Constructing High-Order Runge-Kutta Methods with Embedded Strong-Stability-Preserving Pairs

by

Colin Barr Macdonald

B.Sc., Acadia University, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN THE DEPARTMENT
OF
MATHEMATICS

SIMON FRASER UNIVERSITY

August 2003

APPROVAL

Name: Colin Barr Macdonald

Degree: Master of Science

Title of thesis: Constructing High-Order Runge-Kutta Methods with Embedded Strong-Stability-Preserving Pairs

Examining Committee: Dr. Mary Catherine Kropinski
Chair

Dr. Steve Ruuth
Senior Supervisor

Dr. Bob Russell
Supervisor

Dr. Jim Verner
Supervisor

Dr. Manfred Trummer
Internal/External Examiner

Date Approved:

Abstract

Runge-Kutta methods are one of the fundamental techniques in scientific computing. They are used to compute numerical solutions in a step-by-step fashion for ordinary differential equations (ODEs) and also, via the *method of lines*, for partial differential equations (PDEs).

By sharing information, *embedded* Runge-Kutta methods execute two Runge-Kutta schemes simultaneously while incurring minimal additional cost. Traditionally this is done for the purpose of actively selecting step-sizes for error control. However, in this thesis, we suggest another possible use where the two schemes would be used in different regions of the spatial domain based on local properties of the solution. For example, the solutions of *hyperbolic conservation laws* contain both smooth and non-smooth features. Strong-stability-preserving (SSP) Runge-Kutta schemes are particularly well suited for use near non-smooth or discontinuous behavior such as shocks because they have a *nonlinear stability* property that helps them prevent spurious oscillations (such as the Gibb's phenomenon) and other non-physical behaviour. Unfortunately, SSP schemes have limitations that make them expensive or inappropriate in smooth regions of the solution where a high order of accuracy is desired. In these regions, schemes based on "classical" linear stability analysis are likely a better choice. This motivates the use of high-order Runge-Kutta schemes with embedded SSP pairs, where the higher-order scheme, based on linear stability analysis, would be used to evolve smooth regions of the solution. The lower-order SSP scheme would be used near shocks or other discontinuities to help prevent spurious oscillations. This thesis explores the construction of these new methods.

Following a review of Runge-Kutta methods, strong-stability, and other related concepts, the proprietary BARON optimization software is introduced as a powerful tool for deriving optimal SSP schemes. Various Runge-Kutta methods with embedded SSP pairs are then constructed using a combination of BARON optimization and analytical techniques.

Acknowledgments

The research behind this thesis was supported financially by an NSERC Postgraduate scholarship and the C.D. Nelson Memorial scholarship.

I would like to thank my supervisor Dr. S. Ruuth for suggesting what turned out to be a very interesting and challenging topic. His insights and encouragement were inspiring during this process. Special thanks also to Dr. J. Verner for assistance and discussions on several key points.

Finally, I owe a debt of gratitude to everyone who put up with my down-to-the-wire approach to deadlines.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The original motivation behind this thesis was to construct embedded Runge-Kutta methods for use in computing numerical solutions to hyperbolic conservation laws. The methods would use a high-order linearly stable Runge-Kutta scheme in smooth regions of the spatial domain and, in the vicinity of shocks or other discontinuities, switch to a lower-order scheme possessing a "nonlinear stability" property which would help prevent spurious oscillations and overshoots. The derivation of such a method turned out to be more challenging and interesting than was originally thought and, as such, this thesis has more to do with the *construction* of these embedded methods than it does with the original motivational example.

This chapter begins with an introduction to Runge-Kutta methods and linear stability. It then touches briefly on the topics related to the solution of hyperbolic conservation laws, including nonlinear stability and strong-stability-preserving Runge-Kutta schemes.

Finally, the chapter concludes with a discussion of linearly stable Runge-Kutta methods with embedded strong-stability-preserving Runge-Kutta schemes.

1.1 Runge-Kutta Methods

Runge-Kutta methods are a class of numerical methods for computing numerical solutions to the initial value problem (IVP) consisting of the ordinary differential equation (ODE)

$$\boldsymbol{U}' = \boldsymbol{F}(t, \boldsymbol{U}(t)), \tag{1.1a}$$

and the initial conditions

$$\boldsymbol{U}(t_0) = \boldsymbol{U}^0, \tag{1.1b}$$

where $\boldsymbol{U} \in \mathbb{R}^M$, $\boldsymbol{F} : \mathbb{R} \times \mathbb{R}^M \to \mathbb{R}^M$ and $t \in [t_0, t_f] \subset \mathbb{R}$. A Runge-Kutta method computes a numerical solution, $\boldsymbol{U}^n \approx \boldsymbol{U}(t_n)$, to (1.1) by taking time steps of size $h = \Delta t$ with $t_n = t_0 + nh$. For example, the simplest Runge-Kutta method is Euler's method or *Forward Euler* which computes

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + h\boldsymbol{F}(t_n, \boldsymbol{U}^n). \tag{1.2}$$

Forward Euler is an example of a 1-stage method, that is, $\boldsymbol{F}$ is evaluated once per time step. It is explicit in the sense that no system of equations must be solved to proceed from $\boldsymbol{U}^n$ to $\boldsymbol{U}^{n+1}$.

Although Forward Euler is simple to understand and easy to implement, the global error (the difference between $\boldsymbol{U}^N$ and $\boldsymbol{U}(t_f)$, where $t_f = t_0 + Nh$) is proportional to h. Heuristically speaking, one might expect around 10^6 steps to compute a solution accurate to 6 decimal places [HNW93]. Suppose however, that instead of (1.1), we have a quadrature problem

$$\tilde{\boldsymbol{U}}' = \tilde{\boldsymbol{F}}(t), \quad \tilde{\boldsymbol{U}}(t_0) = \tilde{\boldsymbol{U}}^0, \tag{1.3a}$$

which has the solution

$$\tilde{\boldsymbol{U}}(t_f) = \tilde{\boldsymbol{U}}(t_0) + \int_{t_0}^{t_f} \tilde{\boldsymbol{F}}(\bar{t})\mathrm{d}\bar{t}. \tag{1.3b}$$

Then highly accurate numerical solutions can be calculated using a s-stage quadrature formula (see, for example, [BF01])

$$\tilde{\boldsymbol{U}}^{n+1} = \tilde{\boldsymbol{U}}^n + h \sum_{j=1}^{s} b_j \tilde{\boldsymbol{F}}(t_n + c_j h), \tag{1.4}$$

where the c_j are the nodes or abscissae (typically $c_j \in [0, 1]$) and the b_j are the quadrature weights ($\sum_{j=1}^{s} b_j = 1$). Now reconsider problem (1.1) and note, that to extend the quadrature formula (1.4), we could use

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + h \sum_{j=1}^{s} b_j \boldsymbol{F}(t_n + c_j h, \overline{\boldsymbol{U}}^j), \tag{1.5a}$$

where $\overline{\boldsymbol{U}}^j$ is an estimate for $\boldsymbol{U}$ at the node point c_j. Explicit Runge-Kutta methods build these *stage estimates* recursively using

$$\overline{\boldsymbol{U}}^1 = \boldsymbol{U}^n, \tag{1.5b}$$

$$\overline{\boldsymbol{U}}^2 = \boldsymbol{U}^n + ha_{21}\boldsymbol{F}(t_n, \overline{\boldsymbol{U}}^1), \tag{1.5c}$$

$$\overline{\boldsymbol{U}}^3 = \boldsymbol{U}^n + h\left[a_{31}\boldsymbol{F}(t_n, \overline{\boldsymbol{U}}^1) + a_{32}\boldsymbol{F}(t_n + c_2 h, \overline{\boldsymbol{U}}^2)\right], \tag{1.5d}$$

$$\vdots$$

$$\overline{\boldsymbol{U}}^s = \boldsymbol{U}^n + h\sum_{k=1}^{s-1} a_{sk}\boldsymbol{F}(t_n + c_k h, \overline{\boldsymbol{U}}^k), \tag{1.5e}$$

where a_{jk} are the *stage weights* for $\overline{\boldsymbol{U}}^j$ and $\sum_{k=1}^{j-1} a_{jk} = c_k$. Note in (1.5c) that $\overline{\boldsymbol{U}}^2$ is obtained with a Forward Euler step of size ha_{21}. Although not immediately obvious from (1.5), an s-stage Runge-Kutta method requires exactly s evaluations of $\boldsymbol{F}$. To see how this is possible, consider an alternative formulation which lends itself well to implementation:

$$\boldsymbol{K}^1 = \boldsymbol{F}(t_n, \boldsymbol{U}^n), \tag{1.6a}$$

$$\boldsymbol{K}^2 = \boldsymbol{F}(t_n + c_2 h, \boldsymbol{U}^n + ha_{21}\boldsymbol{K}^1), \tag{1.6b}$$

$$\vdots$$

$$\boldsymbol{K}^s = \boldsymbol{F}\left(t_n + c_s h, \boldsymbol{U}^n + h(a_{s1}\boldsymbol{K}^1 + \cdots + a_{s,s-1}\boldsymbol{K}^{s-1})\right), \tag{1.6c}$$

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + h(b_1\boldsymbol{K}^1 + \cdots + b_s\boldsymbol{K}^s). \tag{1.6d}$$

This formulation also makes it obvious that a general s-stage Runge-Kutta scheme will require s temporary vectors for the $\boldsymbol{K}$'s; this can be a significant amount of storage for very large problems such as those resulting from the discretization of partial differential equations (PDEs) in three dimensions.

