Numerical Solutions of Optimal Control Problems
by S. Lenhart and J. T. Workman

Consider the optimal control problem

$$\max_u \int_{t_0}^{t_1} f(t, x(t), u(t))\, dt$$
$$\text{subject to} \quad x'(t) = g(t, x(t), u(t))$$
$$x(t_0) = x_0,\ x(t_1) \text{ free.}$$

In order to solve this problem numerically, namely, to find a piecewise continuous function $u(t)$ which maximizes the integral, total-enumeration methods or linear programming techniques can be employed. However, as we saw in the previous chapters, any solution to the above optimal control problem must also satisfy

$$x'(t) = g(t, x(t), u(t)),\ x(t_0) = x_0,$$
$$\lambda'(t) = -\frac{\partial H}{\partial x} = -(f_x(t, x, u) + \lambda(t) g_x(t, x, u)),\ \lambda(t_1) = 0,$$
$$0 = \frac{\partial H}{\partial u} = f_u(t, x, u) + \lambda(t) g_u(t, x, u) \text{ at } u^*.$$

The third equation, the optimality condition, can usually be manipulated to find a representation of u^* in terms of t, x, and λ. If this representation is substituted back into the ODEs for x, λ, then the first two equations form a two-point boundary value problem. There exist many numerical methods to solve ordinary differential equations, such as Runge-Kutta or adaptive schemes, and boundary value problems, such as shooting methods. Any of these methods could be used to solve the optimality system, and thus, the optimal control problem (if appropriate existence and uniqueness results are established).

We wish to take advantage of certain characteristics of the optimality system, however. First, we are given an initial condition for the state x but a final time condition for the adjoint λ. Second, g is a function of t, x, and u only. Values for λ are not needed to solve x using a standard ODE solver. Taking this into account, the method we present here is very intuitive. It is generally referred to as the Forward-Backward Sweep method. A rough outline of the algorithm is given below.

Step 1. Make an initial guess for u over the interval. Store the initial guess as u.

Step 2. Using the initial condition $x(t_0) = x_0$ and the stored values for u, solve x forward in time according to its differential equation in the optimality system.

Step 3. Using the transversality condition $\lambda(t_1) = 0$ and the stored values for u and x, solve λ backward in time according to its differential equation in the optimality system.

Step 4. Update the control by entering the new x and λ values into the characterization of u.

Step 5. Check convergence. If values of the variables in this iteration and the last iteration are negligibly small, output the current values as solutions. If values are not small, return to Step 2.

We make a few notes about the algorithm. When making the initial guess, $u \equiv 0$ is almost always sufficient. For certain problems, where division by u occurs, for example, a different initial guess must be used. Occasionally, your initial guess may require adjusting if the algorithm has problems converging. Although it is not necessary, many times in Step 4, a convex combination between the previous control values and values given by the current characterization are used in updating u. This often helps to speed the convergence. As you will see, this is done in the provided codes. For Steps 2 and 3, any standard ODE solver can be used. For the purposes of this text, a Runge-Kutta 4 routine is used. Specifically, if $x'(t) = f(t, x(t))$ and $x(t)$ are known, then the approximation of $x(t+h)$ is given by

$$x(t+h) = x(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{1}$$

where

$$\begin{aligned} k_1 &= f(t, x(t)) \\ k_2 &= f(t + \frac{1}{2}h, x(t) + \frac{1}{2}hk_1) \\ k_3 &= f(t + \frac{1}{2}h, x(t) + \frac{1}{2}hk_2) \\ k_4 &= f(t + h, x(t) + hk_3). \end{aligned} \tag{2}$$

Information on the stability and accuracy of this and other Runge-Kutta routines is found in numerous texts.

