary information, including the fin's tip temperature, the integral heat loss from the fin, and finally, the overall fin efficiency (as defined above).

The final converged solution is plotted in Fig. 5.7 and the output file, **cylfinsh.out**, generated as part of the Matlab coding is listed in Table 5.10. Clearly the normalized profiles are as expected and the summary data in Table 5.10 indicate a fairly efficient fin arrangement, with an overall efficiency of about 93% and a tip temperature of almost 188 F (a drop of only 12 F from the wall temperature).

This example represents a good illustration of the shooting method to a real problem, including some post-processing of the actual solution profiles to obtain additional information about the system. Since simple heat transfer (combined conduction and convection heat transfer in this problem) is a process that is well understood by most engineers, this particular problem is relatively easy to comprehend. Thus, it can serve as a good illustration of how to apply the general shooting method algorithm, and it should help one apply the method to other non-homogeneous BVPs of individual interest. The set of Matlab m-files associated with this problem, **cylfinsh.m** and **cylfinshf.m**, can be used as a basis for other problems of this type.

## Table 5.9 Listing of the Matlab programs for Example 5.3A.

```
%
%   CYLFINSH.M   Heat Transfer in a Cylindrical Fin (Shooting Method)
%
%   This file solves the cylindrical fin heat transfer problem using the
%   Shooting Method. The base problem is defined via the following equation:
%
%     x^2*u'' + x*u' - ALF2*x^2*u = 0    where   ALF2 = 2*h*rs^2/[k*thk]
%
%     with B.C.   u(a) = 1    and    u'(b) = 0
%      and        a = rw/rs   and    b = rs/rs = 1
%
%     where   u = normalized temp = [T(r) - Tinf]/[Tw - Tinf]
%             x = normalized distance = r/rs
%
%     with   rw, rs = inside and outside radius of fin, respectively
%            thk = thickness of fin
%     and    h, k, Tw, and Tinf are all given quantities (fixed)
%
%   From the normalized solution we can construct absolute profiles (if desired):
%       T(r)  =  Tinf + u(x)[Tw - Tinf]
%       T'(r) =  u'(x)[Tw - Tinf]/rs
%
%   The above 2nd order ODE is first converted into a set of two first order
%   ODEs in standard format (see CYLFINSHF.M) and then solved as an IVP as part
%   of the Shooting Method algorithm.  The basic Shooting method has been implemented
%   within the BVP2SH.M function file (requires linear BCs).
%
%   The above development is given as part of the course notes in the Math
%   Methods course (10/24.539).
%
%   File prepared by J. R. White, UMass-Lowell (Aug. 2003)
%


%
%   getting started
      clear all, close all, nfig = 0;
      global alpha
%
%   basic data for the problem
