Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Chapter 3

Optimal Control Preliminaries

3.1 The Transcription Method

The preceding chapters focus on methods for solving NLP problems. In the remainder of the book, we turn our attention to the *optimal control problem*. An NLP problem is characterized by a *finite* set of variables $\mathbf{x}$ and constraints $\mathbf{c}$. In contrast, optimal control problems can involve continuous *functions* such as $\mathbf{y}(t)$ and $\mathbf{u}(t)$. It will be convenient to view the optimal control problem as an infinite-dimensional extension of an NLP problem. However, practical methods for solving optimal control problems require Newton-based iterations with a *finite* set of variables and constraints. This goal can be achieved by *transcribing* or converting the infinite-dimensional problem into a finite-dimensional approximation.

Thus, the *transcription method* has three fundamental steps:

1. convert the dynamic system into a problem with a *finite* set of variables; then

2. solve the finite-dimensional problem using a parameter optimization method (i.e., the NLP subproblem); and then

3. assess the accuracy of the finite-dimensional approximation and if necessary repeat the transcription and optimization steps.

We will begin the discussion by focusing on the first step in the process, namely identifying the NLP variables, constraints, and objective function for common applications. In simple terms, we will focus on how to convert an optimal control problem into an NLP problem.

3.2 Dynamic Systems

A *dynamic system* is usually characterized mathematically by a set of *ordinary differential equations* (ODEs). Specifically, the dynamics are described for $t_I \le t \le t_F$ by a system of n_y ODEs

$$\dot{\mathbf{y}} = \begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \vdots \\ \dot{y}_{n_y} \end{bmatrix} = \begin{bmatrix} f_1[y_1(t),\dots,y_{n_y}(t),t] \\ f_2[y_1(t),\dots,y_{n_y}(t),t] \\ \vdots \\ f_{n_y}[y_1(t),\dots,y_{n_y}(t),t] \end{bmatrix} = \mathbf{f}[\mathbf{y}(t),t]. \tag{3.1}$$

For many applications, the variable t is time and it is common to associate the independent variable with a time scale. Of course, mathematically there is no need to make such an assumption and it is perfectly reasonable to define the independent variable in any meaningful way. The system (3.1) is referred to as an *explicit* first order ODE system.

Example 3.1 ODE EXAMPLE. Consider the following simple system of differential equations:

$$\dot{y}_1 = y_2, \tag{3.2}$$
$$\dot{y}_2 = -y_1 \tag{3.3}$$

defined on the region $0 \le t \le t_F$. It is easy to verify that this system of two first order differential equations is equivalent to the single second order system $\ddot{p}(t) + p(t) = 0$ by identifying $p(t) = y_1(t)$. Now the solution to this system is

$$y_1(t) = \alpha \sin(t+\beta), \tag{3.4}$$
$$y_2(t) = \alpha \cos(t+\beta), \tag{3.5}$$

where α and β are arbitrary constants. Since there are two arbitrary constants one can impose two side conditions, but how we do this determines the nature of the ODE.

First, suppose the side conditions are imposed at the initial time

$$y_1(0) = c_1, \qquad y_2(0) = c_2, \tag{3.6}$$

where c_1 and c_2 are specified constants. Then from (3.4)–(3.5)

$$c_1 = \alpha \sin\beta, \tag{3.7}$$
$$c_2 = \alpha \cos\beta, \tag{3.8}$$

which defines the values for $\beta = \tan^{-1}\frac{c_1}{c_2}$ and $\alpha = c_1/\sin\beta$. The solution is unique for all values of c_1 and c_2, and this is referred to as an *initial value problem* (IVP) because the side conditions are imposed at the initial time. In general for an IVP, one is given a set of initial values for the dependent variables $\mathbf{y}(t_I)$, called the *initial conditions*, and one must determine the values at some other point t_F.

In contrast, for the *boundary value problem*, one must determine the dependent variables such that they have specified values at two or more points, say t_I and t_F. The conditions that define the dependent variables are called *boundary conditions*. So instead of (3.6), suppose the side conditions are imposed at both the initial and final times

$$y_1(0) = c_1, \qquad y_1(t_F) = c_2. \tag{3.9}$$

If we choose $t_F = \pi$ and $c_1 = 0$, then from (3.4) we must have

$$y_1(0) = \alpha \sin(\beta) = 0, \tag{3.10}$$
$$y_1(\pi) = \alpha \sin(\pi+\beta) = c_2. \tag{3.11}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

But $\sin(\beta) = -\sin(\pi + \beta)$, which leads to the conditions

$$\alpha \sin\beta = 0, \tag{3.12}$$

$$-\alpha \sin\beta = c_2. \tag{3.13}$$

Clearly if $c_2 \neq 0$, there is no solution! On the other hand if $c_2 = 0$ and $\beta = 0$, the value of α is arbitrary, and there are an infinite number of solutions! Finally, if we choose $t_F \neq \pi$, then a unique solution can be found. To recapitulate, for the same system of differential equations (3.2)–(3.3), different boundary conditions produce dramatically different results. In contrast to the IVP, for a BVP anything is possible! Although most optimal control problems are BVPs, a number of concepts are usually introduced in the initial value setting.

When the independent variable t does not appear explicitly in the right-hand-side functions, that is, $\mathbf{f} = \mathbf{f}[\mathbf{y}(t)]$, the equations are called an *autonomous system*. Since a nonautonomous system of ODEs can be transformed into an autonomous form by introducing a new variable, say $\mathbf{Y}^\mathsf{T} = (\mathbf{y}, t)^\mathsf{T}$, and a new right-hand side, say $\mathbf{F}^\mathsf{T} = (\mathbf{f}, 1)^\mathsf{T}$, it is sometimes convenient to simply write $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ without loss of generality.

BVPs are often classified by the manner in which the boundary conditions are stated. In general, for a *two-point BVP* the initial and final conditions are given by

$$\boldsymbol{\psi}\left[\mathbf{y}(t_I), \mathbf{y}(t_F)\right] = \mathbf{0}. \tag{3.14}$$

The boundary conditions are said to be *linear* when they can be written as

$$\mathbf{B}_I \mathbf{y}(t_I) + \mathbf{B}_F \mathbf{y}(t_F) = \mathbf{b}, \tag{3.15}$$

where the matrices $\mathbf{B}_I$ and $\mathbf{B}_F$ and vector $\mathbf{b}$ are constant. A boundary condition is said to be *separated* if it involves either $\mathbf{y}(t_I)$ or $\mathbf{y}(t_F)$, but *not* both. Thus for a problem with linear boundary conditions, if row k of $\mathbf{B}_I$ has nonzero values and row k of $\mathbf{B}_F$ is entirely zero, the boundary conditions are both linear and separable. For example, the linear, separable boundary conditions (3.9) can be written using (3.15) as

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_1(0) \\ y_2(0) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_1(t_F) \\ y_2(t_F) \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}. \tag{3.16}$$

For an IVP there is a well-defined sense of direction; that is, it makes perfect sense to propagate *forward* in time. This is not the case for a BVP, and indeed solution methods for the latter are global, rather than local, in nature.

3.3 Shooting Method

Example 3.2 SHOOTING BOUNDARY VALUE PROBLEM. To illustrate the basic concepts, let us consider the simple dynamics:

$$\frac{dy}{dt} = y.$$

Clearly, the analytic solution to the IVP is given by

$$y = y(t_I)e^{t - t_I}.$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

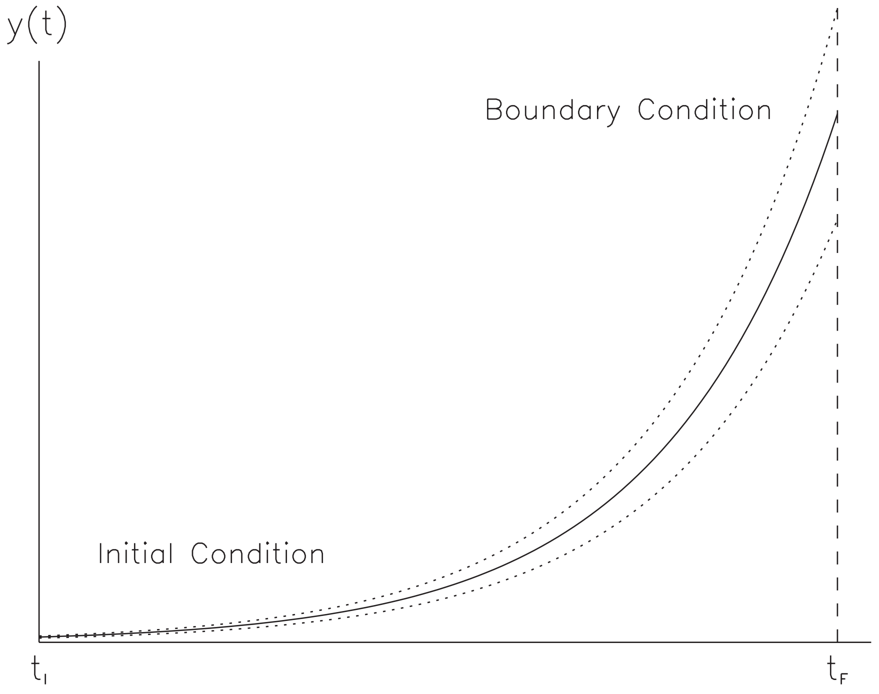

Figure 3.1. *Shooting method.*

However, suppose we want to find $y(t_I) \equiv y_I$ such that $y(t_F) = b$, where b is a specified value. This is called a two-point BVP. In particular, using the transcription formulation, it is clear that we can formulate the problem in terms of the single (NLP) variable $x \equiv y_I$. Then it is necessary to solve the single constraint

$$\begin{aligned} c(x) &= y(t_F) - b \\ &= y_I e^{t_F - t_I} - b \\ &= x e^{t_F - t_I} - b \\ &= 0 \end{aligned}$$

by adjusting the variable x. Figure 3.1 illustrates the problem.

The approach described is referred to as the *shooting method* and is one of the simplest techniques for solving a BVP. An early practical application of the method required that a cannon be aimed such that the cannonball hit its target, hence explaining the colorful name. Of course, the method is not limited to just a single variable and constraint. In general, the shooting method can be summarized as follows:

1. guess initial conditions $\mathbf{x} = \mathbf{y}(t_I)$;
2. propagate the differential equations from t_I to t_F, i.e., “shoot”;
3. evaluate the error in the boundary conditions $\mathbf{c}(\mathbf{x}) = \mathbf{y}(t_F) - \mathbf{b}$;

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

4. use an NLP to adjust the variables $\mathbf{x}$ to satisfy the constraints $\mathbf{c}(\mathbf{x}) = \mathbf{0}$, i.e., repeat steps 1–3.

From a practical standpoint, the shooting method is widely used primarily because the transcribed problem has a small number of variables. Clearly, the number of iteration variables is equal to the number of differential equations. Consequently, any stable implementation of Newton's method is a viable candidate as the iterative technique. Unfortunately, the shooting method also suffers from one major disadvantage. In particular, a small change in the initial condition can produce a very large change in the final conditions. This "tail wagging the dog" effect can result in constraints $\mathbf{c}(\mathbf{x})$ that are *very* nonlinear and, hence, very difficult to solve. The nonlinearity also makes it difficult to construct an accurate estimate of the Jacobian matrix that is needed for a Newton iteration. It should be emphasized that this nonlinear behavior in the boundary conditions can be caused by differential equations that are either nonlinear or *stiff* (or both). Although a discussion of stiffness is deferred temporarily, it should be clear that Example 3.2 would be much harder to solve if we replaced the *linear* differential equation $\dot{y} = y$ by the *linear* differential equation $\dot{y} = 20y$. A more realistic illustration of the shooting method is presented in Example 6.6.

