$$\vdots$$
$$-y_{N-3}+2y_{N-2}-y_{N-1}+\Delta t^2 f(t_{N-2}, y_{N-2}, (y_{N-1}-y_{N-3})/2\Delta t) = 0$$
$$-y_{N-2}+2y_{N-1}-\beta+\Delta t^2 f(t_{N-1}, y_{N-1}, (\beta-y_{N-2})/2\Delta t) = 0.$$

This $(N-1)\times(N-1)$ nonlinear system of equations can be very difficult to solve and imposes a severe constraint on the usefulness of the scheme. However, there may be no other way of solving the problem and a solution to this system of equations must be computed. Further complicating the issue is the fact that for nonlinear systems such as these, there are no guarantees about the existence or uniqueness of solutions. The best approach is to use a *relaxation* scheme which is based upon Newton or Secant method iterations.

## 7.7 Implementing MATLAB for Boundary Value Problems

Both a shooting technique and a direct discretization method have been developed here for solving boundary value problems. More generally, one would like to use a high-order method that is robust and capable of solving general, nonlinear boundary value problems. MATLAB provides a convenient and easy to use routine, known as **bvp4c**, that capable of solving fairly sophisticated problems. The algorithm relies on an iteration structure for solving nonlinear systems of equations. In particular, **bvp4c** is a finite-difference code that implements the three-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a $C^1$-continuous solution that is fourth-order accurate uniformly in $x\in[a,b]$. Mesh selection and error control are based on the residual of the continuous solution. Since it is an iteration scheme, its effectiveness will ultimately rely on your ability to provide the algorithm with an initial guess for the solution.

Two example codes will be demonstrated here. The first is a simple linear, constant coefficient second-order equation with fairly standard boundary conditions. The second example is nonlinear with an undetermined parameter (eigenvalue) that must be also determined. Both illustrate the power and ease of use of the build in boundary value solver of MATLAB.

### Linear Boundary Value Problem

As a simple and particular example of a boundary value problem, consider the following:

$$y''+3y'+6y=5 \tag{7.7.1}$$

on the domain $x\in[1,3]$ and with boundary conditions

$$y(1)=3 \tag{7.7.2a}$$
$$y(3)+2y'(3)=5\,. \tag{7.7.2b}$$

This problem can be solved in a fairly straightforward manner using analytic techniques. However, we will pursue here a numerical solution instead.

As with any differential equations solver, the equation must be first put into the form of a system of first order equations. Thus by introducing the variables

$$y_1 = y(x) \tag{7.7.3a}$$
$$y_2 = y'(x) \tag{7.7.3b}$$

we can rewrite the governing equations as

$$y_1' = y_2 \tag{7.7.4a}$$
$$y_2' = 5 - 3y_2 - 6y_1 \tag{7.7.4b}$$

with the transformed boundary conditions

$$y_1(1) = 3 \tag{7.7.5a}$$
$$y_1(3) + 2y_2(3) = 5\,. \tag{7.7.5b}$$

In order to implement the boundary value problem in MATLAB, the boundary conditions need to be placed in the general form

$$f(y_1, y_2) = 0 \ \text{ at } \ x = x_L \tag{7.7.6a}$$
$$g(y_1, y_2) = 0 \ \text{ at } \ x = x_R \tag{7.7.6b}$$

where $f(y_1, y_2)$ and $g(y_1, y_2)$ are the boundary value functions at the left ($x_L$) and right ($x_R$) boundary points. This then allows us to rewrite the boundary conditions in (7.7.5) as the following:

$$f(y_1, y_2) = y_1 - 3 = 0 \qquad \text{at } \ x_L = 1 \tag{7.7.7a}$$
$$g(y_1, y_2) = y_1 + 2y_2 - 5 = 0 \qquad \text{at } \ x_R = 3\,. \tag{7.7.7b}$$

The formulation of the boundary value problem is then completely specified by the differential equation (7.7.4) and its boundary conditions (7.7.7).

The boundary value solver **bvp4c** requires three pieces of information: the equation to be solved, its associated boundary conditions, and your initial guess for the solution. The first two lines of the following code performs all three of these functions:

```
init=bvpinit(linspace(1,3,10),[0 0]);
sol=bvp4c(@bvp_rhs,@bvp_bc,init);
x=linspace(1,3,100); BS=deval(sol,x);
plot(x,BS(1,:))
```

We dissect this code by first considering the first line of code for generating a MATLAB data structure for use as the initial data. In this initial line of code,

the **linspace** command is used to define the initial mesh that goes from $x = 1$ to $x = 3$ with 10 equally spaced points. In this example, the initial values of $y_1(x) = y(x)$ and $y_2(x) = y'(x)$ are zero. If you have no guess, then this is probably your best guess to start with. Thus you not only guess the initial guess, but the initial grid on which to find the solution.