      rw = 1/12;    rs = 1.5/12;    % inside and outside radius (ft)
      thk = .0625/12;               % thickness of fin (ft)
      Tw = 200;    Tinf = 70;       % inside wall and ambient temps (F)
      h = 20;                       % heat transfer coeff (BTU/hr-ft^2-F)
      k = 75;                       % thermal conductivity (BTU/hr-ft-F)
%
%   derived constants
      a = rw/rs;   b = rs/rs;
      alf2 = (2*h*rs*rs)/(k*thk);    alpha = sqrt(alf2);
      Qideal = 2*pi*h*(rs*rs-rw*rw)*(Tw-Tinf);
%
%   write base data to output file
      fid = fopen('cylfinsh.out','w');
      fprintf(fid,'\n *** CYLFINSH.OUT ***  Data and Results from CYLFINSH.M \n');
      fprintf(fid,'\n \nBASIC DATA FOR PROBLEM \n');
      fprintf(fid,'Inside and outside radius:  rw =  %6.2f ft \t rs =   %5.2f ft \n',rw,rs);
      fprintf(fid,'Thickness of fin:           thk = %6.2f ft \n',thk);
      fprintf(fid,'Inside/ambient temps:       Tw =  %6.2f F  \t Tinf = %5.2f F \n',Tw,Tinf);
      fprintf(fid,'Heat transfer coeff:        h =   %6.2f Btu/hr-ft^2-F \n',h);
      fprintf(fid,'Thermal conductivity:       k =   %6.2f Btu/hr-ft-F \n',k);
%
%   initialize zbc array and call BVP2SH.M to solve problem
      zbc = [1 0 1.0 ;   % u(a) = 1
             0 1 0.0];   % u'(b) = 0
      [x,z] = bvp2sh('cylfinshf',[a b],zbc);
      [nr,nc] = size(z);
%
%   now have desired solution (do auxiliary calcs and plotting)
%   calc heat transferred and fin efficiency (edit key parameters)
      Qactual = -k*(2*pi*a*thk)*(Tw-Tinf)*z(1,2);
      Ttip = Tinf + (Tw-Tinf)*z(nr,1);
      fineff = Qactual/Qideal;
      fprintf(fid,'\n \nFINAL SHOOTING METHOD RESULTS \n');
```

```
      fprintf(fid,'Wall Temp (F) =     %8.3f \n',Tw);
      fprintf(fid,'Ambient Temp (F) =  %8.3f \n',Tinf);
      fprintf(fid,'Tip Temp (F) =      %8.3f \n',Ttip);
      fprintf(fid,'Qideal (BTU/hr) =   %8.3f \n',Qideal);
      fprintf(fid,'Qactual (BTU/hr) =  %8.3f \n',Qactual);
      fprintf(fid,'Fin Eff =           %8.3f \n',fineff);
%
%   plot normalized profiles
      nfig = nfig+1;   figure(nfig)
      subplot(2,1,1),plot(x,z(:,1),'LineWidth',2)
      title('CylFinSH:  Normalized Temp Profile for Cylindrical Fin (Shooting Method)')
      grid,ylabel('temperature')
      subplot(2,1,2),plot(x,z(:,2),'LineWidth',2)
      title('CylFinSH:  Normalized Temp Gradient for Cylindrical Fin (Shooting Method)')
      grid,xlabel('normalized distance'),ylabel('temp gradient')
%
%   close output file
      fclose(fid);
%
%   end simulation




%
%   CYLFINSHF.M   Equations for Circular Fin Problem using the Shooting Method
%                    (called from CYLFINSH.M)
%
%   File prepared by J. R. White, UMass-Lowell (Aug. 2003)
%
      function zp = odefile(x,z)
      global alpha
      zp = zeros(length(z),1);
      zp(1) = z(2);
      zp(2) = -z(2)/x + alpha*alpha*z(1);
%
%   end of function%
```

