

5.3 Boundary value problems: the shooting method

To this point, we have only considered the solutions of differential equations for which the initial conditions are known. However, many physical applications do not have specified initial conditions, but rather some given boundary (constraint) conditions. A simple example of such a problem is the second-order boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \quad (5.3.1)$$

on $t \in [a, b]$ with the general boundary conditions

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1 \quad (5.3.2a)$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2. \quad (5.3.2b)$$

Thus the solution is defined over a specific interval and must satisfy the relations (5.3.2) at the end points of the interval. Figure 4 gives a graphical representation of a generic boundary value problem solution. We discuss the algorithm necessary to make use of the time-stepping schemes in order to solve such a problem.

The Shooting Method

The boundary value problems constructed here require information at the present time ($t = a$) and a future time ($t = b$). However, the time-stepping schemes developed previously only require information about the starting time $t = a$. Some effort is then needed to reconcile the time-stepping schemes with the boundary value problems presented here.

We begin by reconsidering the generic boundary value problem

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right) \quad (5.3.3)$$

on $t \in [a, b]$ with the boundary conditions

$$y(a) = \alpha \quad (5.3.4a)$$

$$y(b) = \beta. \quad (5.3.4b)$$

The stepping schemes considered thus far for second order differential equations involve a choice of the initial conditions $y(a)$ and $y'(a)$. We can still approach the boundary value problem from this framework by choosing the “initial” conditions

$$y(a) = \alpha \quad (5.3.5a)$$

$$\frac{dy(a)}{dt} = A, \quad (5.3.5b)$$

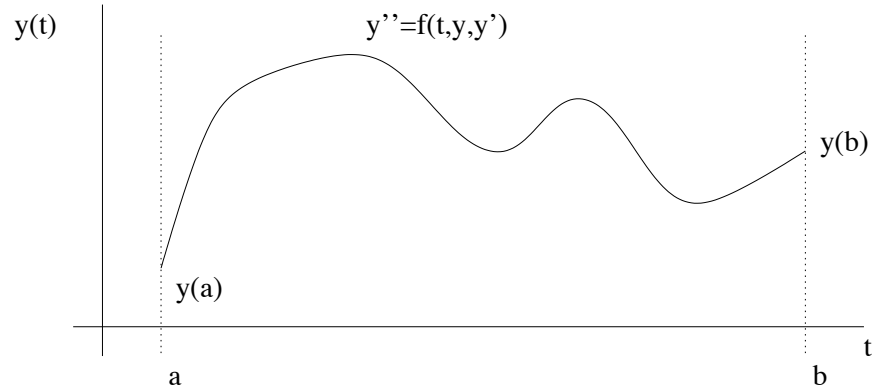


Figure 4: Graphical depiction of the structure of a typical solution to a boundary value problem with constraints at $t = a$ and $t = b$.

where the constant A is chosen so that as we advance the solution to $t = b$ we find $y(b) = \beta$. The shooting method gives an iterative procedure with which we can determine this constant A . Figure 5 illustrates the solution of the boundary value problem given two distinct values of A . In this case, the value of $A = A_1$ gives a value for the initial slope which is too low to satisfy the boundary conditions (5.3.4), whereas the value of $A = A_2$ is too large to satisfy (5.3.4).

Computational Algorithm

The above example demonstrates that adjusting the value of A in (5.3.5b) can lead to a solution which satisfies (5.3.4b). We can solve this using a self-consistent algorithm to search for the appropriate value of A which satisfies the original problem. The basic algorithm is as follows:

1. Solve the differential equation using a time-stepping scheme with the initial conditions $y(a) = \alpha$ and $y'(a) = A$.
2. Evaluate the solution $y(b)$ at $t = b$ and compare this value with the target value of $y(b) = \beta$.
3. Adjust the value of A (either bigger or smaller) until a desired level of tolerance and accuracy is achieved. A bisection method for determining values of A , for instance, may be appropriate.
4. Once the specified accuracy has been achieved, the numerical solution is complete and is accurate to the level of the tolerance chosen and the discretization scheme used in the time-stepping.

