An ODE solver for constrained state spaces: with applications to hybrid-system simulations

Marcos Ángel Donolo

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute of Technology and State University
in partial fulfillment of the requirements for the degree of

Masters of Science in

Electrical Engineering

November 22, 2002
Blacksburg Virginia

Committee:

Pushkin Kachroo, Chairman

Hugh VanLandingham

Lynn Abbott

Keywords: hybrid systems, Zeno effect, ODEs, IVPs, control systems simulation, event detection.

An ODE solver for constrained state spaces: with applications to hybrid-system simulations [†]

by

Marcos Ángel Donolo

mdonolo@vt.edu

Committee: Dr. P. Kachroo, Dr. H. VanLandingham, Dr. L. Abbott.

Bradley Department of Electrical Engineering

ABSTRACT

This thesis presents a solver to handle constrained-state-space ODEs. This solver locates the points where any of the states go outside of the constraining set, and then, transfers the control from the continuous time ODE to the function governing the behavior of the system on the boundaries of the constrained set. The main contribution of this solver is found simulating complex right-hand-sided ODEs for long periods of time.

[†] December 2002.

To my wife Rosana

Acknowledgements

I would like to thank my advisor Pushkin Kachroo for suggesting the research topic, for helping me during the work, and for encouraging me to study mathematics; which has already helped me a great deal. Also, I would like to express my appreciation to Dr. Hugh VanLandingham and Dr. Lynn Abbott for participating in my examination committee, and to Dr. Lawrence Shampine for his useful advice.

I wish to thank Mark Zaldivar of the VT Writing Center, because he is responsible of any sense-making sentence in this thesis.

The help and support of my lab partners both from the Control Laboratory and from Power System Laboratory is much appreciated. Thanks then to Adytia Gadre, Caleb Sylvester, Chris Cannell, David Elizondo, Jared Mach, Juancarlo Depablos, Paolo Dadone, and Xinming Huang.

I am deeply indebted to Virgilio Centeno, for a number of reasons. Here, I want to thank him for proofreading my thesis, which is probably one of the smallest things he has done for me.

I want to thank my parents for giving me the best vacations I can imagine.

Finally, I wish to thank my wife Rosana, for traveling away from her goddaughter to come here with me, and more specially, for rowing with me.

Table of contents

Chapter 1 Introduction 1
1.1 Overview 1
1.2 Thesis outline 1
1.3 References 2
Chapter 2 Motivation 3
2.1 The ramp metering control problem 3
2.2 The general case, the Skorokhod problem and Zeno-effect-exhibiting systems 6
2.2.1 The general problem 6
2.2.2 The Skorokhod problem 7
2.2.3 Hybrid automata and zenoness 7
Hybrid automata 8
Zenoness 9
2.2.4 Bouncing ball example 10
2.2.5 Boundary-detection-error order 11
2.3 Chapter notes 14
2.4 References 14
Chapter 3 Notations and assumptions 16
3.1 Runge-Kutta, Runge-Kutta Fehlberg, and Dormand-Prince methods 16
3.1.1 The general Runge-Kutta method 16
3.1.2 Runge-Kutta Fehlberg and Dormand-Prince methods 18
Runge-Kutta Fehlberg method 19
The Dormand-Prince method 20
3.2 Event detection 21
3.2.1 Continuous extension for Runge-Kutta methods 21
3.2.2 Event detection algorithm 22
General scheme 22
Root-finding methods 24
3.3 Step size control enhancement 25

3.4 Chapter notes 26
3.5 References 26
Chapter 4 Results 28
4.1 Solver outline 28
4.2 ODE solver test 29
Test problem 29
DOPRI54 solver test 30
DOPRI853 solver test 31
4.3 Bouncing ball example 32
4.3.1 Bouncing ball example with DOPRI54c 33
4.3.2 Bouncing ball example with DOPRI853 34
4.4 Ramp metering control example 35
Control function derivation 35
Simulation results 36
4.6 References 38
Chapter 5 Final considerations and Future work 39
5.1 Final considerations 39
5.2 Future work 40
5.3 References 40
Index 41
Appendix A 43
Appendix B 46
Appendix C 50
Vita 52

List of figures

Figure 1 a) Analytic and numerically integrated solution, b) Local speed error ______ 11
Figure 2 Error order ______ 12
Figure 3 Regula falsi and Illinois root-finding methods ______ 25
Figure 4 Solver function's ______ 28
Figure 5 Angular position for 5e4 operations top, and for 5e5 operations bottom ______ 30
Figure 6 Angular velocity for 5e4 operations top, and for 5e5 operations bottom ______ 31
Figure 7 Angular position and velocity obtained with DOPRI853 ______ 32
Figure 8 Bouncing ball example simulation time elapsed ______ 33
Figure 9 Bounce time error ______ 34
Figure 10 Bouncing ball example with DOPRI853 ______ 34
Figure 11 RMCP states and inputs ______ 37
Figure 12 RMCP control action and main traffic flow ______ 38

List of tables

Table 1 Approximation order for Runge-Kutta methods ______ 18
Table 2 Local error bound ______ 30
Table 3 Feature Comparisons ______ 39

Acronyms and symbols

∂K	boundary of K
f^m	mth derivative of $f(\cdot)$
DOPRI54c	Our proposed solver based on Dormand and Prince ODE integrator of orders 5 and 4
DOPRI54d	Our continuous extension implementation of Dormand and Prince ODE integrator of orders 5 and 4
DOPRI853	Our proposed solver based on Dormand and Prince ODE integrator of orders 8, 5 and 3
h	Numerical integrator step length
HA	Hybrid automata
IVP	Initial value problem
K	Subset of $\mathbb{R}^n$
K^0	interior of K
L	Freeway segment length
Li	Lipschitz constant for $f(t,x)$:
ODE	Ordinary differential equation
ODE45	Runge-Kutta-Dormand-Prince ODE solver of order (5)4 implemented by Shampine and Reichelt
RMCP	Ramp metering control problem
SP	Skorokhod problem
$x(t_n)$	Analytic solution at t_n
x_n	Numerical solution at the nth step

Chapter 1 Introduction

In this chapter, we give an overview to the thesis topic, along with a brief description of the contents included in each of the following chapters.

1.1 Overview

A great deal of research is being done regarding hybrid systems and hybrid control systems, including some ordinary differential equation ODE solver extensions to better deal with discontinuities and external events. Among the most recent research, we count the work by Shampine and Thompson in 2000 [2] regarding event location, and the one by Calvo *et. al.* in 2001 [3] regarding the solution of initial value problems IVPs with discontinuities. These researches have materialized in several different solvers. The two best known solvers are those by Shampine and Reichelt from 1997 [1] included in Matlab and Simulink, and Ptolemy which is being developed at The University of California at Berkeley [5].

In this thesis, we present a solver to handle constrained-state-space ODEs which differs from the two mentioned above in the following aspects: First, we use higher order ODE integrators that reduce the computing time when simulating complex right-hand-sided ODEs. And second, we allow the solver's user to define the error bound on the event detection algorithm that can be used to obtain highly accurate solutions or to dramatically reduce the simulation time.

1.2 Thesis outline

In Chapter 2, we introduce the ramp metering control problem as the initial motivation to write the solvers presented in this work. Then, we show the general type of problem we would like to solve. And finally, we review some particular applications for the solver.

In Chapter 3, we present the tools used in our solver along with notations and assumptions that will be used in the following chapters. We give notation for IVPs, and

Runge-Kutta algorithms along with the most well-known implementations by Feldberg and Dormand-Prince. Later, we discuss continuous extensions for these algorithms and root-finding schemes. Finally, we present a novel approach for step size control, specially suited for Zeno-effect-exhibiting IVPs.

Chapter 4 shows the outline of our solvers, and then presents some results obtained from applying the solvers to different problems.

Finally, Chapter 5 presents the conclusions of this research and an outline for future work on the topic.

1.3 References

[1] Lawrence Shampine. Some practical Runge-Kutta formulas. Mathematics of Computation, vol. 46, no. 1, pp. 135-150, Jan, 1986.

[2] L. Shampine, S. Thompson. Event Location for Ordinary Differential Equations. 2000.

[3] M. Calvo, J. Montijano, L. Randez. On the numerical solution of IVPs with discontinuities by adaptive Runge-Kutta codes. 2001.

[4] K.H Johansson, J. Lygeros, S. Sastry, M. Egerstedt. Simulation of Zeno hybrid automata. Proceedings of the 38th conference on decision & control. Phoenix, Arizona USA. 3538-3543. December 1999.

[5] Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Steve Neuendorffer, Yuhong Xiong, Haiyang Zheng. Ptolemy II heterogeneous concurrent modeling and design in java. http://ptolemy.eecs.berkeley.edu.

Chapter 2 Motivation

In this chapter, we first introduce the ramp metering control problem as the initial motivation to write the solvers presented in this work. Then, we move on to generalize the type of problem we would like to solve.

2.1 The ramp metering control problem

The initial motivation for this research came from the need for an efficient solver to simulate the ramp metering control problem (RMCP)[1]. In this class of problem, the traffic flow in a freeway and the queue length in the ramps are to be improved by controlling the ramp's inflow. The simplest RMCP case consists of a one-way freeway segment of length L with an inbound ramp at the beginning of the segment. We can write the conservation equation for the segment as follows:

$$\frac{d\rho(t)}{dt} = \frac{q_{in}(t) - q_{out}(t) + u(t)}{L}, \tag{2.1}$$

where $\rho(t)$ represents the car density in the segment, $q_{in}(t)$ represents the traffic flow coming in from the previous freeway segment, and $q_{out}(t)$ represents the traffic flow going out to the next freeway segment. Lastly, $u(t)$ represents the traffic flow coming in from the ramp; this traffic can be controlled by means of a traffic light, which implies that $u(t)$ is a positive semi-definite function *i.e.* a function returning values bigger or equal to zero.

The traffic flow going out to the next freeway segment at any time ξ can be expressed by:

$$q_{out}(\xi) = v_f \rho(\xi)\left(1 - \frac{\rho(\xi)}{\rho_{\max}}\right), \tag{2.2}$$

where v_f represents the free flow speed, and ρ_{max} represents the jam density for the segment. As a sanity check, note that $\rho(\xi) = \rho_{max}$ and $\rho(\xi) = 0$ yield $q_{out}(\xi) = 0$, which

correspond with an empty and a completely packed freeway respectively; furthermore, by analyzing the derivative of the traffic flow going out to the next freeway segment:

$$q_{out}(\xi) = v_f \rho(\xi) - \frac{v_f}{\rho_{\max}} \rho(\xi)^2 \;\Rightarrow\; \frac{dq_{out}(\xi)}{d\rho} = v_f - \frac{2v_f}{\rho_{\max}} \rho(\xi) \Rightarrow$$
$$0 = v_f - \frac{2v_f}{\rho_{\max}} \rho(\xi) \;\Rightarrow\; \frac{2v_f}{\rho_{\max}} \rho(\xi) = v_f \;\Rightarrow\; \rho(\xi) = \frac{\rho_{\max}}{2}, \tag{2.3}$$

$q_{out}(\xi)$ has a maximum at $\rho(\xi) = \frac{\rho_{\max}}{2}$.

Following with the RMCP model, Equation (2.4) gives an expression for the ramp queue length:

$$s(t) = s_0 + \int_0^t qs_{in}(\varsigma)d\varsigma - \int_0^t u(\varsigma)d\varsigma, \tag{2.4}$$

where $s(t)$ is the ramp queue length at time t, s_0 is the initial queue ramp length, and $qs_{in}(t)$ is the traffic flow arriving at the ramp. Taking the derivative of the ramp queue length:

$$\frac{ds(t)}{dt} = qs_{in}(t) - u(t), \tag{2.5}$$

and choosing as state variables:

$$x(t) = \begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = \begin{pmatrix} \rho(t) - \frac{\rho_{\max}}{2} \\ s(t) \end{pmatrix}, \tag{2.6}$$

and differentiating, one obtains the following system of ordinary differential equations (ODEs):

$$\begin{aligned} \frac{dx_1(t)}{dt} &= \frac{q_{in}(t) - q_{out}(t) + u(t)}{L} \\ \frac{dx_2(t)}{dt} &= qs_{in}(t) - u(t) \\ t_0 &\le t \le t_f. \end{aligned} \tag{2.7}$$

It follows from the constraint on the car density $\rho(t)$ that:

$$\begin{aligned} 0 \le \rho \le \rho_{\max} \Rightarrow \\ 0 \le x_1 + \frac{\rho_{\max}}{2} \le \rho_{\max} \\ -\frac{\rho_{\max}}{2} \le x_1 \le \rho_{\max} - \frac{\rho_{\max}}{2} \\ -\frac{\rho_{\max}}{2} \le x_1 \le \frac{\rho_{\max}}{2}. \end{aligned} \tag{2.8}$$

Besides this constraint, from the ramp queue length $s(t)$ constraints, we have that:

$$0 \le s \le s_{\max} \Rightarrow 0 \le x_2 \le s_{\max}. \tag{2.9}$$

Similarly, from the RMCP definition, we know that the traffic flows only in one direction, i.e. q_{in}, qs_{in}, and u are either positive or zero. It is straightforward that if q_{in} grows big enough, no control $u(t)$ can avoid traffic jams. It is also straightforward that if qs_{in} grows big enough, then the queue length $s(t)$ will grow longer than the maximum allowable value s_{max}, but in order to make the point we are pursuing here, let us assume that q_{in}, and qs_{in}, have nicely small values. Finally, the last constraint on the RMCP not modeled by Equation (2.7) takes into account the fact that at time $t=\xi$ if $s(\xi)=0$ then $u(\xi)$ has to be either smaller than or equal to qs_{in}. Then, by forcing this upper limit on $u(\xi)$, we warrant that $s(t)$ will never become negative.