3.4 Multiple Shooting Method

Example 3.3 MULTIPLE SHOOTING EXAMPLE. In order to reduce the sensitivity present in a shooting method, one approach is to simply break the problem into shorter steps, i.e., don't shoot as far. Thus, we might consider a "first step" for $t_I \le t \le t_2$ and a "second step" for $t_2 \le t \le t_F$. For simplicity, let us assume that $t_2 = \frac{1}{2}(t_F + t_I)$. Since the problem has now been broken into two shorter steps, we must guess a value for y at the midpoint in order to start the second step. Furthermore, we must add a constraint to force continuity between the two steps, i.e.,

$$\hat{y}_1 = y_2,$$

where $y(t_2^-) \equiv y_2^- = \hat{y}_1$ denotes the value at the end of the first step and $y_2 \equiv y(t_2^+) = y_2^+$ denotes the value at the beginning of the second step. As a result of this interval splitting, the transcribed problem now has two variables, $\mathbf{x}^\mathsf{T} \equiv [y_I, y_2]$. Furthermore, we must now solve the two constraints

$$\begin{bmatrix} c_1(\mathbf{x}) \\ c_2(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} y_2 - \hat{y}_1 \\ y(t_F) - b \end{bmatrix} = \begin{bmatrix} x_2 - x_1 e^{(t_2 - t_I)} \\ x_2 e^{(t_F - t_2)} - b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Figure 3.2 illustrates the problem.

The approach described is called *multiple shooting* [55, 123] and the constraints enforcing continuity are called *defect* constraints or, simply, defects. Let us generalize the method as follows: compute the unknown initial values $\mathbf{y}(t_I) = \mathbf{y}_I$ such that the boundary condition

$$\mathbf{0} = \boldsymbol{\psi}[\mathbf{y}(t_F), t_F] \tag{3.17}$$

holds for some value of $t_F > t_I$ that satisfies the ODE system (3.1). The fundamental idea of multiple shooting is to break the trajectory into shorter pieces or segments. Thus, we

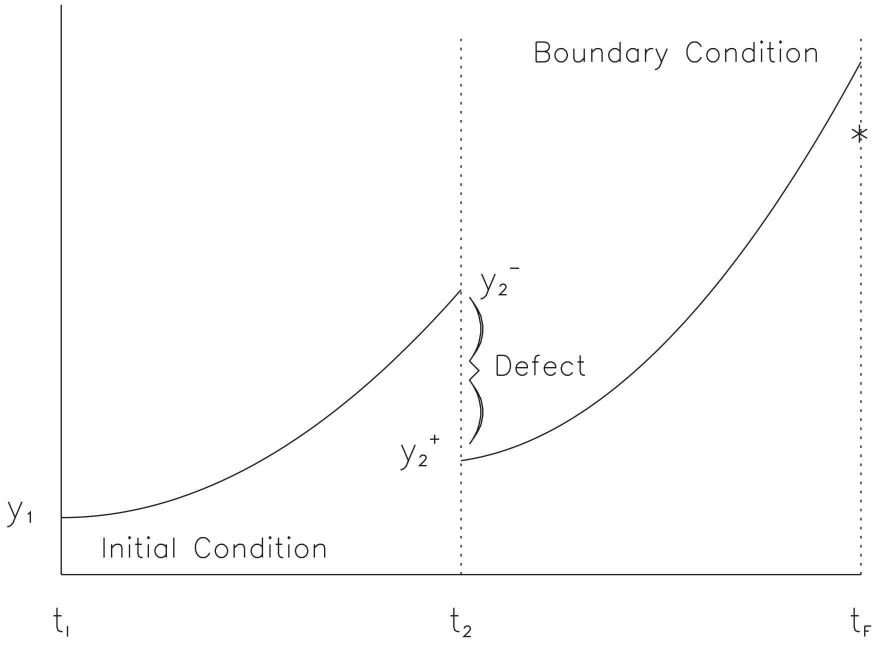

Figure 3.2. *Multiple shooting method.*

break the time domain into smaller intervals of the form

$$t_I = t_1 < t_2 < \cdots < t_M = t_F. \tag{3.18}$$

Let us denote $\mathbf{y}_k$ for $k = 1,\ldots,(M-1)$ as the initial value for the dynamic variable at the beginning of segment k. For segment k we can propagate (integrate) the differential equations (3.1) from t_k to the end of the segment at t_{k+1}. Denote the result of this integration by $\widehat{\mathbf{y}}_k$. Collecting all segments, let us define a set of NLP variables

$$\mathbf{x}^{\mathsf{T}} = \left(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{M-1}\right). \tag{3.19}$$

Now we also must ensure that the segments join at the boundaries; consequently, we impose the constraints

$$\mathbf{c}(\mathbf{x}) = \begin{bmatrix} \mathbf{y}_2 - \widehat{\mathbf{y}}_1 \\ \mathbf{y}_3 - \widehat{\mathbf{y}}_2 \\ \vdots \\ \boldsymbol{\psi}[\widehat{\mathbf{y}}_M, t_F] \end{bmatrix} = \mathbf{0}. \tag{3.20}$$

One obvious result of the multiple shooting approach is an increase in the size of the problem that the Newton iteration must solve. Additional variables and constraints are introduced for each shooting segment. In particular, the number of NLP variables and constraints for a multiple shooting application is $n = n_y(M-1)$, where n_y is the number of dynamic variables $\mathbf{y}$ and $(M-1)$ is the number of segments. Fortunately, the Jacobian matrix, which is needed to compute the Newton search direction, is *sparse*. In particular, only

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

$(M-1)n_y^2$ elements in $\mathbf{G}$ are nonzero out of a possible $[(M-1)n_y]^2$. Thus, the percentage of nonzeros is proportional to $1/(M-1)$, indicating that the matrix gets sparser as the number of intervals grows. This sparsity is a direct consequence of the multiple shooting formulation, since variables early in the trajectory do not change constraints later in the trajectory. In fact, Jacobian sparsity is the mathematical consequence of *un*coupling between the multiple shooting segments. For the simple case described, the Jacobian matrix is banded with $n_y \times n_y$ blocks along the diagonal, and the very efficient methods described in Section 2.3 can be used. It is important to note that the multiple shooting segments are introduced strictly for numerical reasons. A more complete discussion of sparsity is presented in Section 4.6. Example 6.7 describes a practical application of multiple shooting to an orbit transfer problem.

An interesting benefit of the multiple shooting algorithm is the ability to exploit a parallel processor. The method is sometimes called *parallel shooting* because the simulation of each segment can be implemented on an individual processor. This technique was explored for a trajectory optimization application [34] and remains an intriguing prospect for multiple shooting methods in general.

3.5 Initial Value Problems

In the preceding sections, both the shooting and multiple shooting methods require "propagation" of a set of differential equations. For the simple motivational examples, it was possible to analytically propagate the solution from t_I to t_F. However, in general, analytic propagation is not feasible and numerical methods must be employed. The numerical solution of the IVP for ODEs is fundamental to most optimal control methods. The problem can be stated as follows: compute the value of $\mathbf{y}(t_F)$ for some value of $t_I < t_F$ that satisfies (3.1) with the known initial value $\mathbf{y}(t_I) = \mathbf{y}_I$. Numerical methods for solving the ODE IVP are relatively mature in comparison to the other fields in optimal control.

Most schemes can be classified as *one-step methods* or *multistep methods*. Let us begin the discussion with one-step methods. Our goal is to construct an expression over a single step from t_i to t_{i+1}, where h_i is referred to as the *integration stepsize* for step i. We denote the value of the vector $\mathbf{y}$ at t_i by $\mathbf{y}(t_i) \equiv \mathbf{y}_i$. Proceeding formally to integrate (3.1) yields

$$\begin{aligned}\mathbf{y}_{i+1} &= \mathbf{y}_i + \int_{t_i}^{t_{i+1}} \dot{\mathbf{y}}\,dt \\ &= \mathbf{y}_i + \int_{t_i}^{t_{i+1}} \mathbf{f}(\mathbf{y},t)\,dt. \end{aligned} \tag{3.21}$$

To evaluate the integral, first let us subdivide the integration step into K subintervals

$$\tau_j = t_i + h_i \rho_j \tag{3.22}$$

with

$$0 \le \rho_1 \le \rho_2 \le \cdots \le \rho_K \le 1$$

for $1 \le j \le K$. With this subdivided interval, we now apply a quadrature formula within a quadrature formula. Specifically, we have

$$\int_{t_i}^{t_{i+1}} \mathbf{f}(\mathbf{y},t)dt \approx h_i \sum_{j=1}^{K} \beta_j \widehat{\mathbf{f}}_j, \tag{3.23}$$

where $\widehat{\mathbf{f}}_j \equiv \mathbf{f}(\tau_j, \widehat{\mathbf{y}}_j)$. Notice that this approximation requires values for the variables $\mathbf{y}$ at the intermediate points τ_j, i.e., $\widehat{\mathbf{y}}(\tau_j) \equiv \widehat{\mathbf{y}}_j$. Consequently, we construct these intermediate values using the second expression

$$\int_{t_i}^{\tau_j} \mathbf{f}(\mathbf{y},t)dt \approx h_i \sum_{\ell=1}^{K} \alpha_{j\ell} \mathbf{f}_\ell \tag{3.24}$$

for $1 \le j \le K$.

Collecting results, we obtain a popular family of one-step methods called the *K-stage Runge–Kutta* scheme:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \sum_{j=1}^{K} \beta_j \mathbf{f}_{ij}, \tag{3.25}$$

where

$$\mathbf{f}_{ij} = \mathbf{f}\left[\left(\mathbf{y}_i + h_i \sum_{\ell=1}^{K} \alpha_{j\ell} \mathbf{f}_{i\ell}\right), \left(t_i + h_i \rho_j\right)\right] \tag{3.26}$$

for $1 \le j \le K$. K is referred to as the "stage." In these expressions, $\{\rho_j, \beta_j, \alpha_{j\ell}\}$ are known constants with $0 \le \rho_1 \le \rho_2 \le \cdots \le \rho_K \le 1$. A convenient way to define the coefficients is to use the so-called Butcher array

$$\begin{array}{c|ccc} \rho_1 & \alpha_{11} & \dots & \alpha_{1K} \\ \vdots & \vdots & & \vdots \\ \rho_K & \alpha_{K1} & \dots & \alpha_{KK} \\ \hline & \beta_1 & \dots & \beta_K \end{array}.$$

The schemes are called *explicit* if $\alpha_{j\ell} = 0$ for $l \ge j$ and *implicit* otherwise. Four common examples of K-stage Runge–Kutta schemes are summarized below:

Euler Method (explicit, $K = 1$)

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}.$$

Common Representation:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \mathbf{f}_i. \tag{3.27}$$

Classical Runge–Kutta Method (explicit, $K = 4$)

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}.$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Common Representation:

$$\mathbf{k}_1 = h_i \mathbf{f}_i, \tag{3.28}$$

$$\mathbf{k}_2 = h_i \mathbf{f}\left(\mathbf{y}_i + \frac{1}{2}\mathbf{k}_1, t_i + \frac{h_i}{2}\right), \tag{3.29}$$

$$\mathbf{k}_3 = h_i \mathbf{f}\left(\mathbf{y}_i + \frac{1}{2}\mathbf{k}_2, t_i + \frac{h_i}{2}\right), \tag{3.30}$$

$$\mathbf{k}_4 = h_i \mathbf{f}(\mathbf{y}_i + \mathbf{k}_3, t_{i+1}), \tag{3.31}$$