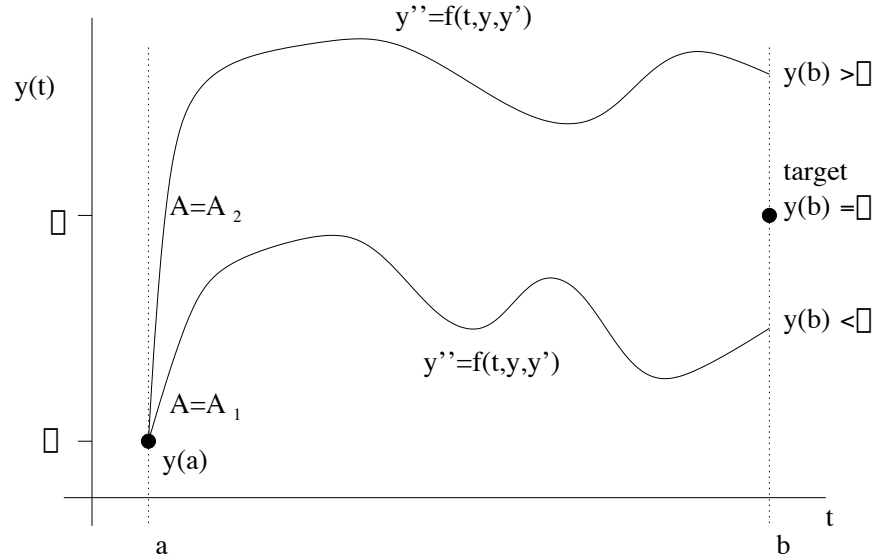


Figure 5: Solutions to the boundary value problem with $y(a) = \alpha$ and $y'(a) = A$. Here, two values of A are used to illustrate the solution behavior and its lack of matching the correct boundary value $y(b) = \beta$. However, the two solutions suggest that a bisection scheme could be used to find the correct solution and value of A .

We illustrate graphically a bisection process in Fig. 6 and show the convergence of the method to the numerical solution which satisfies the original boundary conditions $y(a) = \alpha$ and $y(b) = \beta$. This process can occur quickly so that convergence is achieved in a relatively low amount of iterations provided the differential equation is well behaved.

Implementing MATLAB time-stepping schemes

Up to this point in our discussions of differential equations, our solution techniques have relied on solving initial value problems with some appropriate iteration procedure. To implement one of these solution schemes in MATLAB requires information about the equation, the time or spatial range to solve for, and the initial conditions.

To build on a specific example, consider the third-order, nonhomogeneous, differential equation

$$\frac{d^3 u}{dt^3} + u^2 \frac{du}{dt} + \cos t \cdot u = A \sin^2 t. \tag{5.3.6}$$

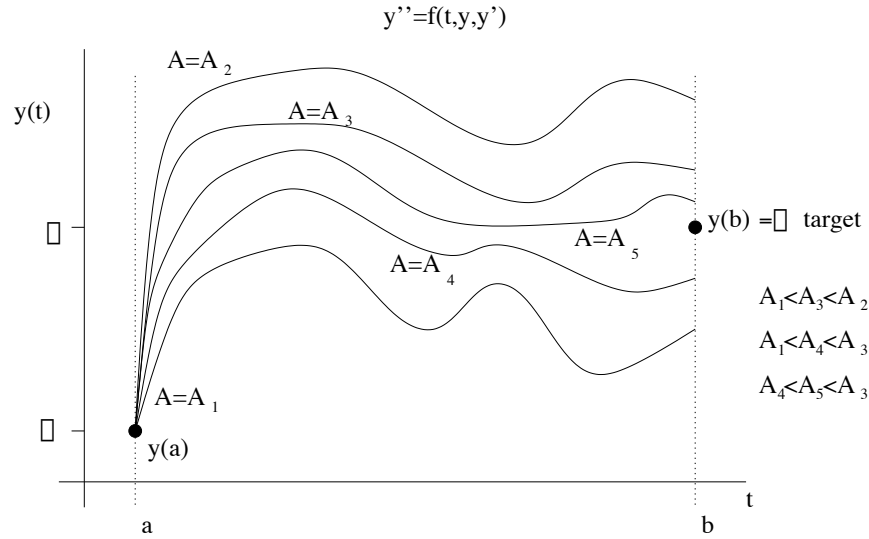


Figure 6: Graphical illustration of the shooting process which uses a bisection scheme to converge to the appropriate value of A for which $y(b) = \beta$.