1.1.1 Butcher Tableaux

The node points c_j, weights b_k, and stage weights a_{jk} are often expressed in Butcher Tableau form using a matrix A, and s-vectors $\boldsymbol{b}$ and $\boldsymbol{c}$:

$$\begin{array}{c|c} \boldsymbol{c} & A \\ \hline & \boldsymbol{b}^T \end{array} = \begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s,1} & a_{s,2} & \ldots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \ldots & b_{s-1} & b_s \end{array} \tag{1.7a}$$

and, to reiterate, the corresponding s-stage Runge-Kutta method is

$$\overline{\boldsymbol{U}}^j = \boldsymbol{U}^n + h \sum_{k=1}^{j-1} a_{jk} \boldsymbol{F}(t_n + c_k h, \overline{\boldsymbol{U}}^k), \quad j = 1, \ldots, s \tag{1.7b}$$

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + h \sum_{j=1}^{s} b_j \boldsymbol{F}(t_n + c_j h, \overline{\boldsymbol{U}}^j). \tag{1.7c}$$

For example, the 2-stage Modified Euler method

$$\overline{\boldsymbol{U}}^1 = \boldsymbol{U}^n, \tag{1.8a}$$

$$\overline{\boldsymbol{U}}^2 = \boldsymbol{U}^n + h\boldsymbol{F}(t_n, \overline{\boldsymbol{U}}^1), \tag{1.8b}$$

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + h \left[\frac{1}{2} \boldsymbol{F}(t_n, \overline{\boldsymbol{U}}^1) + \frac{1}{2} \boldsymbol{F}(t_n + h, \overline{\boldsymbol{U}}^2) \right], \tag{1.8c}$$

has the Butcher tableau

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array}. \tag{1.8d}$$

The simple Forward Euler scheme (1.2) has the Butcher tableau

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}. \tag{1.9}$$

Here we are restricting our discussion to explicit Runge-Kutta methods; that is, methods where each $\overline{\boldsymbol{U}}^j$ can be calculated explicitly from the previous stage estimates $\overline{\boldsymbol{U}}^k$, $k = 1, \ldots, j-1$. In particular, this implies that A must be lower triangular. If A is not lower triangular, then the scheme is known as an implicit Runge-Kutta scheme and requires expensive system solves at each time step. For this thesis, we will concentrate on explicit Runge-Kutta methods and thus use the terms "Runge-Kutta method" and "explicit Runge-Kutta method" interchangeably.

1.1.2 α–β Notation

An alternative notation to Butcher tableaux is α–β notation where the Runge-Kutta method is broken down into a series of Forward Euler steps. The α–β notation uses two matrices

$$\alpha = \begin{bmatrix} \alpha_{10} & & & \\ \alpha_{20} & \alpha_{21} & & \\ \vdots & \vdots & \ddots & \\ \alpha_{s,0} & \alpha_{s,1} & \dots & \alpha_{s,s-1} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_{10} & & & \\ \beta_{20} & \beta_{21} & & \\ \vdots & \vdots & \ddots & \\ \beta_{s,0} & \beta_{s,1} & \dots & \beta_{s,s-1} \end{bmatrix}, \tag{1.10a}$$

and the Runge-Kutta scheme is

$$\widehat{\boldsymbol{U}}^0 = \boldsymbol{U}^n, \tag{1.10b}$$

$$\widehat{\boldsymbol{U}}^i = \sum_{k=0}^{i-1} \left[\alpha_{ik}\widehat{\boldsymbol{U}}^k + h\beta_{ik}\boldsymbol{F}(t_n + c_k h, \widehat{\boldsymbol{U}}^k)\right], \quad i = 1, \dots, s, \tag{1.10c}$$

$$\boldsymbol{U}^{n+1} = \widehat{\boldsymbol{U}}^s, \tag{1.10d}$$

where $\alpha_{ik} \geq 0$ and $\sum_{k=0}^{i-1} \alpha_{ik} = 1$.

Given a particular scheme in α–β notation, there is a unique corresponding Butcher notation. Following [SR02, RS02a], we define the intermediate κ matrix using the following recursive definition

$$\kappa_{ik} = \beta_{ik} + \sum_{j=k+1}^{i-1} \alpha_{ij}\kappa_{jk}. \tag{1.11a}$$

The κ_{ik} coefficients are related to the Butcher tableau coefficients by

$$a_{ik} = \kappa_{i-1,k-1}, \qquad k = 1, \dots, i-1, \quad i = 1, \dots, s-1, \tag{1.11b}$$

$$b_k = \kappa_{s,k-1}, \qquad k = 1, \dots, s. \tag{1.11c}$$

However, for a given Runge-Kutta method in Butcher notation, there is no unique conversion to α–β notation; instead there is a family of α–β representations. For example, consider the

Modified Euler scheme (1.8) which has the following one-parameter family of α–β notations

$$\alpha = \begin{pmatrix} 1 & \\ 1-\lambda & \lambda \end{pmatrix}, \qquad \beta = \begin{pmatrix} 1 & \\ \frac{1}{2}-\lambda & \frac{1}{2} \end{pmatrix}, \tag{1.12}$$

for $\lambda \in [0,1]$ (see [SR02]). The family of α–β representations are algebraically equivalent (see [SO88]) and should produce the same results up to roundoff errors. However, particular members of the family may be easier to implement, require less memory storage or expose certain stability restrictions.

1.1.3 Error and Order

There are typically two reasons for using an s-stage Runge-Kutta method with $s \geq 2$ over Forward Euler: improved accuracy and/or improved stability properties. Here we discuss error and the order of an s-stage Runge-Kutta scheme.

Definition 1.1 (Global Error) *Global error is simply the difference between the exact solution $\boldsymbol{U}(t_f)$ and the numerical approximation $\boldsymbol{U}^N$ measured in some norm. Specifically*

$$g.e. = ||\boldsymbol{U}(t_f) - \boldsymbol{U}^N||, \tag{1.13}$$

where $||\cdot||$ is typically the 1-norm, 2-norm, or ∞-norm.

Definition 1.2 (Local Truncation Error) *If we assume that $\boldsymbol{U}^{n-1}$ is exact, i.e., $\boldsymbol{U}^{n-1} = \boldsymbol{U}(t_{n-1})$, then the local truncation error is the error introduced by the single time step from t_{n-1} to t_n. The local truncation error is the vector*

$$l.t.e. = \boldsymbol{U}^n - \boldsymbol{U}(t_n), \tag{1.14}$$

although we are often interested in $||l.t.e.||$ for some norm.

A Runge-Kutta method is said to be order p if the global error is order p, that is, if

$$\text{g.e.} \leq \tilde{K}h^p, \tag{1.15}$$

for some constant $\tilde{K}$. For sufficiently smooth problems, the global error can be related to the local truncation error and this motivates the following definition.

Definition 1.3 (Order) *A Runge-Kutta method is order p if, for sufficiently smooth problems, the local truncation error is order $p+1$, that is, if*

$$||\text{l.t.e.}|| = Kh^{p+1} + \mathcal{O}\left(h^{p+2}\right), \tag{1.16}$$

for some constant K.

We will often use the notation RK (s,p) to refer to an s-stage, order-p Runge-Kutta scheme.