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4). \tag{3.32}$$

Trapezoidal Method (implicit, $K = 2$)

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array}.$$

Common Representation:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h_i}{2}(\mathbf{f}_i + \mathbf{f}_{i+1}). \tag{3.33}$$

Hermite–Simpson Method (implicit, $K = 3$)

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 5/24 & 1/3 & -1/24 \\ 1 & 1/6 & 2/3 & 1/6 \\ \hline & 1/6 & 2/3 & 1/6 \end{array}.$$

Common Representation:

$$\overline{\mathbf{y}} = \frac{1}{2}(\mathbf{y}_i + \mathbf{y}_{i+1}) + \frac{h_i}{8}(\mathbf{f}_i - \mathbf{f}_{i+1}), \tag{3.34}$$

$$\bar{\mathbf{f}} = \mathbf{f}_i\left(\overline{\mathbf{y}}, t_i + \frac{h_i}{2}\right), \tag{3.35}$$

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h_i}{6}\left(\mathbf{f}_i + 4\bar{\mathbf{f}} + \mathbf{f}_{i+1}\right). \tag{3.36}$$

The Runge–Kutta scheme (3.25)–(3.26) is often motivated in another way. Suppose we consider approximating the solution of the ODE (3.1) by a function $\widetilde{\mathbf{y}}(t)$. As an approximation, let us use a polynomial of degree K (order $K+1$) over each step $t_i \le t \le t_{i+1}$:

$$\widetilde{\mathbf{y}}(t) = a_0 + a_1(t - t_i) + \cdots + a_K(t - t_i)^K \tag{3.37}$$

with the coefficients $(a_0, a_1, \ldots, a_K)$ chosen such that the approximation matches at the beginning of the step t_i, that is,

$$\widetilde{\mathbf{y}}(t_i) = \mathbf{y}_i, \tag{3.38}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

and has derivatives that match at the points (3.22):

$$\frac{d\widetilde{\mathbf{y}}(\tau_j)}{dt} = \mathbf{f}[\mathbf{y}(\tau_j), \tau_j]. \tag{3.39}$$

The conditions (3.39) are called *collocation* conditions and the resulting method is referred to as a *collocation method*. Thus, the Runge–Kutta scheme (3.25)–(3.26) is a collocation method [2], and the solution produced by the method is a piecewise polynomial. While the polynomial representation (3.37) (called a monomial representation) has been introduced for simplicity in this discussion, in practice we will use an equivalent but computationally preferable form called a B-spline representation.

The collocation schemes of particular interest for the remainder of the book, namely (3.33) and (3.36), are both *Lobatto methods*. More precisely, the trapezoidal method is a Lobatto IIIA method of order 2, and the Hermite–Simpson method is a Lobatto IIIA method of order 4 [107, p. 75]. For a Lobatto method, the endpoints of the interval are also collocation points, and consequently $\rho_1 = 0$ and $\rho_K = 1$. The trapezoidal method is based on a quadratic interpolation polynomial. The three coefficients (a_0, a_1, a_2) are constructed such that the function matches at the beginning of the interval (3.38) and the slope matches at the beginning and end of the interval (3.39). For the Hermite–Simpson scheme, the four coefficients (a_0, a_1, a_2, a_3) defining a cubic interpolant are defined by matching the function at the beginning of the interval and the slope at the beginning, midpoint, and end of the interval. A unique property of Lobatto methods is that mesh points are also collocation points. In contrast, *Gauss* schemes impose the collocation conditions strictly interior to the interval, that is, $\rho_1 > 0$ and $\rho_K < 1$. The simplest Gauss collocation scheme is the *midpoint rule*:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \mathbf{f}\left[\frac{1}{2}(\mathbf{y}_{i+1} + \mathbf{y}_i), t_i + \frac{h_i}{2}\right]. \tag{3.40}$$

A *Radau* method imposes the collocation condition at only one end of the interval, specifically $\rho_1 > 0$ and $\rho_K = 1$. The simplest Radau collocation scheme is the *backward Euler method*:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \mathbf{f}_{i+1}. \tag{3.41}$$

An obvious appeal of an explicit scheme is that the computation of each integration step can be performed without iteration; that is, given the value $\mathbf{y}_i$ at the time t_i, the value $\mathbf{y}_{i+1}$ at the new time t_{i+1} follows directly from available values of the right-hand-side functions $\mathbf{f}$. In contrast, for an implicit method, the unknown value $\mathbf{y}_{i+1}$ appears nonlinearly; e.g., the trapezoidal method requires

$$0 = \mathbf{y}_{i+1} - \mathbf{y}_i - \frac{h_i}{2}\left[\mathbf{f}(\mathbf{y}_{i+1}, t_{i+1}) + \mathbf{f}(\mathbf{y}_i, t_i)\right] \equiv \boldsymbol{\zeta}_i. \tag{3.42}$$

Consequently, to compute $\mathbf{y}_{i+1}$, given the values t_{i+1}, $\mathbf{y}_i$, t_i, and $\mathbf{f}[\mathbf{y}_i, t_i]$, requires solving the nonlinear expression (3.42) to drive the *defect* $\boldsymbol{\zeta}_i$ to zero. The iterations required to solve this equation are called *corrector* iterations. An initial guess to begin the iteration is usually provided by the so-called predictor step. There is considerable latitude in the choice of predictor and corrector schemes. For some well-behaved differential equations, a single predictor and corrector step is adequate. In contrast, it may be necessary to perform multiple corrector iterations, e.g., using Newton's method, especially when the differential equations are *stiff*. To illustrate this, suppose that the dynamic behavior is described by

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

two types of variables, namely $\mathbf{y}(t)$ and $\mathbf{u}(t)$. Instead of (3.1), the system dynamics are described by

$$\begin{aligned} \dot{\mathbf{y}} &= \mathbf{f}[\mathbf{y}(t), \mathbf{u}(t), t], \\ \epsilon \dot{\mathbf{u}} &= \mathbf{g}[\mathbf{y}(t), \mathbf{u}(t), t], \end{aligned} \tag{3.43}$$

where ϵ is a "small" parameter. Within a very small region $0 \le t \le t_\epsilon$, the solution displays a rapidly changing behavior and, thereafter, the second equation can effectively be replaced by its limiting form[3]

$$\mathbf{0} = \mathbf{g}[\mathbf{y}(t), \mathbf{u}(t), t]. \tag{3.44}$$

The resulting system given by

$$\dot{\mathbf{y}} = \mathbf{f}[\mathbf{y}(t), \mathbf{u}(t), t], \tag{3.45}$$
$$\mathbf{0} = \mathbf{g}[\mathbf{y}(t), \mathbf{u}(t), t] \tag{3.46}$$

is called a *semiexplicit differential-algebraic equation* (DAE). In modern control theory, the differential variables $\mathbf{y}(t)$ are called *state variables* and the algebraic variables $\mathbf{u}(t)$ are called *control variables*. The system is semiexplicit because the differential variables appear explicitly (on the left-hand side), whereas the algebraic variables appear implicitly in $\mathbf{f}$ and $\mathbf{g}$. The original stiff system of ODEs (3.43) is referred to as a *singular perturbation* problem and in the limit approaches a DAE system.

Example 3.4 VAN DER POL'S EQUATION. An example of a singular perturbation problem that arises in electrical circuits is given in second order form by

$$\epsilon \ddot{z} + (z^2 - 1)\dot{z} + z = 0. \tag{3.47}$$

For a modest value of the parameter $\epsilon = 1$, the second order system can be rewritten as

$$\dot{y}_1 = y_2, \tag{3.48}$$
$$\dot{y}_2 = (1 - y_1^2)y_2 - y_1 \tag{3.49}$$

by identifying $y_1 = z$ and $y_2 = \dot{z}$. Now observe that the first two terms in (3.47)

$$\epsilon \ddot{z} + (z^2 - 1)\dot{z} = \frac{d}{dt}\left[\epsilon \dot{z} + \left(\frac{z^3}{3} - z\right)\right],$$

so if we identify the quantity in brackets as

$$y = \epsilon \dot{z} + \left(\frac{z^3}{3} - z\right)$$

and set $u = z$, we can also write

$$\dot{y} = -u, \tag{3.50}$$
$$\epsilon \dot{u} = y - \left(\frac{u^3}{3} - u\right), \tag{3.51}$$

[3]Strictly speaking $\mathbf{g}_u$ (3.59) must be uniformly negative definite.

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

which is the singular perturbation format (3.43). Clearly, when the parameter $\epsilon = 0$ one obtains the DAE system

$$\dot{y} = -u, \tag{3.52}$$

$$0 = y - \left(\frac{u^3}{3} - u\right). \tag{3.53}$$

The second class of integration schemes are termed *multistep methods* and have the general form

$$\mathbf{y}_{i+k} = \sum_{j=0}^{k-1} \alpha_j \mathbf{y}_{i+j} + h \sum_{j=0}^{k} \beta_j \mathbf{f}_{i+j}, \tag{3.54}$$

where α_j and β_j are known constants. If $\beta_k = 0$, then the method is explicit; otherwise, it is implicit. The *Adams schemes* are members of the multistep class that are based on approximating the functions $\mathbf{f}(t)$ by interpolating polynomials. The Adams–Bashforth method is an explicit multistep method [5], whereas the Adams–Moulton method is implicit [135]. Multistep methods must address three issues that we have not discussed for single-step methods. First, as written, the method requires information at $(k-1)$ previous points. Clearly, this implies some method must be used to start the process, and one common technique is to take one or more steps with a one-step method (e.g., Euler). Second, as written, the multistep formula assumes the stepsize h is a fixed value. When the stepsize is allowed to vary, careful implementation is necessary to ensure that the calculation of the coefficients is both efficient and well-conditioned. Finally, similar remarks apply when the number of steps k (i.e., the order) of the method is changed.

Regardless of whether a one-step or multistep method is used, a successful implementation must address the accuracy of the solution. How well does the discrete solution $\mathbf{y}_i$ for $i = 1, 2, \ldots, M$, produced by the integration scheme, agree with the "real" answer $\mathbf{y}(t)$? All well-implemented schemes have some mechanism for adjusting the integration stepsize and/or order to control the integration error. The reader is urged to consult the works of Dahlquist and Björk [66], Stoer and Bulirsch [163], Hindmarsh [113], Shampine and Gordon [158], Hairer, Norsett, and Wanner [106], and Gear [89] for additional information. It is also worth noting that a great deal of discussion has been given to the distinction between explicit and implicit methods. Indeed, it is often tempting to use an explicit method simply because it is more easily implemented (and understood?). However, the optimal control problem is a BVP, *not* an IVP, and, to quote Ascher, Mattheij, and Russell [2, p. 69],

> … for a boundary value problem … any scheme becomes effectively, implicit. Thus, the distinction between explicit and implicit initial value schemes becomes less important in the BVP context.

Methods for solving IVPs when dealing with a system of DAEs have appeared more recently. For a semiexplicit DAE system such as (3.45)–(3.46), it is tempting to try to "eliminate" the algebraic (control) variables in order to use a more standard method for solving ODEs. Proceeding formally to solve (3.46) one can write

$$\mathbf{u}(t) = \mathcal{G}^{-1}[\mathbf{y}, t], \tag{3.55}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

where $\mathcal{G}^{-1}$ is used to denote the inverse of the function $\mathbf{g}$. When this value is substituted into (3.45), one obtains the nonlinear differential equation

$$\dot{\mathbf{y}} = \mathbf{f}[\mathbf{y}, \mathcal{G}^{-1}[\mathbf{y}, t], t], \tag{3.56}$$

which is amenable to solution using any of the ODE techniques described above.