Adding everything up, we can solve the RCMP problem using an ODE solver, as long as the state variables remain in the interior of the allowed region specified in Equations (2.8) and (2.9). But, whenever any of the states go outside the allowed region we have, first, to find the point at which the state went out, and second, to modify Equation (2.7). It is clear that we do not know beforehand neither how many times the states will reach the boundaries nor at which times it will happen. This problem implies that we have to run partial simulations and need to be prepared to apply the boundary constraints at any time. In our particular problem, with q_{in} and qs_{in} nice enough, and with a control function $u(t)$ driving x_2 to zero, a boundary, we will have to apply the constraints awfully frequently.

In brief, we would like to develop a tool to automatically solve problems like the RMCP with an accuracy comparable to that of the standard ODE solver within the allowed region.

2.2 The general case, the Skorokhod problem and Zeno-effect-exhibiting systems

In the previous section, we explained the problem which motivates this research; the explanation allows us to depict the key features of the general problem for which we want to write a solver. In this section, we will state the general problem and then we will give some particular examples of it.

2.2.1 The general problem

Let K be a closed subset of $\mathbb{R}^n$ and denote its interior and boundaries by K^0 and ∂K respectively. Then, we say that the function $x(t):[a,b]\mapsto K$ is a solution for the initial value problem (IVP):

$$IVP\Big(f\big(t,x(t)\big),H\big(t,x(t)\big),K\Big),\quad x(a)=x_0,\ a\le t\le b,\ \ x_0\in K, \tag{2.10}$$

where $f\big(t,x(t)\big):K^0\times\mathbb{R}\mapsto K$ such that:

$$\dot{x}(t)=f\big(t,x(t),u(t)\big)\qquad x(t)\in K^0, \tag{2.11}$$

governs the behavior of *x(t)* within the closed subset *K*, and where $H\big(t,x(t)\big):\mathbb{R}\times\partial K\mapsto K^0\times\mathbb{R}$ such that:

$$\Big(t^+,x\big(t^+\big)\Big)=H\big(t,x(t)\big),\quad x(t)\in\partial K,\qquad x\big(t^+\big)\in K^0, \tag{2.12}$$

gives *x(t)*'s behavior on *K* boundaries. Further let the interior of:

$$0=k\big(t,x(t)\big), \tag{2.13}$$

define K^0. Then we can unfold it into:

$$k_l\left(t, x(t)\right) \leq x(t) \leq k_u\left(t, x(t)\right), \tag{2.14}$$

where $k_l\left(t, x(t)\right)$ and $k_u\left(t, x(t)\right)$ constitute the lower and upper boundaries for the state vector x. A special case of this type of IVP is the Skorokhod problems explained in the following section.

2.2.2 The Skorokhod problem

Skorokhod presented the Skorokhod problem (SP) in 1961 as a means to study differential equations with so called reflecting boundary conditions [2]. Since then, the SP has been used for applications such as processor sharing [8], [9], queuing models [10], and control of trunk line systems [10]. The SP fits in the general problem definition in the sense that its solution maps unconstrained paths in $\mathbb{R}^n$ to paths constrained within the allowed subset $K \subset \mathbb{R}^n$, by means of a constraining function $H\left(t, x(t)\right) : \mathbb{R} \times \partial K \mapsto K$, which requires the "least amount of effort" possible. Precise definitions of the SP and extensive analytic development can be found in [3], and [8]; but, our research is concerned about simulation strategies to deal with the application of the constraining function $H\left(t, x(t)\right)$. In particular, we are interested in some troublesome cases in which we have to switch to the constraining function often. It turns out that these troublesome cases are referred in literature as cases exhibiting the Zeno phenomenon. Under the next heading, we give some examples of systems showing this behavior.

2.2.3 Hybrid automata and zenoness

In this section, we will first show how hybrid automata fits in our general problem definition, and second, we will explain how our solver reduces the computational effort required to simulate hybrid automata's executions, especially in those extreme cases where the executions turn out to exhibit Zeno properties.

Hybrid automata

Consider an elemental hybrid automaton formally defined in [11], and [12]:

$$HA = \left(Q, \Sigma, D, I, E, \left(q_0, x_0\right)\right), \tag{2.15}$$

where Q is a finite set of vertices; Σ is a finite set of input and output events; D is:

$$D = \left\{d_q : q \in Q\right\}, \tag{2.16}$$

with d_q the dynamics corresponding to the qth vertex given by:

$$\dot{x}_q = f_q\left(x_q(t), u_q(t)\right), \tag{2.17}$$

similarly:

$$I = \left\{I_q : q \in Q\right\}, \tag{2.18}$$

where I_q represents conditions under which the *HA* remains at the qth vertex. Finally, E is a set of transition paths for every allowable transition, and (q_0, x_0) denotes the initial vertex and the initial state, respectively.

A run of the *HA* is a sequence:

$$q_0 \xrightarrow{t_1} q_2 \xrightarrow{t_2} q_2 \xrightarrow{t_3} \cdots \qquad q_i \in Q, \tag{2.19}$$

where t_i denotes the time when the ith transition occurs, and q_i, $i>0$ denotes the vertex reached after the ith transition. The trajectory of a run is the piecewise continuous function given by:

$$T(HA,t) = x_{q_0}, x_{q_1}, x_{q_2}, \ldots \quad x_{q_i} = \left\{x_{q_i}(t) : t \in \left[t_i, t_{i+1}\right)\right\}. \tag{2.20}$$

We say that an *HA* is well-defined if for every time $t = \zeta$, where $x(\zeta) \in I_k$ and $q_k \in Q$ is the active partition, there exists one and only one valid possible transition e_ζ.

Finally, looking at this *HA* from our general problem point of view, we find that at each edge $q_k \in Q$ we can take:

$$\begin{gathered} f\left(t, x(t), u(t)\right) = f_k\left(x_k(t), u_k(t)\right) \\ \partial K = I_k \\ H\left(t, x(t)\right) = e\left(t_k, x(t_k)\right), \end{gathered} \tag{2.21}$$

and right after every evaluation of $H\left(t, x(t)\right)$ swap to the following edge by taking:

$$\begin{gathered} f\left(t, x(t), u(t)\right) = f_{k+1}\left(x_{k+1}(t), u_{k+1}(t)\right) \\ \partial K = I_{k+1} \\ H\left(t, x(t)\right) = e\left(t_{k+1}, x(t_{k+1})\right), \end{gathered} \tag{2.22}$$

allowing us to use our solver to simulate hybrid automata.

Zenoness

We say that a *HA* run, like the one in Equation (2.19), exhibits Zeno behavior if it exhibits an infinite number of transitions in a finite time:

$$\lim_{i \to \infty} t_i = \tau_\infty < \infty, \tag{2.23}$$

where t_i denotes the time when the ith transition happens and τ_∞ is known as the Zeno time. It is clear that we will run into problems trying to simulate an *HA* run close or beyond the Zeno time because in the best case, it involves evaluating an infinite number of times the function $H\left(t, x(t)\right)$. We claim that a solver designed to deal with discontinuities will both reduce errors by pinpointing the boundary crossing points, and will get closer to the Zeno time than regular ODE solvers.

As an aside, note that at least three methods have been developed to take simulations through the Zeno time; namely, continuation by regularization [13], by averaging [13], and by the Filippov solution [14]. These methods go around the Zeno time leaving a small blank around it. Here again, using these methods, along with a solver designed to deal with discontinuities, should yield smaller blanks around the Zeno times.

2.2.4 Bouncing ball example

As a simple example consider a ball bouncing over a flat and hard surface. Letting g be the gravity's acceleration modulus, $x(t)$ be the vertical position of the ball, and K be the allowed region where:

$$\begin{aligned} K &= \left\{ x(t) \in \mathbb{R}^2 \middle| x_1(t) \geq 0 \right\}, \\ K^0 &= \left\{ x(t) \in \mathbb{R}^2 \middle| x_1(t) > 0 \right\}, \\ \partial K &= \left\{ x(t) \in \mathbb{R}^2 \middle| x_1(t) = 0 \right\}, \\ 0 &= k(t, x(t)) = x_2(t) \Rightarrow \\ -\infty &\leq x_1(t) \leq \infty \\ 0 &\leq x_2(t) \leq \infty, \end{aligned} \tag{2.24}$$

then:

$$\ddot{x}(t) = -g \Rightarrow \begin{matrix} x_1(t) = x(t) \\ x_2(t) = \dot{x}(t) \end{matrix} \Rightarrow \begin{matrix} \dot{x}_1(t) = \dot{x}(t) \\ \dot{x}_2(t) = \ddot{x}(t) \end{matrix} \Rightarrow \begin{matrix} \dot{x}_1(t) = x_2(t) \\ \dot{x}_2(t) = -g \end{matrix}, \quad x(t) \in K^0, \tag{2.25}$$

and:

$$f(x(t), t) = \begin{matrix} x_2(t) \\ -g \end{matrix}, \quad x(t) \in K^0. \tag{2.26}$$

Assuming that the ball loses an amount of movement proportional to its speed in every bounce, the behavior at the edges is given by:

$$\begin{aligned} H(t, x(t)) &= \left(t, \begin{matrix} x_1(t) \\ -a x_2(t) \end{matrix} \right), \\ x(t) &\in \partial K, \quad a \in (0,1). \end{aligned} \tag{2.27}$$

Figure 1a) shows both the exact solution, calculated analytically, and the solution obtained using and ODE solver, Runge-Kutta 4, to integrate Equation (2.26) and applying Equation (2.27) every time $x(t) \notin K^0$, and. As can be seen, both solutions are very close to one another until the first bounce where the integration lags behind the analytic solution, as a consequence of the error detecting zero. Another consequence of this detection error is that the speed at which the ball bounces is always higher than the one in

the analytic solution. As a result, the integrated solution bounces back higher than the analytic solution, adding an extra delay to the one incurred in the bounce itself. Figure 1b) shows the difference between the speed calculated analytically and with Runge-Kutta 4; here again the numerical solution closely matches the analytical one until the first bounce where a spike in the error appears because the speed sign change in the numerical solution does not happen at the same time as in the analytic solution.

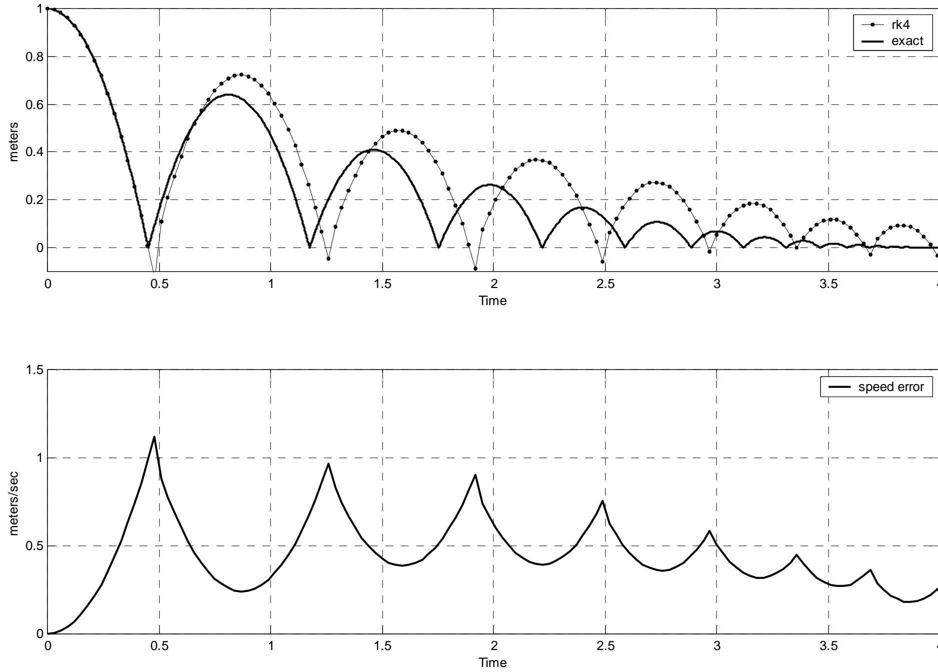

Figure 1 a) Analytic and numerically integrated solution, b) Local speed error

2.2.5 Boundary-detection-error order

In the previous example, we show that when integrating an IVP, like the one in Equation (2.10), the largest errors occur every time the integration runs into the boundaries of the constraining set. In this section, we will prove that in fixed step size algorithms, the error incurred is of the same order as the step. Figure 2 shows a generic step in which the state x has gone outside the allowed set. In general, the state x reached the boundary at time $t_n + \phi h$, where $\phi \in (0,1)$ and h is the step size; however, the trespassing state is detected at a later time, $t_n + h$. The difference between these two times constitutes the time error, e_t, which leads to an error in the state value associated with the boundary e_x.

