1-1-2006

Solving Higher Order Dynamic Equations on Time Scales as First Order Systems

Elizabeth R. Duke
duke6@marshall.edu

SOLVING HIGHER ORDER DYNAMIC EQUATIONS ON TIME SCALES AS FIRST ORDER SYSTEMS

Thesis submitted to
the Graduate College of
Marshall University
In partial fulfillment of
the requirements for the degree of
Master of Arts
in Mathematics

by
Elizabeth R. Duke

Dr. Bonita A. Lawrence, Ph.D., Committee Chair
Dr. Ralph W. Oberste-Vorth, Ph.D.
Dr. Scott A. Sarra, Ph.D.

Marshall University
May 2006

Contents

List of Figures

Abstract

Solving Higher Order Dynamic Equations on Time Scales as First Order Systems

Elizabeth R. Duke

Time scales calculus seeks to unite two disparate worlds: that of differential, Newtonian calculus and the difference calculus. As such, in place of differential and difference equations, time scales calculus uses dynamic equations. Many theoretical results have been developed concerning solutions of dynamic equations. However, little work has been done in the arena of developing numerical methods for approximating these solutions. This thesis work takes a first step in obtaining numerical solutions of dynamic equations—a protocol for writing higher-order dynamic equations as systems of first-order equations. This process proves necessary in obtaining numerical solutions of differential equations since the Runge-Kutta method, the generally accepted, all-purpose method for solving initial value problems, requires that DEs first be written as first-order systems. Our results indicate that whether higher-order dynamic equations can be written as equivalent first-order systems depends on which combinations of which dynamic derivatives are present.

Acknowledgments

This thesis received support from many kind, intelligent, and generous people.

One of these is Dr. Bonita A. Lawrence. Without her intellect and unflagging enthusiasm, this thesis work would not have been attempted or completed. In everything she does, Dr. Lawrence seeks to offer her students the best possible mathematical educations. She transcends her professorial duties, teaching students to be not only mathematicians, but also good and healthy people. I am one of these fortunate students.

Another such person is Dr. Scott Sarra, who taught me the joys of MATLAB and scientific computing and who introduced me to the world of applied mathematics. I owe Dr. Sarra additional gratitude for giving good and helpful advice about this thesis and life in general on several occasions.

Dr. Ralph Oberste-Vorth took me to Greece (Summer 2004) and proved to me that I could prove important mathematical results. This knowledge is empowering and invaluable. I thank Dr. O-V for this and for teaching me the real meaning of "chaos."

I want to thank Dr. Sasha N. Zill, *The* expert on insect campaniform sensilla, for helping me finish this thesis when I should have been digitizing magnets on the dorsi of *Periplaneta americana* (American cockroach) specimens.

Finally, I especially want to thank Dr. Qin Sheng (Baylor University), who so generously suggested this thesis problem and whose numerical work in time scales got my attention and continues to inspire me.

This project received support from the Department of Mathematics and Applied Sciences at Marshall University and also from a summer thesis research award from the Marshall Graduate College through the generosity of Mr. Bill Minner.

Dedication

I dedicate this thesis work to Dr. Linda W. Duke, a first-rate mother and scientist.

Also, I dedicate this work to Boo Kitty, my fine and loyal friend.

Chapter 1

Introduction

Mathematicians have long known that they can rewrite higher-order, ordinary differential equations and higher-order, ordinary difference equations as equivalent first-order systems; and this ability proves important since it facilitates analytical results and since modelling applications often incorporate such equations. As Cleve Moler explains, "Many mathematical models involve more than one unknown function and second or higher order derivatives. These models can be handled by making $x(t)$ a vector-valued function of t. Each component is either one of the unknown functions or one of its derivatives" [Mol04]. In addition to general simplification and analytic benefits, rewriting higher-order equations as first-order systems proves necessary in implementing models that rely on the solutions of ODEs since commercial software requires users to write ODEs as first-order, vector equations for numerical solution approximation [WyBa95].

Using dynamic equations as its tools, time scales calculus seeks to unify differential and difference calculus, or as the oft-quoted E.T. Bell wrote, "to harmonize the continuous and discrete, to include them in one comprehensive mathematics, and to eliminate obscurity from both" [Bel37]. Realizing this and the fact that we can indeed write higher-order equations as equivalent, first-order systems for both cases—differential and difference—we might expect that this result holds for the general case, that indeed, the whole is the sum of its parts.

However, for other results, time scales calculus has proven this natural intuition to be incorrect; at times, complications do arise from juxtaposing the continuous and discrete worlds, the analog and digital universes. Also, the fact that time scales cal-

culus commonly uses two different derivatives—*delta* and *nabla*—and recently, a third dynamic derivative, the *diamond-alpha* derivative, creates further complexity. Unlike differential and difference equations, which rely solely on the standard Newtonian limit and the difference operator, respectively, higher-order dynamic equations can employ a mixture of derivatives. Perhaps, it is this property of time scales calculus that makes its study so appealing but that also makes proving its results so potentially problematic.

Recently, a large body of work has developed in and around time scales calculus. Much of this work has necessarily been concerned with fleshing out the analytic backbone of the discipline, whereas little work has been done concerning the numerical approximation of dynamic equations not solvable in closed-form. Qin Sheng and others at the University of Dayton and Baylor University have conducted important, preliminary studies leading in this direction, writing about the prospect of using dynamic derivatives as continuous derivative approximates and presenting results concerning the validity of approximating dynamic derivatives on a proper subset of points within a given time scale (called a *grid* in numerical analysis) [She05, She06]. This thesis seeks to solve another preliminary problem that stands in the way of approximating the solutions of dynamic equations—that of writing higher-order equations as equivalent first-order systems—with the hope of facilitating the use of existing computational techniques for producing numerical solutions of dynamic equations on time scales.

Analytic work shows and numerical work confirms that nth-order dynamic equations can be written as equivalent systems of n, first-order equations when the choice of derivatives is uniform throughout (*i.e.* nth-order delta, nth-order nabla, or nth-order diamond-alpha dynamic equations). However, this result does not hold for dynamic equations of "mixed" derivatives (*e.g.* the third-order, nabla-delta-nabla dynamic equation).

Chapter 2

Time Scales Calculus

2.1 Introduction to Time Scales Calculus

Paradoxically, humans are finite beings and yet have foggy, ever-present notions concerning the infinite. While we function in a seemingly continuous reality, science tells us that all matter comes in tiny, discrete, atomic packages. The battle between the finite and infinite, the continuous and discrete, perennially plagues and intrigues science, philosophy, and humanity. In keeping with this, mathematicians encounter difficulty when modelling "real-world" scenarios due to nature's tendency to perform in seasons. The growth of plants or insect colonies or bacteria depends heavily on seasons. Nature, at times behaving in an entirely continuous manner and at other times, lying dormant until its next erratic surge forward, balks at neat classification into either the continuous or discrete realm.

These two types of domains—continuous and discrete— represent two separate branches of mathematics. High school and undergraduate college students typically learn continuous calculus (called *differential calculus*) when they study "calculus." However, there are other calculi. The other dominant calculus, difference calculus, is used to model discrete systems. Discrete systems operate on an integer or integer-like domain, whereas differential systems operate on the real numbers. Examples of mathematical problems that rely on discrete domains include calculating compound interest, counting the number of ways to draw green marbles out of a bag, counting the number of green marbles in that bag, calculating population at year n, etc. These problems are inherently discrete since, barring some strange scenarios, we do

not encounter half-marbles or half-people, and we must compound interest at precise instants, rather than infinitely often.

Until Stefan Hilger introduced time scales calculus in his 1988 Ph.D. dissertation, differential and difference calculus remained independent fields despite often producing similar results [Hil90]. Hilger united continuous and discrete calculus under one calculus system—time scales calculus. Employing the delta or Hilger derivative, time scales calculus derives its beauty and inherent usefulness from its ability to find rates of change on a "mixed" domain—one that operates on a combination of discrete and continuous intervals. Hilger's system relies on the idea that we can use the same differentiation and integration systems, changing only the time scale, not the calculus, from discrete to continuous and vice versa.

Critics discredit time scales calculus, threatening that despite its structural elegance, time scales offers the mathematical world no functionality. Paul Glendinning complains, "To be useful the theory must be able to treat a case that is not obviously covered by classical theory, or do something much more simply than before—a unification of formalism isn't enough" [Gle03]. Proponents argue that time scales facilitates analysis, not necessarily computation, for systems with a combination of continuous and discrete domains. Hilger claims that his system eliminates the need for watered-down, discrete versions of analysis, *i.e.*"a parallel presentation of discrete and continuous results, the sometimes boring, sometimes difficult performing of a proof by forming analogies or just its omission" [Hil90]. The purpose of this thesis work is to examine and facilitate time scales calculus's potential computational utility. Before tackling larger questions, I will begin by attempting to explain what a time scale is, anyway.

Definition 1 (Time Scale)**.** A *time scale*, $\mathbb{T}$, is a closed, non-empty subset of the set of real numbers, $\mathbb{R}$.

Examples of time scales include the natural numbers ($\mathbb{N}$), the integers ($\mathbb{Z}$), the real numbers ($\mathbb{R}$) themselves, the Cantor set ($\mathbb{D}$), and any closed interval of real numbers $[a, b]$, where $a, b \in \mathbb{R}$. Sets that are not time scales include $\mathbb{Q}$, $\mathbb{R} - \mathbb{Q}$, and $\mathbb{C}$. $\mathbb{C}$ is not a time scale since it is not a subset of $\mathbb{R}$, while neither $\mathbb{Q}$ nor its irrational counterpart are closed sets.

A set's classification as continuous or discrete relies on the relationships of objects in that set to one another. If the objects are mathematically distinct from each other, *i.e.* each point is measurably far from any other point, then the set is discrete; on the other hand, if no points are measurably far from the other points, the set is continuous. Since time scales can contain points that are on one side measurably distinct from the others but fused to their neighbors on the other side, we must develop a language to describe points and their neighbors. Further, in this respect, to know anything about a point itself is to know where its neighbors are. Time scales calculus, therefore, designates names for neighbors: *the forward jump operator* and *the backward jump operator*.

Definition 2 (Forward Jump Operator)**.** Let $\mathbb{T}$ be a time scale and $\inf \emptyset := \sup \mathbb{T}$. For $t \in \mathbb{T}$,

$$\sigma(t) := \inf\{s \in \mathbb{T} : s > t\}$$

is called the *forward jump operator*.

Definition 3 (Backward Jump Operator)**.** For $t \in \mathbb{T}$ and $\sup \emptyset := \inf \mathbb{T}$,

$$\rho(t) := \sup\{s \in \mathbb{T} : s < t\}$$

is called the *backward jump operator*.

Perhaps the essence of a time scale lies not so much in where its points lie, *per se*, but rather in the magnitude of the distances between them. Time scales calculus specifies functions that describe these distances.

Definition 4 (Graininess Function)**.** The *graininess function*, $\mu : \mathbb{T}^k \to [0, \infty]$ is defined by

$$\mu(t) := \sigma(t) - t$$

for all $t \in \mathbb{T}$.

Definition 5 (Backwards Graininess Function)**.** The *backwards graininess function*, $\nu : \mathbb{T}_k \to [0, \infty]$ is defined by

$$\nu(t) := t - \rho(t)$$

.

Generally speaking, applying the forward jump operator to a point t in a time scale yields the point in the time scale immediately following (or greater than) t, whereas applying the backward jump operator produces the point in the time scale immediately previous to (or less than) t. Thus, if $\mathbb{T} = \mathbb{Z}$, then $\sigma(t) = t + 1$, and $\rho(t) = t - 1$.

Keep in mind that if t exists on a continuous portion of a domain, then its most immediate neighbor must be itself since the neighbors on either side, (i.e. greater than or less than t) are infinitely close and thus indistinguishable from t. Then, if $\mathbb{T} = \mathbb{R}$, the forward jump operator, $\sigma(t) = t$, and the backward jump operator, $\rho(t) = t$.

2.2 A categorization of points

Time scales calculus defines its own vocabulary for referring to the points in continuous, discrete, and mixed domains. In a mixed domain, some points t will occupy the unique position of being both discrete-ish and continuous-ish. Introducing the following language proves necessary in distinguishing set properties (discrete and continuous) from point properties (scattered and dense), giving us words for "discrete-ish" and "continuous-ish," specifically "scattered" and "dense," respectively.

Definition 6 (Scattered)**.** If $\sigma(t) > t$, t is called a *right-scattered* point for $t \in \mathbb{T}$. Likewise, if $\rho(t) < t$, t is called a *left-scattered* point.

Definition 7 (Dense)**.** A point t such that $t \in \mathbb{T}$ is called *right-dense* if $\sigma(t) = t$. Similarly, if $\rho(t) = t$, t is called *left-dense*.

Furthermore, if a point t is scattered on both sides, *i.e.* both left-scattered and right-scattered, it is called *isolated*; but if a point t is dense on both sides, *i.e.* left-dense and right-dense, it is called *dense*.

2.3 Dynamic Derivatives

In order to develop a derivative definition for time scales, we must first define a subset ($\mathbb{T}^k$) of $\mathbb{T}$ since for time scales with left-scattered suprema, a derivative definition that requires $\sigma(t)$ will be undefined at t.

Definition 8 ($\mathbb{T}^k$)**.** If $\mathbb{T}$ has a left-scattered maximum b, then $\mathbb{T}^k = \mathbb{T} - \{b\}$. Otherwise, $\mathbb{T}^k = \mathbb{T}$.

We will see that the nabla derivative uses $\rho(t)$ in its definition. Thus, we must define T_k for right-scattered infima.

Definition 9 ($\mathbb{T}_k$)**.** If $\mathbb{T}$ has a right-scattered minimum a, then $\mathbb{T}_k = \mathbb{T} - \{a\}$. Otherwise, $\mathbb{T}_k = \mathbb{T}$.

Further, $\mathbb{T}^k \cap \mathbb{T}_k$ is denoted $\mathbb{T}_k^k$. To describe a subset of an existing time scale with m left-scattered right endpoints removed and n right-scattered left endpoints removed, we use the notation $\mathbb{T}_{k_n}^{k^m}$, where $m, n = 0, 1, 2 \ldots$, *e.g.* $\mathbb{T}^{k^0} = \mathbb{T}$, $\mathbb{T}^{k^1} = \mathbb{T}^k$. Likewise, $\mathbb{T}^{k^2} = \mathbb{T}^k - \{\rho(b)\}$ if $\rho(b)$ is left-scattered and $\mathbb{T}^{k^2} = \mathbb{T}^k$ otherwise.

With this machinery, we may proceed to define three types of differentiation on time scales.

Definition 10 (The Delta (Hilger) Derivative)**.** Assume $f : \mathbb{T} \to \mathbb{R}$ is a function, and let $t \in \mathbb{T}^k$. Then, if for any given $\epsilon > 0$, there exists a $\delta > 0$ with $s \in (t - \delta, t + \delta) \cap \mathbb{T}$, such that

$$\left|[f(\sigma(t)) - f(s)] - f^{\Delta}(t)[\sigma(t) - s]\right| \leq \epsilon \left|\sigma(t) - s\right|,$$

then $f^{\Delta}(t)$ is called the *delta derivative at* t provided it exists. If $f^{\Delta}(t)$ does exist for all $t \in \mathbb{T}^k$, we say that f is *delta differentiable* on $\mathbb{T}^k$.

Particularly for time scales that contain right-scattered infima, we would like a derivative that uses $\rho(t)$ in its definition rather than $\sigma(t)$. On a more fundamental level, if we are interested in how functions change as time moves backward, rather than forward, the *nabla derivative* is our natural choice.