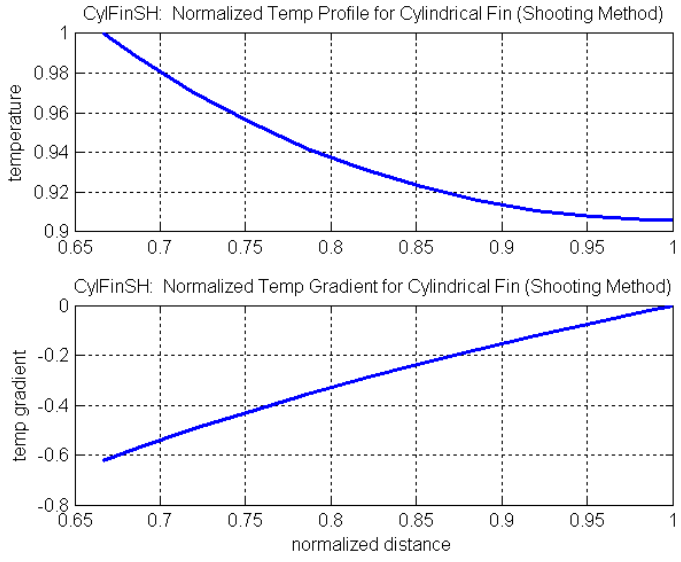



**Fig. 5.7 Converged solution profiles for the circular fin problem (Shooting Method).**

## Table 5.10 Listing of the output file cylfinsh.out.

```
 *** CYLFINSH.OUT ***  Data and Results from CYLFINSH.M


BASIC DATA FOR PROBLEM
Inside and outside radius:  rw =    0.08 ft      rs =    0.13 ft
Thickness of fin:           thk =   0.01 ft
Inside/ambient temps:       Tw =  200.00 F       Tinf = 70.00 F
Heat transfer coeff:        h =    20.00 Btu/hr-ft^2-F
Thermal conductivity:       k =    75.00 Btu/hr-ft-F


FINAL SHOOTING METHOD RESULTS
Wall Temp (F) =      200.000
Ambient Temp (F) =    70.000
Tip Temp (F) =       187.784
Qideal (BTU/hr) =    141.808
Qactual (BTU/hr) =   132.294
Fin Eff =              0.933
```

## Example 5.3B -- Finite Difference Solution to the Circular Fin Problem

**Problem Description:**

With the figure, general notation, and the model development given previously, use the ***Finite Difference Method*** to determine the temperature and temperature gradient profiles for the circular fin problem given the following numerical data:

| | | |
|---|---|---|
| $r_w = 1$ in. | $r_s = 1.5$ in. | $\delta = 0.0625$ in. |
| $T_w = 200$ °F | $T_\infty = 70$ °F | |
| $h = 20$ BTU/hr-ft$^2$-°F | $k = 75$ BTU/hr-ft-°F | |

Evaluate and plot the normalized temperature and gradient profiles and determine the absolute fin edge temperature. Also determine the total heat loss from the fin and compute the fin efficiency, $\eta$, where

$$\eta = \frac{\text{actual heat transfer}}{\text{heat transfer if entire fin is at } T_w}$$

**Problem Solution:**

With the Finite Difference Method, we start the solution process by discretizing the spatial domain and the defining balance equation. The dimensionless form of the steady state energy balance equation was derived previously as,

$$x^2u'' + xu' - \alpha^2x^2u = 0 \qquad \text{where} \qquad \alpha^2 = \frac{2hr_s^2}{k\delta}$$

with boundary conditions,

$$\text{at } x = r_w/r_s = a, \quad u(a) = 1 \qquad \text{and} \qquad \text{at } x = r_s/r_s = b = 1, \quad u'(b) = 0$$

This problem is clearly one dimensional (heat conduction in the normalized x direction) and a side view of a 5-node nodal model is sketched below in Fig. 5.8. Note that, since the left boundary temperature is known, we begin numbering the unknown nodal temperatures with the second control volume, where each temperature is clearly associated with a particular discrete volume element (within the dashed lines). Note also that the nodes associated with the wall temperature and the fin's tip temperature only have half the mesh width as the interior nodes (this fact is important and it will affect any balance equations developed specifically for the boundary nodes). In actual implementation for this problem, N nodes will be used, where N represents the number of unknown temperatures in the discretized model.

Let's start by discretizing the variables using a uniform mesh grid,

$$x \to x_i \qquad \text{where} \qquad x_{i+1} = x_i + \Delta x$$

$$u(x) \to u(x_i) = u_i\ , \qquad u'(x) \to u'(x_i) = u'_i\ , \qquad \text{etc.}$$

For the derivatives let's use the central FD approximation for both u″ and u′, or

$$u_i'' = \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} \quad \text{and} \quad u_i' = \frac{u_{i+1} - u_{i-1}}{2\Delta x}$$

where $\Delta x = (b-a)/N$ represents a uniform mesh grid for this problem.