1.1.4 The Order Conditions

For a Runge-Kutta method to be of order p, it must satisfy certain *order conditions*. These conditions are based on matching leading terms of Taylor expansions of $\boldsymbol{U}(t_n + h)$ and of (1.7c) as a function of h (see [HNW93]). For example, a 2-stage, order-2 Runge-Kutta method satisfies the following order conditions

$$\begin{aligned} b_1 + b_2 &= 1, \qquad (\tau) \\ b_2 a_{21} &= \frac{1}{2}, \qquad (t_{21}) \end{aligned}$$

and a 3-stage order-3 Runge-Kutta method must satisfy

$$\begin{aligned} b_1 + b_2 + b_3 &= 1, \qquad (\tau) \\ b_2 a_{21} + b_3(a_{31} + a_{32}) &= \frac{1}{2}, \qquad (t_{21}) \\ b_2 a_{21}^2 + b_3(a_{31} + a_{32})^2 &= \frac{1}{3}, \qquad (t_{31}) \\ b_3 a_{32} a_{21} &= \frac{1}{6}. \qquad (t_{32}) \end{aligned}$$

As in [HNW93], we denote the order conditions with t_{qr} where r indexes the order conditions of order q (also, τ and t_{11} are used synonymously).

As seen in Table 1.1, the number of order conditions grows exponentially as order increases (see [HNW93]). Specifically, a 5^{th}-order Runge-Kutta method must satisfy the 17 order conditions shown in Table 1.2.

p	1	2	3	4	5	6	7	8	9	10
# of order-p conditions	1	1	2	4	9	20	48	115	286	719

Table 1.1: Number of order conditions up to order 10.

Order-1	τ	$\sum b_j = 1$
Order-2	t_{21}	$\sum b_j c_j = 1/2$
Order-3	t_{31}	$\sum b_j c_j^2 = 1/3$
	t_{32}	$\sum b_j a_{jk} c_k = 1/6$
Order-4	t_{41}	$\sum b_j c_k^3 = 1/4$
	t_{42}	$\sum b_j c_j a_{jk} c_k = 1/8$
	t_{43}	$\sum b_j a_{jk} c_k^2 = 1/12$
	t_{44}	$\sum b_j a_{jk} a_{kl} c_l = 1/24$
Order-5	t_{51}	$\sum b_j c_j^4 = 1/5$
	t_{52}	$\sum b_j c_j^2 a_{jk} c_k = 1/10$
	t_{53}	$\sum b_j c_j a_{jk} c_k^2 = 1/15$
	t_{54}	$\sum b_j c_j a_{jk} a_{kl} c_l = 1/30$
	t_{55}	$\sum b_j a_{jk} c_k a_{jl} c_l = 1/20$
	t_{56}	$\sum b_j a_{jk} c_k^3 = 1/20$
	t_{57}	$\sum b_j a_{jk} c_k a_{kl} c_l = 1/40$
	t_{58}	$\sum b_j a_{jk} a_{kl} c_l^2 = 1/60$
	t_{59}	$\sum b_j a_{jk} a_{kl} a_{lm} c_m = 1/120$

Table 1.2: The 17 order conditions up to order 5.

Each order condition has a tree associated with it and in fact there is a 1-1 mapping between the set of *rooted labeled trees* of order q and the order conditions of order q [HNW93]. Given a rooted labeled tree (see Figure 1.1 for example), we can find the corresponding order condition as follows [Ver03]:

1. Assign an index $i, j, k, \ldots$ to each non-leaf node. Assign the parameter b_i to the root node. Starting at the root, assign a_{ij} to each non-leaf node j adjacent to node i, and c_k to each leaf node connected to node k. The left-hand-side of the order condition is the sum of all products of these parameters.

2. Assign a 1 to each leaf node and assign $n+1$ to each node having n descendent nodes. The right-hand-side is the reciprocal of the product of these integers.

For example, in Figure 1.1 the left-hand-side turns out to be $\sum b_i a_{ij} c_j a_{jk} c_k$ and the right-hand-side is $\frac{1}{40}$. This corresponds to the order condition t_{57}.

We will refer to the trees with only one leaf node as "tall trees" (i.e., τ, t_{21}, t_{32}, t_{44}, and t_{59} in Table 1.2). The "broad trees" are the trees were each leaf node is connected

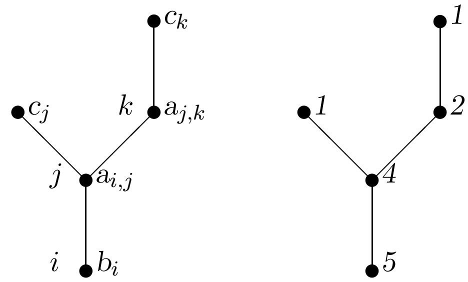


Figure 1.1: The rooted labeled tree corresponding to order condition t_{57}: $\sum b_i a_{ij} c_j a_{jk} c_k = \frac{1}{40}$.

directly to the root. In Table 1.2, the "broad trees" are τ, t_{21}, t_{31}, t_{41}, and t_{51} and they have the special property that the corresponding order conditions are functions of only b_k and c_k; indeed they correspond to the conditions for quadrature methods to be exact for polynomials up to degree 4 (see [Hea97]).

1.1.5 Linear Stability Analysis

The Linear stability analysis of a Runge-Kutta method identifies restrictions on the spectra of the linearized differential operator and on the possible time steps. says add a introduction

The linear stability function $R(z)$ of a Runge-Kutta method (see [HW91]) can be identified with the numerical solution after one step of the method of the scalar *Dahlquist test equation*

$$U' = \lambda U, \quad U^0 = 1, \quad z = h\lambda \tag{1.17}$$

where $\lambda \in \mathbb{C}$. The *linear stability region* or *linear stability domain* is the set

$$S = \{z \in \mathbb{C} : |R(z)| \leq 1\}. \tag{1.18}$$

Let L be the linear operator obtained by linearizing $\boldsymbol{F}$. For a Runge-Kutta method to be linearly stable for (1.1), we must choose h such that $h\lambda_i \in S$ for each of the eigenvalues λ_i of L. Typically this will impose a *time stepsize restriction*.

For an s-stage order-p Runge-Kutta method, $R(z)$ can be determined analytically (see

[HW91]) to be

$$R(z) = \sum_{k=1}^{p} \frac{1}{k!} z^k + \sum_{k=p+1}^{s} tt_k^{(s)} z^k, \tag{1.19}$$

where $tt_k^{(s)}$ are the s-stage, order-k “tall trees”. Figure 1.2 shows the linear stability regions for Runge-Kutta schemes with $s = p$ for $s = 1, \ldots, 4$ (that is, the schemes that do not require tall trees). These plots were created by computing the 1-contour of $|R(z)|$. To quantify the size of these linear stability regions, we measure the *linear stability radius* (see [vdM90]) and the *linear stability imaginary axis inclusion* (for example, as discussed in [SvL85]). These quantities are defined as follows:

Definition 1.4 (Linear Stability Radius) *The linear stability radius is the radius of the largest disc that can fit inside the stability region. Specifically,*

$$\rho = \sup\{\gamma : \gamma > 0 \text{ and } D(\gamma) \subset S\}, \tag{1.20}$$

where $D(\gamma)$ is the disk

$$D(\gamma) = \{z \in \mathbb{C} : |z + \gamma| \leq \gamma\} \tag{1.21}$$

Definition 1.5 (Linear Stability Imaginary Axis Inclusion) *The linear stability imaginary axis inclusion is the radius of the largest interval on the imaginary axis that is contained in the stability region. Specifically,*

$$\rho_2 = \sup\{\gamma : \gamma \geq 0 \text{ and } l(-i\gamma, i\gamma) \subset S\}, \tag{1.22}$$

where $l(z_1, z_2)$ is the line segment connecting $z_1, z_2 \in \mathbb{C}$.