Definition 11 (The Nabla Derivative)**.** A function defined on $\mathbb{T}$ is *nabla differentiable on* $\mathbb{T}_k$ if for every $\epsilon > 0$, there exists a $\delta > 0$ with $s \in (t - \delta, t + \delta) \cap \mathbb{T}$ such that the inequality

$$\left|f(\rho(t)) - f(s) - f^{\nabla}(t)(\rho(t) - s)\right| < \epsilon \left|\rho(t) - s\right|$$

holds. $f^{\nabla}(t)$ is called the *nabla derivative of* f *at* t.

A third dynamic derivative, the diamond-alpha derivative $f^{\diamond_\alpha}$, has attracted recent attention because of its usefulness as a conventional derivative approximate [She05, RoSh06].

Definition 12 (The Diamond-alpha Derivative)**.** $f^{\diamond_\alpha}(t)$ is called the *diamond-alpha derivative* of f at a point t if

$$f^{\diamond_\alpha}(t) = \alpha f^{\Delta}(t) + (1-\alpha) f^{\nabla}(t), \qquad 0 \leq \alpha \leq 1.$$

Note that f is *diamond-alpha differentiable* if and only if f is delta and nabla differentiable.

Chapter 3

Computing Dynamic Derivatives

While the definitions given in the previous chapter precisely define their respective dynamic derivatives, calculating derivatives from these definitions is not necessarily straightforward. This chapter will present computationally-friendly dynamic derivative definitions in order to simplify analytic results and provide formulas for use in numerical computations. The general structure for this section was inspired by Qin Sheng's work [She05].

Dynamic derivative formulae depend on the time scale structures on which derivatives are to be taken. Thus, to simplify the following discussion, we provide notation that decomposes a time scale $\mathbb{T}$ into the following sets:

$\mathbb{T}_I := \{t \in \mathbb{T}$: t is isolated, *i.e.* left-scattered and right-scattered$\}$,

$\mathbb{T}_D := \{t \in \mathbb{T} : t$ is dense, *i.e.* left-dense and right-dense$\}$,

$\mathbb{T}_{DS} := \{t \in \mathbb{T} : t$ is left-dense and right-scattered$\}$,

$\mathbb{T}_{SD} := \{t \in \mathbb{T} : t$ is left-scattered and right-dense$\}$.

3.1 First-order Derivatives

Let f be once delta differentiable for $t \in \mathbb{T}$. Then, we can define formulae for $f^{\Delta}(t)$ as follows:

$$f^{\Delta}(t) = \begin{cases} \frac{f(\sigma(t))-f(t)}{\mu(t)}, & t \in \mathbb{T}_I \bigcup \mathbb{T}_{DS}; \\ f'(t), & \text{otherwise.} \end{cases} \tag{3.1}$$

Next, if f is one-time nabla differentiable for $t \in \mathbb{T}$, then we can write that

$$f^{\nabla}(t) = \begin{cases} \frac{f(t)-f(\rho(t))}{\nu(t)}, & t \in \mathbb{T}_I \bigcup \mathbb{T}_{SD}; \\ f'(t), & \text{otherwise.} \end{cases} \tag{3.2}$$

Finally, assume f is once diamond-alpha differentiable, then we define $f^{\diamond_\alpha}(t)$:

$$f^{\diamond_\alpha}(t) = \begin{cases} \alpha\frac{f(\sigma(t))-f(t)}{\mu(t)} + (1-\alpha)f'(t), & t \in \mathbb{T}_{SD}; \\ \alpha f'(t) + (1-\alpha)\frac{f(t)-f(\rho(t))}{\nu(t)}, & t \in \mathbb{T}_{DS}; \\ \alpha\frac{f(\sigma(t))-f(t)}{\mu(t)} + (1-\alpha)\frac{f(t)-f(\rho(t))}{\nu(t)}, & t \in \mathbb{T}_I; \\ f'(t), & t \in \mathbb{T}_D. \end{cases} \tag{3.3}$$

3.2 Special considerations for Higher-order Derivatives

When calculating higher-order derivatives, we must consider some situations that we do not normally encounter when computing continuous or discrete derivatives separately. Even calculating second order dynamic derivatives gives rise to a number of questions. For example, below we attempt to find a formula for $f^{\Delta\nabla}(t)$ for $t \in \mathbb{T}_I \bigcap \mathbb{T}^{k^2}$, or in other words, for t on which $f^{\Delta\nabla}(t) \neq f''(t)$.

$$\begin{aligned} f^{\Delta\nabla}(t) &= (f^{\Delta}(t))^{\nabla} \\ &= \frac{f^{\Delta}(t)-f^{\nabla}(\rho(t))}{\nu(t)} \\ &= \frac{1}{\nu(t)}\left[\frac{f(\sigma(t))-f(t)}{\mu(t)} - \frac{f(\sigma(\rho(t)))-f(\rho(t))}{\mu(\rho(t))}\right] \end{aligned} \tag{3.4}$$

Simplifying this final expression raises some interesting questions:

$$\text{Is } \sigma(\rho(t)) = t?$$

And,

$$\text{is } \mu(\rho(t)) = \nu(t)?$$

The answer to either questions is yes—provided that t is not left-dense and right-scattered. In the case that t is left-dense and right-scattered, *i.e.*, $t = \rho(t)$ and $\sigma(t) \neq t$, then $\sigma(\rho(t)) = \sigma(t) \neq t$.

Additionally, we might wonder whether $\sigma(t)$ and $\rho(t)$ are commutative. The following example demonstrates that, in general, they are not.

Example 1. Consider the time scale $\mathbb{T} = [1, 2] \cup \{3, 4, 5\}$. First, let's notice that we can decompose this time scale using the notation introduced above:

$$\mathbb{T}_I = \{3, 4, 5\};$$

$$\mathbb{T}_D = [1, 2);$$

$$\mathbb{T}_{DS} = \{2\}.$$

Thus, $\mathbb{T} = \mathbb{T}_I \cup \mathbb{T}_D \cup \mathbb{T}_{DS}$. Note that if we let $t = 2$, then $\rho(\sigma(t)) = \rho(3) = 2$; but $\sigma(\rho(t)) = \sigma(2) = 3$. Therefore, $\sigma(\rho(t)) \neq \rho(\sigma(t))$.

The example illustrates that for certain time scales, $\sigma(t)$ and $\rho(t)$ are not commutative operators. However, for $\mathbb{T} = \mathbb{T}_I$, $\sigma(t)$ and $\rho(t)$ do commute. In the following formulae, the conditions have been set such that $\sigma(\rho(t)) = \rho(\sigma(t)) = t$; $\mu(\rho(t)) = \nu(t)$; and $\nu(\sigma(t)) = \mu(t)$. Moreover, simplified notation is used where appropriate.

In chapter 8 of Bohner and Peterson's second time scales text [BoPe03], *Disconjugacy and Higher Order Dynamic Equations*, Paul Eloe explains a further complication for higher-order dynamic equations:

> *Analysis on time scales provides both a unification and an extension of the theories of differential equations and finite difference equations... Even so, the unification here is quite interesting for the following reason. A standard tool is lost in the study of higher order dynamic equations on time scales. If the sigma function is not differentiable, then one can not take higher order derivatives of products or ratios.*

Indeed, not surprisingly, Eloe is quite right. However, in this manuscript, we have side-stepped this issue somewhat by requiring that dynamic equations be n-times differentiable before giving our conclusions.

3.3 Second-order Derivatives

Let f be twice delta differentiable, then a computational formula for the delta-delta dynamic derivative of f is given below.

$$f^{\Delta\Delta}(t) = \begin{cases} \frac{f(\sigma(\sigma(t)))-f(\sigma(t))}{\mu(\sigma(t))\mu(t)} - \frac{f(\sigma(t))-f(t)}{\mu^2(t))}, & t \in \mathbb{T}_I \bigcap \mathbb{T}^{k^2}; \\ f''(t), & \text{otherwise.} \end{cases} \tag{3.5}$$

Let f be twice nabla differentiable, then the nabla-nabla derivative of f can be calculated as follows:

$$f^{\nabla\nabla}(t) = \begin{cases} \frac{f(t)-f(\rho(t))}{\nu^2(t)} - \frac{f(\rho(t))-f(\rho(\rho(t))}{\nu(\rho(t))\nu(t)}, & t \in \mathbb{T}_I \bigcap \mathbb{T}_{k^2}; \\ f''(t), & \text{otherwise.} \end{cases} \tag{3.6}$$

For a once delta, once nabla differentiable function f, the delta-nabla dynamic derivative is

$$f^{\Delta\nabla}(t) = \begin{cases} \frac{f(\sigma(t))-f(t)}{\mu(t)\nu(t)} - \frac{f(t)-f(\rho(t))}{\nu^2(t)}, & t \in \mathbb{T}_I \bigcap \mathbb{T}_k^k; \\ f''(t), & \text{otherwise.} \end{cases} \tag{3.7}$$

For a once nabla, once delta differentiable function f, the delta-nabla dynamic derivative is

$$f^{\nabla\Delta}(t) = \begin{cases} \frac{f(\sigma(t))-f(t)}{\mu^2(t)} - \frac{f(t)-f(\rho(t))}{\mu(t)\nu(t)}, & t \in \mathbb{T}_I \bigcap \mathbb{T}_k^k; \\ f''(t), & \text{otherwise.} \end{cases} \tag{3.8}$$

If f is twice diamond-alpha differentiable, then formulae for the diamond-alpha-diamond-alpha derivative are given.

$$\begin{aligned} f^{\diamond_\alpha\diamond_\alpha}(t) &= [\alpha f^{\Delta}(t) + (1-\alpha) f^{\nabla}(t)]^{\diamond_\alpha} \\ &= (\alpha f^{\Delta}(t))^{\diamond_\alpha} + [(1-\alpha) f^{\nabla}(t)]^{\diamond_\alpha} \\ &= \alpha[\alpha f^{\Delta\Delta}(t) + (1-\alpha) f^{\Delta\nabla}(t)] + (1-\alpha)[\alpha f^{\nabla\Delta}(t) + (1-\alpha) f^{\nabla\nabla}(t)] \\ &= \alpha^2 f^{\Delta\Delta}(t) + \alpha(1-\alpha) f^{\Delta\nabla}(t) + \alpha(1-\alpha) f^{\nabla\Delta}(t) + (1-\alpha)^2 f^{\nabla\nabla}(t) \end{aligned}$$

For a once diamond-alpha and once delta differentiable function f, the diamond-alpha-delta dynamic derivative is

$$\begin{aligned} f^{\diamond_\alpha\Delta} &= [\alpha f^{\Delta}(t) + (1-\alpha) f^{\nabla}(t)]^{\Delta} \\ &= \alpha f^{\Delta\Delta}(t) + (1-\alpha) f^{\nabla\Delta}(t) \end{aligned}$$

Given f, a once delta, once diamond-alpha differentiable function, the following formula holds for the delta-diamond-alpha dynamic derivative:

$$\begin{aligned} f^{\Delta\diamond_\alpha} &= (f^{\Delta}(t))^{\diamond_\alpha} \\ &= \alpha f^{\Delta\Delta}(t) + (1-\alpha) f^{\Delta\nabla}(t) \end{aligned}$$

Let f be a once diamond-alpha, once nabla differentiable function, the diamond-alpha-nabla dynamic derivative is

$$\begin{aligned} f^{\diamond_\alpha\nabla} &= [\alpha f^{\Delta}(t) + (1-\alpha) f^{\nabla}(t)]^{\nabla} \\ &= \alpha f^{\Delta\nabla}(t) + (1-\alpha) f^{\nabla\nabla}(t) \end{aligned}$$

If f is once nabla, once diamond-alpha differentiable, the nabla-diamond-alpha derivative is

$$\begin{aligned} f^{\nabla\diamond_\alpha} &= (f^{\nabla}(t))^{\diamond_\alpha} \\ &= \alpha f^{\nabla\Delta}(t) + (1-\alpha) f^{\nabla\nabla}(t) \end{aligned}$$

Chapter 4

Ordinary Differential Equations

Assuming that we are studying dynamic equations defined on the real numbers, i.e. $\mathbb{T} = \mathbb{R}$, or some continuous interval therein, a large body of theoretical and computational work exists surrounding the solutions (both exact and approximate) since such dynamic equations are conventional, ordinary differential equations—ODEs. Thus, the discussion of solving dynamic equations on time scales will begin with a summary of the relevant, existing ODE cannon. Most (if not all) of the results of the first section are found in [KePe04, Zil97]. More information about higher-order ODEs is available in [Tre96].

4.1 Initial Value Problems

Definition 13 (ODEs)**.** A *first-order ordinary differential equation* takes the form

$$x'(t) = f(t, x(t)), \tag{4.1}$$

where $x(t)$ is a real scalar, complex scalar, or vector function of t, and $f(t, x(t))$ is continuous on its domain.

The differential equation $x'(t) = f(t, x(t))$ is called *ordinary* since it involves the derivative of a dependent variable x with respect only to t, a single, independent variable (as opposed to a *partial* differential equation, which involves derivatives taken with respect to multiple independent variables). This ODE is of *first-order* because the highest order derivative appearing in (4.1) is first-order.

An *initial value problem*, or IVP, is created when differential equations of the form (4.1) are paired with some initial data $(t_0, x(t_0))$, where $x(t_0)$ is often denoted x_0. Such an initial data point is called an *initial condition*. Solving an initial value problem requires that we find a differentiable function $x(t)$ such that (4.1) is satisfied on some open interval I containing t_0.

4.2 Higher-Order Differential Equations

Despite causing mathematicians headaches, higher-order differential equations often perform instrumental duties in modeling applications and provide a wealth of interesting mathematical material. We define the glorious, headache-producing equations that involve derivatives of arbitrary order below.

Definition 14 (nth-Order Initial Value Problems)**.** Let f be an n times differentiable function of t and $x(t)$, defined on an interval $t \in (a, b)$, where $x(t)$ is a real scalar, complex scalar, or vector function of t. For $t_0 \in (a, b)$, the *nth-order initial value problem* is

$$x^{(n)}(t) = f(t, x(t), x'(t), \ldots, x^{(n-1)}(t)), \tag{4.2}$$

subject to the initial conditions $x(t_0) = x_0$, $x'(t_0) = x_1$, $\ldots$, $x^{(n-1)}(t_0) = x_{n-1}$.

Defining nth-order ODEs is sometimes deemed unnecessary because higher-order ODEs can be written as equivalent systems of first-order ODEs by introducing additional variables, which represent lower-order terms. Consider the following example:

$$x'''(t) = x'(t)x(t) - 2t(x''(t))^2, \tag{4.3}$$

for t in the domain of f with three initial conditions:

$$x(t_0) = x_0; \quad x'(t_0) = x_1; \quad x''(t_0) = x_2. \tag{4.4}$$

To write (4.3) as system of equations of the form (4.1), first, we introduce lower-order

variables:

$$u_0(t) = x(t);$$
$$u_1(t) = x'(t);$$
$$u_2(t) = x''(t).$$

Using these lower-order variables, we write (4.3) as a vector equation:

$$\begin{bmatrix} u_0'(t) \\ u_1'(t) \\ u_2'(t) \end{bmatrix} = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_1(t)u_0(t) - 2t(u_2(t))^2 \end{bmatrix}. \tag{4.5}$$

Then, if we define

$$U(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \\ u_2(t) \end{bmatrix}; \quad F(t, U(t)) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_1(t)u_0(t) - 2t(u_2(t))^2 \end{bmatrix}, \tag{4.6}$$

we have $U'(t) = F(t, U(t))$, with the initial condition

$$U(t_0) = \begin{bmatrix} u_0(t_0) \\ u_1(t_0) \\ u_2(t_0) \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}, \tag{4.7}$$

a first-order, initial value problem. Generally, any single equation of order n can be written as a first-order, vector equation in $\mathbb{R}^n$.