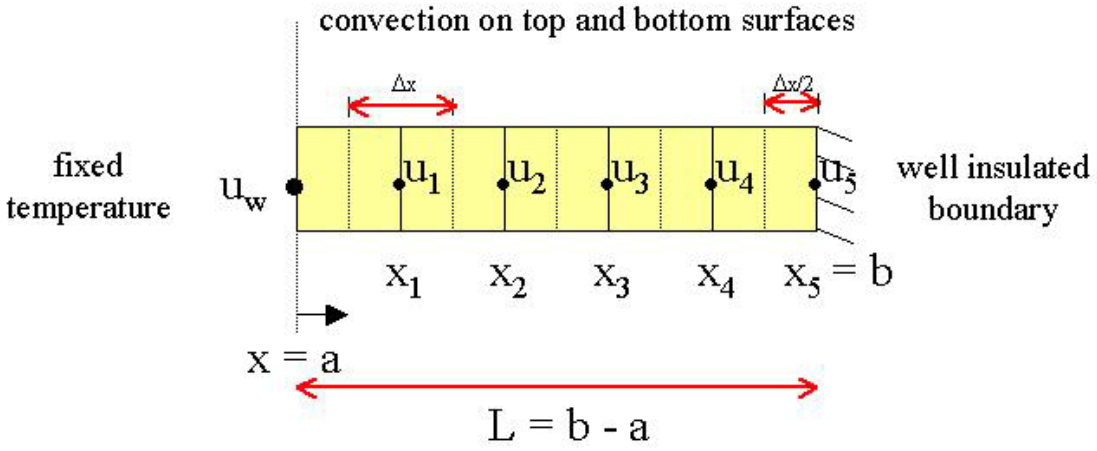



**Fig. 5.8 Discretized representation for the cylindrical fin problem (side view).**

We now substitute these derivative approximations into the discrete form of the original ODE, giving

$$x_i^2\left[\frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2}\right] + x_i\left[\frac{u_{i+1} - u_{i-1}}{2\Delta x}\right] - \alpha^2 x_i^2 u_i = 0$$

Collecting terms and multiplying by $\Delta x^2 / x_i^2$ gives

$$\left(1 - \frac{\Delta x}{2x_i}\right)u_{i-1} - \left(2 + \alpha^2\Delta x^2\right)u_i + \left(1 + \frac{\Delta x}{2x_i}\right)u_{i+1} = 0$$

Therefore, for the general ***internal node*** (row i corresponds to the balance equation for the $i^{th}$ node), we have

$$A(i, i-1) = 1 - \frac{\Delta x}{2x_i}$$

$$A(i, i) = -\left(2 + \alpha^2\Delta x^2\right) \quad \text{and} \quad b(i) = 0$$

$$A(i, i+1) = 1 + \frac{\Delta x}{2x_i}$$

where the A(i, j) and b(i) notation is consistent with the matrix representation used within the Matlab code.

For the boundary nodes, we have to incorporate specific information about the boundary conditions for the problem. For the ***left boundary*** a fixed known temperature is specified, $u_0 = u(a) = 1$. Thus, the nodal balance for node i = 1 gives

$$-\left(2+\alpha^2\Delta x^2\right)u_1+\left(1+\frac{\Delta x}{2x_1}\right)u_2=-\left(1-\frac{\Delta x}{2x_1}\right)u_0$$

where the term on the RHS represents a nonzero entry in the b-vector in the final matrix equations solved within in Matlab.

For the ***right boundary***, things are not so straightforward. The fin's tip is assumed to be well insulated, which implies no heat transfer out the right face. Mathematically, this is represented as $u'(b) = 0$. We could try to incorporate this information into the discretized form of the original ODE but, usually, this introduces a lot of uncertainty and increases the sensitivity to the mesh spacing. Instead, for physically meaningful problems, it is usually much better to perform an explicit balance on the boundary node in question, and incorporate directly the real boundary condition imposed on the problem. In this case, an explicit energy balance on the last nodal volume gives