When analytic elimination is impossible, one alternative is to introduce a nonlinear iterative technique (e.g., Newton's method) that must be executed at every integration step. Consider solving the equations

$$\mathbf{0} = \mathbf{g}[\mathbf{y}_k, \mathbf{u}_k, t_k] \tag{3.57}$$

for the variables $\mathbf{u}_k$ given the variables $\mathbf{y}_k, t_k$. Applying (1.28) and (1.29), we obtain the iteration

$$\overline{\mathbf{u}}_k = \mathbf{u}_k - \mathbf{g}_u^{-1}\mathbf{g}, \tag{3.58}$$

where

$$\mathbf{g}_u = \begin{bmatrix} \frac{\partial g_1}{\partial u_1} & \frac{\partial g_1}{\partial u_2} & \cdots & \frac{\partial g_1}{\partial u_m} \\ \frac{\partial g_2}{\partial u_1} & \frac{\partial g_2}{\partial u_2} & \cdots & \frac{\partial g_2}{\partial u_m} \\ \vdots & & \ddots & \\ \frac{\partial g_m}{\partial u_1} & \frac{\partial g_m}{\partial u_2} & \cdots & \frac{\partial g_m}{\partial u_m} \end{bmatrix}. \tag{3.59}$$

At least in principle, we could repeat the iteration (3.58) until the equations (3.57) are satisfied. This approach is not only very time-consuming, but it can conflict with logic used to control integration error in the dynamic variables $\mathbf{y}$. If an implicit method is used for solving the ODEs, this "elimination" iteration must be performed within each corrector iteration; in other words, it becomes an iteration within an iteration. In simple terms, this is usually *not* the way to solve the problem! However, this discussion does provide one useful piece of information, namely, that successful application of Newton's method requires that $\mathbf{g}_u^{-1}$ can be computed, i.e., that $\mathbf{g}_u$ has full rank.

A word of caution must be interjected at this point. In order to eliminate the variables in the manner presented here, the inverse operation must exist and uniquely define the algebraic variables. However, it is not hard to construct a setting that precludes this operation. For example, suppose the matrix $\mathbf{g}_u$ is rank deficient. In this case, the algebraic constraint does not uniquely specify all of the degrees of freedom, and we could expect some subset (or subspace) of the variables to be undefined. Furthermore, it is not hard to imagine a situation where the rank deficiency in this inverse operation changes with time—full rank in some regions and rank deficient in others.

Instead of using (3.46) to eliminate the control, let us take a slightly different approach. Now if $\mathbf{g}(t) = \mathbf{0}$ must be satisfied over some range of time, then we also expect that the first derivative $\dot{\mathbf{g}} = \mathbf{0}$. Therefore, let us differentiate (3.46) with respect to t yielding

$$0 = \mathbf{g}_y\dot{\mathbf{y}} + \mathbf{g}_u\dot{\mathbf{u}} + \mathbf{g}_t \tag{3.60}$$

$$= \mathbf{g}_y\mathbf{f}[\mathbf{y}, \mathbf{u}] + \mathbf{g}_u\dot{\mathbf{u}} + \mathbf{g}_t, \tag{3.61}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

where (3.61) follows by substituting the expression for $\dot{\mathbf{y}}$ from (3.45). Now if $\mathbf{g}_u$ is nonsingular, we can solve (3.61) for $\dot{\mathbf{u}}$ and replace the original DAE system (3.45)–(3.46) with

$$\dot{\mathbf{y}} = \mathbf{f}[\mathbf{y}(t), \mathbf{u}(t), t], \tag{3.62}$$

$$\dot{\mathbf{u}} = -\mathbf{g}_u^{-1}\left[\mathbf{g}_y \mathbf{f}[\mathbf{y}, \mathbf{u}] + \mathbf{g}_t\right]. \tag{3.63}$$

This is now just a system of differential equations in the new dynamic variables $\mathbf{z}^\mathsf{T} = (\mathbf{y}, \mathbf{u})$. If $\mathbf{g}_u$ is nonsingular, we say the system (3.45)–(3.46) is an *index-one DAE*. On the other hand, if $\mathbf{g}_u$ is rank deficient, we can differentiate (3.60) a second time and repeat the process. If an ODE results, we say the DAE has *index two*. It should also be clear that each differentiation may determine some (but not all) of the algebraic variables. In general, the *DAE index* is the minimum number of times that all or part of the original system must be differentiated with respect to t in order to explicitly determine the algebraic variable. A more complete definition of the DAE index can be found in Brenan, Campbell, and Petzold [48]. For obvious reasons, the process just described is called *index reduction*. It does provide one (not necessarily good) technique for solving DAEs. The example described in Section 4.12 illustrates the use of index reduction.

The first general technique for solving DAEs was proposed by Gear [90] and uses a backward differentiation formula (BDF) in a linear multistep method. The algebraic variables $\mathbf{u}(t)$ are treated the same as the differential variables $\mathbf{y}(t)$. The method was originally proposed for the semiexplicit index-one system described by (3.45)–(3.46) and soon extended to the fully implicit form

$$\mathbf{F}[\dot{\mathbf{z}}, \mathbf{z}, t] = \mathbf{0}, \tag{3.64}$$

where $\mathbf{z} = (\mathbf{y}, \mathbf{u})$. The basic idea of the BDF approach is to replace the derivative $\dot{\mathbf{z}}$ by the derivative of the polynomial that interpolates the solution computed over the preceding k steps. The simplest example is the implicit Euler method, which replaces (3.64) with

$$\mathbf{F}\left[\frac{\mathbf{z}_i - \mathbf{z}_{i-1}}{h_i}, \mathbf{z}_i, t_i\right] = \mathbf{0}. \tag{3.65}$$

The resulting nonlinear system in the unknowns $\mathbf{z}_i$ is usually solved by some form of Newton's method at each time step t_i. The widely used production code DASSL, developed by Petzold [140, 141], essentially uses a variable-stepsize, variable-order implementation of the BDF formulas. The method is appropriate for index-one DAEs with consistent initial conditions. Current research into the solution of DAEs with higher index (≥ 2) has renewed interest in one-step methods, specifically, the implicit Runge–Kutta (IRK) schemes described for ODEs. RADAU5 [107] implements an IRK method for problems of this type. A discussion of methods for solving DAEs can be found in the books by Brenan, Campbell, and Petzold [48] and Hairer and Wanner [107].

Throughout the book, we have adopted the semiexplicit definition of a DAE as given by (3.45)–(3.46). It is worth noting that a fully implicit form such as (3.64) can be converted to the semiexplicit form by simply writing

$$\dot{\mathbf{y}} = \mathbf{u}(t), \tag{3.66}$$

$$\mathbf{0} = \mathbf{F}\left[\mathbf{y}, \mathbf{u}, t\right]. \tag{3.67}$$

However, as a rule of thumb [48], if the original implicit system (3.64) has index ν, then the semiexplicit form has index $\nu + 1$.

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Example 3.5 DAEs are not ODEs. As a final note consider the very simple DAE

$$\dot{y} = u, \tag{3.68}$$

$$0 = y - t. \tag{3.69}$$

Clearly the solution is $y(t) = t$ and $u(t) = 1$. Furthermore it is not necessary to specify any initial or final conditions! The problem is uniquely specified as it stands. In fact if one tries to specify $y(0) = y_0$, there is no solution if $y_0 \neq 0$.

3.6 Boundary Value Example

Example 3.6 Putting Example. To motivate the BVP, Alessandrini [1] describes a problem as follows:

> Suppose that Arnold Palmer is on the 18th green at Pebble Beach. He needs to sink this putt to beat Jack Nicklaus and walk away with the $1,000,000 grand prize. What should he do? Solve a BVP! By modeling the surface of the green, Arnie sets up the equations of motion of his golf ball.

In [1] the acceleration acting on the golf ball is given by

$$\ddot{\mathbf{z}} = -g\mathbf{k} + gn_3\mathbf{n} - \mu_k g n_3 \frac{\dot{\mathbf{z}}}{\|\dot{\mathbf{z}}\|},$$

where the geometry of the green is modeled using a paraboloid with a minimum at $(10, 5)$ given by

$$z_3 = S(z_1, z_2) = \frac{(z_1 - 10)^2}{125} + \frac{(z_2 - 5)^2}{125} - 1, \tag{3.70}$$

where the normal force is in the direction defined by

$$\mathbf{n} = \frac{\mathbf{N}}{\|\mathbf{N}\|}, \qquad \mathbf{N} = \left(-\tfrac{\partial S}{\partial z_1}, -\tfrac{\partial S}{\partial z_2}, 1\right).$$

For this example, distance is given in feet, the constant $g = 32.174$ ft/sec^2 is the acceleration of gravity, and the constant $\mu_k = 0.2$ is called the kinetic coefficient of friction.

Now since the dynamics are described by a second order differential equation, we can construct an equivalent first order system by introducing new state variables $\mathbf{p} = (\mathbf{z}, \dot{\mathbf{z}})$. The dynamics are then described by the first order system

$$\dot{p}_1 = p_4, \tag{3.71}$$

$$\dot{p}_2 = p_5, \tag{3.72}$$

$$\dot{p}_3 = p_6, \tag{3.73}$$

$$\dot{p}_4 = gn_1n_3 - \mu_k g n_3 \frac{p_4}{s}, \tag{3.74}$$

$$\dot{p}_5 = gn_2n_3 - \mu_k g n_3 \frac{p_5}{s}, \tag{3.75}$$

$$\dot{p}_6 = gn_3n_3 - \mu_k g n_3 \frac{p_6}{s} - g, \tag{3.76}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

where $s = \sqrt{p_4^2 + p_5^2 + p_6^2}$ is the speed of the ball. The geometry has been defined such that the ball is located at $\mathbf{z} = (p_1, p_2, p_3) = (0,0,0)$ and the hole is located at $\mathbf{z}_H = (20,0,0)$. In [1], the problem is formulated as a two-point BVP with four variables $\mathbf{x} = (\dot{\mathbf{z}}(0), t_F)$ representing the initial velocity imparted when the golf club strikes the ball and the duration of the putt. Obviously, to win \$1,000,000, Arnold Palmer hopes the final position of the ball is in the hole, thus producing three boundary conditions $\mathbf{z}(t_F) = \mathbf{z}_H$. Since there are four variables and only three conditions, Alessandrini suggests that the final boundary condition should be either $\|\dot{\mathbf{z}}(t_F)\| = 0$ or $\|\dot{\mathbf{z}}(t_F)\| \le s_F$, where s_F is the maximum final speed.

Unfortunately, neither of the proposed boundary conditions is entirely satisfactory. If we use the first suggestion $\|\dot{\mathbf{z}}(t_F)\| = s = 0$, then, clearly, (3.74)–(3.76) have a singularity at the solution! Obviously, this will cause difficulties when the numerical integration procedure attempts to evaluate the ODE at the final time t_F. On the other hand, suppose the second alternative is used and the integration is terminated when $0 < s \le s_F$. This will avoid the singularity at the solution. However, the solution is not unique since there are many possible putts that will yield $s \le s_F$ and it is not clear how to choose the value s_F.