Definition 15 (Corresponding Systems of Equations)**.** A system of equations

$$U'(t) = F(t, U(t)) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ f(t, u_0(t), u_1(t), \ldots, u_n(t)) \end{bmatrix} \text{ with initial condition } U(t_0) = U_0$$

corresponds to the nth order IVP $x^{(n)}(t) = f(t, x(t), x'(t), \ldots, x^{(n-1)}(t))$,
$x(t_0) = x_0$, $x'(t_0) = x_1$, $\ldots$, $x^{(n-1)}(t_0) = x_{n-1}$ whenever

$$U(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \\ \vdots \\ u_{(n-1)}(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ x'(t) \\ \vdots \\ x^{(n-1)}(t) \end{bmatrix}, \quad U_0 = \begin{bmatrix} u_0(t_0) \\ u_1(t_0) \\ \vdots \\ u_{(n-1)}(t_0) \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(n-1)} \end{bmatrix}, \tag{4.8}$$

and $u'_{(n-1)}(t) = f(t, u_0(t), u_1(t), \ldots, u_{(n-1)}(t))$.

Theorem 1 (Equivalence of Solutions)**.** Solving the nth-order initial value problem,
$x^{(n)}(t) = f(t, x(t), x'(t), \ldots, x^{(n-1)}(t))$, $x(t_0) = x_0, x'(t_0) = x_1, \ldots, x^{(n-1)}(t_0) = x_{n-1}$,
is equivalent to solving the corresponding system of n, first-order equations.

Proof. By definition,

$$U'(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \\ \vdots \\ u_{(n-1)}(t) \end{bmatrix}' = \begin{bmatrix} u'_0(t) \\ u'_1(t) \\ \vdots \\ u'_{(n-1)}(t) \end{bmatrix} = \begin{bmatrix} x'(t) \\ x''(t) \\ \vdots \\ x^{(n)}(t) \end{bmatrix} = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_{(n-1)}(t) \\ f(t, x(t), x'(t), \ldots, x^{(n)}(t)) \end{bmatrix}.$$

Since $x^{(n)}(t) = f(t, x(t), x'(t), \ldots, x^{(n-1)}(t))$ and the initial conditions of (4.2) are met, and since all other equations in the system are true by design,
(*e.g.* $u'_0(t) = x'(t) = u_1(t)$), solving

$$U'(t) = F(t, U(t)), \quad U(t_0) = \begin{bmatrix} u_0(t_0) \\ u_1(t_0) \\ \vdots \\ u_{n-1}(t_0) \end{bmatrix}$$

is equivalent to solving the nth-order DE.

Chapter 5

Ordinary Difference Equations

Kelley and Peterson write,"just as the differential operator plays the central role in the differential calculus, the difference operator is the basic component of calculations involving finite differences" [KePe01]. Rather than finding rates of change using a conventional, continuous derivative, difference equations use the *difference operator*; and, as suggested in the introduction, difference equations operate on discrete domains. Refer to [KePe01] for the results of the first section.

Definition 16. If $x(t)$ is a real scalar, complex scalar, or vector function of the variable t, the *difference operator*, Δ, applied to x at t is defined

$$\Delta x(t) = x(t+1) - x(t). \tag{5.1}$$

The goal of this chapter is to demonstrate and prove that initial value problems of higher-order difference equations can be solved by writing them as first-order systems following the same protocol established for differential equations. We begin by introducing initial value problems for first-order difference equations.

5.1 Initial Value Problems

Just as rates of change sometimes describe the behavior of variables of interest in continuous domains, so can we define such relationships using difference equations.

Definition 17. A *first-order ordinary difference equation* can be written in the form

$$\Delta x(t) = f(t,\, x(t)), \tag{5.2}$$

where $t = a,\ a+1,\ a+2, \ldots$.

Defining initial value problems is also similar to the same task for IVPs of differential equations. A difference equation of the form (5.2) together with initial condition $x(t_0) = x_0$ is called an *initial value problem*.

5.2 Higher-Order Difference Equations

Calculating higher-order differences simply involves using the difference operator multiple times. In fact, nth-order differences are obtained by applying the difference operator n times.

Example 2.

$$\begin{aligned}
\Delta^2 x(t) &= \Delta(\Delta x(t)) && (5.3)\\
&= \Delta(x(t+1) - x(t)) && (5.4)\\
&= \Delta(x(t+1)) - \Delta(x(t)) && (5.5)\\
&= x(t+2) - 2x(t+1) + x(t). && (5.6)
\end{aligned}$$

Thus, an *nth-order difference equation* can be written in terms of difference operators as in the upper three equations (5.3), (5.4), (5.5), or can be written in terms of functions of t for $t, t+1, \ldots, t+n$ as in (5.6). Here, we have chosen to use the difference operator notation to define higher-order difference equations in order that it will correspond to differential and dynamic equation notation.

Definition 18. Let $x(t)$ be a function of a discrete set $t = a,\ a+1,\ a+2, \ldots, a+n$ that is n times differenceable. Then, the *nth-order ordinary difference equation* takes the form

$$\Delta^n x(t) = f(t, x(t), \Delta x(t), \ldots, \Delta^{(n-1)} x(t)). \tag{5.7}$$

When the nth-order difference equation is subject to initial conditions

$$x(t_0) = x_0,\ \Delta x(t_0) = x_1,\ \Delta^2 x(t_0) = x_2,\ \ldots,\ \Delta^{(n-1)} x(t_0) = x_{(n-1)}, \tag{5.8}$$

(5.7) and (5.8) together are called an *nth-order initial value problem.*

As with differential equations, ordinary difference equations of order greater than one can be written as systems of first-order equations using a change of variables as described in the following example.

Example 3. Let $\Delta^4 x(t) = 2x(t)\,\Delta x(t) - sin(t)\,\Delta^2 x(t)$. Since this difference equation is fourth order, we will assign four variables, $u_0(t), \ldots, u_3(t)$.

$$\begin{aligned} u_0(t) &= x(t) \\ u_1(t) &= \Delta x(t) \\ u_2(t) &= \Delta^2 x(t) \\ u_3(t) &= \Delta^3 x(t) \end{aligned} \tag{5.9}$$

Then, define a vector function $U(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \\ u_2(t) \\ u_3(t) \end{bmatrix}$. Then, apply the difference operator to the vector function $U(t)$:

$$\Delta U(t) = \begin{bmatrix} \Delta u_0(t) \\ \Delta u_1(t) \\ \Delta u_2(t) \\ \Delta u_3(t) \end{bmatrix} = \begin{bmatrix} \Delta x(t) \\ \Delta^2 x(t) \\ \Delta^3 x(t) \\ \Delta^4 x(t) \end{bmatrix} = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \\ 2u_0(t)u_1(t) - sin(t)u_2(t) \end{bmatrix}. \tag{5.10}$$

Thus, if

$$F(t, U(t)) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \\ 2u_0(t)u_1(t) - sin(t)u_2(t) \end{bmatrix}, \tag{5.11}$$

then we can write (5.9) as a first-order difference equation in $\mathbb{R}^4$, $\Delta U(t) = F(t, U(t))$.

In the previous example, a fourth-order difference equation (5.9) was written as a

first-order difference equation in $\mathbb{R}^4$, or, in other words, as a system of four, first order equations. In general, it is true that an nth-order difference equation can be written as a first-order vector equation in $\mathbb{R}^n$.

Definition 19 (Corresponding Systems of Equations)**.** A system of equations

$$\Delta U(t) = F(t, U(t)) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ f(t, u_0(t), \ldots, u_n(t)) \end{bmatrix} \text{ with initial condition } U(t_0) = U_0$$

corresponds to the nth order IVP $\Delta^n x(t) = f(t, x(t), \Delta x(t), \Delta^2 x(t), \ldots, \Delta^{(n-1)} x(t))$, $x(t_0) = x_0$, $\Delta x(t_0) = x_1, \ldots, \Delta^{(n-1)} x(t_0) = x_{n-1}$ whenever

$$U(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \\ \vdots \\ u_{(n-1)}(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ \Delta x(t) \\ \vdots \\ \Delta^{(n-1)} x(t) \end{bmatrix}, \quad U_0 = \begin{bmatrix} u_1(t_0) \\ u_2(t_0) \\ \vdots \\ u_n(t_0) \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(n-1)} \end{bmatrix}, \tag{5.12}$$

and $\Delta u_n(t) = f(t, u_0(t), u_1(t), \ldots, u_{(n-1)}(t))$.

Theorem 2 (Equivalence of Solutions)**.** Solving the nth-order initial value problem (5.8) is equivalent to solving the corresponding system of n, first-order equations.

Proof.

By definition,

$$\Delta U(t) = \begin{bmatrix} \Delta u_0(t) \\ \Delta u_1(t) \\ \vdots \\ \Delta u_{(n-1)}(t) \end{bmatrix} = \begin{bmatrix} \Delta x(t) \\ \Delta^2 x(t) \\ \vdots \\ \Delta^n x(t) \end{bmatrix} = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ f(t, x(t), \Delta x(t), \ldots, \Delta^n x(t)) \end{bmatrix}.$$

Since $\Delta^n x(t) = f(t, x(t), \Delta x^(t), \ldots, \Delta^{(n-1)} x(t))$ and the initial conditions (5.8) are met, and since all other equations in the system are necessarily true,

(*e.g.* $\Delta u_0(t) = \Delta x(t) = u_1(t)$), solving

$$\Delta U(t) = F(t, U(t)),\ U(t_0) = \begin{bmatrix} u_0(t_0) \\ u_1(t_0) \\ \vdots \\ u_{n-1}(t_0) \end{bmatrix}$$

is equivalent to solving the nth-order difference equation.

Chapter 6

Ordinary Dynamic Equations

Finally, we will discuss Stefan Hilger's unification of differential and difference equations: the dynamic equations of time scales calculus. We begin with their definitions.

A *first-order, ordinary, delta dynamic equation* takes the form

$$x^{\Delta}(t) = f(t, x(t), x(\sigma(t)), \tag{6.1}$$

where $x(t)$ is a real scalar or vector function of $t \in \mathbb{T}$.

A *first-order, ordinary, nabla dynamic equation* is written

$$x^{\nabla}(t) = f(t, x(t), x(\rho(t))), \tag{6.2}$$

where $x(t)$ is a real scalar or vector function of $t \in \mathbb{T}$.

Finally, *first-order, ordinary, diamond-alpha dynamic equations* assume the following form

$$x^{\diamond_\alpha}(t) = f(t, x(t), x(\rho(t)), x(\sigma(t))), \tag{6.3}$$

where $x(t)$ is a real scalar or vector function of $t \in \mathbb{T}$.

6.1 Initial Value Problems

As for differential and difference equations, we define initial value problems for each of the three derivatives on time scales. Care must be taken in defining the time scales on which the solutions of the dynamic equations are defined (*e.g.* $T_{k^3}^{k^3}$) for a

third-order diamond alpha equation; otherwise, these definitions do not differ from those introduced for the traditional cases, $\mathbb{T} = \mathbb{R}$ (differential equations) and $\mathbb{T} = \mathbb{Z}$ (difference equations).

Definition 20. Given $t_0 \in \mathbb{T}$ and $x(t_0)$, the problem

$$x^{\Delta}(t) = f(t, x(t)), \quad x(t_0) = x_0 \tag{6.4}$$

is called an *initial value problem*. A function $x : \mathbb{T} \to \mathbb{R}$ with $x(t_0) = x_0$ that satisfies $x^{\Delta}(t) = f(t, x(t))$ for all $t \in \mathbb{T}^k$ is called a *solution* of this IVP.

Definition 21. A function $x(t)$ is called a *solution* of the nabla dynamic equation if $x^{\nabla}(t) = f(t, x(t))$ is satisfied for all $t \in \mathbb{T}_k$.

If $t_0 \in \mathbb{T}$, the problem

$$x^{\nabla}(t) = f(t, x(t)), \quad x(t_0) = x_0 \tag{6.5}$$

is called an *initial value problem*, and $x(t)$ satisfying the nabla dynamic equation (6.5) is called a *solution* of this IVP.

Definition 22. For $t_0 \in \mathbb{T}$, the problem

$$x^{\diamond_\alpha}(t) = f(t, x(t)), \quad x(t_0) = x_0 \tag{6.6}$$

is called an *initial value problem*, and a solution $x(t)$ that satisfies the diamond-alpha dynamic equation (6.6) for all $t \in \mathbb{T}_k^k$ with $x(t_0) = x_0$ is called a *solution* of this IVP.

6.2 nth-Order Dynamic Equations

At present, there are three types of dynamic derivatives: delta, nabla, and diamond-alpha. nth-order dynamic derivatives may employ any combination or permutation of these three derivatives. Thus, we have nine second-order dynamic derivatives–and thus nine formats for the generalized second order equation. Further, we have 3^n general, nth-order dynamic equations.

As in the two previous chapters, this section aims to show when and/or if nth-order dynamic equations can be written as equivalent first-order systems. First, I define nth-

order dynamic equations of the same type, *i.e.* nth-order delta, nth-order nabla, and nth-order diamond-alpha dynamic equations. Then, I will attempt a generalized structure for any nth-order dynamic equation. In general, higher-order dynamic equations that use a uniform choice of dynamic derivative (*i.e.* delta, nabla, or diamond-alpha) for each of the n derivatives involved can be written as equivalent, first-order systems. Also, any second-order dynamic equation initial value problem (not necessarily of uniform derivative) can be written and solved as a system of two equations. However, a similar statement cannot be made for the "mixed," generalized, higher-order derivative.

Definition 23 (nth-Order Delta Initial Value Problems)**.** Let $x(t)$ be an n times delta differentiable function for $t \in \mathbb{T}^{k^n}$. For $t_0 \in \mathbb{T}$, the nth-order initial value problem is

$$x^{\Delta^n}(t) = f(t, x(t), x^{\Delta}(t), \ldots, x^{\Delta^{(n-1)}}(t)), \tag{6.7}$$

subject to the initial conditions $x(t_0) = x_0$, $x^{\Delta}(t_0) = x_1$, $\ldots$, $x^{\Delta^{(n-1)}}(t_0) = x_{n-1}$.

In *Advances in Dynamic Equations on Time Scales*, the most current text on the subject of time scales calculus, Paul Eloe gives the result for linear, nth-order, delta dynamic equations that we would like to develop in the present paper for general, higher-order dynamic equations:

$$Lx = 0 \quad with \quad Lx := x^{\Delta^n} + \textstyle\sum_{i=1}^{n} q_i (x^{\Delta^{n-i}})^{\sigma}, \tag{6.8}$$

where q_i is rd-continuous (*cf.* appendix for a brief discussion of rd-continuity), $i = 1, \ldots, n$ and where $1 + \mu q_i \neq 0$ on $\mathbb{T}$. For such an equation, we can write an equivalent system of first-order equations, assigning lower-order variables and differentiating [BoPe03]. Later in the section, we will show that this results holds for non-linear equations.

Definition 24 (nth-Order Nabla IVPs)**.** Let x be an n times nabla differentiable function of $t \in \mathbb{T}_{k^n}$. The *nth-order initial value problem* for $t_0 \in \mathbb{T}$ is

$$x^{\nabla^n}(t) = f(t, x(t), x^{\nabla}(t), \ldots, x^{\nabla^{(n-1)}}(t)), \tag{6.9}$$

subject to the initial conditions $x(t_0) = x_0$, $x^{\nabla}(t_0) = x_1$, $\ldots$, $x^{\nabla^{(n-1)}}(t_0) = x_{n-1}$.