$$q_{x_{N-1/2}} - q_{x_N} - q_c = 0$$

In words, this says that the energy coming into node N by conduction must exactly balance the energy leaving the node by conduction and by convection, respectively. However, the formal boundary condition for node N says that the conduction heat transfer out of the node is zero (i.e. $q_{x_N} = 0$). Therefore the energy balance equation reduces to

$$-kA_r\left.\frac{dT}{dr}\right|_{r=r_s-\Delta r/2} - hA_c\left(T_N - T_\infty\right) = 0$$

where the conduction heat transfer area, $A_r$, and convection heat transfer area, $A_c$, for node N are given by

$$A_r = 2\pi\left(r_s-\frac{\Delta r}{2}\right)\delta \qquad \text{and} \qquad A_c = 2\pi\left[r_s^2-\left(r_s-\frac{\Delta r}{2}\right)^2\right] = 2\pi\left(r_s\Delta r-\frac{\Delta r^2}{4}\right)$$

Upon substitution and a little algebra, one has

$$\left.\frac{dT}{dr}\right|_{r_s-\Delta r/2} = -\frac{h\Delta r}{k\delta}f_N\left(T_N-T_\infty\right) \qquad \text{where} \qquad f_N = \frac{\left(r_s-\frac{\Delta r}{4}\right)}{\left(r_s-\frac{\Delta r}{2}\right)}$$

and

$$\left.\frac{du}{dx}\right|_{x_N-\Delta x/2} = \frac{r_s}{T_w-T_\infty}\left.\frac{dT}{dr}\right|_{r_s-\Delta s/2} = -\frac{2hr_s^2}{k\delta}\left(\frac{1}{2}\right)\left(\frac{\Delta r}{r_s}\right)f_N\frac{\left(T_N-T_\infty\right)}{\left(T_w-T_\infty\right)}$$

or

$$\left.\frac{du}{dx}\right|_{x_N-\Delta x/2} = -\alpha^2\left(\frac{\Delta x}{2}\right)f_N u_N$$

We can now apply a standard central finite difference approximation for the derivative evaluated at $x = x_N - \Delta x/2$, or

$$\left.\frac{du}{dx}\right|_{x_N-\Delta x/2} = \frac{u_N - u_{N-1}}{\Delta x}$$

This approximation, coupled with the formal node N balance equation, gives the final form for the discrete representation of the right boundary condition, or

$$-\left(1+\alpha^2\frac{\Delta x^2}{2}f_N\right)u_N + u_{N-1} = 0$$

Finally, the discrete balance equations for the left, central, and right nodes are put into matrix form and solved in Matlab. The final form of the equations is

$$\underline{\underline{A}}\,\underline{u} = \underline{b}$$

where $\underline{u}$ is the desired solution vector.

The above equations, with the numerical values from the above problem description, were implemented into the Matlab file **cylfinfd.m**. This file is listed in Table 5.11. The program first sets up the coefficients of the final matrix equation, solves for the normalized temperature distribution, approximates the gradient profile for plotting purposes, and then evaluates some additional auxiliary information, including the fin's tip temperature, the integral heat loss from the fin, and finally, the overall fin efficiency (as defined above).

In this case, we use both a conduction and convection estimate for the total heat loss calculation. The conduction approximation requires an estimate of the temperature gradient at the inner wall, and the convection calculation uses an integral representation, accounting for the convective heat transfer from each control volume (including the wall node at the left boundary). Since the derivative estimate at the left boundary requires a forward difference approximation, it is often very sensitive to the mesh spacing in the vicinity of the boundary. On the other hand, the convective heat loss computation simply involves a summation of terms containing the actual temperature values (a discrete summation approximation to an integral over the surface of the fin). In general, integral estimates are expected to be more accurate for a given mesh representation.