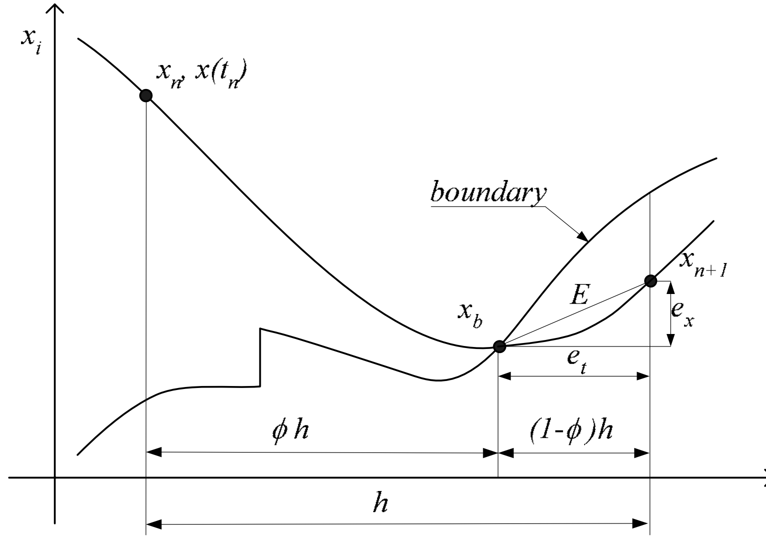

Figure 2 Error order

From Figure 2, the error E is:

$$E = \sqrt{e_t^2 + e_x^2}, \tag{2.28}$$

with:

$$e_t = (1-\phi)h, \quad 0 < \phi < 1, \tag{2.29}$$

expanding e_t we obtain:

$$\begin{aligned} & e_t = (h - \phi h) \Rightarrow e_t = (1-\phi)h \\ & 0 < \phi \le 1; \ \Rightarrow 0 \le (1-\phi) < 1 \\ & \therefore |e_t| \le |h|, \end{aligned} \tag{2.30}$$

then

$$e_t = O(h), \tag{2.31}$$

expanding now e_x we obtain:

$$e_x = x_b - x_{n+1}, \tag{2.32}$$

where:

$$
\begin{aligned}
x_b &= x_n + \sum_{k=1}^{\infty} \frac{(h\phi)^k}{k!} f^k(t_0, x_n) \\
x_b &= x_n + \frac{h\phi}{1!} f'(t_0, x_n) + \frac{(h\phi)^2}{2!} f''(t_0, x_n) + ... + \frac{(h\phi)^k}{(k)!} f^k(t_0, x_n) + ...,
\end{aligned}
\tag{2.33}
$$

and:

$$
x_{n+1} = x_n + \frac{h}{1!} f'(t_0, x_n) + \frac{h^2}{2!} f''(t_0, x_n) + ... + \frac{h^m}{m!} f^m(t_0, x_n), \tag{2.34}
$$

assuming an mth order RK algorithm. Replacing Equations (2.33) and (2.34) into (2.32):

$$
\begin{aligned}
e_x &= x_n + \frac{h\phi}{1!} f'(t_0, x_n) + \frac{(h\phi)^2}{2!} f''(t_0, x_n) + ... + \frac{(h\phi)^k}{(k)!} f^k(t_0, x_n) + ... \\
&\quad - \left(x_n + \frac{h}{1!} f'(t_0, x_n) + \frac{h^2}{2!} f''(t_0, x_n) + ... + \frac{h^m}{m!} f^m(t_0, x_n) \right),
\end{aligned}
\tag{2.35}
$$

and collecting powers of h we obtain:

$$
\begin{aligned}
e_x &= \frac{(\phi - 1)}{1!} f'(t_0, x_n) h + \frac{(\phi - 1)^2}{2!} f''(t_0, x_n) h^2 + ... + \frac{(\phi - 1)^m}{m!} f^m(t_0, x_n) h^m \\
&\quad + \frac{\phi^{m+1}}{(m+1)!} f^{m+1}(t_0, x_n) h^{m+1} + ...,
\end{aligned}
\tag{2.36}
$$

ignoring high order terms we end with:

$$
e_x \cong \frac{(\phi - 1)}{1!} f'(t_0, x_n) h, \tag{2.37}
$$

taking the absolute value we obtain:

$$
\begin{aligned}
|e_x| &\cong |(\phi - 1) f'(t_0, x_n) h| \\
|e_x| &\cong |-(1 - \phi) f'(t_0, x_n) h|,
\end{aligned}
\tag{2.38}
$$

and assuming a Lipschitz constant Li for $f(t, x)$:

$$
Li \geq |f'(t_0, x_n)|, \tag{2.39}
$$

we get an expression for the error in the *x* direction:

$$\begin{aligned} |e_x| &\le Li\left|(1-\phi)h\right| \\ |e_x| &\le Li(1-\phi)|h| \\ e_x &= O(h), \end{aligned} \tag{2.40}$$

now, replacing Equations (2.31) and (2.40) into (2.28):

$$\begin{aligned} E &= \sqrt{\left(O(h)\right)^2 + \left(O(h)\right)^2} \\ E &= \sqrt{K_1^2|h|^2 + K_2^2|h|^2} \\ E &= \sqrt{K_3^2|h|^2} \\ E &= K_3|h| \\ E &= O(h). \end{aligned} \tag{2.41}$$

Which tells us that the error incurred when the integration runs into a boundary is of the same order as the step size *h*. Later on, we will show that the ODE solvers regarded as *bests* [6], tend to increase the step size by using higher order approximations of the solution, which in turn leads to a higher boundary detection error.

2.3 Chapter notes

In this chapter, we introduced the ramp metering control problem as the initial motivation to write the solvers presented in this work. Also, we showed the general problem's type we would like to solve, its application to the Skorokhod problem and to simulate hybrid automata. In the following chapter we will present the numerical techniques involved in the solvers along with the justifications to use them in the solver.

2.4 References

[1] Pushkin Kachroo and Kumar Krishen, System dynamics and feedback control design problem formulations for real time ramp metering. Transactions of the SDPS, vol., pp. 37-54, Mar, 2000.

[2] A. V. Skorokhod, Stochastic equations for diffusions in a bounded region Theor. of prob. and its appl., vol. 6, pp. 264-274, 1961.

[3] Anna Nagurney. Projected dynamical systems and variational inequalities with applications, Kluwer's International Series, 1996.

[4] Edda Eich-Soellner and Claus Führer. Numerical methods on multibody dynamics, B. G. Teubner Stuttgart, 1998.

[5] John R. Dormand. Numerical methods for differential equations. A computational approach, CRC, 1996.

[6] Lawrence Shampine, Some practical Runge-Kutta formulas. Mathematics of Computation, vol. 46, no. 1, pp. 135-150, Jan, 1986.

[7] P.J.Monsterman, F.Zhao, and G.Biswas. Sliding mode model semantics and simulation for hybrid systems. In: In Hybrid system V, Anonymous Springter-Verlag, 1998.

[8] Paul Dupuis and Kavita Ramanan. Convex Duality and the Skorokhod Problem. 1998. Providence, RI 02912, Brown University.

[9] A. K. Parakh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. IEEE/ACM Transaction on Networking, 2:137-150, 1993.

[10] I. C. Paschalidis. Large Deviations in High speed networks. PhD thesis, MIT, Cambridge, Massachusetts, 1996.

[11] Lygeros, J., Johansson, K.H., Sastry, S., Egerstedt, M. On the existence of executions of hybrid automata Decision and Control, 1999. Proceedings of the 38th IEEE Conference on, 3:2249 -2254. 1999.

[12] M. Heymann, F Lin, G Meyer. Viability of controllers for Hybrid Machines. Proceeding of the 36th Conference on Decision & Control. San Diego, California, USA. December 1997.

[13] K.H Johansson, J. Lygeros, S. Sastry, M. Egerstedt. Simulation of zeno hybrid automata. Proceedings of the 38th conference on decision & control. Phoenix, Arizona USA. 3538-3543. December 1999.

[14] A. F. Filippov. Differential equations with discontinuous right hand sides. Kluwer publishers, 1988.

Chapter 3 Notations and assumptions

In this chapter, we present the tools used in our solver along with notations and assumptions that will be used in the following chapters. We start giving notation for IVPs, and then we move on to the Runge-Kutta algorithms along with the most well-known implementations by Feldberg and Dormand-Prince [1], [2], [4], [5], [9]. Later, we discuss continuous extensions for these algorithms [2] and root-finding schemes. Finally, we present a stiffness detection algorithm [2], and a novel approach for step size control specially suited for Zeno-effect-exhibiting IVPs.

3.1 Runge-Kutta, Runge-Kutta Fehlberg, and Dormand-Prince methods

The purpose of this section is to introduce the notation used on IVPs and to give a brief overview to the concepts behind the most used ODE integration algorithms. The notation and the concepts presented here will be useful to explain our ODE solver for constrained state spaces. For details on these algorithms see [1] and [2].

3.1.1 The general Runge-Kutta method

The goal of an ODE solver is to compute an approximated solution for an IVP given by:

$$\begin{gathered}\dot{x}(t) = f\left(t, x(t)\right), \\ x(a) = x_0,\ \ a \le t \le b, \\ f\left(t, x(t)\right): \mathbb{R} \times \mathbb{R}^n \mapsto \mathbb{R}^n,\end{gathered} \tag{3.1}$$

where x_0 is the initial vector. The general one-step Runge-Kutta method computes a sequence of approximated solution vectors $x_i \cong x(t_i)$ by applying:

$$g_0 = f\left(t_n, x(t_n)\right)$$
$$g_j = f\left(t_n + \alpha_j h, x(t_n) + h\sum_{k=1}^{j-1}\beta_{j,k} g_k\right), j = 1,...,s \tag{3.2}$$
$$x_{n+1} = x_n + h\sum_{j=1}^{s} c_j g_j,$$

where h is the step size, and the coefficients α_j, $\beta_{j,k}$, and c_j are chosen in such a way that local error is minimized. Assuming a smooth enough function *f(x(t),t)* we can use the Taylor expansion about (x_0, a) to calculate the local error. For example letting s=2:

$$x_{n+1} = x_n + h\left(c_1 g_1 + c_2 g_2\right)$$
$$x_{n+1} = x_n + h\Big(c_1 f\left(a, x(a)\right) + c_2 f\left(t(a) + \alpha_2 h, x(a) + h\beta_{2,1} f\left(a, x(a)\right)\right)\Big), \tag{3.3}$$

and using:

$$s = 2, c_1 = 0, c_2 = 1, \alpha_2 = \frac{1}{2}, \beta_{2,1} = \frac{1}{2}, \tag{3.4}$$

equation (3.3) becomes:

$$x_{n+1} = x_n + hf\left(a + \frac{h}{2}, x(a) + \frac{h}{2} f\left(a, x(a)\right)\right). \tag{3.5}$$

Computing the Taylor expansion of Equation (3.5) as a function of h:

$$\begin{aligned} x_{n+1} &= x_n + hf\left(a + \frac{h}{2}, x(a) + \frac{h}{2} f\left(a, x(a)\right)\right) \\ &= \left(x_n + hf\left(a + \frac{h}{2}, x(a) + \frac{h}{2} f\left(a, x(a)\right)\right)\right)\Bigg|_{h=0} \\ &+ h\frac{d}{dh}\left(x_n + hf\left(a + \frac{h}{2}, x(a) + \frac{h}{2} f\left(a, x(a)\right)\right)\right)\Bigg|_{h=0} \\ &+ \frac{h^2}{2}\frac{d^2}{dh^2}\left(x_n + hf\left(a + \frac{h}{2}, x(a) + \frac{h}{2} f\left(a, x(a)\right)\right)\right)\Bigg|_{h=0} + O\left(h^3\right), \end{aligned} \tag{3.6}$$

and simplifying:

$$x_{n+1} = x_n + hf\left(a, x(a)\right) + \frac{h^2}{2}\left(\frac{d}{dt} f\left(t, x\right)\Big|_{\substack{t=a \\ x=x(a)}} + \frac{d}{dx} f\left(t, x\right)\Big|_{\substack{t=a \\ x=x(a)}} f\left(a, x(a)\right)\right) + O\left(h^3\right), \quad (3.7)$$

which equals the order 2 Taylor expansion of $x(a+h)$, *i.e.* Equation (3.7) equals the Taylor expansion of the solution of Equation (3.1). In this example, we show that we can obtain a second order approximation of *x(t)* going from *t=a* to *t=a+h* using two right-hand side evaluations and an appropriate set of coefficients α_j, $\beta_{j,k}$, and c_j. Similarly, we can find sets of coefficients α_j, $\beta_{j,k}$, and c_j, for *s*=3, 4,…, *n*, which at most will yield approximations of these orders [3]:

s	3	4	5	6	7	≥ 8
Local approximation order	3	4	4	5	6	*s*-2

Table 1 Approximation order for Runge-Kutta methods [3]

In general, higher order methods result in smaller local truncation errors for the same step size *h*; conversely, the same local truncation errors can be attained with higher order methods using a larger step size *h*, which may lead to a reduced amount of right-hand side evaluations. In these methods, the step size is fixed beforehand; it can be homogeneous through the whole integration, or we can define it to be denser around any area of interest. Using a non-homogeneous preset step size helps to obtain fast solutions by spending little time on areas where the states change slowly and also accurate ones by having small steps where the states change more rapidly. Obviously, there are IVPs in which we do not know in advance where the interesting points will lay. For this kind of IVPs, we would like an algorithm that automatically updates its step size by taking into account the behavior of the states. In the following section, we address those algorithms.

3.1.2 Runge-Kutta Fehlberg and Dormand-Prince methods

In the previous section, we show the general Runge-Kutta method; in this section, we will present two algorithms based on the Runge-Kutta methods that automatically adjust their step size. Automatic step size selection reduces the overall amount of right-hand side evaluations by stretching the step size as much as possible while keeping the

local error under a predefined value. These methods use such a small overhead that they have become the most used ODE solvers.

Runge-Kutta Fehlberg method

Fehlberg presented the first of these two methods in 1970 [4]. The idea behind it is to use two Runge-Kutta algorithms of different orders at each step and use the higher order one to compute an error bound and then modify the step size accordingly. The method is as follows:

INPUT: Initial and final time a and b respectively; initial state x_0; tolerance TOL.
OUTPUT: t, x, h.

Step 1: set $t=a$; $x=x_0$; $h=h_0$;
output *t, x.*
Step 2: ***while*** *(t<b)* ***do*** *Steps 3 to 6*
Step 3: *compute low order solution x;*
compute high order solution $\tilde{x}$*;*
compute the error bound $e=\tilde{x}-x_i$*;*
Step 4: ***if*** *(TOL> e)* ***then do*** *Step 5*
Step 5: *set t =t+h; (approximation accepted)*
output *t, x, h.*
Step 6: *compute the next step size h.*
Step 7: ***stop***

The most widespread implementation of this algorithm uses Runge-Kutta algorithms of orders 4 and 5 with the same coefficients α_j, $\beta_{j,k}$, $j=1...5$ so that, it only requires 6 right-hand side evaluations to obtain both solutions. Fehlberg's algorithm is consistently more computationally expensive than the one presented by Dormand-Prince in [5], and that is why we use the latter in our solver. Under the following heading, we will give details on the Dormand-Prince method.