Definition 25 (nth-Order Diamond-Alpha IVPs)**.** Let x be defined and differentiable for all $t \in \mathbb{T}_{k^n}^{k^n}$. The *nth-order initial value problem* takes the following form when $t_0 \in \mathbb{T}$:

$$x^{\diamond_\alpha{}^n}(t) = f(t, x(t), x^{\diamond_\alpha}(t), \ldots, x^{\diamond_\alpha{}^{(n-1)}}(t)), \tag{6.10}$$

subject to the initial conditions $x(t_0) = x_0$, $x^{\diamond_\alpha}(t_0) = x_1$, $\ldots$, $x^{\diamond_\alpha{}^{(n-1)}}(t_0) = x_{n-1}$.

Before attempting to define truly general nth-order "mixed" derivative dynamic equations, we will define a general form for nth-order IVPs of uniform derivative (*i.e.* all delta, all nabla, or all diamond-alpha) since the results of interest in this paper depend on whether the choices of derivatives are uniform.

Definition 26 (nth-Order IVPs of Uniform Derivative)**.** Let $x(t)$ be an n-times differentiable real scalar or vector function on $t \in \mathbb{T}_{k_n}^{k^n}$ and let $x^{\square}(t)$ be either the delta, nabla, or diamond-alpha derivative of $x(t)$. Then, the general, nth-order initial value problem is

$$x^{\square^n}(t) = f(t, x(t), x^{\square}(t), \ldots, x^{\square^{(n-1)}}(t)), \tag{6.11}$$

subject to the initial conditions $x(t_0) = x_0$, $x^{\square}(t_0) = x_1$, $\ldots$, $x^{\square^{(n-1)}}(t_0) = x_{n-1}$.

For generality, we must define the function $x(t)$ for $t \in \mathbb{T}_{k_n}^{k^n}$. However, when $\square = \Delta$ or $\square = \nabla$, this condition can be relaxed, i.e. $t \in \mathbb{T}^{k^n}$ or $t \in \mathbb{T}_{k^n}$, respectively.

As with differential equations, higher-order dynamic equations can be written as systems of first-order equations using a change of variables as long as the choice of derivative is the same for all n derivatives, *i.e* we have either an nth-order delta, an nth-order nabla, or an nth-order diamond-alpha dynamic equation.

Example 4. Consider the delta dynamic equation

$$x^{\Delta^4}(t) = x^{\Delta\Delta}(t)\, e^t - (x^{\Delta}(t))^3. \tag{6.12}$$

We will assign four variables, $u_0(t), \ldots, u_3(t)$ since this is a fourth-order dynamic

equation:

$$\begin{aligned} u_0(t) &= x(t) \\ u_1(t) &= x^{\Delta}(t) \\ u_2(t) &= x^{\Delta^2}(t) \\ u_3(t) &= x^{\Delta^3}(t) \end{aligned} \tag{6.13}$$

Define a vector function $U(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \\ u_2(t) \\ u_3(t) \end{bmatrix}$, and then apply the delta derivative operator to the vector function $U(t)$:

$$U^{\Delta}(t) = \begin{bmatrix} u_0^{\Delta}(t) \\ u_1^{\Delta}(t) \\ u_2^{\Delta}(t) \\ u_3^{\Delta}(t) \end{bmatrix} = \begin{bmatrix} x^{\Delta}(t) \\ x^{\Delta^2}(t) \\ x^{\Delta^3}(t) \\ x^{\Delta^4}(t) \end{bmatrix} = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \\ u_2(t)e^t - (u_1(t))^3 \end{bmatrix}. \tag{6.14}$$

Thus, if

$$F(t, U(t)) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \\ u_2(t)e^t - (u_1(t))^3 \end{bmatrix}, \tag{6.15}$$

we can write (6.12) as a first-order, dynamic, vector equation, $U^{\Delta}(t) = F(t, U(t))$.

In Example 4, a fourth-order dynamic equation (6.13) was written as a first-order dynamic equation in $\mathbb{T}^4$, or, in other words, as a system of four, first order equations. In general, it is true that nth-order dynamic equations of uniform derivative can be written as first-order, vector equations on $\mathbb{T}^n$.

Definition 27 (Corresponding Systems of Equations)**.** A system of dynamic equations

$$U^{\square}(t) = F(t, U(t)) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ f(t, u_0(t), \ldots, u_{(n-1)}(t)) \end{bmatrix}$$

with initial condition $U(t_0) = U_0$ *corresponds* to the nth-order IVP

$$x^{\square^n}(t) = f(t, x(t),\, x^{\square}(t),\, x^{\square^2}(t),\, \ldots, x^{\square^{(n-1)}}(t)),$$

$x(t_0) = x_0,\, x^{\square}(t_0) = x_1,\, \ldots\, , x^{\square^{(n-1)}}(t_0) = x_{n-1}$ whenever

$$U(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \\ \vdots \\ u_{(n-1)}(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ x^{\square}(t) \\ \vdots \\ x^{\square^{(n-1)}}(t) \end{bmatrix}, \quad U_0 = \begin{bmatrix} u_0(t_0) \\ u_1(t_0) \\ \vdots \\ u_{(n-1)}(t_0) \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(n-1)} \end{bmatrix}, \tag{6.16}$$

and ${u_{(n-1)}}^{\square}(t) = f(t, u_0(t), u_1(t), \ldots, u_{(n-1)}(t))$.

Theorem 3 (Equivalence of Solutions)**.** Solving the nth-order initial value problem (6.11) is equivalent to solving the corresponding system of n, first-order equations.

Proof. By definition,

$$U(t)^{\square} = \begin{bmatrix} u_0^{\square}(t) \\ u_1^{\square}(t) \\ \vdots \\ u_{(n-1)}^{\square}(t) \end{bmatrix} = \begin{bmatrix} x^{\square}(t) \\ x^{\square^2}(t) \\ \vdots \\ x^{\square^n}(t) \end{bmatrix} = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ f(t, x(t), x^{\square}(t), \ldots, x^{\square^{(n-1)}}(t)) \end{bmatrix}.$$

Since $x^{\square^n}(t) = f(t, x(t), x^{\square}(t), \ldots, x^{\square^{(n-1)}}(t))$ and the initial conditions in definition 26 are met, and since all other equations in the system are true by design, (*e.g.* $u_0^{\square}(t) = x^{\square}(t) = u_1(t)$), solving

$$U^{\square}(t) = F(t, U(t)),\, U(t_0) = \begin{bmatrix} u_0(t_0) \\ u_1(t_0) \\ \vdots \\ u_{n-1}(t_0) \end{bmatrix}$$

is equivalent to solving the nth-order dynamic equation.

Second-order IVPs of uniform derivative are simply special cases of the general nth-order IVP of uniform derivative just described in definition (26). As such, they are indeed rewritable and equivalently solvable as systems of two equations. But when attempting to rewrite a mixed-derivative, second order equation as a system of lower-order variables, we run into trouble. Consider this example:

Example 5. Given a once nabla, once delta differentiable function $x(t)$, defined on $\mathbb{T}_k^k$ and the initial value problem

$$x^{\nabla\Delta}(t) = x(t); \quad x(t_0) = x_0, \tag{6.17}$$

where $t_0 \in \mathbb{T}$, we make substitutions for $x(t)$ and $x^{\nabla}(t)$:

$$U(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ x^{\nabla}(t) \end{bmatrix} \tag{6.18}$$

Differentiating yields

$$U^{\Delta}(t) = \begin{bmatrix} u_0^{\Delta}(t) \\ u_1^{\Delta}(t) \end{bmatrix} = \begin{bmatrix} x^{\Delta}(t) \\ x^{\nabla\Delta}(t) \end{bmatrix} = \begin{bmatrix} ? \\ u_0 \end{bmatrix} \tag{6.19}$$

We see that the problem lies in the fact that when the derivative operator is applied to the lower-order derivatives, they no longer correspond to the actual arguments of the original dynamic equation. In the example above, we needed $u_0^{\Delta}(t) = x^{\nabla}(t)$, but we cannot set up the system of n equations such that this will happen. Delta-differentiating the vector $U(t)$ such that the $x^{\nabla}(t)$ term gives $f(t, x(t), x^{\nabla}(t))$ does not yield lower-order terms found in the original IVP. Indeed, we find that delta differentiating $u_0(t)$ gives $x^{\Delta}(t)$, which is not present in the dynamic equation or in the lower-order variable scheme we introduced. The desired result holds for differential, difference, and dynamic equations of uniform derivative precisely because the derivative operator applied to obtain the dynamic equation equivalent is the only derivative operator used in all lower-order derivative arguments of the equation.

However, if we allowed ourselves to use more equations in the equivalent system, might we be able to write a vector equivalent for the original dynamic equation? The

resulting system would consist of more than n equations, some percentage of which would be extraneous to the final solution, creating additional computational costs if a Runge-Kutta-like scheme were employed. However, when attempting to rewrite the dynamic equation from the previous example in this way, we discover another problem:

Recall that in (6.19), we found that delta-differentiating $u_0(t)$ gave $x^{\Delta}(t)$, which was not present in the lower-order variable scheme. So, relaxing the usual protocol, we attempt to introduce a third lower-order variable, $x\Delta(t)$. However, upon differentiating,

$$U^{\Delta}(t) = \begin{bmatrix} u_0^{\Delta}(t) \\ u_1^{\Delta}(t) \\ u_2^{\Delta}(t) \end{bmatrix} = \begin{bmatrix} x^{\Delta}(t) \\ x^{\nabla\Delta}(t) \\ x^{\Delta\Delta}(t) \end{bmatrix} = \begin{bmatrix} u_2 \\ u_0 \\ ? \end{bmatrix},$$

we find that we have created another variable not present in our scheme. We might attempt to solve this problem by further introducing another variable, $u_3(t)$, but upon differentiating, we find ourselves trapped in an endless loop, creating an infinitely large system. Devising a system for rewriting dynamic equations of mixed derivatives is certainly not a trivial task, and in fact, might prove impossible. Without a uniform choice of derivative, we cannot rewrite dynamic equations as equivalent systems of n equations—at least not in the same, standard way. Whether this can be done in a non-standard way remains to be seen. We introduce the generalized, mixed, higher-order IVP on time scales to formalize this finding.

Definition 28 (nth-Order General Dynamic Equations of Non-uniform Derivative)**.** Let $x(t)$ be an n times continuously differentiable real scalar or vector function on $t \in \mathbb{T}_{k_n}^{k^n}$ and let $x^{\square_i}(t)$ be any one of the delta, nabla, or diamond-alpha derivatives. Then, the general, nth-order initial value problem is

$$x^{\square_1\square_2\ldots\square_n}(t) = f(t, x(t), x^{\square_1}(t), \ldots, x^{\square_1\square_2\ldots\square_{(n-1)}}(t)), \tag{6.20}$$

subject to the initial conditions $x(t_0) = x_0$, $x_1^{\square}(t_0) = x_1$, $\ldots$, $x^{\square_1\square_2\ldots\square_{(n-1)}}(t_0) = x_{n-1}$. Note that the choice of the derivative $\square_i$ need not be uniform for $i = 1, 2, 3, \ldots, n$.

In general, it is not the case that these generalized, higher-order dynamic equations can be rewritten as equivalent systems of n equations. However, for certain choices of time scale, we can in fact rewrite higher-order equations as equivalent systems for

the general initial value problem where the choice of derivatives need not be uniform. One such instance occurs when $\mathbb{T} = \mathbb{R}$.

Chapter 7

Numerical Solutions of ODEs

It's one thing to define ordinary differential equations and the initial value problems they inspire; however, it's quite another thing to find the solutions of these equations. In fact, in many instances, finding closed-form solutions to IVPs proves impossible. Usually, we must satisfy ourselves with approximate solutions, which are themselves difficult to generate. As LeVeque writes,"Our goal is to approximate solutions to differential equations, i.e. to find a function (or some discrete approximation to this function) which satisfies a given relationship between various of its derivatives on some given region of space and/or time ... In general this is a difficult problem" [LeV98]. Further information on these topics is available in [LeV98, Tre96, WyBa95]. Cleve Moler gives a particularly good introduction to Runge-Kutta methods in [Mol04].

7.1 Solving ODEs with Finite Difference Methods

Finite difference methods give us straight-forward protocol for approximating solutions for these difficult problems. Finite difference methods substitute finite difference formulas for the actual derivatives in ODEs so that ODEs can be algebraically manipulated to yield approximate solutions. LeVeque explains,"A finite difference method proceeds by replacing the derivatives in the differential equation by finite difference approximations, which gives a large algebraic system of equations to be solved in place of the differential equation" [LeV98]. A finite difference formula is a formula or "stencil" that generates an approximation of a derivative at some point t using a specified number of neighboring data values (and may include the data value at t).

If we let $x(t)$ be a function of one variable t that is sufficiently smooth to be differentiable several times and such that each derivative is a well-defined, bounded function over some interval that contains a particular point of interest, t_k, $k \in \mathbb{Z}$, we can approximate the derivative of x at t_k. In approximating such a derivative, perhaps the most immediate approach would be to find the slope of the secant line between t_k and the next point in the domain, $t_{k+1} = t_k + h$, for some real value of h for the function, $x(t)$, that we want to "differentiate." We obtain

$$x'(t_k) \approx \frac{x(t_k + h) - x(t_k)}{h}. \tag{7.1}$$

This approximation for the derivative is called *one-sided* since x is evaluated only at points "on one side of t_k", i.e. $t \geq t_k$. An analogous, one-sided approximation attempt simply finds the slope of the secant line that intersects $(t_k, x(t_k))$ and $(t_k - h, x(t_k - h))$

$$x'(t_k) \approx \frac{x(t_k) - x(t_k - h)}{h} \tag{7.2}$$

for some small h value not necessarily equal to h in (7.1). Note that (7.1) is the derivative approximation used in the finite difference method known as *Forward Euler* or *explicit Euler*, and (7.2) approximates the derivative in the *Backward Euler* or *implicit Euler* method.

Further, a third and more accurate attempt at approximating derivatives comes from averaging the two one-sided approximations to obtain a *centered approximation*:

$$x'(t_k) \approx \frac{(\frac{x(t_k+h)-x(t_k)}{h}) + (\frac{x(t_k)-x(t_k-h)}{h})}{2}. \tag{7.3}$$

Note that if the points in the domain are uniformly spaced, averaging yields

$$x'(t_k) \approx \frac{x(t_k + h) - x(t_k - h)}{2h}, \tag{7.4}$$

the derivative approximation used in the *Central Euler* finite difference method.

These formulae are the classic choices for approximating a first derivative of one variable. However, there are infinitely many ways to construct such a formula. To increase the accuracy of a certain finite difference formula, we might increase the

number of data points used to approximate the derivative. For instance, the four step Adams-Bashforth formula for approximating $x'(t_k)$ yields a fourth-order accurate approximation on a uniform grid:

$$x'(t_k) \approx \frac{1}{24}[9x(t_{k+1}) + 19x(t_k) - 5x(t_{k-1}) + x(t_{k-2})] \tag{7.5}$$

(refer to the appendix for information concerning the accuracy of an approximation). However, while increasing the number of data points used at each step of the approximation calculation does increase the accuracy of an approximation in theory, it also requires more known data values at start-up. When such data are not available, using more accurate methods can be less accurate in practice although methods do exist for "getting around" the problem of missing start-up values.

Notice also that we can construct finite difference formulae for higher order derivatives similarly. The standard centered difference approximation of the second derivative, which yields a second-order accurate approximation on a uniform grid, is

$$x''(t_k) \approx \frac{1}{h^2}[x(t_{k-1}) - 2x(t_k) + x(t_k)] \tag{7.6}$$

Once an appropriate finite difference formula is chosen for the IVP at hand, we employ the corresponding finite difference method to solve. Consider the next example.