There is a second, less obvious, difficulty with the original formulation that was pointed out by R. Vanderbei. Since the surface of the green is given by (3.70), it follows that the vertical position of the center of mass for the golf ball while on the green is just

$$p_3 = \frac{(p_1 - 10)^2}{125} + \frac{(p_2 - 5)^2}{125} - 1 + r_b, \tag{3.77}$$

where r_b is the radius of the ball. Differentiating, we find that

$$\begin{aligned}\dot{p}_3 &= \frac{2}{125}(p_1 - 10)\dot{p}_1 + \frac{2}{125}(p_2 - 5)\dot{p}_2 \\ &= \frac{2}{125}(p_1 - 10)p_4 + \frac{2}{125}(p_2 - 5)p_5 = p_6. \end{aligned}\tag{3.78}$$

When this equation is differentiated a second time, the resulting expression for $\dot{p}_6$ is not consistent with (3.76) unless the surface of the green given by (3.70) is a plane. Essentially, the formulation in [1] is "too simple" to produce a well-posed numerical problem!

A better approach is to model the motion of the golf ball in two different regions, namely on the green and in the hole. While the ball is on the green, the motion can be defined by four state variables $\mathbf{y} = (p_1, p_2, p_4, p_5)$ and the differential equations

$$\dot{y}_1 = y_3, \tag{3.79}$$

$$\dot{y}_2 = y_4, \tag{3.80}$$

$$\dot{y}_3 = g n_1 n_3 - \mu_k g n_3 \frac{y_3}{s}, \tag{3.81}$$

$$\dot{y}_4 = g n_2 n_3 - \mu_k g n_3 \frac{y_4}{s}. \tag{3.82}$$

By using (3.78) to compute p_6, the speed of the ball $s = \sqrt{y_1^2 + y_2^2 + p_6^2(\mathbf{y})}$. One also finds that

$$\mathbf{N} = \left[-\frac{2}{125}(y_1 - 10), -\frac{2}{125}(y_2 - 5), 1 \right].$$

Thus, all of the quantities in the system (3.79)–(3.82) are completely specified by the four elements in $\mathbf{y}$. If we assume that the hole has a diameter of 4.25 in and the ball has a

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

diameter of 1.68 in, then, when the distance from the center of the ball to the center of the hole is greater than 1.29 in, the ball is on the green. The location of the hole is just $\mathbf{y}_H = (20,0)$. Thus, mathematically, when $\|\mathbf{y}(t)-\mathbf{y}_H\| \geq R_H$, where $R_H = (4.25/2-1.68/2)/12$, the ball is rolling on the green and the dynamics are given by (3.79)–(3.82).

On the other hand, when the golf ball is inside the hole, there is no frictional force term, and the surface geometry constraint is not needed. Consequently, inside the hole, the complete dynamics are described by six (not four) state variables and

$$\dot{y}_1 = y_4, \tag{3.83}$$

$$\dot{y}_2 = y_5, \tag{3.84}$$

$$\dot{y}_3 = y_6, \tag{3.85}$$

$$\dot{y}_4 = 0, \tag{3.86}$$

$$\dot{y}_5 = 0, \tag{3.87}$$

$$\dot{y}_6 = -g. \tag{3.88}$$

Since the time when the ball leaves the green is unknown, we must introduce an additional variable t_2, where $t_2 < t_F$, and an additional constraint $\|\mathbf{y}(t_2)-\mathbf{y}_H\| = R_H$. Furthermore, as long as $\sqrt{(y_1(t_F)-20)^2+y_2^2(t_F)} \leq R_H$, the ball is in the hole at the end of the trajectory. However, the final state is still not uniquely defined. Although there are other possibilities, it seems reasonable to minimize the horizontal velocity at the final time, i.e., minimize $\dot{y}_1^2+\dot{y}_2^2$. To summarize, the four NLP variables $\mathbf{x}^\mathsf{T} = (\dot{\mathbf{y}}(0), t_2, t_F)$ must be chosen to satisfy the two NLP constraints

$$\|\mathbf{y}(t_2)-\mathbf{y}_H\| = R_H, \tag{3.89}$$

$$\sqrt{(y_1(t_F)-20)^2+y_2^2(t_F)} \leq R_H \tag{3.90}$$

and minimize the objective

$$F(\mathbf{x}) = \dot{y}_1^2+\dot{y}_2^2. \tag{3.91}$$

Figure 3.3 illustrates two (local) solutions to the problem. It is interesting that the long-time solution takes $t_F = 4.46564$ sec with an optimal objective value $F^* = 0.1865527926$, whereas the short-time solution, which takes $t_F = 2.93612$ sec, yields an almost identical objective value of $F^* = 0.1865528358$.

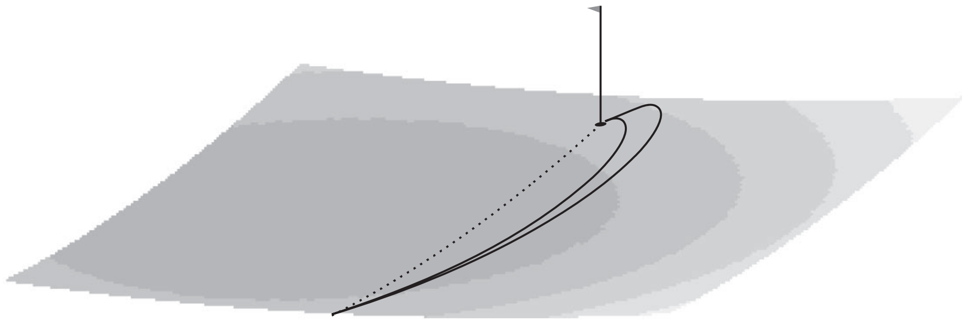

Figure 3.3. *Putting example.*

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

3.7 Dynamic Modeling Hierarchy

The golf-putting example described in the previous section suggests that it may be necessary to break a problem into one or more pieces in order to correctly model the physical situation. In the putting example, it made sense to do this because the differential equations were different on and off the green. In fact, in general, it is convenient to view a dynamic trajectory as made up of a collection of *phases*. Specifically, let us define a *phase* as a portion of a trajectory in which the system of DAEs remains *unchanged*. Conversely, different sets of DAEs *must* be in different phases.

The complete description of a problem may require one or more phases. When necessary, phases may be *linked* together by linkage conditions. For the golf-putting example, the first phase (on the green) was linked to the second phase (in the hole) by forcing continuity in the position and velocity across the phase boundary. A phase terminates at an *event*. Thus, for the golf-putting example, phase 1 was terminated at the boundary time t_2 by the event *condition* or criterion $\|\mathbf{y}(t_2)-\mathbf{y}_H\| = R_H$. Although it is common for phases to occur sequentially in time, this is not always the case. Rather, a phase should be viewed as a fundamental building block that defines a "chunk" of the dynamics that is needed to construct a complete problem description. In fact, multiphase trajectories with nontrivial linkage conditions permit very complex problem definitions.

Now to reiterate, we have stated that

1. the DAEs must be unchanged within a phase and
2. different sets of DAEs must be in different phases.

However, this does not imply that the DAEs *must* change across a phase boundary. In other words, a phase may be introduced strictly for numerical reasons, as opposed to modeling different physical phenomena. In particular, multiple shooting segments may (or may not) be treated as phases. Furthermore, within a phase, we have a spectrum of possibilities:

1. the phase may have a single multiple shooting segment and a multiple number of integration steps;
2. the phase may be subdivided into a limited number of multiple shooting segments (e.g., 5), with multiple integration steps per segment; or
3. the number of multiple shooting segments may be *equal* to the number of integration steps (i.e., one step per segment).

Traditionally, the first approach is what we have described as the *shooting method* and the second what we have described as *multiple shooting*. Although the remainder of the book will concentrate on the third approach, it is often useful to recall this modeling hierarchy.

3.8 Function Generator

3.8.1 Description

The whole focus of this chapter has been to present different methods for transcribing a continuous problem described by DAEs into a finite-dimensional problem that can be solved using an NLP algorithm. In fact, we have described different ways to construct a

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

function generator for the NLP algorithm. The input to the function generator is the set of NLP variables $\mathbf{x}$. The purpose of the function generator is to compute the constraints $\mathbf{c}(\mathbf{x})$ and objective function $F(\mathbf{x})$ by using one of the transcription methods we have outlined. If it is not possible to compute this information, the function generator can communicate this status by setting a *function error* flag. Thus, the output of the function generator is either the requested constraint and objective function values or a flag indicating that this information cannot be computed. In fact, it is often convenient to view the function generator as a giant subroutine or "black box." Figure 3.4 illustrates this concept.

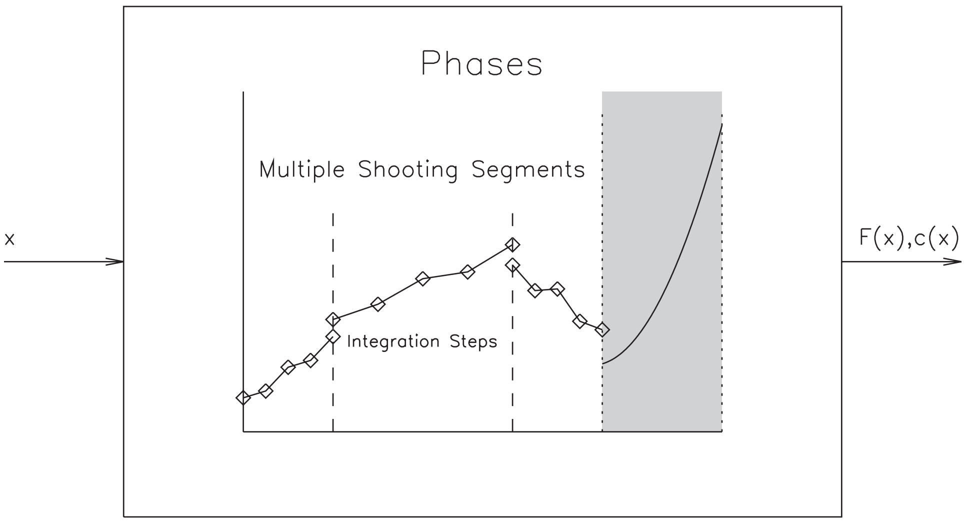

Figure 3.4. *Function generator.*

3.8.2 NLP Considerations

Section 1.16 described a number of considerations that should be addressed in order to construct a problem that is well-posed and amenable to solution by an NLP algorithm. How do those goals relate to the formulation of an optimal control problem? Clearly, a major goal is to construct a function generator, i.e., select a transcription method that is *noise free*. However, before proceeding any further, it is important to clarify the distinction between *consistency* and *accuracy*. To be more precise

- a *consistent function generator* executes the same sequence of arithmetic operations for all values of $\mathbf{x}$;
- an *accurate* function generator computes accurate approximations to the dynamics $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$.

Thus, an adaptive quadrature method, which implements a sophisticated variable-order and/or stepsize technique, will appear as "noise" to the NLP algorithm. This approach is accurate, but it is not consistent! In contrast, a fixed number of steps with an explicit integration method is a consistent process, but it is not necessarily accurate.