The Dormand-Prince method

The Dormand-Prince method (DOPRIx) differs from the one by Fehlberg in two key points. The first one is that DOPRIx uses the lower order solution to compute the error bound and the higher order solution to obtain the following solution approximation. This allows DOPRI to attain a higher order solution without having to do further right-hand side evaluations. The second difference is that, for the high order solution of DOPRI, α_s is set to one, so that the last right-hand side evaluation of the nth step matches the first right-hand side evaluation of the nth+1 step, and after the first step, we save one evaluation. The most widespread implementation of DOPRI is the one by Shampine and Reichelt [6] for Matlab 5.x. This implementation uses Runge-Kutta algorithms of orders 5 and 4, requiring $6 + 1/n$ right-hand side evaluations per step and yielding order 5 approximations. For our solver, we use two different implementations of the DOPRI method; the lower order one uses Runge-Kutta methods of orders 5 and 4 (DOPRI5), and the higher order one uses Runge-Kutta methods of orders 8 and 5 (DOPRI8), yielding an eighth order approximation. The DOPRI5 routines were written for Matlab following the guidelines in [5] and the DOPRI8 routines were borrowed from [13].

In this section, we show two algorithms that automatically adjust their step sizes. These methods achieve accurate solutions requiring a reduced number of right-hand side evaluations by stretching the step length. However, stretching the step size may cause a severe loss of resolution in the solution output, i.e. even when the approximated solutions are close enough to the actual solution, the solution path cannot be inferred from the output data. This problem becomes unbearable when using high order solvers such as Runge-Kutta Fehlberg of order (4)5, DOPRI5, or DOPRI8. Also, as we show in Equation (2.41), the error incurred every time the states reach a boundary is of the same order as the step size. One straitght forward solution to both of these problems is to limit the maximum step size so that the approximated solution is at least as dense as we need, but this approach is only effective when the right-hand side evaluations are computationally cheap. In the following section, we present a method that solves the resolution issue and can help to reduce the boundary detection error.

3.2 Event detection

This section first explains how continuous extensions for Runge-Kutta methods solve resolution problems caused by high order methods. Secondly, it explains the general root-finding scheme used to plug in to the root-finding methods within the ODE solver in order to accurately detect states going outside the allowed region.

3.2.1 Continuous extension for Runge-Kutta methods

To overcome the resolution problem, explained in Section 3.1.2, continuous extensions for most common Runge-Kutta methods have been developed [7], [8]. Continuous extensions for Dormand-Prince methods provide computationally cheap approximation to the solution at $t = \left(t_n + \phi h\right)$, where $0 < \phi < 1$ by applying:

$$
\begin{aligned}
g_0 &= f\left(t_n, x\left(t_n\right)\right) \\
g_j &= f\left(t_n + \alpha_j h, x\left(t_n\right) + h\sum_{k=1}^{j-1}\beta_{j,k}\, g_k\right),\ j = 1,\ldots,s,\ldots,\tilde{s} \\
x_n\left(t_n + \phi h\right) &= x_n + \phi\sum_{i=0}^{\tilde{s}} b_i g_i,
\end{aligned}
\tag{3.8}
$$

where the coefficients α_j, $\beta_{j,k}$, and c_j for $j \le s$ match those in Equation (3.2) and the remaining coefficients α_j, $\beta_{j,k}$, and c_j for $j>s$, and b_j for $j = 1,\ldots,s,\ldots,\tilde{s}$ are chosen to improve the approximation order of the continuous extension. Once implemented, the continuous extension allows us to generate a higher resolution output solution by evaluating Equation (3.8) at the points of interest. In addition, having a continuous extension allows us to use a root-finding scheme to find the boundary-crossing points with an error comparable to that of the continuous extension. We leave, for the next section, a discussion regarding root-finding schemes.

For our DOPRI5 solver, we use a continuous order 4 extension, and for the DOPRI8 solver, we use a continuous order 6 extension. These two algorithms do not require any extra right-hand-side evaluation [9].

3.2.2 Event detection algorithm

In the previous section, we addressed the loss of resolution issue caused by high order methods. In this section, we first explain the modifications made to the ODE solver so that it accommodates a root-finding algorithm, and second, we explain the root-finding algorithms used in our solver to locate the boundary crossing points.

General scheme

Consider Equation (2.14):

$$k_l\left(x(t),t\right) \le x(t) \le k_u\left(x(t),t\right), \tag{2.14}$$

where $k_l\left(x(t),t\right)$ and $k_u\left(x(t),t\right)$ constitute the lower and upper boundaries for the state vector *x*, or component-wise:

$$k_{li}\left(x(t),t\right) \le x_i(t) \le k_{ui}\left(x(t),t\right) \;\; i = 1...p, \tag{3.9}$$

where *p* is vector *x* length, and where $k_{li}\left(x(t),t\right)$ and $k_{ui}\left(x(t),t\right)$ constitute the lower and upper boundaries for the ith entry of the state vector *x.* then we are interested in locating the first points on which any of the states go either under $k_{li}\left(x(t),t\right)$ or over $k_{ui}\left(x(t),t\right)$, i.e. we are seeking the first time $t=\xi$ that satisfies either:

$$0 = x_i(\xi) - k_{li}\left(x(\xi),\xi\right) \;\; i = 1...p, \tag{3.10}$$

or:

$$0 = x_i(\xi) - k_{ui}\left(x(\xi),\xi\right) \;\; i = 1...p. \tag{3.11}$$

In order to find this point, we modify Step 5 in the algorithm in Section 3.1.2 to include the root-finding scheme:

INPUT: Initial and final time a and b respectively; initial state x_0; tolerance TOL;
# of inter-sample evaluations ISE.

OUTPUT: t, x.

Step 1: *set t=a; x=x_0; h=h_0;*
***output** t, x.*

Step 2: ***while*** *(t<b)* ***do*** *Steps 3 to 11*

Step 3: *compute low order solution x;*

compute high order solution $\tilde{x}$ *;*

compute the error bound $e= \tilde{x} - x_i$*;*

Step 4: ***if*** *(TOL> e)* ***then do*** *Step 5 to 10*

Step 5: *set hs=h/ISE (approx. accepted)*

set $t_u=t$*;*

compute $k_l(x(t), t)$*;* $k_u(x(t), t)$

Step 6: ***while*** *(j<ISE)* ***do*** *Steps 7 to 9*

Step 7: *set* $t_u=t+hs$*;*

compute $x(t_u)$*;*

compute $k_l(x(t_u), t_u)$*;* $k_u(x(t_u), t_u)$

$$\textbf{\textit{if}}\Big(sign\big(x_i(t)-k_{li}(x(t),t)\big)\neq sign\big(x_i(t_u)-k_{li}(x(t_u),t_u)\big)\ \vee$$

$$sign\big(x_i(t)-k_{ui}(x(t),t)\big)\neq sign\big(x_i(t_u)-k_u(x(t_u),t_u)\big),i=1\ldots p\Big)$$

then do step **8**

Step 8: $(t,x)=\text{root-findingsolver}(t,t_u)$ *;*

break;

Step 9: *set* $t = t_u$*;*

compute $k_l(x(t), t)$*;* $k_u(x(t), t)$*;*

Step 10: ***output*** *t, x.*

Step 11: *compute the next step size h.*

Step 12: ***stop***

The rationale behind this algorithm is to use a continuous extension to obtain smaller steps, and then to use a root-finding solver to pinpoint the crossing time within the first interior step exhibiting a sign change in either Equation (3.10) or (3.11). For our solvers, we implemented a variation of this algorithm in which, instead of requiring a fixed number of interior steps, we limit the maximum length of these interior steps. This variation provides an homogeneous interior-step size, which saves some computations

when the steps become small. It also prevents interior steps from becoming too long, which may lead to misplacing two sign changes by none.

Root-finding methods

Once a sign change has been detected in either Equation (3.10) or (3.11) between $t=\xi_l$ and $t=\xi_u$ then a root-finding technique should be used to accurately locate the zero. For our solver, we tried several root-finding methods, namely linear interpolation, bisection [11], [12], false position [11], [12], the secant method [11], [12], Newton Ralphon [11], [12], and the Illinois method [12]. As we show in the following chapter, we obtain good results with linear interpolation for the problem which motivates this research, but since the method's accuracy is extremely dependant on the boundary's shape we decide to use the Illinois method for the solver.

For the following discussion, on root-finding methods, let:

$$f(\xi) = x_i(\xi) - k_{li}\left(x(\xi), \xi\right), \tag{3.12}$$

then Equation (3.10) can be rewritten as:

$$0 = f(\xi), \tag{3.13}$$

where $f(\cdot)$ is a continuous but "non smooth" function.

Linear interpolation

The idea with using linear interpolation is to obtain a really quick approximation of the root. The root is calculated using:

$$\xi = t_l + \frac{f(t_l)}{f(t_l) - f(t_u)}(t_u - t_l). \tag{3.14}$$

Illinois method

The Illinois method is a slight variation of the regula falsi method or the false position method [11], [12]. Yet, this method is fast converging and provides the reliability of the regula falsi method. Figure 3 shows the difference between both algorithms.

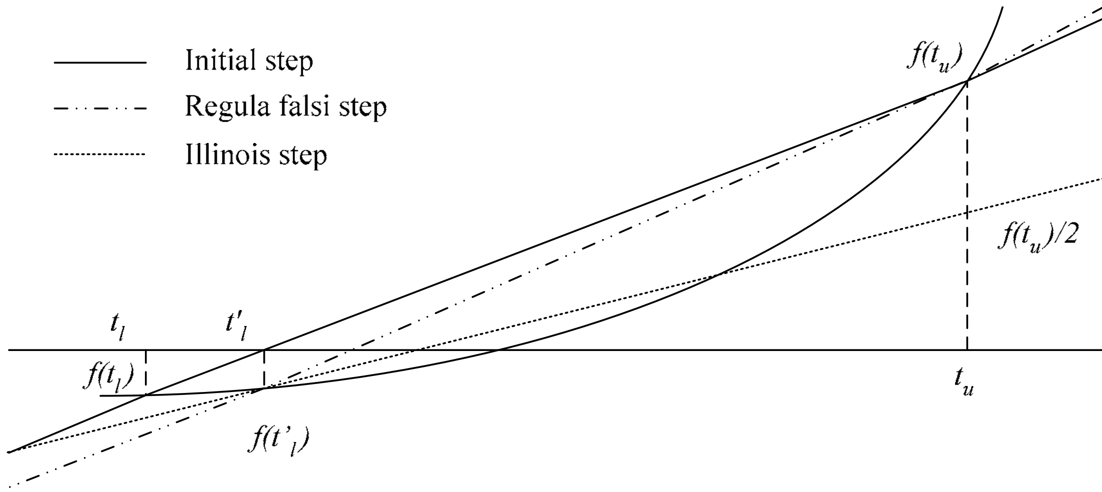

Figure 3 Regula falsi and Illinois root-finding methods

Illinois algorithm proceeds in the same way as in the regula falsi but when reusing a function value it is divided by two, what notably reduce the convergence time.

3.3 Step size control enhancement

In this section, we present a modification made to DOPRI's step size control algorithm. This modification allows the solver to gain several orders of precision when simulating Zeno-behavior-exhibiting systems by conveniently reducing the step size. This enhancement uses the fact that every time the solver reaches a boundary it discards part of the step and the fact that we can find a bound on the time between two consecutive transitions. Here, we start by showing how the step size influences the local error, and then we explain the rationale used to find a bound for the step size.

As explained in Section 3.1.1, the error incurred in each integration step is:

$$e = O\left(h^N\right), \tag{3.15}$$

where *N* is the approximation order. Then by reducing the step size, we attain a smaller local error. Also, consider the simulation of a *HA* run exhibiting Zeno behavior like the one shown below:

$$q_a \xrightarrow{t_1} q_a \xrightarrow{t_2} q_a \xrightarrow{t_3} \cdots \quad q_a = Q, \tag{3.16}$$

where:

$$\lim_{i\to\infty} t_i = \tau_\infty < \infty \,. \tag{3.17}$$

Defining the time between two consecutive transitions as:

$$\Delta t_i = t_{i+1} - t_i \,, \tag{3.18}$$

then Equation (3.17) implies that:

$$\lim_{i\to\infty} \Delta t_i = 0 \,, \tag{3.19}$$

and for big-enough values of *i* we can assert that:

$$\Delta t_i < \Delta t_{i+1} < .... < \Delta t_{i+m} \,. \tag{3.20}$$

Therefore, after the ith transition there is no point on allowing the step size to go over Δt_i so we limit the initial step size after a transition to the time between the last two transitions. Obviously, we can do much better than this by using information from several previous transitions to estimate the following one, or by switching to a lower order integration algorithm, but since doing it requires more overhead, we stick to the simpler algorithm, and we leave the implementation of these more complex algorithms for a later time.

3.4 Chapter notes

In this chapter, we introduced the DOPRI methods as the selected integration method for the solvers presented in this work. Also, we presented the continuous extension for these methods and a novel approach to control the integration step size. In the following chapter, we review these methods to show how they interact, and then we present some results obtained from applying our solver to different problems.

3.5 References

[1] John R. Dormand. Numerical methods for differential equations. A computational approach, CRC, 1996.

[2] Lawrence Shampine, Some practical Runge-Kutta formulas. Mathematics of Computation, vol. 46, no. 1, pp. 135-150, Jan, 1986.

[3] Butcher, J. R. On the attainable order of Runge-Kutta methods. *Mathematics of computation*, 19, 408-417. 1965.

[4] Fehlberg, E., New high-order Runge-Kutta formulas with an arbitrary small truncation error. *Computing* 6, 61-71. 1970.