Example 6. Suppose we would like to use the *Forward Euler* finite difference method to solve the first-order IVP

$$x'(t) = x(t),\ t \in [t_0, t_n],\ x(t_0) = x_0.$$

As it turns out, we can obtain the exact, closed-form solution for this particular example problem by integrating and solving for the unknown constant using initial value data. In fact, if we suppose that $t_0 = 0$ and $x_0 = 1$, the solution is $x(t) = e^t$, which we can use to compare to our approximate solution in the discussion that follows.

Since we cannot approximate this solution for every value in the interval (because there are infinitely many numbers in that interval), we must choose some values in the interval, often called a "grid" in numerical analysis terms, at which we can approximate solutions for an IVP. Wylie and Barrett write, "our objective is to obtain satisfactory

approximations to the values of the solution ... on a specified set of values" [WyBa95]. Here, we shall construct an evenly-spaced grid with step size h of $n+1$ points $t_0, t_0 + h, \ldots, t_0 + nh = t_n$.

To solve the IVP using the Forward Euler finite difference method, we substitute the one-sided, forward difference approximate (7.1) for $x'(t)$ in the ODE and use the value of t for which the solution is known, t_0:

$$\frac{x(t_0 + h) - x(t_0)}{h} \approx x(t_0). \tag{7.7}$$

Then, solving for $x(t_0 + h)$ yields

$$x(t_0 + h) = (1 + h)x(t_0). \tag{7.8}$$

Now that we have obtained a value for $x(t_0+h)$, we can march forward in time and calculate $x(t_0 + 2h)$ in the same way; then, we can use $x(t_0 + 2h)$ to find $x(t_0 + 3h)$, etc, until we have obtained values for all $n + 1$ points in our grid. Note that the more general form for solving the first-order IVP $x'(t) = f(t, x(t)),\ t \in [t_0, t_n],\ x(t_0) = x_0$, using Euler's method where $k \in \mathbb{Z}$ is

$$x(t_k + h) = x(t_k) + hf(t_k, x(t_k)). \tag{7.9}$$

For the specific example in question, letting $t \in [0, 2]$ and using a grid with step-size $h = 0.20$, we compare the results in Fig. 7.1 from Forward Euler with those of the actual solution using MATLAB 6.5.

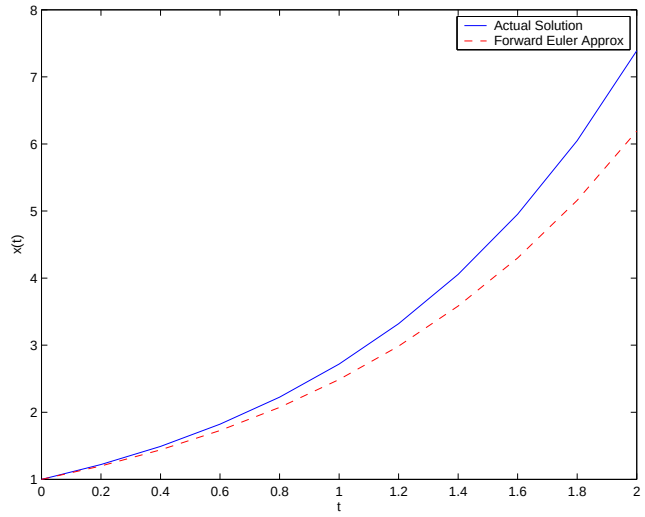

Figure 7.1: Exact Solution and Solution Approximated by Euler's Method for the IVP $x'(t) = x(t)$

The Forward Euler finite difference method is extremely useful for demonstration purposes, but as shown in the figure, gives relatively large error values. In fact, compared with modern differential equation solvers, Forward Euler is rather primitive. As we shall see in the next section, however, it is often used in intermediate steps of one of the best modern solution techniques, the family of Runge-Kutta methods.

7.2 Runge-Kutta Methods

In the world of DE solvers, finite difference methods are categorized as *multi-step methods*. They are so called because FD methods use function evaluations at one or more points in the grid to generate each point in the solution approximation. There are other families of solvers, however, that perform many calculations between grid points. These are called *single-step* and/or *multi-stage* methods. The family of Runge-Kutta methods belongs to this latter category.

The basic idea behind the Runge-Kutta methods is that we can obtain more accurate solution estimates if we use Euler's method several times between each grid point, t_k, and compute a weighted average of the results, rather than reaching further

behind or beyond the immediate grid point at hand. Thus, we might perform four separate function evaluations and a linear combination of their results to move from t_k to t_{k+1}. To find $x(t_k + h)$ using a finite difference method, we rewrite the relevant ODE substituting the appropriate finite difference formula (here, I'm using Forward Euler once again) and rearrange the terms of the equation to solve for $x(t_k + h)$:

$$\begin{aligned} \tfrac{x(t_0+h)-x(t_0)}{h} &\approx f(t_k, x_k) \\ \Rightarrow x(t_k + h) &\approx hf(t_k, x_k) + x_k \end{aligned} \tag{7.10}$$

To solve the ODE using a Runge-Kutta method, on the other hand, we will perform several calculations. The details of these calculations vary enormously, but what is typically considered the *classical Runge-Kutta method* follows.

Rather than assume the slope of the line tangent to the first grid point, t_k, is the slope of the tangent lines to all grid points throughout the interval $[t_k, t_{k+1}]$, classical Runge-Kutta makes several intermediate slope calculations denoted $s_1, s_2, \ldots$ (again, we will assume that h is the distance between two grid points, *i.e* $h = t_{k+1} - t_k$, though not necessarily uniform).

$$\begin{aligned} s_1 &= f(t_k, x_k), \\ s_2 &= f(t_k + \tfrac{h}{2}, x_k + \tfrac{h}{2}s_1), \\ s_3 &= f(t_k + \tfrac{h}{2}, x_k + \tfrac{h}{2}s_2), \\ s_4 &= f(t_k + h, x_k + hs_3). \end{aligned} \tag{7.11}$$

The actual step to the next grid point, t_{k+1} is taken via a linear combination of these "slope" values:

$$x(t_k + h) = x(t_k) + \frac{h}{6}(s_1 + 2s_2 + 2s_3 + s_4) \tag{7.12}$$

The classical (fourth-order) Runge Kutta method is used to solve the example problem given for finite difference methods (Example 6). A plot (Fig. 7.2) of the Runge-Kutta and Forward Euler solutions is given below (left); the exact solution is not shown here because, for this simple example, Runge-Kutta approximates the solution so closely that giving the exact solution makes seeing the RK solution difficult. An error plot (right) compares the error obtained using classical Runge-Kutta versus

the Forward Euler finite difference method.

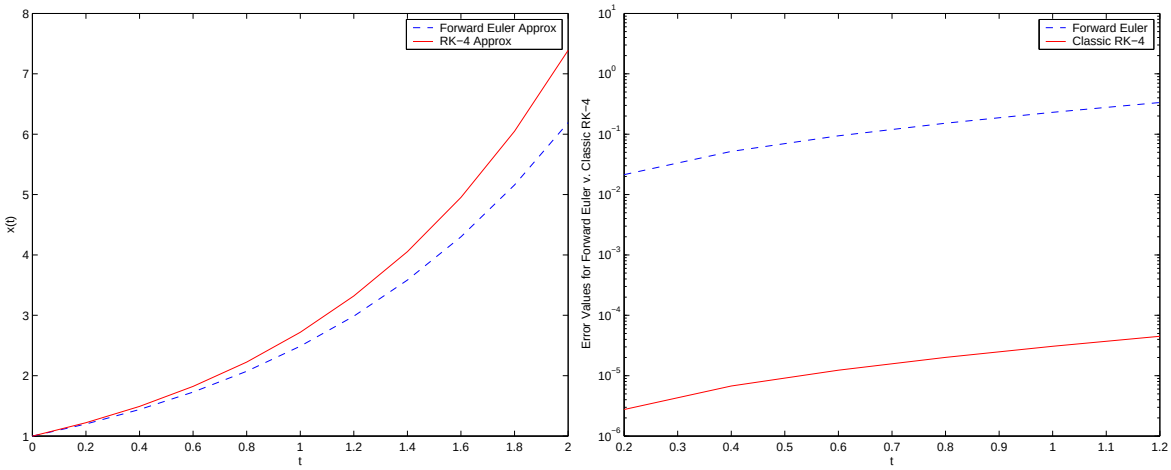

Figure 7.2: Left: Approx Solns of the IVP $x'(t) = x(t)$, $x_0 = 1$, Right: Error Plot

Classical Runge-Kutta is a four-stage method because it makes four intermediate slope calculations between steps. These steps need not be uniform. In fact, when computer software (in this case, MATLAB 6.5) employs a Runge-Kutta method to solve an ODE, the sizes of the intermediate steps and the coefficients used in the linear combination in (7.12) are not predetermined. Instead, these parameters depend on error estimates that are found by matching terms in a Taylor Series expansion of the slopes. The generalized R-K method of m stages, along with a generalized form for the error estimate, is given by a number of parameters: $\alpha_i, \beta_{i,j}, \gamma_i$, and δ_i.

$$s_i = f(t_k + \alpha_i h, x_k + h \sum_{j=1}^{i-1} \beta_{i,j}\, s_j), \quad i = 1, \ldots, m. \tag{7.13}$$

The proposal for the solution approximate at the next time-step is

$$x_{k+1} = x_k + h \sum_{i=1}^{k} \alpha_i\, s_i, \tag{7.14}$$

with an error estimate

$$e_{k+1} = h \sum_{i=1}^{k} \delta_i\, s_i. \tag{7.15}$$

If this error estimate falls below a specified tolerance, then the proposed solution approximate, x_{k+1}, is accepted; if not, then the proposed x_{k+1} is rejected, and another is generated with an h value modified by the results of the error estimate.

Once the parameters are chosen and implemented, the overall *order of accuracy* for the R-K method is given by the smallest power of the step-size h that cannot be "matched" in the Taylor Series approximate or "cancelled" if subtracting the s_i value from the Taylor expansion [Mol04]. Following the generalized procedure outlined above, the one, two, three, and four stage methods (often denoted RK-1, RK-2, etc.) give first, second, third, and fourth order accuracy, respectively. Including more stages does increase the accuracy though not at the same linear rate (*i.e.* RK-6 gives fifth order accuracy; RK-9 gives seventh order accuracy)[Tre96].

The MATLAB solver `ode45`, a four-stage RK method, uses error estimates (called *Dormand-Prince pairs*) from both fourth and fifth stage methods between steps to obtain a kind of fourth-fifth order hybrid Runge-Kutta method. `ode45` is what you might call the "bread-and-butter" of numerical differential equation solvers. In his book, *Numerical Computing in MATLAB*, Cleve Moler suggests, "in general, `ode45`, is the first function to try for most problems." Moler refers to `ode45` as "the workhorse of the MATLAB ordinary differential equations suite" [Mol04].

Below (Fig. 7.3), we use `ode45` to solve the IVP in Example 6. Notice that the plot also shows the exact solution of the IVP; and moreover, the `ode45` solution is almost indistinguishable. These results are given alongside an error plot that compares `ode45`, classic RK-4, and the Forward Euler finite difference method to the exact solution. We can see that while the classical RK-4 method gives excellent results, MATLAB's use of variable step size and Dormand-Prince pairs improves accuracy significantly.

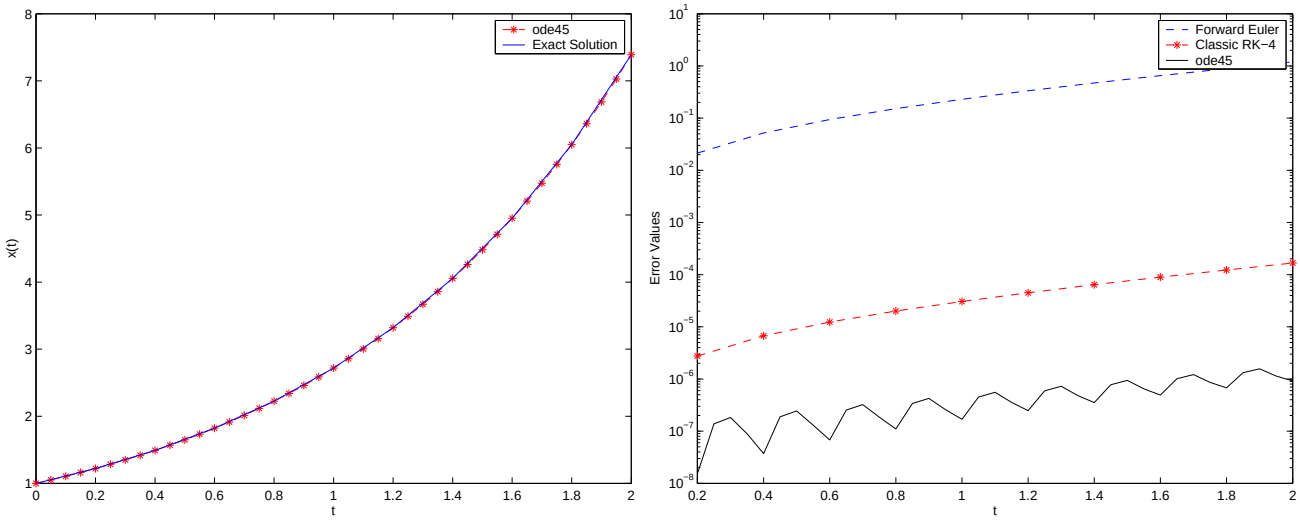

Figure 7.3: Right: Exact and Approx Solns of $x'(t) = x(t)$, $x_0 = 1$, Left: Error

The driving force behind this thesis problem (writing higher-order dynamic equations on time scales as systems of first-order DEs) is the fact that software implementations of the Runge-Kutta family of methods (including `ode45`) require that higher-order DEs be written as systems of first-order equations [WyBa95]. Before we can develop software for solving dynamic equations on time scales, we must determine whether such transformations are possible since the assumptions that underly commercial software applications may not hold for dynamic equations.

Chapter 8

Numerical Solutions of Dynamic Equations

For applied mathematicians, time scales calculus holds great appeal because of its potential to model natural phenomena more accurately (and perhaps more efficiently) that behave sometimes in a continuous manner and at others times, in a discrete manner. Also, from a computational standpoint, time scales calculus provides a formal, mathematically-sound framework in which analysts could reduce computational expense by constructing 'mixed domain' models. For instance, discrete domains with few points could be used when modeled behavior was regular and predictable and when making calculations for additional grid points in the domain might be unnecessary. On the other hand, continuous domains could be employed when modeled behaviors were erratic or difficult to predict. Certainly, these kinds of techniques have been employed before (especially for such things as the resolution of Runge phenomena); however, time scales calculus provides a formalized mathematical structure that could be used to justify such techniques analytically and to suggest optimum protocol.

Previously, most of the research in time scales calculus has been analytically-based. This approach seems logical since the pioneers of times scales calculus needed to establish the ground rules of the new discipline before applying them. Recently though, several mathematicians have been laying the groundwork for computing numerical solutions of dynamic equations on time scales [BaOt05, ESH03, ElSh04, RoSh06, SFHD06, She05, She06]. Also, our research group at Marshall University has used

time scales calculus theory to show that the solution of a first-order delta dynamic equation converges to the differential solution as the graininess function, $\mu(t)$, approaches zero [DHOv04a, LaOv06] and furthermore, that there is a homeomorphism that maps the graininess function of the dynamic equation $x^{\Delta}(t) = f(t, x(t))$ to the parameter space of the logistic equation [DHOv04b].

In the previous chapters, we have shown analytically that certain initial value problems for higher-order dynamic equations can be written as corresponding systems of equations and that solving these systems is equivalent to solving the original IVPs. This paper seeks also to show this result numerically. However, in undertaking this project, we have encountered some uncharted waters and have discovered that trying to "solve" a dynamic equation numerically is not necessarily a straight-forward procedure.