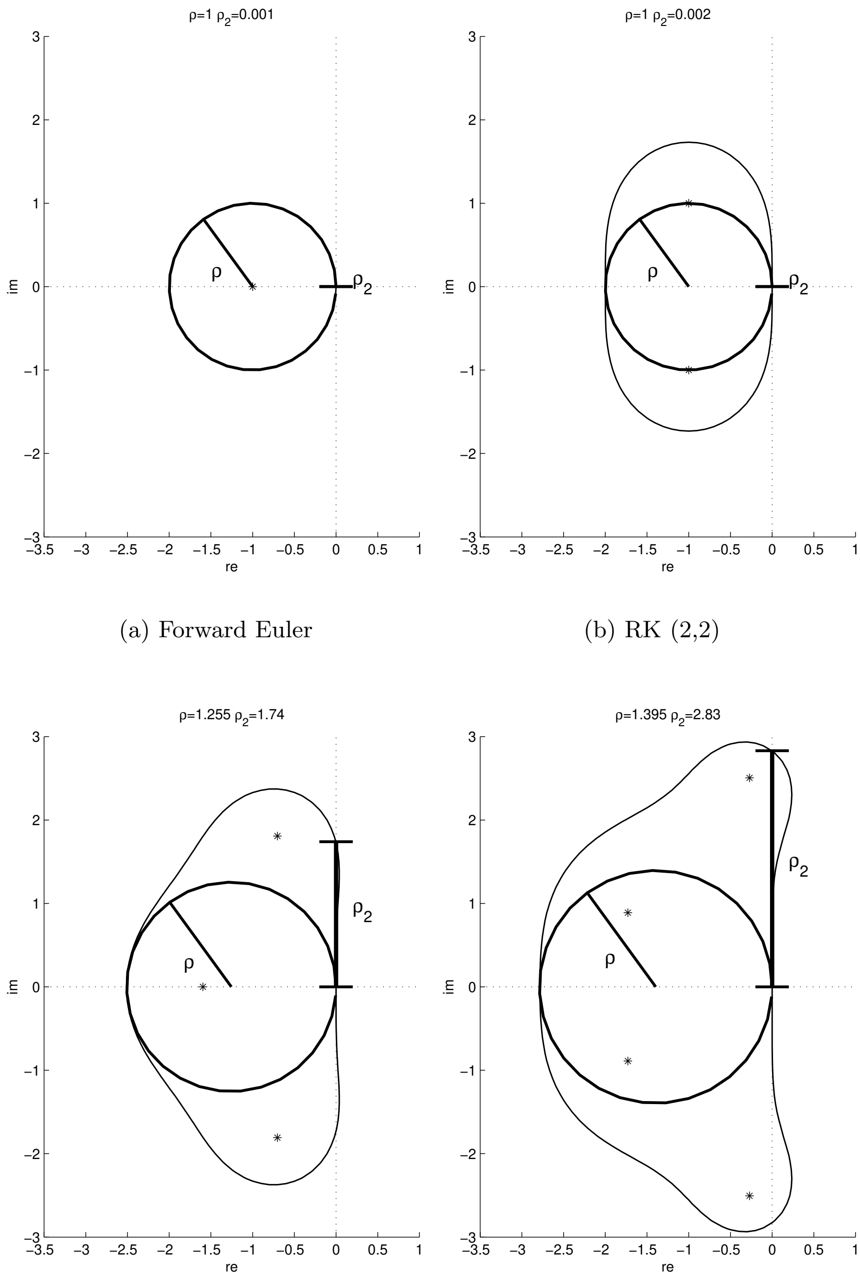
In Figure 1.2, the linear stability radius and the linear stability imaginary axis inclusion are noted. When $s \neq p$, the linear stability region is determined by the value of the additional tall trees. For 6-stage order-5 Runge-Kutta methods, the linear stability function is

$$R(z) = 1 + z + \tfrac{1}{2}z^2 + \tfrac{1}{3!}z^3 + \tfrac{1}{4!}z^4 + \tfrac{1}{5!}z^5 + tt_6^{(6)} z^6, \tag{1.23a}$$

where

$$tt_6^{(6)} = b_6 a_{65} a_{54} a_{43} a_{32} a_{21}. \tag{1.23b}$$

Figure 1.3 shows some examples of the linear stability regions for RK (6,5) methods and in Figure 1.4, the values of ρ and ρ_2 are plotted against $tt_6^{(6)}$. Two important values on this



(a) Forward Euler

(b) RK (2,2)

(c) RK (3,3)

(d) RK (4,4)

Figure 1.2: Linear stability regions for Runge-Kutta schemes with $s = p$. The roots of $R(z)$ are marked with $*$'s and the linear stability radius ρ and linear stability imaginary axis inclusion ρ_2 are labeled.

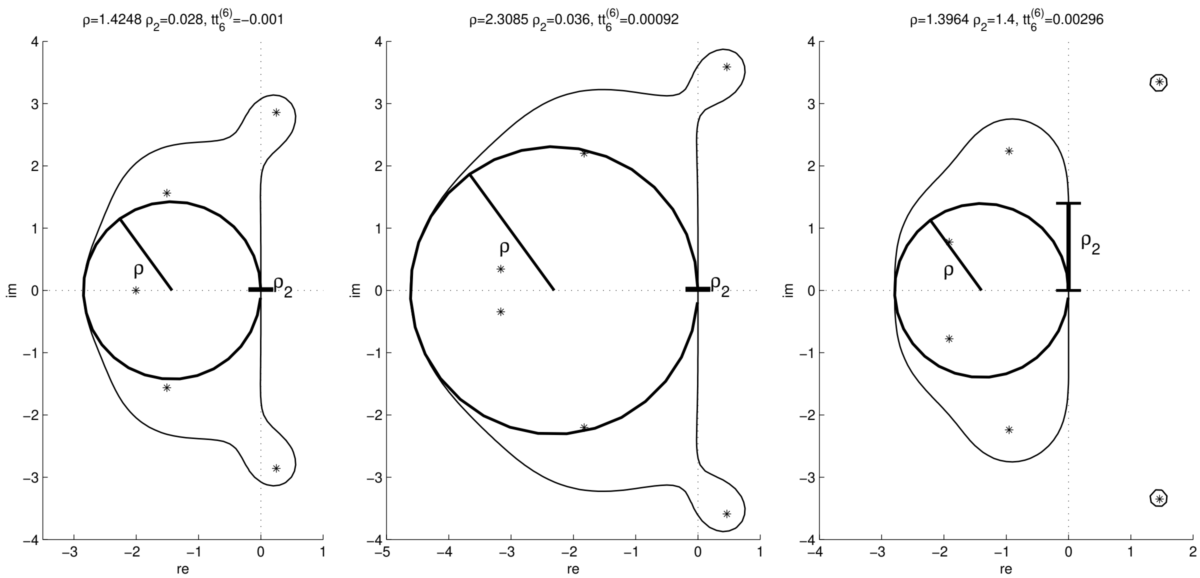
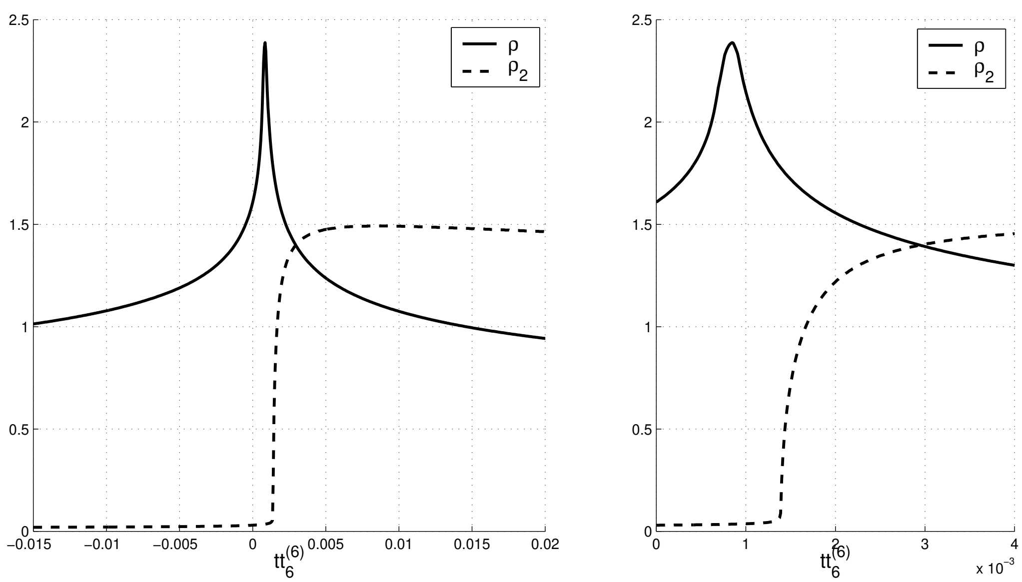


Figure 1.3: Various linear stability regions for RK (6,5) schemes. The roots of $R(z)$ are marked with $*$'s and the linear stability radius ρ and linear stability imaginary axis inclusion ρ_2 are labeled.



(a) Wide-angle

(b) Magnified

Figure 1.4: Linear stability radius (solid) and linear stability imaginary axis inclusion (dashed) for RK (6,5) schemes.

plot are the global maximum of $\rho \approx 2.3868$ at $tt_6^{(6)} \approx 0.00084656$ and the global maximum of $\min(\rho, \rho_2) \approx 1.401$ at $tt_6^{(6)} \approx 0.0029211$.