[5] J. Dormand, P. Prince. A family of embedded Runge-Kutta formulae. *Journal of computational and applied Mathematics.* 6, 19-26. 1980.

[6] L. Shampine, M Reichelt. The Matlab Ode Suite, SIAM J. Sci. Comput., 18 1-22, 1997.

[7] M. Horn, Fourth- and fifth-order, scaled Runge-Kutta algorithms for treating dense output, SIAM. J. Numer. Anal., v. 20, pp. 558-568, 1983.

[8] L. Shampine, Interpolation for Runge-Kutta methods, SIAM. J. Numer. Anal., v. 22, pp. 1014-1027 1985.

[9] S. Papakostas, CH. Tsitouras, Highly continuous interpolants for one step ODE solvers and their application to Runge-Kutta Methods, SIAM J. Numer. Anal., v. 34, No. 1, pp. 22-47, 1997.

[10] Math/Library T. M. Fortran subroutines for mathematical Applications. IMSL. 1989.

[11] J. Stoer, R. Bulirsch, Introduction to numerical analysis. Second edition. Texts in applied mathematics 12. Springer-Verlag. 1993

[12] G. Engeln-Müllges, F. Uhlig. Numerical Algorithms with C. Springer. 1996.

[13] E. Hairer, S.P. Norsett and G. Wanner, Solving ordinary differential equations I, non-stiff problems, 2nd edition, Springer Series in Computational Mathematics, Springer-Verlag. 1993.

Chapter 4 Results

In this chapter, we first show the outline of our solver, and then, we present some results obtained from applying our solver to different problems. We choose the first of these problems to expose troubles in the integration scheme, and the second one to show the overall behavior of the solver when simulating Zeno-effect-exhibiting systems. Finally, we show the results obtained from simulating the problem that originally motivated this research.

4.1 Solver outline

We organize our solver in three blocks: the main block, which runs the algorithms detailed in Section 3.2.2, the Illinois algorithm block in charge of detecting the crossing points, and the continuous extension block used by the previous two blocks to obtain inter-step approximations of the solution. Besides these blocks, three problem-specific functions are required: one hosting the ODE right-hand-side as in Equation (2.10), another one hosting the boundary definition function as described in Equation (2.14), and finally, one hosting the function $H\left(x\left(t\right),t\right)$ which defines the behavior of the states once they reach a boundary, as explained in Section 2.2.1. Figure 4 shows the use relationship among the blocks and the problem-specific functions:

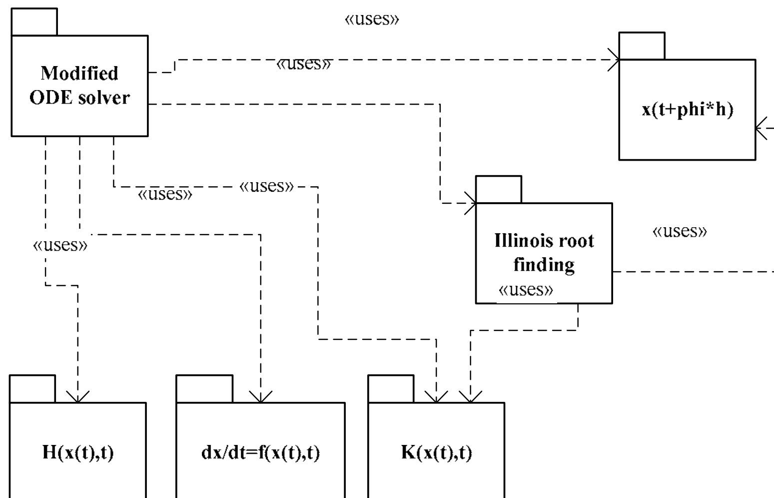

Figure 4 Solver function's

Following this three-block three-function scheme, we developed two ODE solvers for constrained state spaces. The first one –written for Matlab– is based on our own implementation of DOPRI5 [2]. The second one –written in C– is based on the order 8 implementation of Dormand and Prince's method by Hairer & Wanner [4]. Both methods use our own implementations of the Illinois method explained in Section 3.2.2 and the continuous extensions algorithm explained in [4]. We refer here to the first of these two algorithms as DOPRI54c, and to the second one as DOPRI853. Appendix A shows our implementation of DOPRI5 for Matlab; Appendix B shows the code corresponding to DOPRI54c, and Appendix C shows the C code corresponding to DOPRI853.

4.2 ODE solver test

This section shows the validation runs we performed on the solvers to make us sure that the integration algorithm is working properly. First, we show the problem used to analyze the solvers. Second, we compare the results obtained from DOPRI54 and DOPRI54c with those from ODE45 [1]. Third and last, we show the simulation results obtained with the DOPRI85 solver.

Test problem

We analyzed the performance of the implemented ODE solver by using it to integrate the pendulum equations shown below:

$$\ddot{\phi} = -\sin(\phi) \Rightarrow \begin{matrix} x_1 = \phi \\ x_2 = \dot{\phi} \end{matrix} \Rightarrow \begin{matrix} \dot{x}_1 = \dot{\phi} = x_2 \\ \dot{x}_2 = \ddot{\phi} = -\sin(\phi), \end{matrix} \tag{4.1}$$

with initial conditions:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \phi \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}, \tag{4.2}$$

where ϕ is the angle measured from the stable equilibrium point and $\dot{\phi}$ is the angular velocity. It is important to note that, with these initial conditions, the pendulum should swing once all the way up to the unstable equilibrium point and then remain there, but since numerical integrators do not keep energy invariant, the pendulum in the integrated

solution eventually falls. Clearly, the longer the simulation remains close to the instable equilibrium point, the better the simulation.

DOPRI54 solver test

To test this solver, we adjusted the allowed local error of both our DOPRI54 solver and the ODE45 so that the number of operations required in simulating the pendulum behavior for 40 s is roughly the same. We start tuning the local error to simulate the system using 5e4 operations and we obtain local error bounds of 1e-4, 1e-6, and 1/3e-4 for ODE45, DOPRI54 and for DOPRI54d respectively. Then, we proceed similarly but allowing 8e4 and 5e5 operations and we obtain the results shown in Table 2:

	Local error bound		
# of operations	ODE45	DOPRI54	DOPRI54d
5e4	1e-4	1e-6	$0.\widehat{3}$ e-4
8e4	1e-7	1e-9	$0.\widehat{3}$ e-7
5e5	1e-11	1e-13	$0.\widehat{6}$ e-11

Table 2 Local error bound

In Figure 5 and Figure 6, we show the simulation's outcome corresponding to the cases where we allow 5e4 and 5e5 operations:

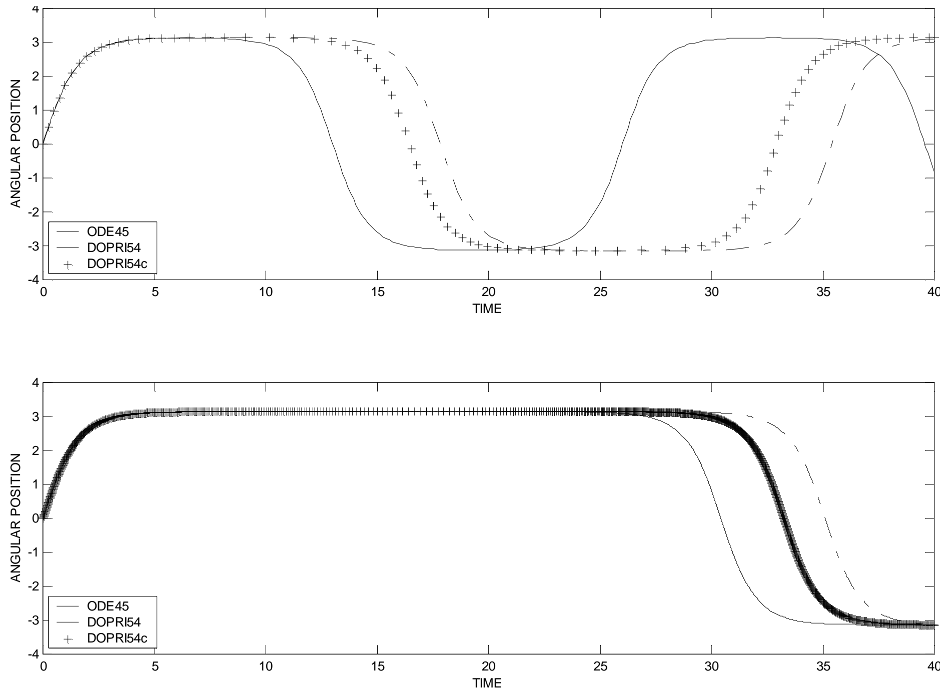

Figure 5 Angular position for 5e4 operations top, and for 5e5 operations bottom

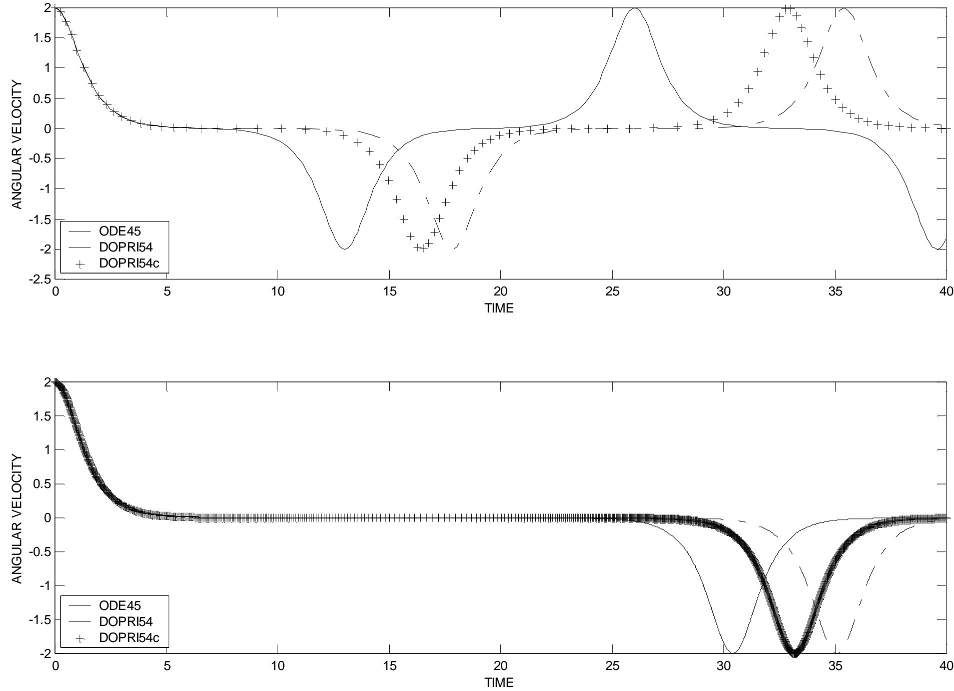

Figure 6 Angular velocity for 5e4 operations top, and for 5e5 operations bottom

We attribute the differences between the solutions obtained with DOPRI54 and with DOPRI54d to the operations required to compute the inter-steps solutions. Following, we attribute the differences between the solutions obtained with DOPRI54d and with ODE45 to the overhead required by the stiffness detection algorithm built in the ODE45 routine.

DOPRI853 solver test

To evaluate this solver, we set the allowed local error bound to 1e-5 and to 1e-12, and then we simulated the pendulum behavior. As Figure 7 shows, the results obtained from these simulations match those obtained with DOPRI5d when allowing similar local error bounds. We expect these matches to happen, because we are allowing the same amount of error in both algorithms, and with this setting, higher order methods yield larger step sizes in such a way that the local error remains close to the prefixed bound.

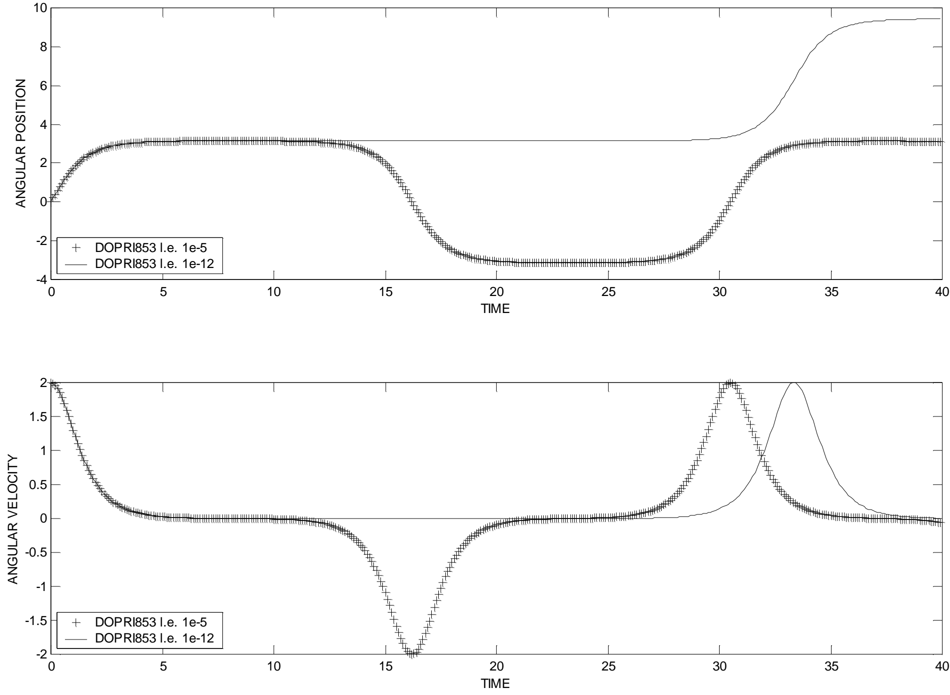

Figure 7 Angular position and velocity obtained with DOPRI853

4.3 Bouncing ball example

In this section, we show our solver's solution to the bouncing ball example given in Section 2.2.4 governed by Equations (2.26):

$$f\left(x(t),t\right)=\begin{matrix} x_2(t) \\ -g \end{matrix},\quad x(t)\in K^0, \tag{2.26}$$

and (2.27):

$$\begin{gathered} H\left(x(t),t\right)=\left(\begin{matrix} x_1(t) \\ -ax_2(t) \end{matrix},t\right), \\ x(t)\in\partial K,\qquad a\in(0,1), \end{gathered} \tag{2.27}$$

with initial conditions:

$$x(0)=\begin{pmatrix} 0.2 \\ 0 \end{pmatrix}. \tag{4.3}$$

First, we give the solution from DOPRI54c and then the solution from DOPRI853.