8.1 What does it mean to "solve" a dynamic equation numerically?

Given the ordinary differential equation $x'(t) = f(t, x)$, solving for $x(t)$ is equivalent to integrating $f(t, x)$ and thus to finding the area of the region under the curve of the graph that represents $f(t, x)$. Even if this cannot be done analytically, we can always draw rectangles of tinier and tinier widths under that curve, find their areas, and calculate the Riemann Sum, achieving a solution that becomes more and more accurate as the rectangles get thinner. For ODEs of higher-order, this physical solution becomes more obscure, but the objective is clear: finding numerical values of a function (perhaps unknown) whose nth instantaneous rate of change is $f(t, x, x', \ldots, x^{(n)})$.

Typically, we don't hear much about approximating the solutions of difference equations. After all, we use the solutions of difference equations as approximations for the solutions of differential equations. We describe one family of these procedures, *i.e.* finite difference methods, in the previous chapter. In fact, as long as a forward difference equation is explicitly solvable for its highest-order term, a numerical approximation of its solution amounts to recursion (solving for $x(t+1)$ using the value of $x(t)$ and solving for $x(t+2)$ using $x(t+1)$, etc.). Thus, we can solve difference equations numerically at machine accuracy as long as they are explicitly solvable. However, when difference equations are not explicitly solvable for the highest-order terms, *i.e.*

when they are non-linear, techniques such as *linearization* and *quasilinearization* are employed.

Despite differences in the details for approximating the solutions of differential and difference equations, the overall goal of the various procedures is straightforward, as are the physical representations of the solutions. Also, for those who want to use time scales derivatives and grids to obtain more accurate solutions of differential equations, the goal is also clear—to approximate the DE solution more closely, using the flexibility of time scales grids and derivatives to cause the solution of the dynamic equation to emulate the solution of the ODE. But, what about the dynamic equations themselves? How do we calculate or approximate their solutions? And what do these solutions represent physically?

Previous research has developed many tools for solving linear dynamic equations analytically [BoPe01, BoPe03]. Recent numerical research on dynamic equations has demonstrated successful techniques for approximating solutions of ordinary differential equations using time scales derivatives and grids [She05, SFHD06, ElSh04, ESH03]. Solutions of non-linear dynamic equations been solved using a method of *quasilinearization* in which non-linear equations are approximated by equations more linear in form and more easily solved. These methods provide monotonically convergent lower and upper solution sequences [ESH03, ElSh04].

8.2 tsSolver

Two students at Baylor University, Brian Ballard and Bumni Otegbade, have written a time scales toolbox for MATLAB that includes a delta dynamic equation solver called *tsode* [BaOt05]. In the manual that accompanies the toolbox, the authors explain that their solution protocols differ for continuous and discrete time scales and for continuous and discrete intervals within mixed time scales. On discrete time scales and on discrete intervals within mixed time scales, *tsode* solves for $x(\sigma(t))$ (called $y(t+1)$ in the manual) recursively using (3.1). For continuous time scales and on continuous intervals, *tsode* solves using MATLAB's built-in differential equation solver, `ode23`. Ballard and Bumni's protocol inspired our procedure for approximating solutions of first-order and second-order delta, nabla, and diamond-alpha dynamic equations on

time scales.

In an attempt to create such a protocol for approximating solutions of dynamic equations on time scales, we developed a GUI (Graphical User Interface) in MATLAB 6.5. The GUI, called *tsSolver*, displays time scales that users choose from a drop-down menu and solutions of dynamic equations (that users also choose from a menu) on the chosen time scale. The GUI follows the protocol described below.

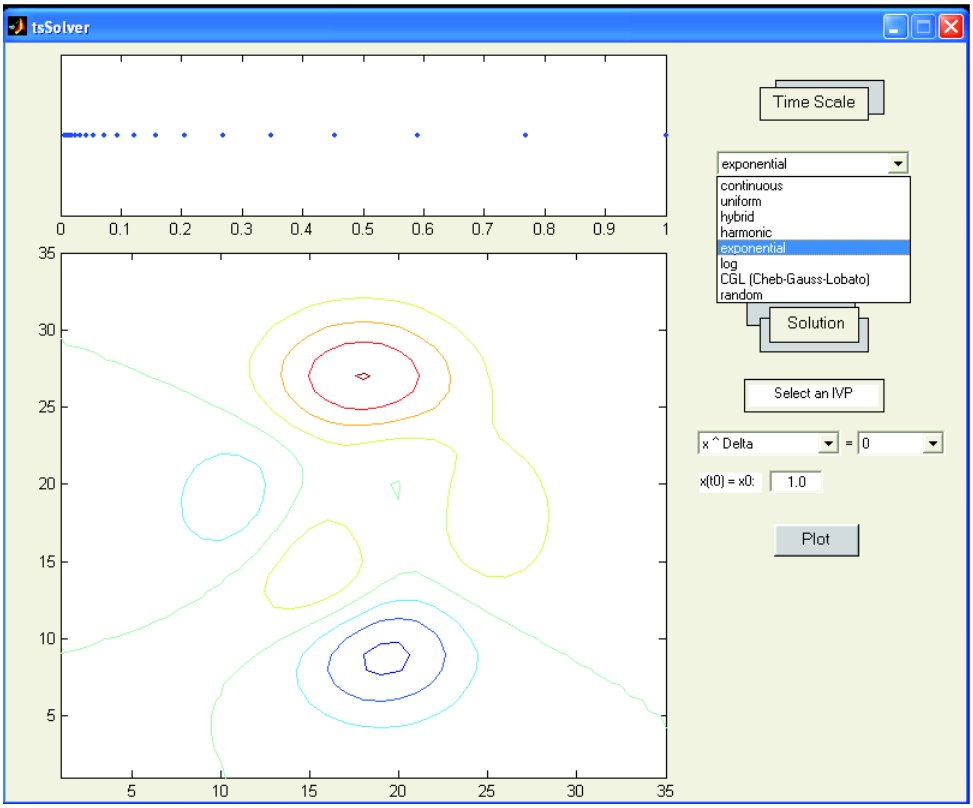

Figure 8.1: A screen shot from *tsSolver* that demonstrates the time scales available in the drop-down menu.

On an isolated time scale, *i.e.* $\sigma(t) > t > \rho(t)$, for first and second order delta IVPs, our approach is similar to the difference equation strategy described above and similar to *tsode*'s recursion strategy—solve the dynamic equation explicitly for the "highest-order" version of the solution x in the dynamic equation. Then, generate the solution values for each grid point on which the solution is defined by repeated recursion. Consider the following example.

Example 7. Let $x^{\Delta}(t) = -x(t); \quad x(t_0) = 1; \quad \mathbb{T} = \{0,\ 0.2,\ 0.4,\ 0.6,\ 0.8,\ 1.0\}$. Recall from chapter 3, the computational formula (3.1) for $x^{\Delta}(t)$ when $\mathbb{T} = \mathbb{T}_I$:

$$x^{\Delta}(t) = \frac{x(\sigma(t)) - x(t)}{\mu(t)}$$

Solving for the"highest-order" term $x(\sigma(t))$ gives

$$x(\sigma(t)) = x(t) + \mu(t)x^{\Delta}(t). \tag{8.1}$$

Applying this computational formula to the IVP on the given time scale gives us a numerical value of the solution $t = 0.2$:

$$x(0.2) = x(0) + \mu(0)(-x(0)) = 1.0 + 0.2(-1.0) = 0.8.$$

We repeat the recursion, obtaining $x(0.4)$,

$$x(0.4) = 0.8 + 0.2(-0.8) = 0.64.$$

and continue this process to obtain the complete solution vector $x(t)$ for $t \in \mathbb{T}^k$

$$x(t) = \begin{bmatrix} 1.0 \\ 0.8 \\ 0.64 \\ 0.5120 \\ 0.4096 \end{bmatrix}. \tag{8.2}$$

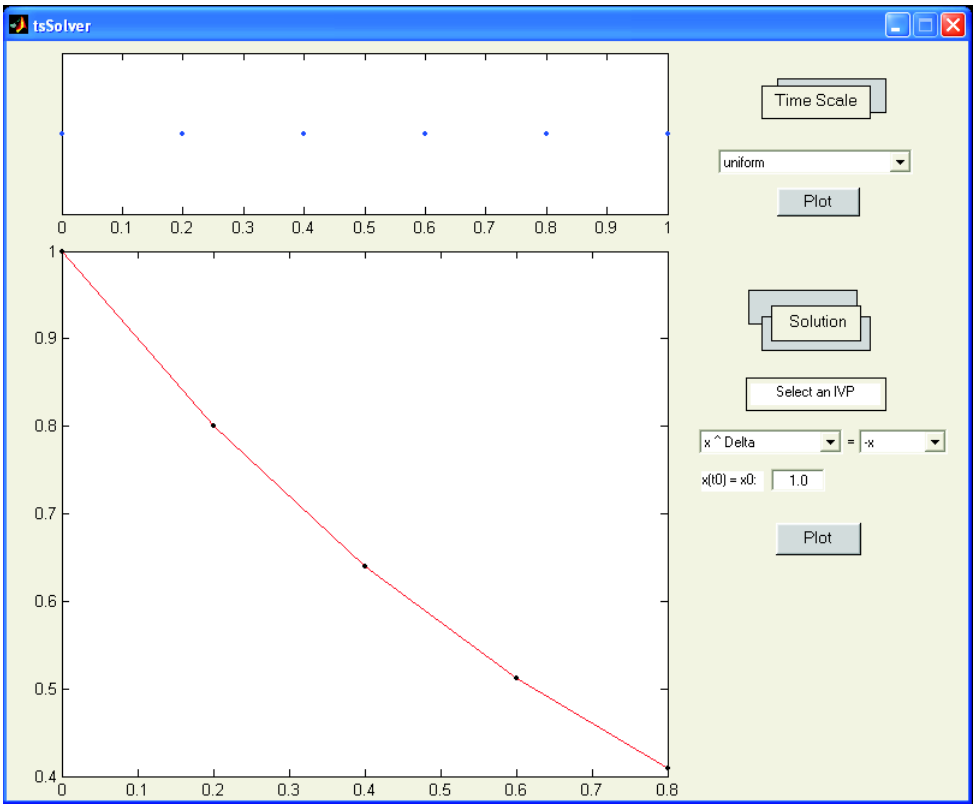

Figure 8.2: A screen shot from *tsSolver*'s solution of the given delta dynamic equation (below) for the given $\mathbb{T}$ (above).

Notice that unlike approximating solutions of differential equations with finite difference methods, on discrete time scale intervals, this method of solving first-order delta and nabla IVPs is accurate to machine epsilon since the derivative actually is defined as the computational formula given for isolated time scales.

But, what if instead, the time scale at hand was a combination of discrete and continuous intervals? Just as *tsode* uses `ode23` to solve delta equations on continuous intervals, *tsSolver* uses MATLAB's built-in `ode45` (discussed in the previous chapter) since `ode45`'s approximations offer more accuracy than `ode23` for non-stiff problems.

These procedures seem fairly straightforward and in fact, are. However, we meet a bit of difficulty (familiar to numerical analysts and engineers who use finite difference methods to solve differential equations) when we attempt to solve diamond-alpha dynamic equations in this way. For an isolated time scale or an isolated interval within a mixed time scale, the diamond-alpha computational formula (3.3) with $\alpha = 0.5$ is equivalent to a weighted version of the central Euler formula of chapter seven. When solving such a diamond-alpha dynamic equation using repeated recurrences or when solving a differential equation with a central Euler formula, we run into the problem of not having enough initial data at the beginning of the recurrence process. In the following example, we see that we need to calculate $x(\sigma(t))$ from $x(t)$ and $(x(\rho(t))$, but have only $x(\rho(t))$.

Example 8.

$$x^{\diamond_\alpha}(t) = t; \quad x(0) = 0; \quad \alpha = 0.5; \quad \mathbb{T} = \{0,\ 0.05,\ 0.10, \ldots,\ 0.25,\ 0.30\} \tag{8.3}$$

(3.3) gives us that

$$\alpha x^{\Delta}(t) + (1-\alpha)x^{\nabla}(t) = t.$$

Further rearrangement and substitution yields

$$x(\sigma(t)) = \frac{\mu(t)\nu(t)t - 0.5\mu(t)(x(t) - x(\rho(t))) + 0.5\nu(t)x(t)}{0.5\nu(t)}$$

In this case, $t_0 = 0$ and thus, $x(\rho(t)) = 0$, but we need to solve for $x(\sigma(t)) = x(0.10)$ and lack the necessary $x(t) = x(0.05)$ data value.

Numerical analysts and engineers alike have developed schemes, such as choosing

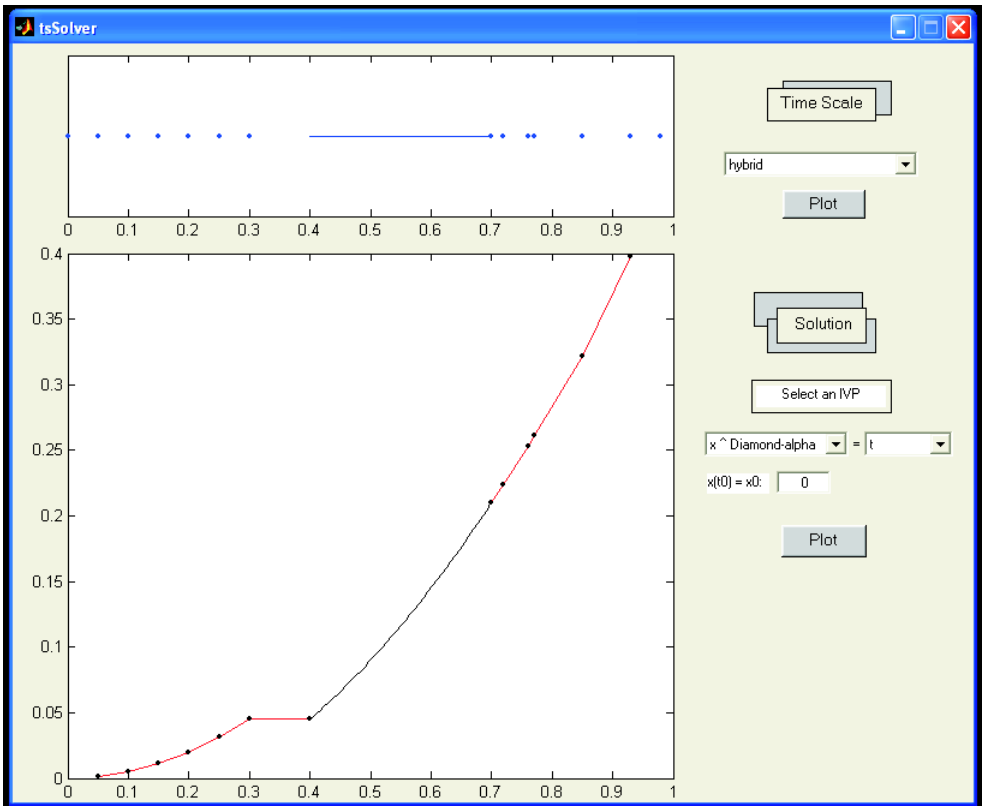

Figure 8.3: *tsSolver* plots the solution of $x^{\diamond_\alpha}(t) = t;\ x(0) = 0;\ \alpha = 0.5$ on the given time scale.

"ghost points" or using circular boundaries for BVPs. *tsSolver* employs another of these techniques: using classical RK4 to advance from the known data value to the unknown value using sufficiently small time steps (not included in the time scale) between the two points. Thus, in the case of example 8, *tsSolver* calculates data values at several time steps between $t = 0$ and $t = 0.1$ before determining an appropriate start-up value for $x(0.1)$. The figure that follows shows *tsSolver*'s solution for this dynamic equation on $\mathbb{T}_k^k$ for the hybrid, discrete-continuous-discrete time scale
$\mathbb{T} = \{0,\ 0.05,\ 0.10, \ldots,\ 0.25,\ 0.30\} \cup [0.4, 0.7] \cup \{0.71,\ 0.72,\ 0.76,\ 0.77,\ 0.85,\ 0.93,\ 0.98\}$.