The results from **cylfinfd.m** are summarized in Fig. 5.9 and in Table 5.12. Again, the profiles given in Fig. 5.9 appear reasonable, and they agree nicely with the profiles obtained for this same problem using the Shooting Method (see Fig. 5.7 in the discussion for Example 5.3A). Table 5.12 includes a listing of the output file, **cylfinfd.out**, and it summarizes the integral results for this problem for several different mesh representations (N varies from 25 to 400). In all cases, the results are pretty good, but one does see some sensitivity to the mesh spacing, especially for the computation of the overall heat loss using the wall conduction representation. This is not unexpected, and it simply implies that integral expressions should be used wherever possible in problems solved via the Finite Difference method.

This example represents a good illustration of the finite difference method to a real problem, including some post-processing of the actual solution profiles to obtain additional information about the system. It can serve as a good illustration of how to apply the general FD algorithm for linear problems, and it should help one apply the method to other non-homogeneous linear BVPs of individual interest. Thus, the student should study the **cylfinfd.m** Matlab file carefully, since it can be used as a framework for other problems of this type.

**Table 5.11 Listing of the cylfinfd.m Matlab program.**

```
%
%   CYLFINFD.M   Heat Transfer in a Cylindrical Fin (Finite Difference Solution)
%
%   This file solves the cylindrical fin heat transfer problem using the finite
%   difference method. The base problem is defined via the following equation:
%
%     x^2*u'' + x*u' - ALF2*x^2*u = 0    where   ALF2 = 2*h*rs^2/[k*thk]
%
%     with B.C.   u(a) = 1    and    u'(b) = 0
%      and        a = rw/rs   and    b = rs/rs = 1
%
%     where   u = normalized temp = [T(r) - Tinf]/[Tw - Tinf]
%             x = normalized distance = r/rs
%
%     with   rw, rs = inside and outside radius of fin, respectively
%            thk = thickness of fin
%     and    h, k, Tw, and Tinf are all given quantities (fixed)
%
%   From the normalized solution we can construct absolute profiles (if desired):
%       T(r)  =  Tinf + u(x)[Tw - Tinf]
%       T'(r) =  u'(x)[Tw - Tinf]/rs
%
%   The above 2nd order ODE is first converted into a set of N simultaneous
%   algebraic equations (one for each node in the physical system) and then
%   these are solved using Matlab's standard equation solver.  The mesh interval
%   is constant for this example.
%
%   The above development is given as part of the course notes in the Math
%   Methods course (10/24.539).
%
%   File prepared by J. R. White, UMass-Lowell (Aug. 2003)
%


%
%   getting started
      clear all, close all, nfig = 0;
%
%   basic data for the problem
      rw = 1/12;    rs = 1.5/12;    % inside and outside radius (ft)
      thk = .0625/12;               % thickness of fin (ft)
      Tw = 200;    Tinf = 70;       % inside wall and ambient temps (F)
      h = 20;                       % heat transfer coeff (BTU/hr-ft^2-F)
      k = 75;                       % thermal conductivity (BTU/hr-ft-F)
%
%   derived constants
      a = rw/rs;   b = rs/rs;
      alf2 = (2*h*rs*rs)/(k*thk);    alpha = sqrt(alf2);
      Qideal = 2*pi*h*(rs*rs-rw*rw)*(Tw-Tinf);
%
%   write base data to output file
      fid = fopen('cylfinfd.out','w');