4.3.1 Bouncing ball example with DOPRI54c

In this section, we compare the outcome of our solver with that from ODE45 [1]. Figure 8 shows the simulation-elapsed time against the bouncing time *i.e.* it shows the time between the beginning of the simulation and each of the detected ball bounces. For this simulation we set the boundary detection error of our solver to 1e-14 which, for this particular simulation yields a more accurate solutions than ODE45.

We want to call attention to three aspects shown in this figure: The first one is that the time elapsed until the detection of the first bounce is mainly due to the memory allocation tasks. The second one is that after the first two integration steps the "actual" time between two consecutive bounces is smaller than the integrators-attempted step, so most of the computing effort is due to the boundary detection algorithms. The third, and the most important aspect, is that the solid line corresponding to the time elapsed by our solver goes under the doted one corresponding to ODE45, which means that for this particular application our solver is faster than ODE45.

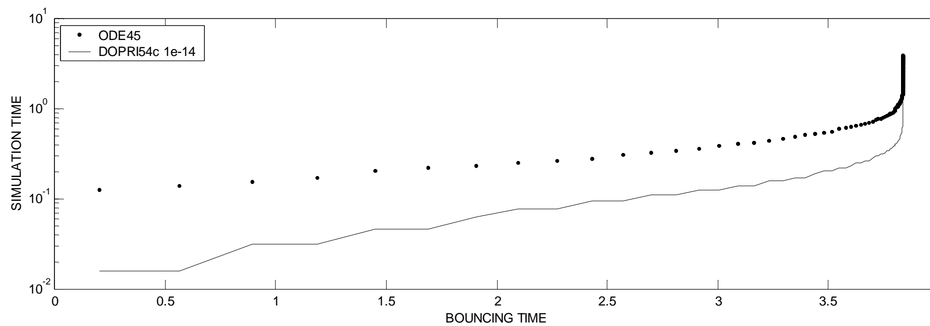

Figure 8 Bouncing ball example simulation time elapsed

Figure 9 shows the difference between the analytically-computed times corresponding to the first 200 bounces, and the corresponding times in the numerical solution. This difference is computed for four simulation runs: one with ODE45 and 3 others with DOPRI54c setting the boundary-detection-error bound to 1e-12, 1e-13 and 1e-14 respectively. As shown in Figure 9, the errors from the first two runs are quite similar, and notably bigger than those in the last two runs.

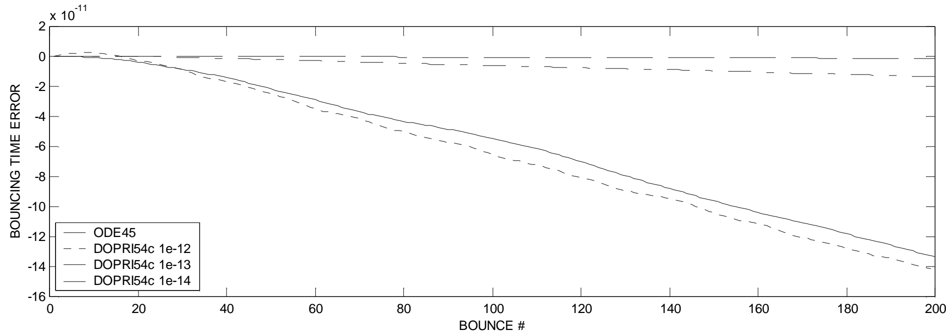

Figure 9 Bounce time error

It is important to note that the elapsed-simulation times displayed in Figure 8 correspond to the first and fourth simulation runs from Figure 9, which implies that for this particular application, our solver not only is faster, but also about two orders of magnitude more accurate.

4.3.2 Bouncing ball example with DOPRI853

Figure 10 shows the simulation result computed with DOPRI853 when setting the boundary-detection-error bound to 1e-14. As can be seen the results are similar to those obtained with DOPRI54c.

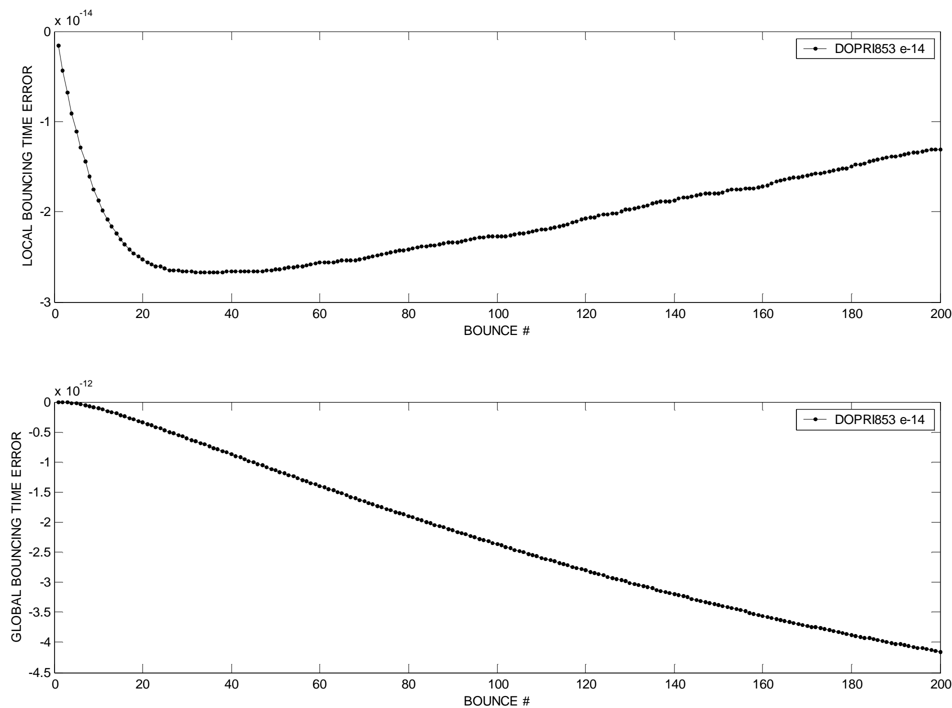

Figure 10 Bouncing ball example with DOPRI853

4.4 Ramp metering control example

In this section we first obtain a control function for the RMCP detailed in Chapter 2 and then we show some results obtained from simulating the resulting system.

Control function derivation

In Chapter 2, we explained that for the RMCP we require a positive semi-definite control action u(t). Here, we derive an expression for the difference between the desired system state and the actual state. Then, we use it to obtain a control action.

Equation (2.6) represents the state variables for the RMCP:

$$x(t) = \begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = \begin{pmatrix} \rho(t) - \frac{\rho_{\max}}{2} \\ s(t) \end{pmatrix}, \tag{2.6}$$

and Equation (2.7) represents the derivatives of these states:

$$\begin{aligned} &\frac{dx_1(t)}{dt} = \frac{q_{in}(t) - q_{out}(t) + u(t)}{L} \\ &\frac{dx_2(t)}{dt} = qs_{in}(t) - u(t) \\ &t_0 \le t \le t_f . \end{aligned} \tag{2.7}$$

From Equation (2.6), we can write an expression for the error:

$$e(t) = \left| w_1 x_1(t) + w_2 x_2(t) \right| \qquad -\frac{\rho_{\max}}{2} \le x_1 \le \frac{\rho_{\max}}{2}, 0 \le x_2, \tag{4.4}$$

where w_1 and w_2 are positive weights. For values of x_1 greater than zero, we can write the error derivative using Equation (2.7) to obtain:

$$\frac{de(t)}{dt} = w_1 \frac{q_{in}(t) - q_{out}(t) + u(t)}{L} + w_2 \left(qs_{in}(t) - u(t) \right), \tag{4.5}$$

now, in order to minimize this error, we take:

$$\frac{de(t)}{dt} = -k \cdot e(t)$$
$$\frac{de(t)}{dt} = -k\left(w_1 x_1(t) + w_2 x_2(t)\right), \tag{4.6}$$

and replacing the error derivative:

$$w_1 \frac{q_{in}(t) - q_{out}(t) + u(t)}{L} + w_2\left(qs_{in}(t) - u(t)\right) = -k\left(w_1 x_1(t) + w_2 x_2(t)\right), \tag{4.7}$$

solving now for $u(t)$, we obtain:

$$u^{+}(t) = \frac{-k\left(w_1 x_1(t) + w_2 x_2(t)\right) - \frac{w_1}{L} q_{in}(t) + \frac{w_1}{L} q_{out}(t) - w_2 qs_{in}(t)}{\frac{w_1}{L} - w_2}, \tag{4.8}$$

where $u^{+}(t)$ denotes the control action when x_1 is greater than zero. Similarly we can obtain the control action $u^{-}(t)$ for x_1 smaller than zero:

$$u^{-}(t) = \frac{k\left(-w_1 x_1(t) + w_2 x_2(t)\right) - \frac{w_1}{L} q_{in}(t) + \frac{w_1}{L} q_{out}(t) + w_2 qs_{in}(t)}{\frac{w_1}{L} + w_2}. \tag{4.9}$$

Then the control action result:

$$u(t) = \begin{cases} \dfrac{-k\left(w_1 x_1(t) + w_2 x_2(t)\right) - \frac{w_1}{L} q_{in}(t) + \frac{w_1}{L} q_{out}(t) - w_2 qs_{in}(t)}{\frac{w_1}{L} - w_2}, x_1 \geq 0 \\ \dfrac{k\left(-w_1 x_1(t) + w_2 x_2(t)\right) - \frac{w_1}{L} q_{in}(t) + \frac{w_1}{L} q_{out}(t) + w_2 qs_{in}(t)}{\frac{w_1}{L} + w_2}, x_1 < 0 \end{cases} \tag{4.10}$$

then, plugging Equation (4.10) into (2.7) we can run a simulation using the *HA* schemata explained under heading 2.2.3.

Simulation results

In this section we show the simulation outcome obtained using equations (2.8), (2.9), and (4.10) with initial conditions:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -\dfrac{\rho_{\max}}{4} \\ 1 \end{pmatrix}, \qquad \rho_{\max} = 2. \tag{4.11}$$

These initial conditions mimic a situation where the car density in the highway is low, and the queue length in the ramp is rather long. Assuming flow inputs q_{in} and qs_{in} as Figure 11 shows with a dashed line, and running the simulation we obtain the values shown with solid lines in Figure 11 and Figure 12.

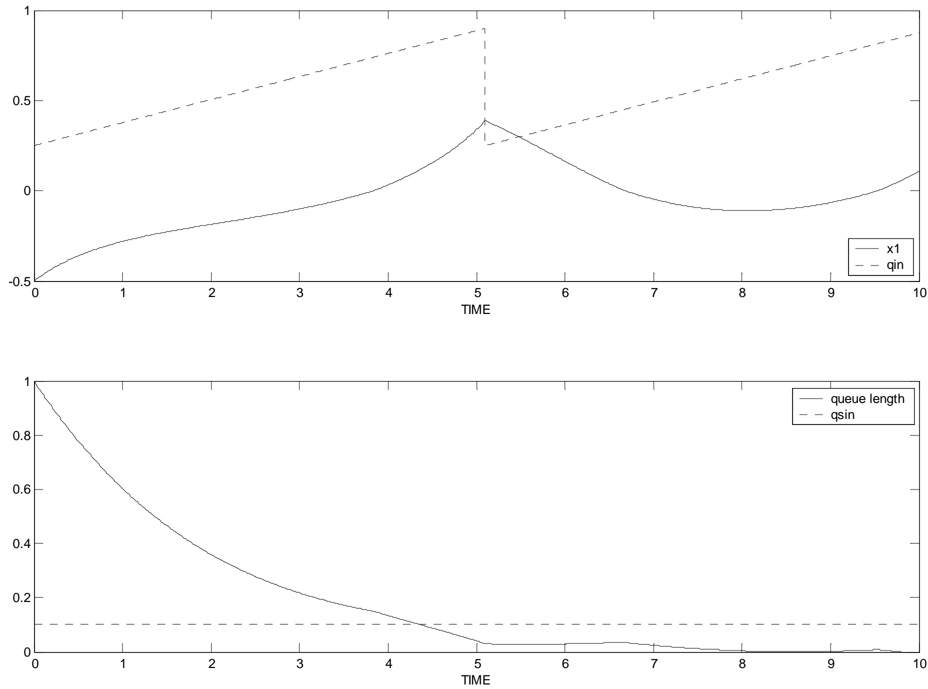

Figure 11 RMCP states and inputs

Note that in Figure 11 the state x_1 grows from it's initial value up to a maximum at time equal to five, and then stays around zero. On the other hand, x_2, the ramp queue length, decreases rapidly and then remains close to zero. Following, Figure 12 shows the control action $u(t)$ and the traffic going out at the end of the freeway segment. In the control action $u(t)$ note four discontinuities. Also note, that the first, third and fourth correspond to x_1 equaling zero and the second one to the discontinuity in the incoming flow.