Note that the problem of requiring additional data values does not plague us only in the case of the diamond-alpha equations but instead for all higher-order dynamic equations. This problem presents a significant obstacle in approximating solutions of higher-order dynamic equations accurately using these procedures and makes the need to develop some sort of Runge-Kutta-like protocol more pressing. The author hopes that since the present study has developed criteria for writing higher-order dynamic IVPs as systems of equations, future work will offer such numerical methods.

Appendix A

Determining Orders of Accuracy

The standard method for determining the order of accuracy of a certain finite difference formula (and the consequent finite difference method) is to obtain an expression for the difference formula by expanding the function evaluations, $x(t_k + h), x(t_k - h), x(t_k - 2h)$, etc. in Taylor Series and then subtracting $x'(t)$ from the expanded expression. Often, many terms will (or at least should, for an accurate method) drop out due to cancellation, leaving only constant terms multiplied by powers of h, the step-size in our domain, or grid. The least power of h, typically called p appearing in the expression is called the formula's *order of accuracy*. A finite difference method derived from this derivative approximation is said to be *pth order accurate*. Let us use the one-sided forward difference formula as an example.

We begin by expanding $x(t_k + h)$ in a Taylor Series about the point t_k:

$$x(t_k + h) = x(t_k) + hx'(t_k) + \frac{h^2}{2}x''(t_k) + \frac{h^3}{6}x'''(t_k) + \mathcal{O}(h^4) \tag{A.1}$$

Making this substitution in (7.1) yields

$$x(t_k + h) = \frac{x(t_k) + hx'(t_k) + \frac{h^2}{2}x''(t_k) + \frac{h^3}{6}x'''(t_k) + \mathcal{O}(h^4) - x(t_k)}{h} \tag{A.2}$$

(see discussion of $\mathcal{O}(h^p)$ below)

Note heavy cancellation as (A.2) reduces nicely to (A.3):

$$x(t_k + h) = x'(t_k) + \frac{h}{2}x''(t_k) + \frac{h^2}{6}x'''(t_k) + \mathcal{O}(h^3) \tag{A.3}$$

Since this expression should approximate $x'(t_k)$, we subtract $x'(t_k)$ to obtain the error the FD formula generates, leaving the error as a function of h,

$$E(h) = \frac{h}{2}x''(t_k) + \frac{h^2}{6}x'''(t_k) + \mathcal{O}(h^3). \tag{A.4}$$

Thus, $E(h) \to 0$ at least as quickly as h does. Then, (7.1) gives a first-order accurate approximation of $x'(t_k)$. As a result, in the next section we shall see that the Forward Euler finite difference method approximates the solution of the corresponding ODE with first-order accuracy.

A.1 "Big-Oh" Notation, $\mathcal{O}(h^p)$

Let f and g be two functions of h. Then,

$$f(h) = \mathcal{O}(g(h)) \text{ as } h \to 0 \tag{A.5}$$

implies that there is some constant C such that

$$\left|\frac{f(h)}{g(h)}\right| < C \tag{A.6}$$

for all h sufficiently small [Lev98]. Equivalently, if $|g(h)|$ is bounded, then

$$|f(h)| < C\,|g(h)| \tag{A.7}$$

for all h sufficiently small. In other words, (A.5) tells us that $f(h)$ goes to 0 *at least as fast* as the function $g(h)$ does. Thus, when we speak of a FD method converging to a solution with pth-order accuracy, this is equivalent to writing the error as a function of h using big-oh notation

$$E(h) = \mathcal{O}(h^p), \tag{A.8}$$

meaning that as $h \to 0$, the error diminishes as least as quickly as h^p does. Thus, the higher the order of accuracy, the more quickly the error goes to 0 as the step-size h decreases.

Appendix B

MATLAB Programming Code

B.1 Figure Generators

The following MATLAB 6.5 code was used to generate figures in the manuscript.

B.1.1 `forwardEuler.m`, Fig. 7.1

```
function forwardEuler(h)
t = 0:h:2;
t = t';

%computes soln of IVP x' = x using Forward Euler
xEul(1) = 1;

for k = 1:length(t) - 1
xEul(k + 1) = (1 + h)*xEul(k);
end
xEul = xEul';

%computes actual soln.
x = exp(t);

%plots actual soln and Euler solution in same fig for comparison
figure('Color', 'white')
```

```
h = plot(t, x, 'b', t, xEul, 'r--')
legend('Actual Solution', 'Forward Euler Approx')
```

B.1.2 classicRK4.m, Fig. 7.2

```
function y = classicRK4(h);
% applies classic 4th order RK method to IVP x'=x, x0 = 1;

t = (0:h:2)'; tLength = length(t);

xRK(1) = 1;

for m = 1:tLength - 1
s1 = xRK(m);
s2 = xRK(m) + 0.5*h*s1;
s3 =xRK(m) + 0.5*h*s2;
s4 = xRK(m) + h*s3;
xRK(m + 1) = xRK(m) + h/6*(s1 + 2*s2 + 2*s3 + s4);
end

xRK = xRK';

%computes actual soln.
x = exp(t);

%computes soln of IVP x' = x using Forward Euler
xEul(1) = 1;

for k = 1:tLength - 1;
xEul(k + 1) = (1 + h)*xEul(k);
end

xEul = xEul';
```

```
%plots actual soln, Euler soln, and RK soln in same fig for comparison
figure('Color', 'white')
h = plot(t, xEul, 'b--', t, xRK, 'r');
legend('Forward Euler Approx', 'RK-4 Approx')
xlabel('t')
ylabel('x(t)')

%gives error plot for fEul v. RK4
figure('Color', 'white')

errRK = abs(xRK - x);
errFEul = abs(xEul -x);
semilogy(t, errFEul, 'b--', t, errRK, 'r')
xlim([0.2:2])
legend('Forward Euler', 'Classic RK-4')
xlabel('t')
ylabel('Error Values')
```

B.1.3 ode45Demo.m, Fig. 7.3

```
function y = ode45Demo(h)
% applies classic 4th order RK method to IVP x'=x, x0 = 1 and compares it to
%ode45 solution and Forward Euler

t = (0:h:2)';
tLength = length(t);
xRK(1) = 1;

for m = 1:tLength - 1
s1 = xRK(m);
s2 = xRK(m) + 0.5*h*s1;
s3 = xRK(m) + 0.5*h*s2;
```

```
s4 = xRK(m) + h*s3;
xRK(m + 1) = xRK(m) + h/6*(s1 + 2*s2 + 2*s3 + s4);
end

xRK = xRK';

%computes soln of IVP x' = x using Forward Euler
xEul(1) = 1;

for j = 1:length(t) - 1
xEul(j + 1) = (1 + h)*xEul(j);
end

xEul = xEul';

%computes actual soln.
x = exp(t);

%plots exact soln in same fig with ode45 for comparison
figure('Color', 'white')
hold on
F = inline('k','r','k');

[r,k] = ode45(F, [0 2], 1);
plot(r, k, 'r--', t, x, 'b', t, xRK, 'k-*', t, xEul, 'g.');
legend('ode45', 'Exact Solution')
axis on
xlabel('t')
ylabel('x(t)')
hold off

%computes actual soln. on ode45 grid
```

```
x45 = exp(r);


%gives error plot for ForEul v. ode45 v. RK4
errFEUL = abs(xEul - x);
errRK = abs(xRK - x);
err45 = abs(k - x45);
semilogy(t, errFEUL, 'b--', t, errRK, 'r--*', r, err45, 'k')
xlim([0.2 2.0])
legend('Forward Euler', 'Classic RK-4', 'ode45')
```

B.1.4 `tsSolver.m`, Fig. 8.1, Fig. 8.2, Fig. 8.3

cf. the following section for the programming code that produced the `tsSolver` figures.

B.2 Programming Code for `tsSolver`

B.2.1 `tsSolver.m`

```
function varargout = tsSolver(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1; gui_State = struct('gui_Name',       mfilename,
...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @tsSolver_OpeningFcn, ...
                   'gui_OutputFcn',  @tsSolver_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end


if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
```

```
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

function tsSolver_OpeningFcn(hObject, eventdata, hand, varargin)

% Create the data to plot.
hand.peaks=peaks(35); hand.membrane=membrane; [x,y] =
meshgrid(-8:.5:8); r = sqrt(x.^2+y.^2) + eps; sinc = sin(r)./r;
hand.sinc = sinc;

% Set the current data value.
hand.current_data = hand.peaks; contour(hand.current_data)

% Choose default command line output for tsSolver
hand.output = hObject; hand.type = 'continuous'; hand.deriv = 'x ^
delta'; hand.eq = '0'; hand.x0 = 1.0;
%hand.udf = '0';
x0 = hand.x0; ts = getTS(hand.type); x =
solveIt(hand.type,hand.deriv,hand.eq,hand.x0);

% Update handles structure
guidata(hObject, hand); hold off

% UIWAIT makes tsSolver wait for user response (see UIRESUME)
% uiwait(hand.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = tsSolver_OutputFcn(hObject, eventdata, hand);
varargout{1} = hand.output;

%CREATES
```

```
function chooseTS_CreateFcn(hObject, eventdata, hand) function
chooseEQ_CreateFcn(hObject, eventdata, hand) function
chooseDeriv_CreateFcn(hObject, eventdata, hand) function
axesTS_CreateFcn(hObject, eventdata, hand) function
axesSol_CreateFcn(hObject, eventdata, hand) function
x0_CreateFcn(hObject, eventdata, hand)

%CALLBACKS
%POP-UP MENUS
function chooseTS_Callback(hObject, eventdata, hand) stringTS =
get(hObject,'String'); hand.type =
lower(char(stringTS(get(hObject, 'Value')))); guidata(hObject,
hand);

function chooseDeriv_Callback(hObject, eventdata, hand)
stringDeriv = get(hObject, 'String'); hand.deriv =
lower(char(stringDeriv(get(hObject, 'Value')))); guidata(hObject,
hand);

function chooseEQ_Callback(hObject, eventdata, hand) stringEQ =
get(hObject,'String'); hand.eq = lower(char(stringEQ(get(hObject,
'Value')))); guidata(hObject, hand);

%PUSHBUTTON CALLBACKS
function plotTS_Callback(hObject, eventdata, hand) ts =
getTS(hand.type); len = length(ts); t = zeros(len);
axes(hand.axesTS) if isequal(hand.type, 'continuous')
    plot(ts, t, 'b')
    set(hand.axesTS,'YTick', [])
    hold off
elseif isequal(hand.type, 'hybrid')
    tdi = 0:.05:0.3;
```

```
    xdi = zeros(1,length(tdi));
    tci = 0.4:0.01:0.7;
    xci = zeros(1,length(tci));
    td2 = [0.7000 0.7200 0.7600 0.7700 0.8500 0.9300 0.9800];
    xd2 = zeros(1,length(td2));
    plot(tdi, xdi, '.')
    hold on
    plot(tci, xci, 'b')
    hold on
    plot(td2,xd2, '.')
    set(hand.axesTS,'YTick', [])
    hold off
else
    plot(ts,t,'b.')
    set(hand.axesTS,'YTick', [])
    hold off
end

function plotSol_Callback(hObject, eventdata, hand)
axes(hand.axesSol) if isequal(hand.type, 'continuous')
    [ts, x] = solveIt(hand.type, hand.deriv, hand.eq, hand.x0);
    plot(ts, x, 'r')
elseif isequal(hand.type, 'hybrid')
    [tdi,xdi,tci,xci,td2,xd2,q,q2] = hybrid(hand.deriv, hand.eq, hand.x0);
    plot(tdi(1:q + 1), xdi(1:q + 1), 'r')
    hold on
    plot(tdi(1:q + 1), xdi(1:q + 1), 'k.')
    hold on
    plot(tci, xci, 'k')
    hold on
    plot(td2(1:q2),xd2(1:q2), 'r')
    hold on
```

```
    plot(td2(1:q2),xd2(1:q2), 'k.')
else
    [ts, x] = solveIt(hand.type, hand.deriv, hand.eq, hand.x0);
    plot(ts, x, 'r-')
    hold on
    plot(ts, x, 'k.')
end hold off

%EDIT TEXT BOXES
function x0_Callback(hObject, eventdata, hand) hand.x0 =
str2double(get(hObject, 'String')); guidata(hObject, hand);
```

B.2.2 getTS.m

```
function ts = getTS(type)

switch type
    case 'continuous'
        a = 0;
        b = 1;
        ts = a:0.01:b;
    case 'uniform'
        ts = linspace(0,1,20);
    case 'hybrid'
        tdi = 0:.05:0.3;
        tci = linspace(0.4,0.7,7);
        td2 = [0.7100 0.7200 0.7600 0.7700 0.8500 0.9300 0.9800];
        a = length(tdi); b = length(tci); c = length(td2); d= a+b; e = a+b+c;
        ts = zeros(1,e);
        ts(1:a)= tdi; ts(a+1:d)=tci; ts(d+1:e) = td2;
        ts = ts;
    case 'harmonic'
        t = 1:100;
```

```
        for j = 1:100;
        ts(j) = 1 - 1/t(j);
        end
        ts(100) = 1;
    case 'exponential'
        t = linspace(0,5,20);
        ts = exp(t);
        val = norm(ts, inf);
        ts = ts./val;
    case 'log'
        t = linspace(0.01,exp(1),20);
        ts = log(t);
    case 'cgl (cheb-gauss-lobato)'
        t = linspace(0,pi,20);
        ts = cos(t);
    case 'random'
        t = randn(1,20);
        ts = sort(t);
        maxTS = max(abs(ts));
        ts = ts./maxTS;
end
```