      fprintf(fid,'\n *** CYLFINFD.OUT ***  Data and Results from CYLFINFD.M \n');
      fprintf(fid,'\n \nBASIC DATA FOR PROBLEM \n');
      fprintf(fid,'Inside and outside radius:  rw =  %6.2f ft \t rs =   %5.2f ft \n',rw,rs);
      fprintf(fid,'Thickness of fin:           thk = %6.2f ft \n',thk);
      fprintf(fid,'Inside/ambient temps:       Tw =  %6.2f F  \t Tinf = %5.2f F \n',Tw,Tinf);
      fprintf(fid,'Heat transfer coeff:        h =   %6.2f Btu/hr-ft^2-F \n',h);
      fprintf(fid,'Thermal conductivity:       k =   %6.2f Btu/hr-ft-F \n',k);
%
```

```
%   loop over different N if desired
      u0 = 1;
      N = input('Input number of unknowns/nodes for problem (zero to quit) = ');
      while N ~= 0
        clear A B u x xx zz zzp
        A = zeros(N,N);  B = zeros(N,1);
%
        dx = (b-a)/N;   dx2 = dx*dx;   x = linspace(a+dx,b,N);
        fn = (b-dx/4)/(b-dx/2);
%
%   setup matrix eqns. (treat boundary terms as special cases)
        for n = 2:N-1
          A(n,n-1) = 1-dx/(2*x(n));   A(n,n) = -(2+alf2*dx2);
          A(n,n+1) = 1+dx/(2*x(n));   B(n) = 0;
        end
%   left boundary (fixed temp)
        A(1,1) = -(2+alf2*dx2);   A(1,2) = 1+dx/(2*x(1));
        B(1) = -(1-dx/(2*x(1)))*u0;
%   right boundary (zero heat flux)
        A(N,N-1) = 1;     A(N,N) = -(1+alf2*dx2*fn/2);   B(N) = 0;
%
%   solve system of eqns
        u = A\B;
%
%   add left boundary point to solution for plotting
        xx = [a x];      zz = [u0 u'];
%
%   estimate temperature gradient for plotting
        zzp(1) = (zz(2)-zz(1))/dx;   zzp(N+1) = 0;
        for n = 2:N,   zzp(n) = (zz(n+1)-zz(n-1))/(2*dx);   end
%
%   calc heat transferred and fin efficiency (using cond. & conv. estimates)
        Qactual1 = -k*(2*pi*a*thk)*(Tw-Tinf)*zzp(1);   dr = dx*rs;
        Qactual2 = h*2*pi*((rw+dr/2)^2-rw^2)*(Tw-Tinf)*zz(1) +    ...
                   h*2*pi*(rs^2-(rs-dr/2)^2)*(Tw-Tinf)*zz(N+1);
        for n = 2:N
          r2 = (xx(n)+dx/2)*rs;   r1 = (xx(n)-dx/2)*rs;
          area = 2*pi*(r2^2-r1^2);
          Qactual2 = Qactual2+h*area*(Tw-Tinf)*zz(n);
        end
%
%   edit key parameters
        Ttip = Tinf + (Tw-Tinf)*zz(N+1);    fineff = Qactual2/Qideal;
        fprintf(fid,'\n \nFINITE DIFFERENCE METHOD RESULTS (N = %3i) \n',N);
        fprintf(fid,'Wall Temp (F) =     %8.3f \n',Tw);
        fprintf(fid,'Ambient Temp (F) =  %8.3f \n',Tinf);
        fprintf(fid,'Tip Temp (F) =      %8.3f \n',Ttip);
        fprintf(fid,'Qideal (BTU/hr) =   %8.3f \n',Qideal);
        fprintf(fid,'Qactual (BTU/hr) =  %8.3f (conduction)\n',Qactual1);
        fprintf(fid,'Qactual (BTU/hr) =  %8.3f (convection)\n',Qactual2);
        fprintf(fid,'Fin Eff =           %8.3f \n',fineff);
%
%   plot normalized profiles
        nfig = nfig+1;   figure(nfig)
        subplot(2,1,1),plot(xx,zz,'LineWidth',2)
        title(['CylFinFD:  Normalized Temp Profile for Cylindrical Fin (FD Method N = ', ...
               num2str(N),')'])
        grid,ylabel('temperature')
        subplot(2,1,2),plot(xx,zzp,'LineWidth',2)
        title(['CylFinFD:  Normalized Temp Gradient for Cylindrical Fin (FD Method N = ', ...
               num2str(N),')'])
        grid,xlabel('normalized distance'),ylabel('temp gradient')
%
%   loop back for a different N if desired
        N = input('Enter another value of N if desired (zero to quit)? ');
      end
%
%   close output file
      fclose(fid);
%
%   end simulation
%
```