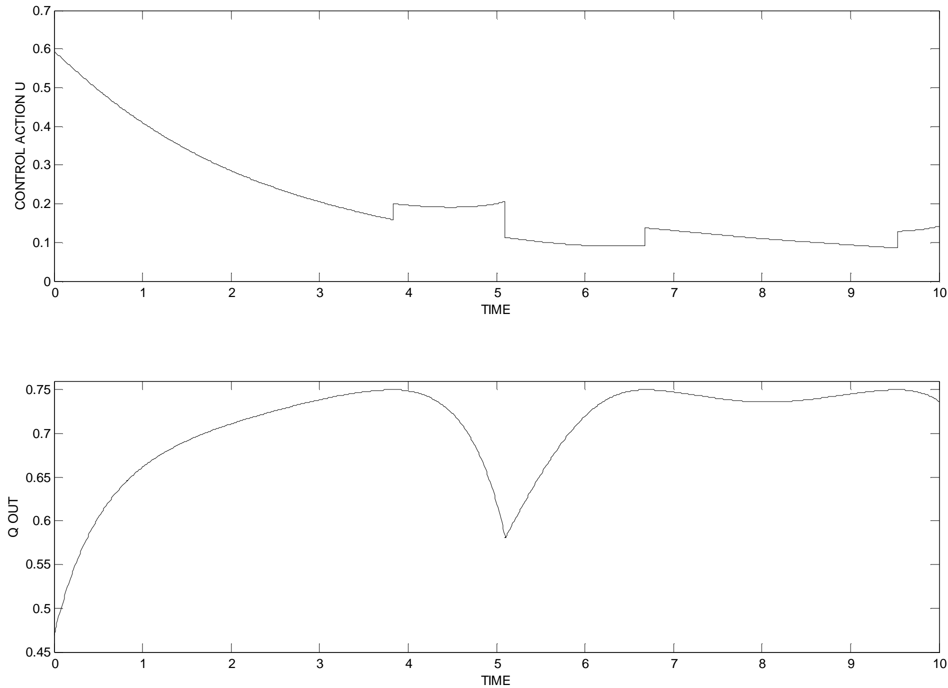

Figure 12 RMCP control action and main traffic flow

4.6 References

[1] L. Shampine, M Reichelt. The Matlab Ode Suite, SIAM J. Sci. Comput., 18 1-22, 1997.

[2] J. Dormand, P. Prince. A family of embedded Runge-Kutta formulae. *Journal of computational and applied Mathematics.* 6, 19-26. 1980.

[3] S. Papakostas, CH. Tsitouras, Highly continuous interpolants for one step ODE solvers and their application to Runge-Kutta Methods, SIAM J. Numer. Anal., v. 34, No. 1, pp. 22-47, 1997.

[4] E. Hairer, S.P. Norsett and G. Wanner, Solving ordinary differential equations I, non-stiff problems, 2nd edition, Springer Series in Computational Mathematics, Springer-Verlag. 1993.

[5] The Mathworks, Inc. Using Matlab. 2000.

Chapter 5 Final considerations and Future work

This chapter presents some final considerations regarding our solvers and an outline for future work on the topic.

5.1 *Final considerations*

In this thesis, we presented a solver to handle constrained-state-space ODEs. This solver locates the points where any of the states go outside of the constraining set, and then, transfers the control from the continuous time ODE to the function governing the behavior of the system on the boundaries of the constrained set. For our solver we implemented state-of-the-art high order integrations methods, specially suited for complex right-hand-sided ODEs and low local-truncations-errors. We also implemented continuous extensions for each of the integrations methods, and a zero crossing detection algorithm. In Table 3, we show some key features of the most well known ODE solvers and hybrid systems packages. It is important to note that the only solver designed to handle hybrid systems, which includes an 8th order solver, is the one we presented in this thesis.

Feature	DVERK (IMSL)	RKSUITE	Matlab ODE Simulink	Ptolemy II 2.0.1	Our's
Pair(s)	(5,6)	(3,2), (5,4), (8,7)	(5,1)s, (2,3)s (3,2), (5,4)	(2,3)	(5,4), (8,5,3)
Dense output	Hit t_{out}	(3,2): interp. (5,4): interp. (8,7): hit t_{out}	(3,2): interp. (5,4): interp.	At least hit t_{out}	(5,4): interp. (8,7): interp.

Table 3 Feature Comparisons[†]

[†] We borrow the first three columns of this table from [2].

5.2 Future work

At this point in the research, we are facing two different research paths: The first one points toward getting a complete ODE integration suite. Working in this direction will require to:

- improve the continuous extensions up to same approximation order as the ODE solver, doing the required extra right hand side evaluations.
- add lower order ODE solver to the suite as well as solvers for stiff problems.
- include the work by Calvo *et. al.* [3] to handle a wider range of problems.
- dedicate lots of time to debugging.

The other path of research, the one we are interested the most, points to improve the solver by:

- finding a cheap way to analyze the behavior of the constraining-set boundaries so that we do not have to scan slowly changing areas.
- switching automatically from ODE853, to ODE54, and to ODE23, when, due to Zeno behavior, the step size falls low enough so that the shift is worth it.
- finding a better approximation for the bound in the Zeno effect relaying step size control algorithm.
- including the work by Calvo *et. al.* [3] to handle a wider range of problems.

This second path of research requires keeping the solver in development stage for at least another semester.

5.3 References

[1] L. Shampine, M Reichelt. The Matlab Ode Suite, SIAM J. Sci. Comput., 18 1-22, 1997.

[2] W. Enright, D. Higham, B. Owren, P. Sharp. A survey of the explicit Runge-Kutta method. TR 291/94, Department of Computer Science, University of Toronto, 1994.

[3] M. Calvo, J. Montijano, L. Randez. On the numerical solution of IVPs with discontinuities by adaptive Runge-Kutta codes. 2001.

Index

Bouncing ball example ... vi, 17, 39, 40, 41
boundary-detection-error ... 40, 41
Boundary-detection-error ... 18
C ... 22, 34, 36, 51, 52, 53, 54, 55, 56, 59
Calvo ... 8, 9, 47
Dense output ... 46
discontinuity ... 8, 9, 16, 44, 47
DOPRI5 ... 27, 28, 36
DOPRI54 ... 36, 37, 38
DOPRI8 ... 27, 28
DOPRI853 ... vi, vii, 36, 38, 39, 40, 41
Dormand ... vii, 9, 22, 23, 25, 26, 27, 28, 33, 34, 36, 45, 52, 53, 54, 56, 59
Dormand-Prince ... vii, 9, 23, 25, 26, 27, 28, 52, 54, 56
DVERK ... 46
Fehlberg ... 23
HA ... *See* hybrid
Hybrid ... vii, 14, 15, 22
IMSL ... 34, 46
Lipschitz ... vii, 20
Matlab ... 8, 27, 34, 36, 45, 46, 47
pendulum ... 36, 37, 38
Ptolemy ... 8, 9, 46
Reichelt ... vii, 8, 27, 34, 45, 47
RKSUITE ... 46
RMCP ... vi, vii, 10, 11, 12, 13, 42, 44, 45
Root-finding
  Illinois ... vi, 31, 32, 35, 36

Linear Interpolation 31
Runge-Kutta .... vi, vii, 9, 17, 22, 23, 25, 26, 27, 28, 34, 45, 47
Shampine .... vii, 8, 9, 22, 27, 34, 45, 47, 50
Skorokhod .... vii, 13, 14, 21, 22
Taylor expansion 24, 25
Thompson 8, 9
Zeno .... ii, 9, 13, 14, 16, 23, 32, 35, 47

Appendix A

Dormand and Prince solver of orders 5 and 4 with continuous extension:

```
function [tout,xout]=dopri54(FUN,tspan,x0,tol)

hmax = (tspan(2) - tspan(1))/2.5;
[A,B4,B5,C]=setcoefs;
t0 = tspan(1);
tfinal = tspan(2);
t = t0;
hmin = (tfinal - t)/1e20;
h = (tfinal - t)/100; % initial step size guess
x = x0(:);            % ensure x is a column vector

Nstates = size(x,1);                        % Number of states
NEstSteps = (tspan(2)-tspan(1))*1e3;  % Estimated number of steps.
tout = zeros(NEstSteps ,1);               % pre-allocating memory
xout = zeros(NEstSteps ,Nstates);         % pre-allocating memory
StepN = 1;
tout(StepN) = t;                          % first output time
xout(StepN,:) = x.';                      % first output solution = IC's

k = x*zeros(1,7);                           % pre-allocating memory
k(:,1)=feval(FUN,t,x);                    % First RHS evaluation...
while (t < tfinal) & (h >= hmin),
   if t + h > tfinal, h = tfinal - t; end

   for j = 1:6,                             % RHS evaluations here!
      k(:,j+1) = feval(FUN, t+C(j+1)*h, x+h*k(:,1:j)*A(j+1,1:j)');
   end
   xl=x + h* (k*B4);                    %  4th order approximation
   xh=x + h*(k*B5);                     %  5th order approximation
   gamma1 = xh - xl;                        %  local truncation error estimation (vector
form)
   delta = norm(gamma1,'inf');       %  local truncation error estimation (absolute
value)
   tau = tol*max(norm(x,'inf'),1.0); %  maximum local truncation error.
   if (delta<=tau),
      StepN=StepN+1;
      t=t+h;
      x = xh;                               %Local extrapolation.
      tout(StepN) = t;
      xout(StepN,:) = x.';
      k(:,1)=k(:,7);                      %FSAL
   else                                     % if (delta<=tau)
      %Do I want to count the number of rejected steps.... no, I don't think so!
   end                                      % if (delta<=tau)

                                            %step size update block down
   if (delta==0.0),                         %this takes care of steady state solutions
      delta = 1e-16;                        %this takes care of steady state solutions
   end % if (delta==0.0)                  %this takes care of steady state solutions
   h = min(hmax, 0.8*h*(tau/delta)^(1/6));   %step size update block up

end % while (t < tfinal) & (h >= hmin)

if (StepN<NEstSteps),      %this deletes unnecessary zeros....
   tout = tout(1:StepN);
   xout = xout(1:StepN,:);
end % if (Nsteps<NEstSteps),

function [A,B4,B5,C]=setcoefs;
% Use the Dormand-Prince 4(5) coefficients:
A(1,1)=0; A(2,1)=1/5; A(3,1)=3/40; A(3,2)=9/40; A(4,1)=44/45; A(4,2)=-56/15; A(4,3)=32/9;
A(5,1)=19372/6561; A(5,2)=-25360/2187; A(5,3)=64448/6561; A(5,4)=-212/729;
A(6,1)=9017/3168;
A(6,2)=-355/33; A(6,3)=46732/5247; A(6,4)=49/176; A(6,5)=-5103/18656;A(7,1)=35/384;
```

```
A(7,2)=0; A(7,3)=500/1113; A(7,4)=125/192; A(7,5)=-2187/6784; A(7,6)=11/84;
% 4th order coefficients
B4(1,1)=5179/57600; B4(2,1)=0; B4(3,1)=7571/16695; B4(4,1)=393/640; B4(5,1)=-
92097/339200;
B4(6,1)=187/2100; B4(7,1)=1/40;
% 5th order coefficients
B5(1,1)=35/384; B5(2,1)=0; B5(3,1)=500/1113; B5(4,1)=125/192; B5(5,1)=-2187/6784;
B5(6,1)=11/84; B5(7,1)=0;

for i=1:7,
   C(i)=sum(A(i,:)); %time coefficients.
end
```

Dormand and Prince solver of orders 5 and 4 with continuous extension:

```
function [tout,xout]=dopri54d(FUN,tspan,x0,tol,meshsize)

hmax = (tspan(2) - tspan(1))/2.5;
[A,B4,B5,C]=setcoefs;
t0 = tspan(1);
tfinal = tspan(2);
t = t0;
hmin = (tfinal - t)/1e20;
h = (tfinal - t)/100; % initial step size guess
x = x0(:);            % ensure x is a column vector

Nstates = size(x,1);                         % Number of states
NEstSteps = (tspan(2)-tspan(1))*1e3;  % Estimated number of steps.
tout = zeros(NEstSteps ,1);               % pre-allocating memory
xout = zeros(NEstSteps ,Nstates);         % pre-allocating memory
StepN = 1;
tout(StepN) = t;                          % first output time
xout(StepN,:) = x.';                      % first output solution = IC's

k = x*zeros(1,7);                            % pre-allocating memory
k(:,1)=feval(FUN,t,x);                    % First RHS evaluation...
while (t < tfinal) & (h >= hmin),
   if t + h > tfinal, h = tfinal - t; end
   for j = 1:6,                              % RHS evaluations here!
      k(:,j+1) = feval(FUN, t+C(j+1)*h, x+h*k(:,1:j)*A(j+1,1:j)');
   end
   xl=x + h* (k*B4);                    %  4th order approximation
   xh=x + h*(k*B5);                        %  5th order approximation
   gamma1 = xh - xl;                        %  local truncation error estimation (vector
form)
   delta = norm(gamma1,'inf');       %  local truncation error estimation (absolute
value)
   tau = tol*max(norm(x,'inf'),1.0); %  maximum local truncation error.
   if (delta<=tau),                         % Accepted step
      ninterp=ceil(h/meshsize);
      for i=1:ninterp,
         sigma=i/ninterp;
         [yn]=interpolant(k,t,h,x,sigma);
         StepN=StepN+1;
         T=t+h*sigma;
         X = yn;                               %Local extrapolation.
         tout(StepN) = T;
         xout(StepN,:) = X.';
      end
      t=t+h;
      x = xh;                                %Local extrapolation.
      k(:,1)=k(:,7);                      %FSAL
   else                                      % if (delta<=tau)
      %Do I want to count the number of rejected steps.... no, I don't think so!
   end                                       % if (delta<=tau)

   %step size update block down
   if (delta==0.0),                          %this takes care of steady state solutions
      delta = 1e-16;                         %this takes care of steady state solutions
   end % if (delta==0.0)                  %this takes care of steady state solutions
```

```
   h = min(hmax, 0.8*h*(tau/delta)^(1/6));   %step size update block up

end % while (t < tfinal) & (h >= hmin)

if (StepN<NEstSteps),     %this deletes unnecessary zeros....
   tout = tout(1:StepN);
   xout = xout(1:StepN,:);
end % if (Nsteps<NEstSteps),

function [A,B4,B5,C]=setcoefs;
% Use the Dormand-Prince 4(5) coefficients:
A(1,1)=0; A(2,1)=1/5; A(3,1)=3/40; A(3,2)=9/40; A(4,1)=44/45; A(4,2)=-56/15; A(4,3)=32/9;
A(5,1)=19372/6561; A(5,2)=-25360/2187; A(5,3)=64448/6561; A(5,4)=-212/729;
A(6,1)=9017/3168;
A(6,2)=-355/33; A(6,3)=46732/5247; A(6,4)=49/176; A(6,5)=-5103/18656;A(7,1)=35/384;
A(7,2)=0; A(7,3)=500/1113; A(7,4)=125/192; A(7,5)=-2187/6784; A(7,6)=11/84;
% 4th order coefficients
B4(1,1)=5179/57600; B4(2,1)=0; B4(3,1)=7571/16695; B4(4,1)=393/640; B4(5,1)=-
92097/339200;
B4(6,1)=187/2100; B4(7,1)=1/40;
% 5th order coefficients
B5(1,1)=35/384; B5(2,1)=0; B5(3,1)=500/1113; B5(4,1)=125/192; B5(5,1)=-2187/6784;
B5(6,1)=11/84; B5(7,1)=0;

for i=1:7,
   C(i)=sum(A(i,:)); %time coefficients.
end

function [yn]=interpolant(K,t,h,y,s)
%continuous extensions file
b(1,1) = s*(1+s*(-1337/480+s*(1039/360+s*(-1163/1152))));
b(2,1) = 0;
b(3,1) = 100*s*s*(1054/9275+s*(-4682/27825+s*(379/5565)))/3;
b(4,1) = -5*s*s*(27/40+s*(-9/5+s*(83/96)))/2;
b(5,1) = 18225*s*s*(-3/250+s*(22/375+s*(-37/600)))/848;
b(6,1) = -22*s*s*(-3/10+s*(29/30+s*(-17/24)))/7;
b(7,1) = 0;
yn = y + h*(K*b);
```