For 7-stage 5$^{\text{th}}$-order Runge-Kutta methods, the linear stability function is

$$R(z) = 1 + z + \tfrac{1}{2}z^2 + \tfrac{1}{3!}z^3 + \tfrac{1}{4!}z^4 + \tfrac{1}{5!}z^5 + tt_6^{(7)}z^6 + tt_7^{(7)}z^7, \tag{1.24a}$$

for the tall trees

$$tt_7^{(7)} = b_7 a_{76} a_{65} a_{54} a_{43} a_{32} a_{21}, \tag{1.24b}$$

$$\begin{aligned} tt_6^{(7)} = b_6 a_{65} a_{54} a_{43} a_{32} c_2 + b_7 \big(& a_{75} a_{54} a_{43} a_{32} c_2 + a_{76} a_{64} a_{43} a_{32} c_2 \\ &+ a_{76} a_{65} a_{53} a_{32} c_2 + a_{76} a_{65} a_{54} a_{42} c_2 + a_{76} a_{65} a_{54} a_{43} c_3 \big). \end{aligned} \tag{1.24c}$$

Figure 1.5 shows several example RK (7,5) linear stability regions and in Figure 1.6, the values of ρ and ρ_2 are plotted against $tt_6^{(7)}$ and $tt_7^{(7)}$.

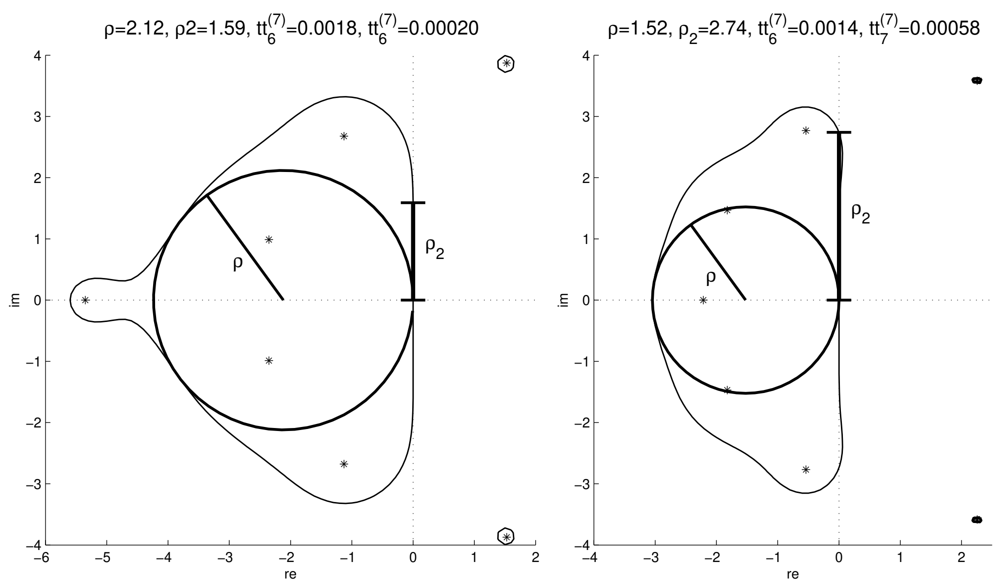
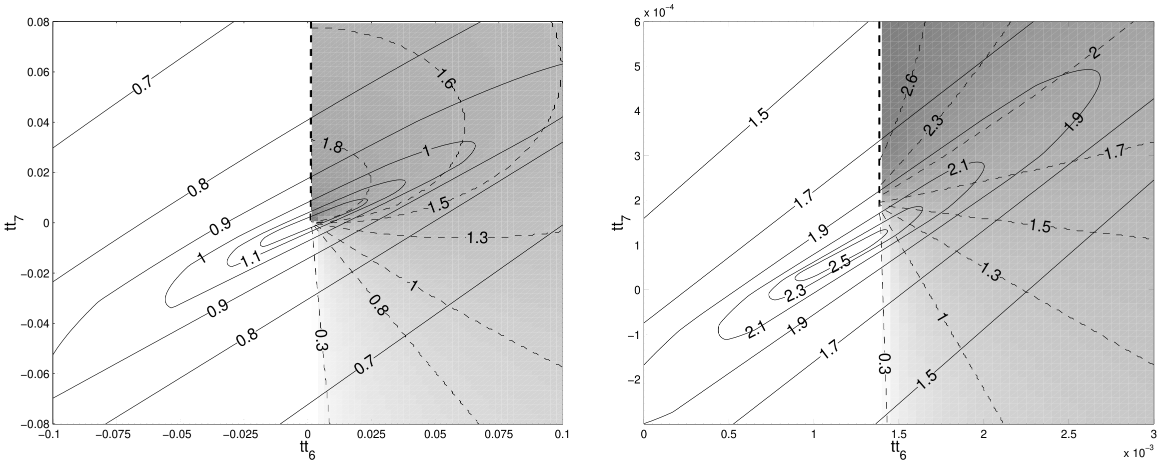


Figure 1.5: Various linear stability regions for RK (7,5) schemes. The roots of $R(z)$ are marked with $*$'s and ρ and ρ_2 are labeled.

1.1.6 Embedded Runge-Kutta Pairs

Two Runge-Kutta schemes can be *embedded* and, by sharing common stages, the resulting *embedded Runge-Kutta method* will be computationally cheaper then running the two



(a) Wide-angle (b) Magnified

Figure 1.6: The linear stability radius (solid contours) and linear imaginary axis inclusion (dashed contours and shading) of RK (7,5) for various values of $tt_6^{(7)}$ and $tt_7^{(7)}$.

schemes independently. We refer to the two schemes of an embedded Runge-Kutta method as *pairs*. The Butcher Tableau for an embedded Runge-Kutta method has two s-vectors of weights $\hat{\boldsymbol{b}}$ and $\boldsymbol{b}$ and is expressed as

$$\begin{array}{c|c} \boldsymbol{c} & A \\ \hline & \hat{\boldsymbol{b}}^T \\ & \boldsymbol{b}^T \end{array} = \begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s,1} & a_{s,2} & \dots & a_{s,s-1} & \\ \hline & \hat{b}_1 & \hat{b}_2 & \dots & \hat{b}_{s-1} & \hat{b}_s \\ & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array} \tag{1.25a}$$

and the schemes are

$$\overline{\boldsymbol{U}}^j = \boldsymbol{U}^n + h \sum_{k=1}^{j-1} a_{jk} \boldsymbol{F}(t_n + c_k h, \overline{\boldsymbol{U}}^k), \quad j = 1, \ldots, s, \tag{1.25b}$$

$$\tilde{\boldsymbol{U}}^{n+1} = \boldsymbol{U}^n + h \sum_{j=1}^{s} \hat{b}_j \boldsymbol{F}(t_n + c_j h, \overline{\boldsymbol{U}}^j), \tag{1.25c}$$

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + h \sum_{j=1}^{s} b_j \boldsymbol{F}(t_n + c_j h, \overline{\boldsymbol{U}}^j), \tag{1.25d}$$

where $\tilde{\boldsymbol{U}}^{n+1}$ and $\boldsymbol{U}^{n+1}$ are the two solutions. After each time step, one of the two solutions is typically propagated and the other discarded. Traditionally, embedded Runge-Kutta methods are used for error control for ODEs; the two schemes typically differ in order, where the higher-order scheme provides a way to estimate the error in the lower-order scheme. If the error estimate is within acceptable tolerances, then the step passes and the lower-order scheme is propagated to the next timestep.[1] Otherwise, the step is rejected and a new stepsize is selected.

In this thesis we present another possible use for embedded schemes in a partial differential equation context. Depending on spatially local characteristics of the solution, one of the two embedded schemes (or a convex combination of the two) could be used to propagate that component of the solution. That is, each scheme could be used in different regions of the spatial domain depending on characteristics of the solution. Over the next several sections, we discuss this idea in more detail.

1.2 The Method of Lines

The method of lines is a widely used technique for approximating partial differential equations (PDEs) with large systems of ODEs in time. A numerical solution to the PDE is then calculated by solving each ODE along a line in time (see Figure 1.7).

Consider a general PDE problem with one temporal derivative

$$u_t = \bar{f}(u, u_x, u_y, u_{xx}, u_{xy}, u_{yy}, \ldots), \tag{1.26}$$

where $\bar{f}$ is some function. The method of lines begins with a semi-discretization of the problem. First, the spatial domain is partitioned into a discrete set of points. In one

[1] Some methods, such as Dormand-Prince 5(4) propagate the higher-order result and use the lower-order for the error estimate [HNW93].