B.2.3 SolveIt.m

```
function [ts, x] = solveIt(TS, Deriv, eqn, x0)

ts = getTS(TS)'; n = length(ts);

if isequal(TS, 'continuous')
    a = ts(1); b = ts(n);
    F = inline(eqn,'t','x');
    [ts, x] = ode45(F, [a, b], x0);
```

```
else
%calculates missing data point for appropriate DyEqns using classic RK4
h = (ts(2)-ts(1))/24; tTemp = ts(1):h:ts(2); xRK(1) = x0; for m =
1:24
        s1 = equation(eqn, tTemp(m), xRK(m));
        s2 = equation(eqn, tTemp(m), xRK(m)) + 0.5*h*s1;
        s3 = equation(eqn, tTemp(m), xRK(m)) + 0.5*h*s2;
        s4 = equation(eqn, tTemp(m), xRK(m)) + h*s3;
        xRK(m + 1) = xRK(m) + h/6*(s1 + 2*s2 + 2*s3 + s4);
end
%calculates a second missing data point for appropriate DyEqns using classic RK4
h = (ts(3)-ts(2))/24; tTemp = ts(2):h:ts(3); xRK2(1) = xRK(25);
for m = 1:24
        s1 = equation(eqn, tTemp(m), xRK2(m));
        s2 = equation(eqn, tTemp(m), xRK2(m)) + 0.5*h*s1;
        s3 = equation(eqn, tTemp(m), xRK2(m)) + 0.5*h*s2;
        s4 = equation(eqn, tTemp(m), xRK2(m)) + h*s3;
        xRK2(m + 1) = xRK2(m) + h/6*(s1 + 2*s2 + 2*s3 + s4);
end

%calculates missing data point in backwards direction using a modified RK4
h = (ts(2)-ts(1))/24; tTemp = ts(1):h:ts(2); xRKb(1) = x0;

for m =1:24
        s1 = equation(eqn, tTemp(m), xRKb(m));
        s2 = equation(eqn, tTemp(m), xRKb(m)) - 0.5*h*s1;
        s3 = equation(eqn, tTemp(m), xRKb(m)) - 0.5*h*s2;
        s4 = equation(eqn, tTemp(m), xRKb(m)) - h*s3;
        xRKb(m + 1) = xRKb(m) - h/6*(s1 + 2*s2 + 2*s3 + s4);
end

t1 = ts(1:n-2); t2 = ts(2:n-1); t3 = ts(1:n-1); t4 = ts(2:n); mu =
```

```
t2 - t1; nu = t4 - t3;


switch Deriv
    case 'x ^ delta'
        x(1) = x0;
        for j = 1:n-2
           x(j + 1) = x(j) + mu(j)*equation(eqn, ts(j), x(j));
       end
       ts = ts(1:n-1);
       x = x';
    case 'x ^ nabla'
       x(n) = x0;
       for j = 1:n-1
       x(n - j) = x(n - j + 1) - nu(n - j)*equation(eqn, ts(n - j + 1), x(n - j + 1));
       end
       ts = ts(2:n);
       x = x(2:n)';
    case 'x ^ diamond-alpha'
       alpha = 0.5;
       x(1) = x0; x(2) = xRK(25);
       for j = 1:n-3
       x(j + 2) = (1/(alpha*nu(j +1)))*(equation(eqn,ts(j+1),x(j+1))*mu(j+1)*nu(j+1)
       - mu(j + 1)*(1 - alpha)*(x(j + 1) - x(j)) + nu(j + 1)*alpha*x(j + 1));
       end
       ts = ts(2:n-1)
       x = x(2:n-1)'
    case 'x ^ delta delta'
        x(1) = x0; x(2) = xRK(25);
        for j = 1:n-4;
        x(j + 2)= (1/mu(j))*(mu(j)*x(j + 1) + mu(j + 1)*x(j + 1) - mu(j + 1)*x(j) +
        equation(eqn, ts(j), x(j))*mu(j)^2*mu(j+1));
        end
```

```
        ts = ts(1:n-2);
        x = x';
    case 'x ^ nabla nabla'
       x(n) = x0; x(n - 1) = xRKb(25);
       for j = 1:n-2
       x(n - j - 1) = (1/nu(n-j))*(equation(eqn,ts(n-j+1),x(n-j+1))*nu(n-j)^2*nu(n-j-1)
       - nu(n-j-1)*(x(n-j+1)-x(n-j)) + nu(n-j)*x(n-j));
       end
       ts = ts(2:n);
       x = x(2:n)';
       end
end
```

B.2.4 equation.m

```
function eq = equation(eqn, ts, x)

a = eqn; b = ts; c = x;

switch eqn
    case '0'
        f = 0;
    case 't'
        f = ts;
    case 'x'
        f = x;
    case '-x'
        f = -1*x;
    case 'x^2'
        f = x.^2;
    case 'cos(t)'
        f = cos(ts);
end eq = f;
```

B.2.5 hybrid.m

```
function [tdi,xdi,tci,xci,td2,xd2,q,q2] = hybrid(Deriv,eqn,x0)

tdi = 0:.05:0.3; tci = linspace(0.4,0.7,7); td2 = [0.7000 0.7200
0.7600 0.7700 0.8500 0.9300 0.9800]; n = length(tdi);

%calculates missing data point for appropriate DyEqns using classic RK4
h = (tdi(2)-tdi(1))/24; tTemp = tdi(1):h:tdi(2); xRK(1) = x0; for
m = 1:24
        s1 = equation(eqn, tTemp(m), xRK(m));
        s2 = equation(eqn, tTemp(m), xRK(m)) + 0.5*h*s1;
        s3 = equation(eqn, tTemp(m), xRK(m)) + 0.5*h*s2;
        s4 = equation(eqn, tTemp(m), xRK(m)) + h*s3;
        xRK(m + 1) = xRK(m) + h/6*(s1 + 2*s2 + 2*s3 + s4);
end
%calculates a second missing data point for appropriate DyEqns using classic RK4
h = (tdi(3)-tdi(2))/24; tTemp = tdi(2):h:tdi(3); xRK2(1) =
xRK(25); for m = 1:24
        s1 = equation(eqn, tTemp(m), xRK2(m));
        s2 = equation(eqn, tTemp(m), xRK2(m)) + 0.5*h*s1;
        s3 = equation(eqn, tTemp(m), xRK2(m)) + 0.5*h*s2;
        s4 = equation(eqn, tTemp(m), xRK2(m)) + h*s3;
        xRK2(m + 1) = xRK2(m) + h/6*(s1 + 2*s2 + 2*s3 + s4);
end

%calculates missing data point in backwards direction using a modified RK4
h = (tdi(2)-tdi(1))/24; tTemp = tdi(1):h:tdi(2); xRKb(1) = x0;

for m = 1:24
        s1 = equation(eqn, tTemp(m), xRKb(m));
        s2 = equation(eqn, tTemp(m), xRKb(m)) - 0.5*h*s1;
```

```
        s3 = equation(eqn, tTemp(m), xRKb(m)) - 0.5*h*s2;
        s4 = equation(eqn, tTemp(m), xRKb(m)) - h*s3;
        xRKb(m + 1) = xRKb(m) - h/6*(s1 + 2*s2 + 2*s3 + s4);
end

t1 = tdi(1:n-1); t2 = tdi(2:n); t3 = zeros(1, n+1); t4 = t3;
t3(2:n+1) = tdi; t4(1:n) = tdi;
mu = t2 - t1; nu = t4 - t3;

t12 = td2(1:n-1); t22 = td2(2:n); t32 = zeros(1,n+1); t42 = t32;
t32(2:n+1) = td2; t42(1:n) = td2;

mu2 = t22 - t12; nu2 = t42 - t32;

switch Deriv
    case 'x ^ delta'
       xdi(1) = x0;
       for j = 1:n-1
       xdi(j + 1) = xdi(j) + mu(j)*equation(eqn, tdi(j), xdi(j));
       end
       tdi = tdi';
       xdi = xdi'
    case 'x ^ nabla'
       xd2(n) = x0;
       for j = 1:n-1
       xd2(n - j) = xd2(n-j+1) - nu2(n-j+1)*equation(eqn,td2(n-j+1),xd2(n-j+1));
       end
       td2 = td2';
       xd2 = xd2';
   case 'x ^ diamond-alpha'
       alpha = 0.5;
       xdi(1) = x0; xdi(2) = xRK(25);
```

```
        for j = 1:n-2
        xdi(j + 2) = (1/(alpha*nu(j+1)))*(equation(eqn,tdi(j+1),xdi(j+1))*mu(j+1)*nu(j+1)
        - mu(j + 1)*(1 - alpha)*(xdi(j + 1) - xdi(j)) + nu(j + 1)*alpha*xdi(j + 1));
        end
        tdi = tdi(2:n)';
        xdi = xdi(2:n)';
    case 'x ^ delta delta'
        xdi(1) = x0; xdi(2) = xRK(25);
        for j = 1:n-4;
        xdi(j + 2)= (1/mu(j))*(mu(j)*xdi(j + 1) + mu(j + 1)*xdi(j + 1) - mu(j + 1)*xdi(j)
        + equation(eqn, tdi(j), xdi(j))*mu(j)^2*mu(j+1));
        end
        tdi = tdi(1:n-2)';
        xdi = xdi';
    case 'x ^ nabla nabla'
        xdi(n) = x0; xdi(n - 1) = xRKb(25);
        for j = 1:n-2
        xdi(n-j-1) = (1/nu(n-j))*(equation(eqn,tdi(n-j+1),xdi(n-j+1))*nu(n-j)^2*nu(n-j-1)
        - nu(n-j-1)*(xdi(n-j+1)-xdi(n-j)) + nu(n-j)*xdi(n-j));
        end
        tdi = tdi(2:n)';
        xdi = xdi(2:n)';
end

q = length(tdi); q2 = length(td2); q3 = length(tci);

%solves on continuous part
if isequal(Deriv, 'x ^ nabla')
    a = tci(q3); b = tci(1); c = xd2(1);
    F = inline(eqn,'t','x');
    [tci,xci] = ode45(F, [a, b], c)
    sizer = length(tci);
```

```
    for i = 1:sizer
    rev(i) = sizer - i + 1;
    end
    xci = xci(rev);
    tci = tci(rev);
else
    a = tci(1); b = tci(q3); c = xdi(q);
    F = inline(eqn,'t','x');
    [tci,xci] = ode45(F, [a, b], c);
    sizer = length(tci);
end

tdi(q + 1:sizer) = 0; xdi(q + 1:sizer) = 0;

%calculates missing data point for appropriate DyEqns using classic RK4
h = (td2(2)-td2(1))/24; tTemp = td2(1):h:td2(2); xRK(1) =
xci(sizer); for m = 1:24
        s1 = equation(eqn, tTemp(m), xRK(m));
        s2 = equation(eqn, tTemp(m), xRK(m)) + 0.5*h*s1;
        s3 = equation(eqn, tTemp(m), xRK(m)) + 0.5*h*s2;
        s4 = equation(eqn, tTemp(m), xRK(m)) + h*s3;
        xRK(m + 1) = xRK(m) + h/6*(s1 + 2*s2 + 2*s3 + s4);
end

switch Deriv
    case 'x ^ delta'
       xd2(1) = xci(sizer);
       for j = 1:n-2
       xd2(j + 1) = xd2(j) + mu2(j)*equation(eqn, td2(j), xd2(j));
       end
       td2 = td2(1:n-1)';
       xd2 = xd2';
```

```
case 'x ^ nabla'
   xdi(n) = xci(1);
   for j = 1:n-1
   xdi(n - j) = xdi(n - j + 1) -
   nu(n - j + 1)*equation(eqn,tdi(n - j + 1),xdi(n - j + 1));
   end
   tdi = tdi(2:n)';
   xdi = xdi(2:n)';
   q = length(tdi);
   tdi(q + 1:sizer) = 0;
   xdi(q + 1:sizer) = 0;
case 'x ^ diamond-alpha'
   alpha = 0.5;
   xd2(1) = xci(sizer); xd2(2) = xRK(25);
   for j = 1:n-2
   xd2(j+2) = (1/(alpha*nu2(j+1)))*(equation(eqn,td2(j+1),xd2(j+1))*mu2(j+1)*nu2(j+1)
   - mu2(j + 1)*(1 - alpha)*(xd2(j + 1) - xd2(j)) + nu2(j + 1)*alpha*xd2(j + 1));
   end
   td2 = td2(1:n-1)';
   xd2 = xd2(1:n-1)';
case 'x ^ delta delta'
   xd2(1) = xci(sizer); xd2(2) = xRK(25);
   for j = 1:n-4;
   xd2(j+2) = (1/mu2(j))*(mu2(j)*xd2(j+1) + mu2(j+1)*xd2(j+1) - mu2(j+1)*xd2(j)
   + equation(eqn, td2(j), xd2(j))*mu2(j)^2*mu2(j+1));
   end
   td2 = td2(1:n-2)';
   xd2 = xd2';
case 'x ^ nabla nabla'
   xd2(n) = xci(sizer); xd2(n - 1) = xRKb(25);
   for j = 1:n-2
   xd2(n-j-1)=(1/nu2(n-j))*(equation(eqn,td2(n-j+1),xd2(n-j+1))*nu2(n-j)^2*nu2(n-j-1)
```

```
        - nu2(n-j-1)*(xd2(n-j+1)-xd2(n-j)) + nu2(n-j)*xd2(n-j));
        end
        td2 = td2(2:n)';
        xd2 = xd2(2:n)';
        end
q2 = length(td2); td2(q2 + 1:sizer) = 0; xd2(q2 + 1:sizer) = 0;
xdi(q + 1) = xci(1) tdi(q+1) = tci(1);
```

Bibliography

[BaOt05] B. Ballard and B. Otegbade, *Manual of Time Scales Tool Box for MATLAB*, 2005, web-publication and MATLAB toolbox available at http://www.timescales.org.

[Bel37] E.T. Bell, *The Men of Mathematics*, Simon and Schuster, 1937.

[BoPe01] M. Bohner and A. Peterson, *Dynamic Equations on Time Scales*, Birkhäuser, 2001.

[BoPe03] M. Bohner and A. Peterson, eds. *Advances in Dynamic Equations on Time Scales*, Birkhäuser, 2003.

[DHOv04a] E. R. Duke, K. J. Hall, and R. W. Oberste-Vorth, Changing time scales I: The continuous case as a limit, *Proc. of Sixth WSEAS Internat. Conf. on Appl. Math.*, (CD-ROM), 2004.

[DHOv04b] E. R. Duke, K. J. Hall, and R. W. Oberste-Vorth, Changing time scales II: Bifurcations in second degree equations, *Proc. of Sixth WSEAS Internat. Conf. on Appl. Math.*, (CD-ROM), 2004.

[ElSh04] P. W. Eloe and Q. Sheng, Approximating crossed symmetric solutions of nonlinear dynamic equations via quasilinearization, *Nonlinear Analysis*, **56** (2004), 253–272.

[ESH03] P.W. Eloe, Q. Sheng, and J. Henderson, Notes on crossed symmetry solutions of boundary value problems on time scales, *J. of Difference Equations and Applications*, **9** (2003), 29–48.

[Gle03] P. Glendinning, A View from the Pennines: Analysis on Time Scales, *Mathematics Today*, **39**, (2003), 156–159.

[Hil90] S. Hilger, Analysis on measure chains—a unified approach to continuous and discrete calculus, *Results in Mathematics*, **18** (1990), 18–56.

[KePe04] W. Kelley and A. Peterson, *The Theory of Differenital Equations: Classical and Quantitative*, Pearson Education, 2004.

[KePe01] W. Kelley and A. Peterson, *Difference Equations: an introduction with applications*, 2nd ed., Harcourt/Academic Press, 2001.

[LaOv06] B. A. Lawrence and R. W. Oberste-Vorth, Solutions of dynamic equations with varying time scales, *Proc. of Internat. Conf. on Difference Eqns, Special Funcs. and Appl.* (to appear)

[LeV98] R. J. LeVeque, *Finite Difference Methods for Differential Equations*, lecture notes for AMath585-6, 1998, available at http://www.amath.washington.edu/ rjl/booksnotes.html.

[Mol04] C. Moler, *Numerical Computing in MATLAB*, SIAM, 2004, available at http://www.mathworks.com/moler/index.html.

[RoSh06] J. Rogers, Jr. and Q. Sheng. Notes on the Diamond-α Derivative on Time Scales, *J. of Math. Analysis and App.*, 2006.

[She05] Q. Sheng, A view of dynamic derivatives on time scales from approximations, *J. of Difference Equations and Applications*, **11** (2005), 63–81.

[She06] Q. Sheng, Some Preliminary Results on Dynamic Derivative Approximations on Time Scales, Abstract, *AMS/MAA Joint Meetings*, San Antonio, Texas, 2006.

[SFHD06] Q. Sheng, M. Fadag, J. Henderson and J. M. Davis, An exploration of combined dynamic derivatives on time scales and their applications, *Nonlinear Anal.: Real World Appl.*, **7** 2006, 395–413.

[Tre96] L. N. Trefethen, *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations*, unpublished text, 1996, available at http://web.comlab/ox.ac.uk/oucl/work/nick.trefethen/pdetext.html.

[WyBa95] C. R. Wylie and L.C. Barrett, *Advanced Engineering Mathematics*, 6th ed., McGraw-Hill, 1995.

[Zil97] D. Zill, *A First Course in Differential Equations with modeling applications*, 6th ed., Brooks/Cole, 1997.