Appendix B

In this appendix we attach the solver code for DOPRI54C and a sample of the problem specific functions as explained in Section 4.3.

```
function [tout,xout]=mysolver54(FUN,tspan,x0,tol,pmeshsize,bmeshsize)
hmax = (tspan(2) - tspan(1))/2.5;
[A,B4,B5,C]=setcoefs;
t0 = tspan(1);
tfinal = tspan(2);
t = t0;
hmin = (tfinal - t)/1e20;
h = (tfinal - t)/100; % initial step size guess
x = x0(:);            % ensure x is a column vector
boundary = 0;           % boundary flag


Nstates = size(x,1);                             % Number of states
NEstSteps = (tspan(2)-tspan(1))*1e3;  % Estimated number of steps.
tout = zeros(NEstSteps ,1);                % preallocating memory
xout = zeros(NEstSteps ,Nstates);          % preallocating memory
StepN = 1;
tout(StepN) = t;                           % first output time
xout(StepN,:) = x.';                       % first output solution = IC's

k = x*zeros(1,7);                                % preallocating memory
k(:,1)=feval(FUN,t,x);                     % First RHS evaluation...
while (t < tfinal) & (h >= hmin),
   if t + h > tfinal, h = tfinal - t; end
   for j = 1:6,                                  % RHS evaluations here!
      k(:,j+1) = feval(FUN, t+C(j+1)*h, x+h*k(:,1:j)*A(j+1,1:j)');
   end
   xl=x + h* (k*B4);                    %  4th order aproximation
   xh=x + h*(k*B5);                        %  5th order aproximation
   gamma1 = xh - xl;                        %  local truncation error estimation (vector
form)
   delta = norm(gamma1,'inf');        %  local truncation error estimation (absolute
value)
   tau = tol*max(norm(x,'inf'),1.0); %  maximum local truncation error.
   if (delta<=tau),                         % Accepted step
      ninterp=ceil(h/bmeshsize);
      plotinginterp=ceil(h/pmeshsize);
      if plotinginterp>ninterp, plotinginterp=ninterp; end
      I1=ceil(ninterp/plotinginterp);
      I2=1;
      for i=1:ninterp,
         sigma=i/ninterp;
         [yn]=interpolant(k,t,h,x,sigma);
         [bln,bun]=b(t+h*sigma,yn);
         if (sum(bln>bun))
            disp('the space of some of the states is zero')
            StepN=StepN+1;
            tout(StepN) = t+(h*sigma);
            xout(StepN,:) = yn.';
            if (StepN<NEstSteps),         %this deletes unnecesary zeros....
               tout = tout(1:StepN);
               xout = xout(1:StepN,:);
            end % if (Nsteps<NEstSteps),
            return
         end
         if ((sum(yn<bln))|((sum(yn>bun))&(sigma~=1)))
            %root finding down
            [T,yn]=illinois(t,x,yn,h,tol,k);
            %root finding up
```

```
            StepN = StepN + 1;
            tout(StepN ) = T;
            xout(StepN,:) = yn.';
            %function B  down
            [t,x]=fb(T,yn);
            %function B  up
            k(:,7)=feval(FUN,t,x); %  it will be to the first stage later on.
            boundary = 1;
            break;
         end %end         if (sum(x5<bln)|sum(x5>bun))
         if I2>I1
            T=t+h*sigma;
            StepN = StepN + 1;
            tout(StepN ) = T;
            xout(StepN,:) = yn.';
            I2=0;
         end
         I2=I2+1;
      end
      if ~boundary,
         t=t+h;
         x = xh;                                  %Local extrapolation.
      end

      k(:,1)=k(:,7);                          %FSAL
   else                                       % if (delta<=tau)
      %Do I want to count the number of rejected steps.... no, I don't think so!
   end                                        % if (delta<=tau)

   %step size update block down
   if (delta==0.0),                           %this takes care of steady state solutions
      delta = 1e-16;                          %this takes care of steady state solutions
   end % if (delta==0.0)                  %this takes care of steady state solutions
   if boundary
      h = min([hmax, 0.8*h*(tau/delta)^(1/6),h*sigma]);   %step size update block up
      boundary = 0;                                      %reset boundary flag.
   else
      h = min(hmax, 0.8*h*(tau/delta)^(1/6));   %step size update block up
   end


end % while (t < tfinal) & (h >= hmin)

if (StepN<NEstSteps),     %this deletes unnecessary zeros....
   tout = tout(1:StepN);
   xout = xout(1:StepN,:);
end % if (Nsteps<NEstSteps),

function [A,B4,B5,C]=setcoefs;
% Use the Dormand-Prince 4(5) coefficients:
A(1,1)=0; A(2,1)=1/5; A(3,1)=3/40; A(3,2)=9/40; A(4,1)=44/45; A(4,2)=-56/15; A(4,3)=32/9;
A(5,1)=19372/6561; A(5,2)=-25360/2187; A(5,3)=64448/6561; A(5,4)=-212/729;
A(6,1)=9017/3168;
A(6,2)=-355/33; A(6,3)=46732/5247; A(6,4)=49/176; A(6,5)=-5103/18656;A(7,1)=35/384;
A(7,2)=0; A(7,3)=500/1113; A(7,4)=125/192; A(7,5)=-2187/6784; A(7,6)=11/84;
% 4th order coefficients
B4(1,1)=5179/57600; B4(2,1)=0; B4(3,1)=7571/16695; B4(4,1)=393/640; B4(5,1)=-
92097/339200;
B4(6,1)=187/2100; B4(7,1)=1/40;
% 5th order coefficients
B5(1,1)=35/384; B5(2,1)=0; B5(3,1)=500/1113; B5(4,1)=125/192; B5(5,1)=-2187/6784;
B5(6,1)=11/84; B5(7,1)=0;
for i=1:7,
   C(i)=sum(A(i,:)); %time coeficients.
end


function [yn]=interpolant(K,t,h,y,s)
b(1,1) = s*(1+s*(-1337/480+s*(1039/360+s*(-1163/1152))));
b(2,1) = 0;
b(3,1) = 100*s*s*(1054/9275+s*(-4682/27825+s*(379/5565)))/3;
```

```
b(4,1) = -5*s*s*(27/40+s*(-9/5+s*(83/96)))/2;
b(5,1) = 18225*s*s*(-3/250+s*(22/375+s*(-37/600)))/848;
b(6,1) = -22*s*s*(-3/10+s*(29/30+s*(-17/24)))/7;
b(7,1) = 0;
yn = y + h*(K*b);



function [t,y]=illinois(t,y,yn,h,TOL,K)
%[t,y]=illinois(t,y,yn,h,TOL,K)
%[crosing time,y value at crossing point]=illinois(initial time, initial y value, end of
the step value, step size....))
[bln,bun]=b(t+h,yn);
for a=1:length(bln)
   if (yn(a)<bln(a)| yn(a)>bun(a))
      [Tlu(a),ylu(:,a)]=illinoisscallar(t,y,yn,h,TOL,K,a);
   else
      Tlu(a)=t+h;ylu(:,a)=yn;
   end
end
[t,a]=min(Tlu);
y=ylu(:,a);
function [t,y]=illinoisscallar(t,y,yn,h,TOL,K,a,upperorlower)
T=t; Y=y; H=h; %i need fixed values for the interpolant....

[bln,bun]=b(t+h,yn);
[bl,bu]=b(t,y);

if (yn(a)<bln(a))
   ftl=y(a)-bl(a);
   fthl=yn(a)-bln(a);
   for b=0:1e5,
      if ((ftl-fthl)~=0)
         sigma=ftl/(ftl-fthl); %sigma corresponding with the next approximation
      else
         return
      end
      sigmainterp=((t+h*sigma)-T)/H;
      [ynn]=interpolant(K,T,H,Y,sigmainterp); %y corresponding to the approximation
      [bln,bun]=b(t+sigma*h,ynn);
      if (ynn(a)<bln(a))
         yn=ynn;
         h=sigma*h;
         [bln,bun]=b(t+h,yn);
         ftl=ftl/2;
         fthl=yn(a)-bln(a);
      else
         y=ynn;
         t=t+sigma*h;
         h=(1-sigma)*h;
         [bl,bu]=b(t,y);
         ftl=y(a)-bl(a);
         fthl=fthl/2;
      end
      if h<TOL
         return
      end
   end
else %it means that  yn(a)>bun(a) so I run the algorithm on it.
   ftu=y(a)-bu(a);
   fthu=yn(a)-bun(a);
   for b=0:1e5,
      if ((ftu-fthu)~=0)
         sigma=ftu/(ftu-fthu); %sigma corresponding with the next approximation
      else
         return
      end
      sigma=ftu/(ftu-fthu); %sigma corresponding with the next approximation
      sigmainterp=((t+h*sigma)-T)/H;
      [ynn]=interpolant(K,T,H,Y,sigmainterp); %y corresponding to the approximation
  %    figure(1)      Debugging plots
```

```
  %    hold on
  %    plot(T+H*sigmainterp,ynn(1),'.r')
  %    plot([t,t+h],[1,1]*0.3,'.b')
  %    pause
      [bln,bun]=b(t+sigma*h,ynn);
      if (ynn(a)>bun(a))
         yn=ynn;
         h=sigma*h;
         [bln,bun]=b(t+h,yn);
         ftu=ftu/2;
         fthu=yn(a)-bun(a);
      else
         y=ynn;
         t=t+sigma*h;
         h=(1-sigma)*h;
         [bl,bu]=b(t,y);
         ftu=y(a)-bu(a);
         fthu=fthu/2;
      end
      if h<TOL
         return
      end
   end
end


function [dy]=f(t,y)
%ODE right hand side
g=-9.8;
dy(1)=y(2);
dy(2)=g;


function [t,y]=fb(t,y)
y(1)=y(1);
y(2)=-y(2);
t=t;


function [bl,bu]=b(t,y)
bl=y;
bu=y;
bl(1)=0;%t
bl(2)=-10;
bu(1)=1-t^2;
bu(2)=10;
```

Appendix C

In this appendix we attach the solver code for DOPRI853. This solver is based on the order 8 implementation of Dormand and Prince's method by E. Hairer & G. Wanner [1].

[1] E. Hairer, S.P. Norsett and G. Wanner, Solving ordinary differential equations I, non-stiff problems, 2nd edition, Springer Series in Computational Mathematics, Springer-Verlag. 1993.

```
void illinois(double *yn,double *t, double told,int state, int
nstates)
{
   double yu;
   double yl;
   double dt;
   double t1;
   double ynew;
   if ((boundary(  yn, *t,state))!=0) return(0);
   yu=contd8(state,*t);
   yl=yn[0];
   dt=*t-told;

   do
   {
      t1=*t+yl/(yl-yu)*dt;
      ynew=contd8(state,t1);
      for (int a=0;a<nstates,a++)
         yn[a]=contd8(a,t1);
      if (~boundary(yn, told,state))
      {
         *t=t1;
         yl=ynew;
         yu=yu/2;
      }
      else
      {
         told=t1;
         yu=ynew;
         yl=yl/2;
      }

   }while(*t-told>1e-14);
   *t=t1;
   ynew=contd8(0,t1);
   yn[0]=ynew;
```

```
   yn[1]=contd8(1,t1);
   return (1);
}

int boundary( double *y,double t,int state)
{
   int boundaryreached;
   if (y[0]<0) boundaryreached=1; else boundaryreached=0;
   return (boundaryreached);
}
```

Vita

Marcos Donolo was born in Río Cuarto, Córdoba, Argentina, on July 7th, 1975. He received his B.S. degree in Electrical Engineering at the Facultad de Ingeniería from the Universidad Nacional de Río Cuarto in the year 2000. Since 2001, he has been an MS. student in the Electric Engineering Department in Virginia Polytechnic Institute and State University, Blacksburg, Virginia. His interests are hybrid control systems, electric power systems analysis, software development, Visual C++, numerical methods, and artificial intelligence.