Many types of convergence tests exist for Step 5. Often times, it is sufficient to require $\sum_{i=1}^{n} |u(i) - u_{old}(i)|$ to be small, where $u(i)$ is the vector of estimated values of the control during the current iteration, and $u_{old}(i)$ is the vector of estimated values from the previous iteration. Both these vectors are of length n, which is the number of time steps. In this text, we use a slightly stricter convergence test. Namely, we will require the percentage error to be negligibly small, i.e.,

$$\frac{|u - u_{old}|}{|u|} \leq \delta$$

where δ is the accepted tolerance. Generally in numerical analysis, this would be required for all vector values of u and u_{old}. However, we must make two small adjustments. First, we must allow for $u = 0$, so we multiply both sides by u to remove it from the denominator. Second, if the estimates of $u(i)$ converge to 0 for some i, then this condition will usually not be met at i. Therefore, we only require the condition be met for the averages of u and u_{old}. Of course, considering the averages is equivalent to considering the sums, so our convergence criterion is

$$\delta \sum_{i=1}^{n} |u(i)| - \sum_{i=1}^{n} |u(i) - u_{old}(i)| \geq 0.$$

We will actually make this requirement of all variables, not just the control u. In the lab programs, we take $n = N + 1$ and $\delta = 0.001$. (with N usually 1000)

The remainder of this chapter will be devoted to further explanation of the Forward-Backward Sweep algorithm by way of example. Consider the optimal control problem

$$\max_u \int_0^1 Ax(t) - Bu^2(t)\,dt$$
$$\text{subject to} \quad x'(t) = -\frac{1}{2}x^2(t) + Cu(t)$$
$$x(0) = x_0 \text{ fixed}, \; x(1) \text{ free}.$$

We require $B > 0$ so that this is a maximization problem. Develop the optimality system of this problem by first noting the Hamiltonian is

$$H = Ax - Bu^2 - \frac{1}{2}\lambda x^2 + C\lambda u.$$

Using the optimality condition,

$$0 = \frac{\partial H}{\partial u} = -2Bu + C\lambda \;\Rightarrow\; u^* = \frac{C\lambda}{2B}.$$

By calculating $\frac{\partial H}{\partial x}$, we find that

$$x'(t) = -\frac{1}{2}x^2 + Cu, \; x(0) = x_0$$
$$\lambda'(t) = -A + x\lambda, \; \lambda(1) = 0.$$

Using these two differential equations and representation of u^*, we generate the numerical code as described above, written in MATLAB. The code can be viewed in its entirety in the file *code1.m*, and is also shown in increments below.

code1.m

```
1  function y = code1(A,B,C,x0)
2
3  test = -1;
4  N=1000
5  t = linspace(0,1,N+1);
6  h = 1/N;
7  h2 = h/2;
8
9  u = zeros(1,N+1);
10
11 x = zeros(1,N+1);
12 x(1) = x0;
13 lambda = zeros(1,N+1);
14
15 while(test < 0)
```

Line 1 establishes the MATLAB function code1 and variables A, B, C, and x_0 as inputs. The variable y is the output. The variable *test* created in Line 3 is the convergence test variable. It begins the *while* loop in Line 15. The loop, as we will see, contains the forward-backward sweep. Once convergence occurs, *test* will become non-negative, and the *while* loop will end. In Line 5, a vector t representing the time variable is created. The MATLAB function *linspace* creates $N+1$ equally spaced nodes between 0 and 1. In Line 6, the spacing between these nodes is assigned as h. Line 7 establishes a convenient short-hand which is used in the Runge-Kutta subroutine. Line 9 is our initial guess for the control u, namely, $u = 0$ at each of the $N+1$ nodes. Lines 11 and 13 declare the variables x and λ and their size. These are not guesses, as these values will be overwritten during the sweep process. The initial value of x is stored in Line 12.

code1.m

```
17    oldu = u;
18    oldx = x;
19    oldlambda = lambda;
```

Line 17 - 19 are the first lines inside the *while* loop, which begins the sweep process. These lines store the vectors u, x, and λ as previous values, which we denote as *old_*. Recall that in our convergence test, we require the values of the current and previous iterations. After storing the current values as the previous ones here, new values are generated in the following lines.

code1.m

```
21    for i = 1:N
22        k1 = -0.5*x(i)^2 + C*u(i);
23        k2 = -0.5*(x(i) + h2*k1)^2 + C*0.5*(u(i) + u(i+1));
24        k3 = -0.5*(x(i) + h2*k2)^2 + C*0.5*(u(i) + u(i+1));
25        k4 = -0.5*(x(i) + h*k3)^2 + C*u(i+1);
26        x(i+1) = x(i) + (h/6)*(k1 + 2*k2 + 2*k3 + k4);
27    end
```