Once the guess and initial mesh is generated, two functions are called with the **@bvp_rhs** and **@bvp_bc** function calls. These are functions representing the differential equation and boundary conditions (7.7.4) and (7.7.7) respectively. These functions are easily constructed as the following:

**bvp_rhs.m**

```
function rhs=bvp_rhs(x,y)
rhs=[y(2);  5 - 3*y(2) - 6*y(1)];
```

Likewise, the boundary conditions are constructed from the code

**bvp_rhs.m**

```
function bc=bvp_bc(yL,yR)
bc=[yL(1)-3;  yR(1)+2*yR(2)-5];
```

Note that what is passed into the boundary condition function is the value of the vector **y** at the left (**yL**) and right (**yR**) of the computational domain, i.e. at the values of $x = 1$ and $x = 3$.

What is produced by **bvp4c** is a data structure **sol**. This data structure contains a variety of information about the problem, including the solution $y(x)$. To extract the solution, the final two lines of code in the main program are used. Specifically, the **deval** command evaluates the solution at the points specified by the vector **x**, which in this case is a linear space of 100 points between one and three. The solution can then simply be plotted once the values of $y(x)$ have been extracted. Figure 60 shows the solution and its derivative that are produced from the boundary value solver.

### Nonlinear eigenvalue problem

As a more sophisticated example of a boundary value problem, we will consider a fully nonlinear eigenvalue problem. Thus consider the following:

$$y'' + (100 - \beta)y + \gamma y^3 = 0 \tag{7.7.8}$$

on the domain $x \in [-1, 1]$ and with boundary conditions

$$y(-1) = 0 \tag{7.7.9a}$$
$$y(1) = 0\,. \tag{7.7.9b}$$

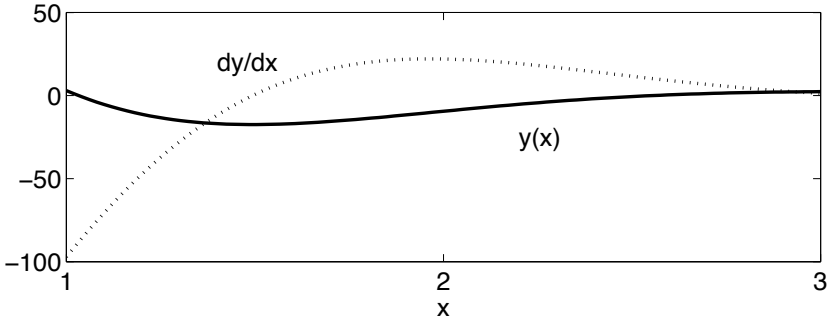


Figure 60: The solution of the boundary value problem (7.7.4) with (7.7.7). The solid line is $y(x)$ while the dotted line is its derivative $dy(x)/dx$.

This problem cannot be solved using analytic techniques due to the complexity introduced by the nonlinearity. But a numerical solution can be fairly easily constructed. Note the similarity between this problem and that considered in the shooting section.

As before, the equation must be first put into the form of a system of first order equations. Thus by introducing the variables

$$y_1 = y(x) \tag{7.7.10a}$$
$$y_2 = y'(x) \tag{7.7.10b}$$

we can rewrite the governing equations as

$$y_1' = y_2 \tag{7.7.11a}$$
$$y_2' = (\beta - 100)y_1 - \gamma y_1^3 \tag{7.7.11b}$$

with the transformed boundary conditions

$$y_1(-1) = 0 \tag{7.7.12a}$$
$$y_1(1) = 0\,. \tag{7.7.12b}$$

The formulation of the boundary value problem is then completely specified by the differential equation (7.7.11) and its boundary conditions (7.7.12). Note that unlike before, we do not know the parameter $\beta$. Thus it must be determined along with the solution. As with the initial conditions, we will also guess an initial value of $\beta$ and let **bvp4c** converge to the appropriate value of $\beta$. Note that for such nonlinear problems, the effectiveness of **bvp4c** relies almost exclusively on providing a good initial guess.