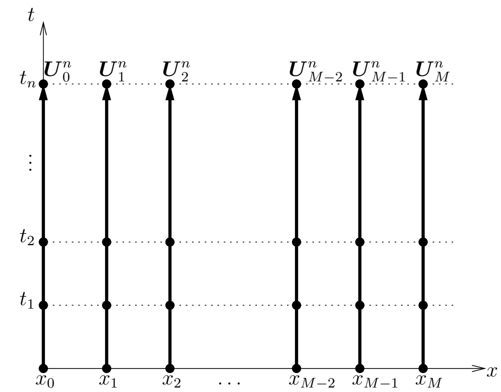


Figure 1.7: The method of lines.

dimension, for example, the domain $x \in [0, 1]$ could be discretized with constant spatial stepsize $\Delta x = \frac{1}{M}$ such that $x_j = j\Delta x$ for $j = 0, \ldots, M$. For higher-dimensions, a suitable ordering of the spatial points $\boldsymbol{x}_j$ for $j = 0, \ldots, M$ is chosen. Then we associate the time dependent vector $\boldsymbol{U}(t)$ with each of these spatial points, specifically

$$\boldsymbol{U}_j(t) \approx u(t, \boldsymbol{x}_j). \tag{1.27}$$

Here we consider a finite difference approach where all of the spatial partial derivatives are replaced with finite difference equations. For example, the spatial partial derivative u_x could be approximated with the simple forward difference

$$u_x \approx \frac{\boldsymbol{U}_{j+1} - \boldsymbol{U}_j}{\Delta x}, \tag{1.28}$$

or the with the essentially non-oscillatory schemes of the next section. After all spatial partial derivatives have been replaced with appropriate finite differences, and any boundary conditions have been discretized or otherwise dealt with[2], we are left with a system of ODEs

$$\boldsymbol{U}' = \boldsymbol{F}(t, \boldsymbol{U}(t)), \tag{1.29}$$

where the operator $\boldsymbol{F}$ depends on the particular spatial discretizations and often also on the value of the solution itself.

Usually the spatial stepsize imposes a stability requirement upon the time stepsize. In the case of hyperbolic conservation laws, this restriction is known as the Courant-Friedrichs-Lewy or *CFL condition* and is the requirement that the numerical domain of dependence

[2]Dealing with boundary conditions in a method of lines framework is non-trivial particularly for higher-order schemes. For this thesis, we will deal with periodic boundary conditions to avoid these additional complications.

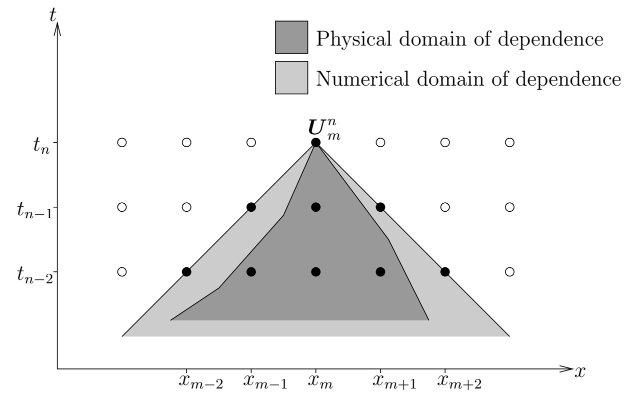


Figure 1.8: The physical and numerical domains of dependence for $\boldsymbol{U}_m^n$.

must contain the physical domain of dependence (see Figure 1.8 and [Lan98]). In other words, the time stepsize must be chosen so that all pertinent information about the solution at t_n has an influence on the solution at t_{n+1}.

1.3 Hyperbolic Conservation Laws

Hyperbolic conservation laws (HCLs) are fundamental to the study of computational gasdynamics and other areas of fluid dynamics. They also play an important role in many other areas of scientific computing, physics and engineering.

HCLs are PDES which express conservation of mass, momentum or energy and the interactions between such quantities. The general HCL initial value problem (IVP) is the PDE

$$\boldsymbol{u}_t + \mathrm{div}\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{0}, \tag{1.30}$$

coupled with boundary conditions and initial conditions, where $\boldsymbol{u}$ is a vector of conserved quantities and $\boldsymbol{f}$ is a vector-valued flux function. From a mathematical point of view, and particularly from a computational point of view, HCLs pose difficulties because they can generate non-smooth (or *weak*) solutions even from smooth initial conditions. These solutions are typically not unique and can include both physically relevant non-smooth features (like shocks or contact discontinuities) and non-physical features such as expansion shocks. Specifying an *entropy condition* (see [Lan98]) will enforce unique and physically correct features of the solution (such as correct shock speeds and smooth expansion fans rather then expansion shocks). Because of the importance of dealing with these phenomena correctly within the computational fluid dynamics (CFD) community, there is a lot of interest in computing the correct entropy satisfying solution to HCLs.

The general one-dimensional scalar conservation law is

$$u_t + f(u)_x = 0, \tag{1.31}$$

with appropriate boundary conditions and initial conditions, where u is some conserved quantity and $f(u)$ is the flux function. Scalar conservation laws exhibit much of the same behavior as general HCLs such as shocks and other discontinuities. The computation of their solutions also involves finding the correct entropy satisfying solution. For this reason, scalar conservation laws such as the linear advection equation

$$u_t + au_x = 0, \tag{1.32}$$

or the nonlinear inviscid Burger's equation

$$u_t + uu_x = 0, \tag{1.33}$$

are often exploited in the development and refinement of numerical techniques.

1.4 Essentially Non-Oscillatory Discretizations

Consider the scalar conservation law

$$u_t + f(u)_x = 0, \tag{1.34}$$

where physical flux $f(u)$ is convex (that is, $f'(u) \geq 0$ for all relevant values of u). A method of lines approach to solving (1.34) using finite differences usually involves the *conservative form*

$$u_t = -\frac{1}{\Delta x}\left(\hat{f}(u_{j+\frac{1}{2}}) - \hat{f}(u_{j-\frac{1}{2}})\right), \tag{1.35}$$

where $\hat{f}(u_{j+\frac{1}{2}}) = \hat{f}(u_{j-K_1+1}, \ldots, u_{j+K_2})$ is the *numerical flux*. The numerical flux should be Lipschitz continuous and must be consistent with the physical flux in the sense that $\hat{f}(u, \ldots, u) = f(u)$ [Lan98].

Often $\hat{f}_{j+\frac{1}{2}} = h(u_j, u_{j+1})$ where $h(a, b)$ is a *Riemann solver* such as the Lax-Friedrichs approximate Riemann solver

$$h^{\mathrm{LF}}(a, b) = \frac{1}{2}\left[f(a) + f(b) - \alpha(b - a)\right], \quad \alpha = \max_u |f'(u)|, \tag{1.36}$$

or one of many others (see [Jia95, Shu97, Lan98]). Unfortunately, schemes built using these Riemann solvers are at most first-order for multi-dimensional problems [GL85].

Essentially non-oscillatory (ENO) discretizations take a different approach from most discretization techniques. They are based on a *dynamic stencil* instead of a fixed stencil. Given a set of *candidate stencils*, ENO discretizations attempt to pick the stencil corresponding to the smoothest possible polynomial interpolate. Geometrically speaking, ENO discretizations choose stencils that avoid discontinuities by biasing the stencils toward smoother regions of the domain.

1.4.1 ENO Schemes

The ENO numerical flux $\hat{f}_{j+\frac{1}{2}}$ is a high-order approximation to the function $h(x_{j+\frac{1}{2}})$ defined implicitly by

$$f(u(x)) = \frac{1}{\Delta x}\int_{x-\frac{\Delta x}{2}}^{x+\frac{\Delta x}{2}} h(\xi)\mathrm{d}\xi, \tag{1.37}$$

as in [Jia95].

Assuming a constant spatial stepsize Δx, we compute the third-order ENO numerical flux $\hat{f}_{j+\frac{1}{2}}$ as follows [Jia95]:

1. Construct the *undivided* (or forward) differences (see [BF01]) of $f(u_j)$ for each j

$$\begin{aligned}
f[j,0] &:= f(u_j),\\
f[j,1] &:= f[j+1,0] - f[j,0],\\
f[j,2] &:= f[j+1,1] - f[j,1],\\
f[j,3] &:= f[j+1,2] - f[j,2],
\end{aligned}$$

2. Choose the stencil based on comparing the magnitude of the undivided differences. Using the smallest (in magnitude) undivided differences will typically lead to the smoothest possible approximation for $h(x_{j+\frac{1}{2}})$. The left most index of the stencil is chosen by computing

$$i := j; \tag{1.38a}$$

$$i := i-1 \quad \text{if } |f[i,1]| > |f[i-1,1]|, \tag{1.38b}$$

$$i := i-1 \quad \text{if } |f[i,2]| > |f[i-1,2]|, \tag{1.38c}$$

$$i := i-1 \quad \text{if } |f[i,3]| > |f[i-1,3]|. \tag{1.38d}$$

3. Finally, we compute the interpolating polynomial evaluated at $x_{j+\frac{1}{2}}$

$$\hat{f}_{j+\frac{1}{2}} = \sum_{m=0}^{3} c(i-j,m) f[i,m], \tag{1.39a}$$

where

$$c(q,m) = \frac{1}{(m+1)!} \sum_{l=q}^{q+m} \prod_{\substack{\mathrm{r=q} \\ r \neq l}}^{\mathrm{q+m}} (-r). \tag{1.39b}$$

If Δx is not constant, then *divided differences* could be used instead of undivided differences and $c(q,m)$ changed accordingly.