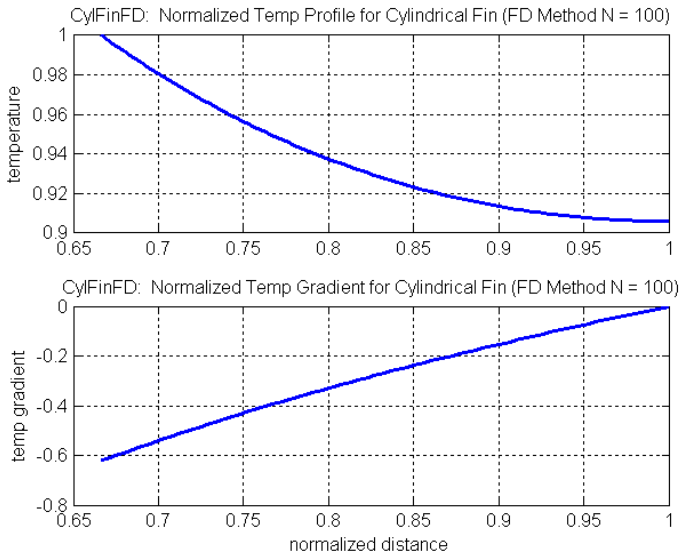



**Fig. 5.9 Solution profiles for the circular fin problem (FD Method with N = 100).**

**Table 5.12 Listing of the output file cylfinfd.out.**

```
 *** CYLFINFD.OUT ***  Data and Results from CYLFINFD.M


BASIC DATA FOR PROBLEM
Inside and outside radius:  rw =    0.08 ft      rs =    0.13 ft
Thickness of fin:           thk =   0.01 ft
Inside/ambient temps:       Tw =  200.00 F       Tinf = 70.00 F
Heat transfer coeff:        h =    20.00 Btu/hr-ft^2-F
Thermal conductivity:       k =    75.00 Btu/hr-ft-F


FINITE DIFFERENCE METHOD RESULTS (N =  25)
Wall Temp (F) =      200.000
Ambient Temp (F) =    70.000
Tip Temp (F) =       187.785
Qideal (BTU/hr) =    141.808
Qactual (BTU/hr) =   128.729 (conduction)
Qactual (BTU/hr) =   132.297 (convection)
Fin Eff =              0.933


FINITE DIFFERENCE METHOD RESULTS (N =  50)
Wall Temp (F) =      200.000
Ambient Temp (F) =    70.000
Tip Temp (F) =       187.785
Qideal (BTU/hr) =    141.808
Qactual (BTU/hr) =   130.504 (conduction)
Qactual (BTU/hr) =   132.294 (convection)
Fin Eff =              0.933
```

```
FINITE DIFFERENCE METHOD RESULTS (N = 100)
Wall Temp (F) =      200.000
Ambient Temp (F) =    70.000
Tip Temp (F) =       187.784
Qideal (BTU/hr) =    141.808
Qactual (BTU/hr) =   131.396 (conduction)
Qactual (BTU/hr) =   132.293 (convection)
Fin Eff =              0.933


FINITE DIFFERENCE METHOD RESULTS (N = 200)
Wall Temp (F) =      200.000
Ambient Temp (F) =    70.000
Tip Temp (F) =       187.784
Qideal (BTU/hr) =    141.808
Qactual (BTU/hr) =   131.844 (conduction)
Qactual (BTU/hr) =   132.293 (convection)
Fin Eff =              0.933


FINITE DIFFERENCE METHOD RESULTS (N = 400)
Wall Temp (F) =      200.000
Ambient Temp (F) =    70.000
Tip Temp (F) =       187.784
Qideal (BTU/hr) =    141.808
Qactual (BTU/hr) =   132.068 (conduction)
Qactual (BTU/hr) =   132.293 (convection)
Fin Eff =              0.933
```