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

To fully appreciate the importance of this matter, it is worth reviewing our approach. Recall that we intend to choose the NLP variables $\mathbf{x}$ to optimize an NLP objective function $F(\mathbf{x})$ and satisfy a set of NLP constraints defined by the functions $\mathbf{c}(\mathbf{x})$. Newton's method requires first and second derivative information, i.e., the Jacobian and Hessian matrices. When the NLP functions are computed by a function generator, it is clear that we must differentiate the function generator in order to supply the requisite Jacobian and Hessian. The most common approach to computing gradients is via finite difference approximations as described in Section 2.2. A *forward difference* approximation to column j of the Jacobian matrix $\mathbf{G}$ is

$$\mathbf{G}_{\cdot j} = \frac{1}{\delta_j}\big[\mathbf{c}(\mathbf{x}+\boldsymbol{\Delta}_j) - \mathbf{c}(\mathbf{x})\big], \tag{3.92}$$

where the vector $\boldsymbol{\Delta}_j = \delta_j \mathbf{e}_j$ and $\mathbf{e}_j$ is a unit vector in direction j. From (2.5), a *central difference* approximation is

$$\mathbf{G}_{\cdot j} = \frac{1}{2\delta_j}\big[\mathbf{c}(\mathbf{x}+\boldsymbol{\Delta}_j) - \mathbf{c}(\mathbf{x}-\boldsymbol{\Delta}_j)\big]. \tag{3.93}$$

In order to calculate gradient information this way, it is necessary to integrate the differential equations for each perturbation. Consequently, at least n trajectories are required to evaluate a finite difference gradient, and this information may be required for each NLP iteration. A less common alternative to finite difference gradients is to integrate the so-called variational equations. For this technique, described in Section 3.9.4, an additional differential equation is introduced for each derivative and the augmented system of differential equations must be solved along with the state equations. Unfortunately, the variational equations must be derived for each application and, consequently, are used far less in general-purpose software.

While the cost of computing derivatives by finite differences is a matter of practical concern, a more important issue is the accuracy of the gradient information. Forward difference estimates are of order δ, whereas central difference estimates are $\mathcal{O}(\delta^2)$. Of course, the more accurate central difference estimates are twice as expensive as forward difference gradients. Typically, numerical implementations use forward difference estimates until nearly converged and then switch to the more accurate derivatives for convergence. While techniques for selecting the finite difference perturbation size might seem to be critical to accurate gradient evaluation, a number of effective methods are available to deal with this matter [99]. A more crucial matter is the interaction between the gradient computations and the underlying numerical interpolation and integration algorithms. A complete discussion of the matter is given in Section 3.9. We have already discussed how linear interpolation of tabular data can introduce gradient errors. However, it should be emphasized that sophisticated predictor-corrector variable-step, variable-order numerical integration algorithms also introduce "noise" into the gradients. Why? Because the evaluation of a perturbed trajectory is *not* consistent with the nominal trajectory. Thus, while these sophisticated techniques enhance the efficiency of the integration, they degrade the efficiency of the optimization. In fact, a simple fixed-step, fixed-order integrator may yield better overall efficiency in the optimization because the gradient information is more accurate. Two integration methods that are suitable for use inside the function generator are described in Brenan [47], Gear and Vu [91], and Vu [170]. Another approach, referred to as *internal differentiation*, will be described in Section 3.9.3.

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Another issue arises in the context of an optimal control application when the final time t_F is defined implicitly by a boundary or event condition instead of explicitly. In this case, we are not asking to integrate from t_I to t_F but rather from t_I *until* $\psi[\mathbf{y}(t_F), t_F] = 0$. Most numerical integration schemes *interpolate* the solution to locate the final point. On the other hand, if the final point is found by *iteration* (e.g., using a root-finding method), the net effect is to introduce noise into the external Jacobian evaluations. A better alternative is to simply add an extra variable and constraint to the overall NLP problem and avoid the use of an "internal" iteration.

In order to avoid the aforementioned difficulties, the approach we will describe in the next chapter does two things. First, we select a transcription method with one integration step per multiple shooting segment. This ensures the function generator is *consistent*. Second, we will treat the solution accuracy *outside* the NLP problem.

3.9 Dynamic System Differentiation

Let us consider a slightly more general definition of the dynamics than given in (3.1), namely

$$\dot{\mathbf{y}} = \mathbf{f}[\mathbf{y}, \mathbf{p}, t], \tag{3.94}$$

where $\mathbf{y}$ is the n dimension state vector, and $\mathbf{p}$ is an m dimension vector of parameters. The initial conditions at the fixed initial time t_I are

$$\boldsymbol{\psi}[\mathbf{y}_I, \mathbf{p}, t_I] = \mathbf{0}, \tag{3.95}$$

where the initial state $\mathbf{y}(t_I)$ is denoted as $\mathbf{y}_I$. Now the solution to (3.94) is a function of the $n+m$ variables

$$\mathbf{x} = \left[\mathbf{y}_I, \mathbf{p}\right] \tag{3.96}$$

as well as time t. Let us denote this solution by $\mathbf{y}(\mathbf{x}, t)$. In order to solve a BVP it is necessary to compute derivatives of this ODE solution with respect to the variables $\mathbf{x}$, and clearly the numerical integration and differentiation processes interact. The goal of this section is how to deal effectively with this interaction.

3.9.1 Simple Example

To illustrate the alternatives first consider the simple ODE

$$\dot{y} = y^2 \tag{3.97}$$

with initial condition $y(t_I) = y_I$ at the initial time $t = t_I = 0$. The analytic solution to the IVP is given by

$$y^*(t) = \frac{-y_I}{y_I t - 1}. \tag{3.98}$$

Suppose we want to choose y_I to satisfy the boundary condition

$$c(y_I) = y_F^* - a = \left[\frac{-y_I}{y_I t_F - 1}\right] - a = 0. \tag{3.99}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Clearly the analytic solution to this simple BVP is just

$$y_I^* = \frac{a}{1+at_F}. \tag{3.100}$$

The derivative of the boundary condition (3.99) with respect to the variable y_I is just

$$c' = (y_I t_F - 1)^{-2}. \tag{3.101}$$

Let us now consider an approximate solution to the dynamic system and subdivide the domain into M steps, where $h = \frac{t_F}{M}$. A simple Euler integration leads to the approximation

$$y_{i+1} = y_i + hf(y_i) \tag{3.102}$$

for $i = 1, \dots, M-1$. Now if the formula is applied recursively with $y_1 = y_I$, one finds

$$\begin{aligned} y_2 &= y_1 + hy_1^2, \\ y_3 &= y_2 + hy_2^2, \\ &\vdots \\ y_M &= y_{M-1} + hy_{M-1}^2. \end{aligned} \tag{3.103}$$

Differentiating (3.103) yields the derivative of the boundary condition when the dynamics are computed using the Euler approximation

$$\begin{aligned} y_2' &= 1 + 2hy_1, \\ y_3' &= y_2' + 2hy_2y_2', \\ &\vdots \\ c_E' &= y_M'. \end{aligned} \tag{3.104}$$

Clearly the (exact) derivative of the approximate dynamics given by (3.104) is not the same as the (exact) derivative of the true solution given by (3.101).

Now, let us consider solving this very simple BVP in four different ways. For all cases treating the initial condition y_I as the iteration variable x let us use a simple Newton iteration scheme (1.11)

$$x^{(k+1)} = x^{(k)} - \frac{c[x^{(k)}]}{c'[x^{(k)}]}, \tag{3.105}$$

and for simplicity let us choose $t_F = a_F = 1$. With these values the true solution from (3.100) is $x^* = \frac{1}{2}$, and so let us begin the iteration with $x^{(1)} = .6$.

Example 3.7 Analytic ODE Solution, Analytic Derivative. First let us consider solving the single constraint (3.99) using the Newton iteration (3.105). For the derivative we use the exact expression given by (3.101). Table 3.1 presents the error between the true solution $|\bar{x} - y_I^*|$ and the current iterate, as well as the error in the constraint $|c(x)|$. As expected the iteration converges quadratically to the correct solution.

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Table 3.1. *BVP iteration: analytic ODE solution, analytic derivative.*

k	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$
1	0.100	0.500
2	0.200×10^{-1}	0.833×10^{-1}
3	0.800×10^{-3}	0.321×10^{-2}
4	0.128×10^{-5}	0.512×10^{-5}
5	0.328×10^{-11}	0.131×10^{-10}
6	0.00	0.00

Example 3.8 Discretized ODE Solution, Analytic Derivative of Discretization. For realistic problems it is seldom possible to express the ODE solution in closed form as in (3.98). So let us repeat the experiment using the discretized approximation to the solution given by (3.103). A derivative is needed for the Newton iteration (3.105), and this is given by (3.104). Table 3.2 summarizes the iteration history for discretizations with 10, 100, and 1000 grid points, respectively. Since the Newton iteration uses the exact derivative of the discretization, quadratic convergence of the iterates is still observed. However, now the accuracy of the solution when compared to y_I^* is dictated by the number of grid points.

Table 3.2. *BVP iteration: discretized ODE solution with analytic derivative of discretization.*

M	10		100		1000	
k	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$
1	0.100	0.200	0.100	0.459	0.100	0.496
2	0.474×10^{-1}	0.138×10^{-1}	0.217×10^{-1}	0.708×10^{-1}	0.202×10^{-1}	0.820×10^{-1}
3	0.432×10^{-1}	0.761×10^{-4}	0.483×10^{-2}	0.230×10^{-2}	0.120×10^{-2}	0.310×10^{-2}
4	0.432×10^{-1}	0.236×10^{-8}	0.424×10^{-2}	0.257×10^{-5}	0.425×10^{-3}	0.478×10^{-5}
5	0.432×10^{-1}	0.222×10^{-15}	0.424×10^{-2}	0.325×10^{-11}	0.423×10^{-3}	0.114×10^{-10}
6	0.432×10^{-1}	0.00	0.424×10^{-2}	-0.666×10^{-15}	0.423×10^{-3}	-0.167×10^{-14}
7			0.424×10^{-2}	0.444×10^{-15}	0.423×10^{-3}	0.888×10^{-15}
8			0.424×10^{-2}	0.00	0.423×10^{-3}	0.00

Example 3.9 Discretized ODE Solution, Finite Difference Derivative of Discretization. The results illustrated in Example 3.8 demonstrate rapid convergence of the BVP iteration provided exact (analytic) derivatives of the ODE discretization are used. What if finite difference derivatives are used instead? So let us repeat the experiment using the discretized approximation to the solution given by (3.103). However, let us use the forward difference derivative

$$c' \approx \frac{y_M[x^{(k)}+\delta] - y_M[x^{(k)}]}{\delta} \tag{3.106}$$

with $\delta = 10^{-3}$ for the Newton iteration (3.105) instead of the exact expression given by (3.104). Table 3.3 summarizes the iteration history for discretizations with 10, 100, and 1000 grid points, respectively. Since the Newton iteration uses the forward difference approximation to the derivative of the discretization, convergence of the iterates is degraded somewhat when compared to Example 3.8. Nevertheless, the overall behavior compares favorably in all other respects.

Table 3.3. *BVP iteration: discretized ODE solution with finite difference derivative of discretization.*

M	10		100		1000	
k	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$
1	0.100	0.200	0.100	0.459	0.100	0.496
2	0.474×10^{-1}	0.140×10^{-1}	0.219×10^{-1}	0.716×10^{-1}	0.204×10^{-1}	0.829×10^{-1}
3	0.432×10^{-1}	0.972×10^{-4}	0.487×10^{-2}	0.248×10^{-2}	0.125×10^{-2}	0.332×10^{-2}
4	0.432×10^{-1}	0.133×10^{-6}	0.424×10^{-2}	0.774×10^{-5}	0.426×10^{-3}	0.121×10^{-4}
5	0.432×10^{-1}	0.176×10^{-9}	0.424×10^{-2}	0.149×10^{-7}	0.423×10^{-3}	0.242×10^{-7}
6	0.432×10^{-1}	0.233×10^{-12}	0.424×10^{-2}	0.285×10^{-10}	0.423×10^{-3}	0.481×10^{-10}
7	0.432×10^{-1}	0.222×10^{-15}	0.424×10^{-2}	0.540×10^{-13}	0.423×10^{-3}	0.888×10^{-13}
8	0.432×10^{-1}	0.00	0.424×10^{-2}	0.00	0.423×10^{-3}	0.933×10^{-14}
9					0.423×10^{-3}	-0.233×10^{-14}
10					0.423×10^{-3}	0.222×10^{-15}
11					0.423×10^{-3}	0.00

Example 3.10 Discretized ODE Solution, Analytic Derivative of Exact Solution. As a final experiment let us consider solving the discretized ODE problem using the analytic (exact) derivative of the true solution (3.101). Table 3.4 summarizes the iteration history for discretizations with 10, 100, and 1000 grid points, respectively. It might be expected that the analytic derivative should approximate the derivative of the discretization, especially as $M \to \infty$. In fact the BVP iteration converges rapidly for $M = 1000$. However, the analytic derivative (3.101) is *not* the derivative of the discrete approximation (3.103), and for small values of M the BVP iteration does not converge at all!