The ENO discretization technique is quite general and can be extended to any order (at the cost of increased computation of undivided differences and wider candidate stencils). However, for the purposes of this discussion, we will use the term "ENO" to refer to third-order ENO discretizations.

1.4.2 WENO Schemes

Weighted essentially non-oscillatory (WENO) numerical fluxes build upon ENO schemes by taking a convex combination of all the possible ENO numerical fluxes. WENO uses *smoothness estimators* to choose the weights in the combination in such a way that it achieves 5^{th}-order in smooth regions and automatically falls back to a 3^{rd}-order ENO choice near shocks or other discontinuities.

Note that we have used the term "WENO" when discussing fifth-order WENO discretizations (which in turn are based on third-order ENO discretizations) but that higher-order WENO discretizations are possible and indeed ninth-order WENO discretizations have been constructed (see [QS02]).

WENO discretizations must compute all possible ENO stencils and are therefore more computationally expensive then ENO discretizations *on single-processor computer architectures*. However, WENO schemes can be more efficient on vector-based or multi-processor architectures because they avoid the plethora of "if" statements typically used to implement the stencil choosing step (1.38) of ENO schemes [Jia95].

1.4.3 Other ENO/WENO Formulations

There are also ENO and WENO formulations for Hamilton-Jacobi equations such as the *level set equation*

$$\phi_t + \boldsymbol{V} \cdot \nabla \phi = 0, \tag{1.40}$$

where ϕ implicitly captures an interface with its zero-contour and $\boldsymbol{V}$ may depend on many quantities. Hamilton-Jacobi equations do not contain shocks or discontinuities but they do contain kinks (i.e., discontinuities of the first spatial derivatives) and as such their numerical solution can benefit from schemes like ENO/WENO which help minimize spurious oscillations. See [OF03] for a detailed and easy-to-follow description of Hamilton-Jacobi ENO/WENO. Additional information on level set equations and their applications can be found in [Set99] and [OF03].

For the purposes of this thesis, in either the hyperbolic conservation law or Hamilton-Jacobi formulations, ENO discretizations provide uniformly third-order spatial stencils almost everywhere in the domain. WENO discretizations provide fifth-order spatial stencils in smooth regions and third-order spatial stencils near shocks and other discontinuities.

1.5 Nonlinear Stability

For hyperbolic conservation laws where solutions may exhibit shocks, contact discontinuities and other non-smooth behavior, linear stability analysis may be insufficient because it is based upon the assumption that the linearized operator $\boldsymbol{L}$ is a good approximation to $\boldsymbol{F}$. Numerical solutions using methods based on linear stability analysis often exhibit spurious oscillations and overshoots near shocks and other discontinuities. These unphysical behaviors are known as *weak* or *nonlinear instabilities* and they often appear before a numerical solution becomes completely unstable (i.e., blows up) and in fact they may contribute to a linear instability. We are interested in methods which satisfy certain *nonlinear stability* conditions. ENO and WENO are examples of spatial discretization schemes that satisfy a nonlinear stability condition in the sense that the magnitudes of any oscillations decay at $\mathcal{O}(\Delta x^r)$ where r is the order of accuracy (see [HEOC87]). The strong-stability-preserving time schemes discussed next satisfy a (different) nonlinear stability condition. Finally, a survey of nonlinear stability conditions is presented in [Lan98].

1.6 Strong-Stability-Preserving Runge-Kutta Methods

Strong-stability-preserving (SSP) Runge-Kutta methods satisfy a nonlinear stability requirement that helps suppress spurious oscillations and overshoots and prevent loss of positivity. We begin with the definition of strong-stability.

Definition 1.6 (Strong-Stability) *A sequence of solutions $\{\boldsymbol{U}^n\}$ to (1.1) is strongly stable if, for all $n \geq 0$,*

$$||\boldsymbol{U}^{n+1}|| \leq ||\boldsymbol{U}^n||, \tag{1.41}$$

for some given norm $||\cdot||$.

We say that a Runge-Kutta method is strong-stability-preserving if it generates a strong-stable sequence $\{\boldsymbol{U}^n\}$. The following theorem (see [SO88], [GST01], and [SR02]) makes α–β notation very useful for constructing SSP methods.

Theorem 1.1 (SSP Theorem) *Assuming Forward Euler is SSP with a CFL restriction $h \leq \Delta t_{F.E.}$, then a Runge-Kutta method in α–β notation with $\beta_{ij} \geq 0$ is SSP for the modified CFL restriction*

$$h \leq C\Delta t_{F.E.},$$

where $C = \min \frac{\alpha_{ij}}{\beta_{ij}}$ is the CFL coefficient.

The proof of this theorem is illustrative and we include it for the case when $s = 2$.

Proof The general s-stage Runge-Kutta method in α–β notation is

$$\begin{aligned}
u^{(0)} &= u^n, \\
u^{(1)} &= \alpha_{10}u^{(0)} + h\beta_{10}F(u^{(0)}), \\
u^{(2)} &= \alpha_{20}u^{(0)} + h\beta_{20}F(u^{(0)}) + \alpha_{21}u^{(1)} + h\beta_{21}F(u^{(1)}), \\
u^{n+1} &= u^{(2)}.
\end{aligned}$$

where $\alpha_{ik} \geq 0$, $\sum_{k=0}^{i-1} a_{ik} = 1$. Assume $\beta_{ik} \geq 0$ and that Forward Euler is SSP for some time stepsize restriction. That is $\|u^n + hF(u^n)\| \leq \|u^n\|$ for all $h \leq \Delta t_{\text{F.E.}}$. Now $\|u^{(1)}\| = \|u^{(0)} + h\beta_{10}F(u^{(0)})\|$ and thus $||u^{(1)}|| \leq ||u^{(0)}||$ for

$$h\beta_{10} \leq \Delta t_{\text{F.E.}}. \tag{1.42}$$

Now consider

$$\|u^{(2)}\| = \left\|\alpha_{20}u^{(0)} + h\beta_{20}F(u^{(0)}) + \alpha_{21}u^{(1)} + h\beta_{21}F(u^{(1)})\right\|,$$
$$\|u^{(2)}\| \leq \alpha_{20}\left\|u^{(0)} + h\frac{\beta_{20}}{\alpha_{20}}F(u^{(0)})\right\| + \alpha_{21}\left\|u^{(1)} + h\frac{\beta_{21}}{\alpha_{21}}F(u^{(1)})\right\|,$$

and, provided that

$$h\frac{\beta_{20}}{\alpha_{20}} \leq \Delta t_{\text{F.E.}}, \tag{1.43}$$
$$h\frac{\beta_{21}}{\alpha_{21}} \leq \Delta t_{\text{F.E.}}, \tag{1.44}$$

then

$$\begin{aligned}
\|u^{(2)}\| &\leq \alpha_{20}\|u^{(0)}\| + \alpha_{21}\|u^{(1)}\|,\\
\|u^{(2)}\| &\leq \alpha_{20}\|u^{(0)}\| + \alpha_{21}\|u^{(0)}\|,\\
\|u^{(2)}\| &\leq (\alpha_{20} + \alpha 21)\|u^{(0)}\|,\\
\|u^{(2)}\| &\leq \|u^{(0)}\|.
\end{aligned}$$

Note that the three restrictions (1.42), (1.43), and (1.44) are exactly the condition in the theorem. $\blacksquare$

The SSP property holds for a particular Runge-Kutta scheme *regardless of the form it is written in.* In this sense, α–β notation should be interpreted as a form that makes the SSP property and time step restriction evident. Also note that a given α–β notation may not expose the optimal C value for a particular Runge-Kutta method (recall the Modified Euler example from Section 1.1.2).

We will use the notation SSP (s,p) to refer to an s-stage, order-p strong-stability-preserving Runge-Kutta method.