Lines 21 - 27 contain the Runge-Kutta sweep solving x forward in time. Line 21 begins the *for* loop and Line 27 ends it. Line 22 calculates the k_1 value, which is simply the RHS of the differential equation. In Line 23, to find k_2, x is replaced with $x + \frac{h}{2}k_1$. We are also to adjust the time variable t by replacing it with $t + \frac{h}{2}$. There is no explicit dependence on t in the differential equation, but u is a function of t. So, we should replace $u(i)$ with $u(i + \frac{h}{2})$. However, this value is not assigned by our vector. There are many ways to approximate this value. An interpolating polynomial or spline of u could be generated, for example. However, it usually suffices to approximate it with the average $\frac{u(i)+u(i+1)}{2}$. In Line 24, the prescribed changes in x and u are made. In Line 25, a full time step is called for, so $u(i+1)$ is used. Line 26 generates the next iterated value of the state x. Note, as $x(1)$ is used to find $x(2)$, $x(2)$ to find $x(3)$, and so on, $x(N)$ is used to find $x(N+1)$. This is why in Line 21, i only runs to N, not $N+1$.

code1.m

```
29      for i = 1:N
30          j = N+2 - i;
31          k1 = -A + lambda(j)*x(j);
32          k2 = -A + (lambda(j) - h2*k1)*0.5*(x(j)+x(j-1));
33          k3 = -A + (lambda(j) - h2*k2)*0.5*(x(j)+x(j-1));
34          k4 = -A + (lambda(j) - h*k3)*x(j-1);
35          lambda(j-1) = lambda(j) - (h/6)*(k1 + 2*k2 + 2*k3 + k4);
36      end
```

Lines 29 - 36 consist of the Runge-Kutta sweep solving λ backward in time. The *for* loop begins in Line 29, while the new index is introduced in Line 30. Notice that as i counts forward from 1 to N, j counts backward from $N+1$ to 2. Line 31 is the k_1 calculation, which comes directly from the differential equation. In Line 32, λ is replaced by $\lambda - \frac{h}{2}k_1$. Notice the minus sign, as we are moving backward in time, so the time increment should actually be $-1/N$. As before, we approximate a backward half time-step of x by an average. Lines 33, 34, and 35 are as prescribed by the Runge-Kutta routine. Line 36 ends the *for* loop. Note, the λ value at each node is used to find the one before it, so that $\lambda(2)$ is used to find $\lambda(1)$. This is why we only need to count backward to 2, not 1.

code1.m

```
38      u1 = C*lambda/(2*B);
39      u = 0.5*(u1 + oldu);
40
41      temp1 = 0.001*sum(abs(u)) - sum(abs(oldu - u));
42      temp2 = 0.001*sum(abs(x)) - sum(abs(oldx - x));
43      temp3 = 0.001*sum(abs(lambda)) - sum(abs(oldlambda - lambda));
44      test = min(temp1, min(temp2, temp3));
45  end
```

Line 38 is the representation of u^* using the new values for λ. This is not stored as the control u, but as a temporary variable $u1$. The control u is set as the average of the last iteration of u, named *oldu*, and the new representation. This is the convex combination described earlier. Lines 41, 42, and 43 are the convergence test parameters of each variable, where $\delta = 0.001$. Recall, we require these three values to be non-negative. In Line 44, the variable *test* is reassigned as the minimum of these three values. The MATLAB function *min* is a binary operation, so, in order to find the minimum of three values, two applications of *min* are necessary. The *end* in Line 45 marks the end of the while loop. If the minimum is non-negative, i.e. if all three values are non-negative, then convergence has been achieved, and the while loop ends. If it is not, then the program returns to the beginning of the while loop and performs another sweep. Once convergence occurs, the values of the final vectors are stored in the output matrix y.

code1.m

```
47  y(1,:) = t;
48  y(2,:) = x;
49  y(3,:) = lambda;
50  y(4,:) = u;
```

Note, this technique can only be used to solve problems where the state is fixed at the initial time and free at the terminal time. An obvious adjustment to the code allows states fixed at the terminal time and free at the initial time. However, different algorithms, specifically ones based on shooting methods, must be employed if the state has two or no boundary conditions.

Note that there are more complicated methods of updating the controls by using convex combinations of the current and new values of the control.