As before, the boundary value solver **bvp4c** requires three pieces of information: the equation to be solved, its associated boundary conditions, and your

initial guess for the solution and the parameter $\beta$. The first line of the code gives the initial guess for $\beta$ while the next two lines of the following code perform the three remaining functions:

```
beta=99;
init2=bvpinit(linspace(-1,1,50),@mat4init,beta);
sol2=(bvp4c(@bvp_rhs2,@bvp_bc2,init2));
x2=linspace(-1,1,100); BS2=deval(sol2,x2);
figure(2), plot(x2,BS2(1,:))
```

In this case, there are now three function calls: **@bvp_rhs2**, **@bvp_bc2** and **@mat4init**. This calls the differential equation, its boundary values and its initial guess respectively.

We begin with the initial guess. Our implementation example of the shooting method showed that the first linear solution behaved like a cosine function. Thus we can guess that $y_1(x) = y(x) = \cos[(\pi/2)x]$ where the factor of $\pi/2$ is chosen to make the solution zero at $x = \pm 1$. Note that initially fifty points are chosen between $x = -1$ to $x = 1$. The following function generates the initial data to begin the iteration process.

**mat4init.m**

```
function yinit = mat4init(x)
yinit = [cos((pi/2)*x); -(pi/2)*sin((pi/2)*x)];
```

Again, it must be emphasized the success of **bvp4c** relies almost exclusively on the above subroutine and guess. Without a good guess, especially for nonlinear problems, you may find a solution, but just not the one you want.

The equation itself is handled in the subroutine **bvp_rhs2.m**. The following is the construction of the right hand side function with $\gamma = 1$.

**bvp_rhs2.m**

```
function rhs=bvp_rhs2(x,y,beta)
rhs=[y(2); ((beta-100)*y(1)-y(1)^3)];
```

Finally, the implementation of the boundary conditions, or constraints, must be imposed. There is something very important to note here: we started with two constraints since we have a second order differential equation, i.e. $y(\pm 1) = 0$. However, since we do not know the value of $\beta$, the system is currently an underdetermined system of equations. In order to remedy this, we need to impose one more constraint on the system. This is somewhat arbitrary unless there is a natural constraint for you to choose in the system. Here, we will simply choose $dy(-1)/dx = 0.1$, i.e. we will impose the launch angle at the left. This results in the boundary value imposition routine:

**bvp_bc2.m**

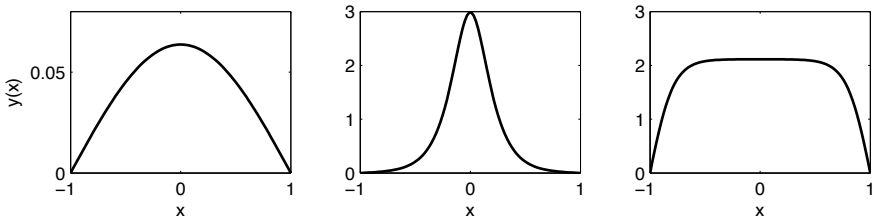


Figure 61: Three different nonlinear eigenvalue solutions to (7.7.11) and its boundary conditions (7.7.12). The left panel has the constraint condition $y'(-1) = 0.1$ with $\gamma = 1$, the middle panel has $y'(-1) = 0.1$ with $\gamma = 10$ and the right panel has $y'(-1) = 10$ with $\gamma = -10$. Such solutions are called *nonlinear ground states* of the eigenvalue problem.

```
function bc=bvp_bc2(yl,yr,beta)
bc=[yl(1); yl(2)-0.1; yr(1)];
```

With these three constraints, the **bvp4c** iteration routine not only finds the solution, but also the appropriate value of $\beta$ that satisfies the above constraints.

Executing the routine and subroutines generates the solution to the boundary value problem. One can also experiment with different guesses and values of $\beta$ to generate new and interesting solutions. Three such solutions are demonstrated in Fig. 61 as a function of the $\beta$ value and the guess angle $dy(-1)/dx$.

## 7.8 Linear Operators and Computing Spectra

As a final application of boundary value problems, we will consider the ability to accurately compute the spectrum of linear operators. Linear operators often arise in the context of evaluating the stability of solutions to partial differential equations. Indeed, stability plays a key role in many branches of science and engineering, including aspects of fluid mechanics, pattern formation with reaction-diffusion models, high-speed transmission of optical information, and the feasibility of MHD fusion devices, to name a few. If one can find solutions for a given particle differential equation, then the stability of that solution becomes critical in determining the ultimate behavior of the system. Specifically, if a physical phenomenon is observable and persists, then the corresponding solution to a valid mathematical model should be stable. If, however, instability is established, the nature of the unstable modes suggest what patterns may develop from the unstable solutions. Finally, for many problems of physical interest, fundamental mathematical models are well established. However, in many cases these fundamental models are too complicated to allow for detailed analysis, thus leading to the study of simpler approximate (linear) models using reductive perturbation methods.