Table 3.4. *BVP iteration: discretized ODE solution, analytic derivative of exact solution.*

M	10		100		1000	
k	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$	$\lvert x^{(k)} - y_I^*\rvert$	$\lvert c(x)\rvert$
1	0.100	0.200	0.100	0.459	0.100	0.496
2	0.680×10^{-1}	0.836×10^{-1}	0.266×10^{-1}	0.914×10^{-1}	0.207×10^{-1}	0.843×10^{-1}
3	0.524×10^{-1}	0.304×10^{-1}	0.611×10^{-2}	0.734×10^{-2}	0.132×10^{-2}	0.360×10^{-2}
4	0.463×10^{-1}	0.102×10^{-1}	0.432×10^{-2}	0.306×10^{-3}	0.428×10^{-3}	0.205×10^{-4}
5	0.442×10^{-1}	0.333×10^{-2}	0.424×10^{-2}	0.117×10^{-4}	0.423×10^{-3}	0.795×10^{-7}
6	0.435×10^{-1}	0.107×10^{-2}	0.424×10^{-2}	0.446×10^{-6}	0.423×10^{-3}	0.308×10^{-9}
7	0.433×10^{-1}	0.344×10^{-3}	0.424×10^{-2}	0.170×10^{-7}	0.423×10^{-3}	0.120×10^{-11}
8	0.432×10^{-1}	0.110×10^{-3}	0.424×10^{-2}	0.647×10^{-9}	0.423×10^{-3}	0.488×10^{-14}
9	0.432×10^{-1}	0.353×10^{-4}	0.424×10^{-2}	0.246×10^{-10}	0.423×10^{-3}	-0.167×10^{-14}
10	0.432×10^{-1}	0.113×10^{-4}	0.424×10^{-2}	0.939×10^{-12}	0.423×10^{-3}	0.888×10^{-15}
11	0.432×10^{-1}	0.362×10^{-5}	0.424×10^{-2}	0.344×10^{-13}	0.423×10^{-3}	0.00
12	0.432×10^{-1}	0.116×10^{-5}	0.424×10^{-2}	0.00		
$\vdots$						
100	0.432×10^{-1}	0.222×10^{-15}				

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

3.9.2 Discretization versus Differentiation

When solving a BVP it is necessary to address two distinct numerical tasks. Specifically a discretization and/or quadrature technique is needed to solve the differential equations (3.94). Furthermore to solve the BVP it is also necessary to compute derivatives of the boundary conditions. But which comes first? The previous examples suggest that it is important to

- first discretize the ODE and then
- compute derivatives of the discretization.

Conversely, reversing the order of these operations as illustrated by Example 3.10 is both contraindicated and counterproductive! It is worth noting that similar observations can be found in Gill, Murray, and Wright [99, pp. 266–267]. This illustrates a principle we refer to as *discretize then optimize* which will be revisited in Section 4.12.

What is less obvious is an explanation of this behavior. The key notion illustrated here is the concept of a *consistent function generator*. In all four examples the evaluation of the ODE solution is performed consistently—that is, with the same number of additions, subtractions, multiplications, and divisions—regardless of what value the iteration variable x takes on. For Examples 3.7, 3.8, and 3.9 the Newton iteration used the derivative of the function generator. For Example 3.10—which behaves poorly—the Newton iteration did not use the derivative of the function generator. It is important that

- the function generator be consistent and
- the iterative process use derivatives of the function generator.

Does this preclude the use of adaptive quadrature procedures for numerical integration error control? Not necessarily, as we shall see in the next section.

3.9.3 External and Internal Differentiation

The examples in the preceding section illustrate different techniques for computing derivatives of the ODE solution $\mathbf{y}(\mathbf{x}, t_F)$ with respect to the variables $\mathbf{x}$. More precisely we are interested in constructing the matrix

$$\mathbf{G}(t_F) = \frac{\partial \mathbf{y}(t_F)}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1(t_F)}{\partial x_1} & \frac{\partial y_1(t_F)}{\partial x_2} & \cdots & \frac{\partial y_1(t_F)}{\partial x_{n+m}} \\ \frac{\partial y_2(t_F)}{\partial x_1} & \frac{\partial y_2(t_F)}{\partial x_2} & \cdots & \frac{\partial y_2(t_F)}{\partial x_{n+m}} \\ \vdots & & \ddots & \\ \frac{\partial y_n(t_F)}{\partial x_1} & \frac{\partial y_n(t_F)}{\partial x_2} & & \frac{\partial y_n(t_F)}{\partial x_{n+m}} \end{bmatrix}. \tag{3.107}$$

If we define perturbation vectors

$$\boldsymbol{\delta}_k^{\mathsf{T}} = [0, \dots, 0, \delta_k, 0, \dots, 0]^{\mathsf{T}} \tag{3.108}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

for $k = 1,2,\dots,(n+m)$, then a forward difference approximation to column k is of the form

$$\mathbf{G}_{*,k}(t_F) \approx \frac{1}{\delta_k}\left[\mathbf{y}(\mathbf{x}+\boldsymbol{\delta}_k, t_F) - \mathbf{y}(\mathbf{x}, t_F)\right]. \tag{3.109}$$

The computational implementation of (3.109) is deceptively simple and is referred to as *external differentiation*. The basic algorithm is as follows:

```
                         External Differentiation

Nominal Point    Compute $\mathbf{y}(\mathbf{x}, t_F)$
    Integrate the ODEs (3.94) from $t_I$ to $t_F$ with initial conditions $\mathbf{x}$

Perturbations    For $k = 1,\dots,(n+m)$
    · Set $\overline{\mathbf{x}} = \mathbf{x} + \boldsymbol{\delta}_k$
    · Compute $\mathbf{y}(\overline{\mathbf{x}}, t_F)$ ... Integrate the ODEs (3.94) from $t_I$ to $t_F$
    with initial conditions $\overline{\mathbf{x}}$
    · Construct $\mathbf{G}_{*,k}(t_F)$ from (3.109)
```

In effect this approach wraps a "loop" around the numerical integration of the ODE system. Since there are many effective software packages for solving the IVP this approach is very appealing because the dynamic simulation can be treated as a "black box." It is a serial process, in effect simulating each perturbation one at a time. Unfortunately the approach is not consistent. To control integration error sophisticated software may alter the number of integration steps and/or the order of the method. In particular, there is no reason to expect that the integration history will be consistent for all of the perturbations.

Instead of integrating the perturbed trajectories one at a time, let us consider doing all of them at the same time. To do so, define an augmented system of ODEs by making $(1+n+m)$ copies of (3.94)

$$\dot{\mathbf{z}}_k = \mathbf{f}[\mathbf{z}_k, \mathbf{q}_k, t] \qquad \text{for} \quad k = 0,1,\dots,(n+m). \tag{3.110}$$

By construction the augmented system (3.110) involves $n(1+n+m)$ differential variables; i.e., $\mathbf{z}$ is the $n(1+n+m)$-dimensional vector $\mathbf{z}^\mathsf{T} = (\mathbf{z}_0^\mathsf{T},\dots,\mathbf{z}_{n+m}^\mathsf{T}) = (\mathbf{y}^\mathsf{T},\dots,\mathbf{y}^\mathsf{T})$. However, now define the initial conditions for this augmented system as

$$\begin{bmatrix} \mathbf{z}_0(t_I) \\ \mathbf{z}_1(t_I) \\ \vdots \\ \mathbf{z}_n(t_I) \\ \mathbf{z}_{n+1}(t_I) \\ \vdots \\ \mathbf{z}_{n+m}(t_I) \end{bmatrix} = \begin{bmatrix} \mathbf{y}_I \\ \mathbf{y}_I + \delta_1 \widehat{\mathbf{e}}_1 \\ \vdots \\ \mathbf{y}_I + \delta_n \widehat{\mathbf{e}}_n \\ \mathbf{y}_I \\ \vdots \\ \mathbf{y}_I \end{bmatrix}, \tag{3.111}$$

where the n-vector $\widehat{\mathbf{e}}_k$ has a one in row k and zero elsewhere. In a similar fashion we define

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

the parameter vector

$$\begin{bmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \vdots \\ \mathbf{q}_n \\ \mathbf{q}_{n+1} \\ \vdots \\ \mathbf{q}_{n+m} \end{bmatrix} = \begin{bmatrix} \mathbf{p} \\ \mathbf{p} \\ \vdots \\ \mathbf{p} \\ \mathbf{p} + \delta_{n+1}\widetilde{\mathbf{e}}_{n+1} \\ \vdots \\ \mathbf{p} + \delta_{n+m}\widetilde{\mathbf{e}}_{n+m} \end{bmatrix}, \tag{3.112}$$

where the m-vector $\widetilde{\mathbf{e}}_k$ has a one in row $k-n$ and zero elsewhere. In particular if the augmented system is propagated from t_I to the final time t_F, we find that

$$\begin{bmatrix} \mathbf{z}_0(t_F) \\ \mathbf{z}_1(t_F) \\ \vdots \\ \mathbf{z}_{n+m}(t_F) \end{bmatrix} = \begin{bmatrix} \mathbf{y}(\mathbf{x}, t_F) \\ \mathbf{y}(\mathbf{x}+\boldsymbol{\delta}_1, t_F) \\ \vdots \\ \mathbf{y}(\mathbf{x}+\boldsymbol{\delta}_{n+m}, t_F) \end{bmatrix}. \tag{3.113}$$

Observe that all of the vectors needed to compute the matrix $\mathbf{G}(t_F)$ as given by (3.109) are available at the final point in (3.113). Hairer, Norsett, and Wanner [106] refer to this as a "stepsize freeze" technique for *internal differentiation*. The original idea for internal differentiation was proposed by Bock [43] using a slightly different implementation.

```
                         Internal Differentiation

Propagation     Compute $\mathbf{y}(\mathbf{x},t_F),\dots,\mathbf{y}(\mathbf{x}+\boldsymbol{\delta}_{n+m},t_F)$
    Integrate the augmented ODE system (3.110) from $t_I$ to $t_F$ with
    · initial conditions (3.111)
    · and parameter values (3.112)

Derivative Evaluation     For $k = 1,\dots,(n+m)$
    · Construct $\mathbf{G}_{*,k}(t_F)$ from (3.109) and (3.113)
```