By defining

$$y_1 = u \tag{5.3.7a}$$

$$y_2 = \frac{du}{dt} \tag{5.3.7b}$$

$$y_3 = \frac{d^2u}{dt^2}, \tag{5.3.7c}$$

we find that $dy_3/dt = d^3u/dt^3$. Using the original equation along with the definitions of y_i we find that

$$\frac{dy_1}{dt} = y_2 \tag{5.3.8a}$$

$$\frac{dy_2}{dt} = y_3 \tag{5.3.8b}$$

$$\frac{dy_3}{dt} = \frac{d^3u}{dt^3} = -u^2 \frac{du}{dt} - \cos t \cdot u + A \sin^2 t = -y_1^2 y_2 - \cos t \cdot y_1 + A \sin^2 t \tag{5.3.8c}$$

which results in the first-order system (5.1.1) considered previously

$$\frac{dy}{dt} = \frac{d}{dt} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ -y_1^2 y_2 - \cos t \cdot y_1 + A \sin^2 t \end{pmatrix} = f(\mathbf{y}, t). \tag{5.3.9}$$

At this point, all the time-stepping techniques developed thus far can be applied to the problem. It is imperative to write any differential equation as a first-order system before solving it numerically with the time-stepping schemes developed here.

Before solving the equation, the initial conditions and time span must be defined. The initial conditions in this case specify $y_1(t_0)$, $y_2(t_0)$ and $y_3(t_0)$, or equivalently $u(t_0)$, $u_t(t_0)$ and $u_{tt}(t_0)$. The time span specifies the beginning and end times for solving the equations. Thus $t \in [t_0, t_f]$, where t_0 is the initial time and t_f the final time. For this example, let's consider solving the equations with initial conditions $y_1(0) = 1$, $y_2(0) = 0$ and $y_3(0) = 0$ over the interval $t \in [t_0, t_f] = [0, 30]$ with $A = 2$. In MATLAB this would be accomplished with

```
y0=[1 0 0];      % initial conditions
tspan=[0 30];    % time interval
A=2;             % parameter value
```

These can now be used in calling the differential equation solver.

The basic command structure and function call to solve this equation is given as follows:

```
[t,y]=ode45('rhs',tspan,y0,[],A)
```

This piece of code calls a function **rhs.m** which contains information about the differential equations. It sends to this function the time span for the solution, the initial conditions, and the parameter value of A . The brackets are placed in a position for specifying accuracy and tolerance options. By placing the brackets with nothing inside, the default accuracy will be used by **ode45**. To access other methods, simply **ode45** with **ode23**, **ode113**, or **ode15s** as needed.

The function **rhs.m** contains all the information about the differential equation. Specifically, the right hand side of (5.3.9) is of primary importance. This right hand side function looks as follows:

```
function rhs=rhs(t,y,dummy,A) % function call setup
rhs=[y(1);
     y(2);
     -(y(1)^2)*y(2)-cos(t)*y(1)+A*sin(t)]; % rhs vector
```

This function imports the variable t which takes on values between those given by $tspan$. Thus t starts at $t = 0$ and ends at $t = 30$ in this case. In between these points, t gives the current value of time. The variable y stores the solution. Initially, it takes on the value of y_0 . The *dummy* slot is for sending in information about the tolerances and the last slot with A allows you to send in any parameters necessary in the right hand side.

Note that the output of **ode45** are the vectors t and y . The vector t gives all the points between $tspan$ for which the solution was calculated. The solution

at these time points is given as the columns of matrix y . The first column corresponds to the solution y_1 , the second column contains y_2 as a function of time, and the third column gives y_3 . To plot the solution $u = y_1$ as a function of time, we would plot the vector t versus the first column of y :

```
figure(1), plot(t,y(:,1)) % plot solution u
figure(2), plot(t,y(:,2)) % plot derivative du/dt
```

A couple of important points and options. The **inline** command could also be used in place of the function call. Also, it may be desirable to have the solution evaluated at specified time points. For instance, if we wanted the solution evaluated from $t = 0$ to $t = 30$ in steps of 0.5, then choose

```
tspan=0:0.5:30 % evaluate the solution every dt=0.5
```

Note that MATLAB still determines the step sizes necessary to maintain a given tolerance. It simply interpolates to generate the values at the desired locations.