There are three additional points that deserve comment. First, although internal differentiation was described using forward differences, clearly the approach is applicable to any of the higher-order methods outlined in Section 1.17. Second, the same techniques for choosing an optimal perturbation size, e.g., (1.166), can be applied in this setting. Finally, although the initial and final times t_I and t_F were assumed fixed in the discussion, variable time applications can also be treated. Specifically, a problem with variable initial and/or final time can be converted to one on the fixed domain $0 \le \tau \le 1$ by introducing the transformation

$$t = t_I + (t_F - t_I)\tau. \tag{3.114}$$

Example 3.11 BRUSSELATOR PROBLEM. To demonstrate the difference between external and internal differentiation let us consider the *Brusselator* problem with dynamics given by the ODEs

$$\dot{y}_1 = 1 + y_1^2 y_2 - 4y_1, \qquad y_1(0) = 1.5, \tag{3.115}$$

$$\dot{y}_2 = 3y_1 - y_1^2 y_2, \qquad y_2(0) = y_{2I} \tag{3.116}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

for $0 \le t \le 20$. For the sake of illustration let us compute forward difference approximations to

$$\left.\frac{\partial y_2(t)}{\partial y_{2I}}\right|_{t=20} = \frac{1}{\delta}\left\{y_2(20)[y_{2I}+\delta] - y_2(20)[y_{2I}]\right\} \tag{3.117}$$

for different values of the initial condition $y_{2I} = 2.90, 2.91, \ldots, 3.09, 3.1$. The finite difference perturbation size is chosen to be $\delta = |y_{2I}| \times 10^{-3}$. The ODEs were solved using a variable-step, eighth order Dormand–Prince Runge–Kutta integrator DOP853 [107] for three different values of the absolute and relative error tolerances, $\delta_A = \delta_R = 1\times 10^{-4}, 1\times 10^{-6}, 1\times 10^{-8}$. The results were computed using external and internal differentiation, and are illustrated in Figure 3.5. The left-hand column in the figure illustrates the results for external differentiation and the right-hand column presents the corresponding derivatives computed by internal differentiation. Each row in the figure corresponds to a different integration tolerance. As the nominal initial value for y_{2I} changes smoothly from a small to large value, one intuitively expects the derivative to change accordingly. Indeed, this consistent behavior is observed for the internal derivatives. In contrast, the external derivatives shown in the left-hand column change erratically. Furthermore when using internal differentiation, the derivative values appear consistent regardless of whether a large or small numerical integration tolerance is used. In contrast, the integration tolerance must be very tight to compute a reasonable derivative estimate using external derivatives.

An explanation of this behavior is afforded by examining the details of one particular case at $y_{2I} = 2.91$. Specifically the details of external and internal differentiation are given in Tables 3.5 and 3.6, respectively. In Table 3.5 the nominal trajectory required 31 integration steps, with 484 evaluations of the right-hand-side functions $\mathbf{f}(\mathbf{y},t)$. However, the positive perturbation required 30 integration steps, with 461 right-hand-side evaluations. Even though both the nominal and perturbed trajectories satisfy the relative integration error tolerance, the trajectories are clearly inconsistent. The same sequence of arithmetic operations was not performed on the nominal and perturbed trajectories. When the finite difference derivative is calculated as shown in (3.119) this inconsistent behavior is accentuated leading to the value -1.4344680. In contrast for internal differentiation the nominal and perturbed trajectories have the same number of steps and right-hand-side evaluations as shown in Table 3.6. The finite difference derivative computations given in (3.120) lead to the value -0.13319669. A consistent function generator yields a consistent derivative!

3.9.4 Variational Derivatives

Generalizing the techniques in the previous section, let us consider computing the derivatives of the state at time t with respect to the variables $\mathbf{x}$. Specifically define the $n \times (n+m)$ matrix of *variational derivatives*

$$\mathbf{G}(t) = \frac{\partial \mathbf{y}(t)}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1(t)}{\partial x_1} & \frac{\partial y_1(t)}{\partial x_2} & \cdots & \frac{\partial y_1(t)}{\partial x_{n+m}} \\ \frac{\partial y_2(t)}{\partial x_1} & \frac{\partial y_2(t)}{\partial x_2} & \cdots & \frac{\partial y_2(t)}{\partial x_{n+m}} \\ \vdots & & \ddots & \\ \frac{\partial y_n(t)}{\partial x_1} & \frac{\partial y_n(t)}{\partial x_2} & & \frac{\partial y_n(t)}{\partial x_{n+m}} \end{bmatrix}. \tag{3.118}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

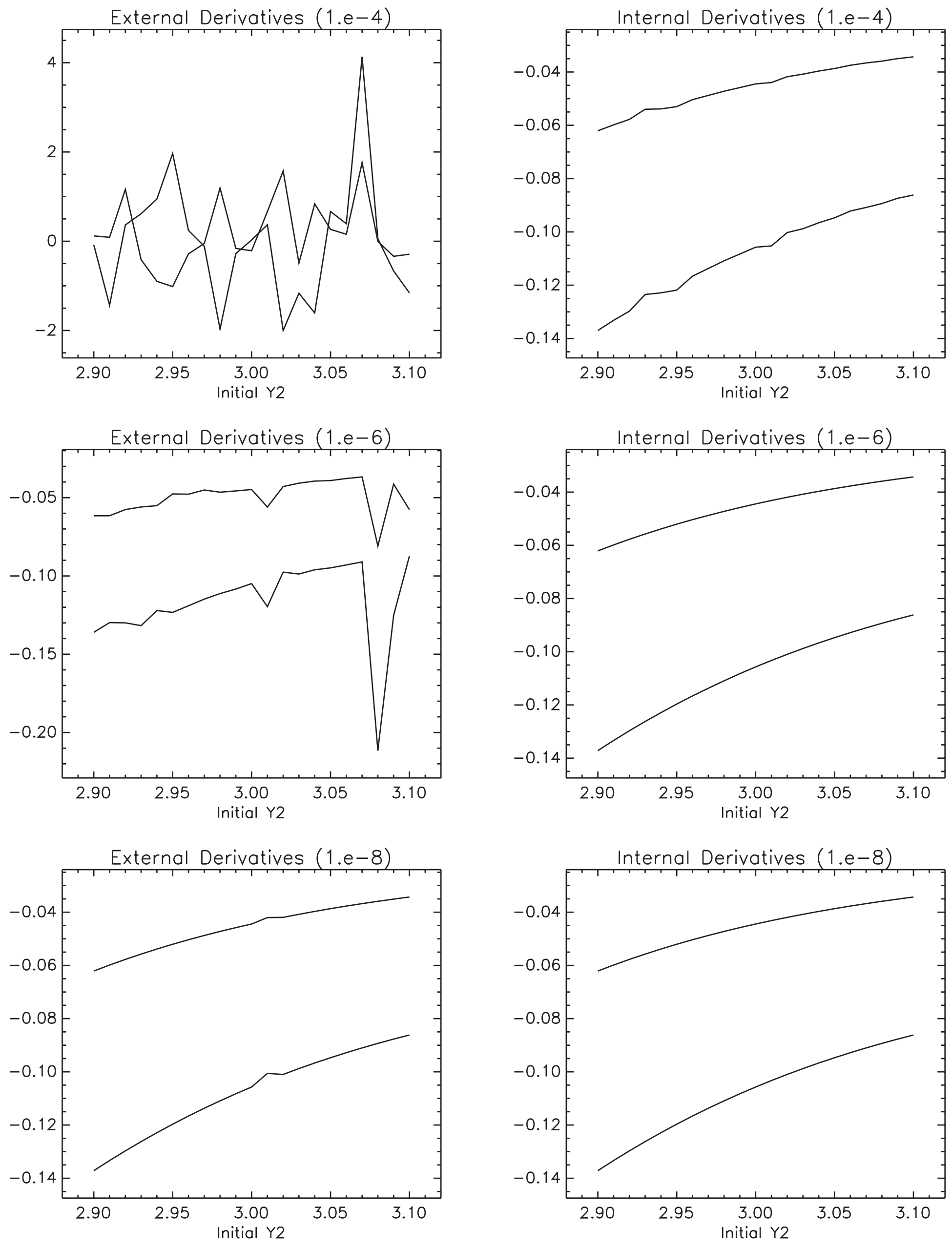

Figure 3.5. *Differentiation: external (left) versus internal (right).*

Table 3.5. *External differentiation behavior at $y_{2I} = 2.91$.*

$y_2(0)$	NSTEPS	NRHSFE	$y_2(20)$
2.91000000000000	31	484	4.605741850415169
2.91003162277660	30	461	4.605696488553900

$$\left.\frac{\partial y_2(t)}{\partial y_{2I}}\right|_{t=20} = \frac{4.605696488553900 - 4.605741850415169}{2.91003162277660 - 2.91000000000000} = -1.4344680 \tag{3.119}$$

Table 3.6. *Internal differentiation behavior at $y_{2I} = 2.91$.*

$y_2(0)$	NSTEPS	NRHSFE	$y_2(20)$
2.91000000000000	31	$1485/3 = 495$	4.607480645589384
2.91003162277660	*	*	4.607476433540143

$$\left.\frac{\partial y_2(t)}{\partial y_{2I}}\right|_{t=20} = \frac{4.607476433540143 - 4.607480645589384}{2.91003162277660 - 2.91000000000000} = -0.13319669 \tag{3.120}$$

Differentiating (3.94) leads to the *variational differential equations*

$$\dot{\mathbf{G}} = \mathbf{f}_y[\mathbf{y}(t), \mathbf{p}, t]\mathbf{G} + \mathbf{f}_p[\mathbf{y}(t), \mathbf{p}, t], \tag{3.121}$$

where

$$\mathbf{f}_y[\mathbf{y}(t), \mathbf{p}, t] = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial y_n} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial y_n} \\ \vdots & & \ddots & \\ \frac{\partial f_n}{\partial y_1} & \frac{\partial f_n}{\partial y_2} & & \frac{\partial f_n}{\partial y_n} \end{bmatrix} \tag{3.122}$$

and

$$\mathbf{f}_p[\mathbf{y}(t), \mathbf{p}, t] = \begin{bmatrix} \frac{\partial f_1}{\partial p_1} & \frac{\partial f_1}{\partial p_2} & \cdots & \frac{\partial f_1}{\partial p_m} \\ \frac{\partial f_2}{\partial p_1} & \frac{\partial f_2}{\partial p_2} & \cdots & \frac{\partial f_2}{\partial p_m} \\ \vdots & & \ddots & \\ \frac{\partial f_n}{\partial p_1} & \frac{\partial f_n}{\partial p_2} & & \frac{\partial f_n}{\partial p_m} \end{bmatrix}. \tag{3.123}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

In general the initial conditions for (3.121) are

$$\mathbf{G}(t_I) = \frac{\partial \boldsymbol{\psi}(t_I)}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \psi_1(t_I)}{\partial x_1} & \frac{\partial \psi_1(t_I)}{\partial x_2} & \cdots & \frac{\partial \psi_1(t_I)}{\partial x_{n+m}} \\ \frac{\partial \psi_2(t_I)}{\partial x_1} & \frac{\partial \psi_2(t_I)}{\partial x_2} & \cdots & \frac{\partial \psi_2(t_I)}{\partial x_{n+m}} \\ \vdots & & \ddots & \\ \frac{\partial \psi_n(t_I)}{\partial x_1} & \frac{\partial \psi_n(t_I)}{\partial x_2} & & \frac{\partial \psi_n(t_I)}{\partial x_{n+m}} \end{bmatrix}. \tag{3.124}$$

However, when the initial conditions (3.95) take the simple form

$$\mathbf{y}(t_I) = \mathbf{y}_I, \tag{3.125}$$

then the initial conditions for the variational equations are just

$$\mathbf{G}(t_I) = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{3.126}$$

Downloaded 10/25/15 to 128.6.218.72. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php