1.6.1 Optimal SSP Runge-Kutta Methods

For a given order and number of stages, we would like to find the "best" strong-stability-preserving Runge-Kutta scheme. As in [SR02], we define an optimal s-stage, order-p, $s \geq p$, SSPRK scheme as the one with the largest possible CFL coefficient C. That is, an optimal SSPRK method is the global maximum of the optimization problem

$$\max_{\alpha_{ik},\beta_{ik}} \min \frac{\alpha_{ik}}{\beta_{ik}}, \tag{1.45a}$$

subject to the constraints

$$\alpha_{ik} \geq 0, \tag{1.45b}$$

$$\beta_{ik} \geq 0, \tag{1.45c}$$

$$\sum_{k=0}^{i-1} \alpha_{ik} = 1, \tag{1.45d}$$

$$t_{qr}(\alpha, \beta) = \gamma_{qr}, \tag{1.45e}$$

where $t_{qr}(\alpha, \beta)$ and γ_{qr} represent, respectively, the left- and right-hand sides of the order conditions up to order p written in terms of α_{ik} and β_{ik}. The order conditions in Butcher notation are polynomial expressions of b_k and a_{ik} and thus, using (1.11), $t_{qr}(\alpha, \beta)$ are polynomial expressions in α_{ik} and β_{ik}.

This optimization problem is difficult to solve numerically because of the highly nonlinear objective function (1.45a). In [SR02], the problem is reformulated, with the addition of a dummy variable z, as

$$\max_{\alpha_{ik}, \beta_{ik}} z, \tag{1.46a}$$

subject to the constraints

$$\alpha_{ik} \geq 0, \tag{1.46b}$$

$$\beta_{ik} \geq 0, \tag{1.46c}$$

$$\sum_{k=0}^{i-1} \alpha_{ik} = 1, \tag{1.46d}$$

$$\alpha_{ik} - z\beta_{ik} \geq 0, \tag{1.46e}$$

$$t_{qr}(\alpha, \beta) = \gamma_{qr}, \tag{1.46f}$$

where $t_{qr}(\alpha, \beta)$ and γ_{qr} are again the left- and right-hand sides of the order conditions, respectively. Notice that the dummy variable z is just the CFL coefficient we are looking for.

In Chapter 2, we will find some optimal strong-stability-preserving Runge-Kutta methods using a numerical optimizer to find a global maximum for (1.46).

In Theorem 1.1 we assumed that $\beta_{ik} \geq 0$. While it is possible to have SSP schemes with negative β coefficients, these schemes are more complicated. For each $\beta_{ik} < 0$, the *downwind-biased operator* $\tilde{\boldsymbol{F}}$ is used in (1.10c) instead of $\boldsymbol{F}$. The downwind-biased operator is a discretization of the same spatial derivatives as $\boldsymbol{F}$ but discretized in such a way that Forward Euler (using $\tilde{\boldsymbol{F}}$) and solved *backwards* in time generates a strongly-stable sequence $\{\boldsymbol{U}^n\}$ for $h \leq C\Delta t_{\mathrm{FE}}$ (see [SO88, Shu88, SR02, RS02a]). At best, the use of $\tilde{\boldsymbol{F}}$ complicates a method because of the additional coding required to discretize the downward biased operator. At worst, if both $\boldsymbol{F}$ and $\tilde{\boldsymbol{F}}$ are required in a particular stage, then the computational cost and storage requirements of that stage are doubled! In [RS02a] and [Ruu03] schemes are found with negative β coefficients that avoid this latter limitation. Also, schemes involving $\tilde{\boldsymbol{F}}$ may not be appropriate for any PDE problems with artificial viscosity (or other dissipative terms), such as the viscous Burger's equation $u_t + uu_x = \varepsilon u_{xx}$, because these terms are unstable when integrated backwards in time.[3] However, as is proven in [RS02b], strong-stability-preserving Runge-Kutta schemes of order five and higher must involve contain some negative β coefficients in order to satisfy the order conditions. In summary, there are significant reasons to avoid the use of negative β coefficients although this is not possible for fifth- and higher-order schemes.

1.7 Motivation for a Embedded RK/SSP Pair

Recall that weighted essentially non-oscillatory (WENO) spatial discretizations provide fifth-order spatial discretizations in smooth regions of the solution and third-order spatial discretizations near shocks or other discontinuities.

Because of the fifth-order spatial regions, it is natural to use a fifth-order time solver with WENO spatial discretizations. In fact, we should use a strong-stability-preserving Runge-Kutta method because the solution may contain shocks or other discontinuities. Unfortunately, as noted above, fifth-order SSPRK methods are complicated by their use of the downwind-biased operator $\tilde{\boldsymbol{F}}$. However, *the SSP property is only needed in the vicinity of non-smooth features and in these regions WENO discretizations provide only third-order.* This idea motivates the construction of fifth-order linearly stable Runge-Kutta schemes with third-order strong-stability-preserving embedded pairs. The fifth-order scheme would be used in smooth regions whereas the third-order scheme SSP scheme would be used near shocks or other discontinuities.

We could also use these embedded methods or build others like them for error control. To construct an error estimator for a SSP scheme, we could embed it in a higher-order

[3] For example, it is well known (see [Str92]) that the heat equation $u_t - u_{xx} = 0$ is ill-posed for $t < 0$.

linearly stable Runge-Kutta scheme and use the difference between the schemes as the error estimator. Although the error estimator scheme would not necessarily be strongly stable, its results would not be propagated and thus any spurious oscillations produced could not compound over time.

1.7.1 On Balancing z and ρ

Recall that within a PDE context, the CFL coefficient C (or z) measures the time stepsize restriction of an SSP scheme in *multiples of a strongly-stability-preserving Forward Euler stepsize* $\Delta t_{F.E.}$. Now, because $\rho = 1$ for Forward Euler, ρ effectively measures the time stepsize restriction of an linearly stable Runge-Kutta scheme in *multiples of a linearly stable Forward Euler stepsize*, say $\widehat{\Delta t}_{\text{F.E.}}$. Thus, assuming that these two fundamental stepsizes are the same ($\Delta t_{\text{F.E.}} = \widehat{\Delta t}_{\text{F.E.}}$), the *overall* CFL coefficient for method consisting of embedded Runge-Kutta and SSP pairs will be simply the minimum of z and ρ. We use the following working definition of the *CFL coefficient* for a RK method with embedded SSP pair:

Definition 1.7 *The CFL coefficient $\hat{C}$ for an embedded RK / SSP method is the minimum of the CFL coefficient of the SSP scheme and the linear stability radius of the RK scheme. That is,*

$$\hat{C} = \min(C, \rho) = \min(z, \rho). \tag{1.47}$$

Methods typically cannot be compared solely on the basis of CFL coefficients; to achieve a fair comparison, one must account for the number of stages each method uses. This motivates the following definition

Definition 1.8 (Effective CFL Coefficient) *A* effective CFL coefficient $\hat{C}_{\mathit{eff}}$ *of a method is*

$$\hat{C}_{\mathit{eff}} = \hat{C}/s, \tag{1.48}$$

where $\hat{C}$ is the CFL coefficient of the method and s is the number of stages (or more generally function evaluations).

In the next chapter, we use an optimization software package to find optimal strong-stability-preserving Runge-Kutta schemes. We investigate embedded methods further in Chapters 3 and 4.

Chapter 2

Finding Strong-Stability-Preserving Runge-Kutta Schemes

In this chapter, we present a technique for deriving strong-stability-preserving Runge-Kutta schemes using the proprietary software package GAMS/BARON. Some optimal SSPRK methods are then shown.

2.1 GAMS/BARON

The General Algebraic Modeling System (GAMS) [GAM01] is a proprietary high-level modeling system for optimization problems. The Branch and Reduce Optimization Navigator (BARON) is a proprietary solver available to GAMS that is particularly well-suited to factorable global optimization problems. BARON guarantees global optimality provided that the objective function and constraint functions are bounded and factorable and that all variables are suitably bounded above and below. The optimization problem (1.46) has polynomial constraints and a linear objective function so if appropriate bounds are provided, BARON will guarantee optimality given sufficient memory and CPU time (at least to within the specified tolerances).

2.1.1 Using GAMS/BARON to Find SSPRK Schemes

We use a hybrid combination of Butcher and α–β notation using the A, $\boldsymbol{b}$, and α coefficients. This allows the order conditions (by far the most complicated constraints of the optimization problem) to be written in a slightly simpler form. Each β_{ik} can be written as a polynomial expression in the Butcher tableau coefficients. The optimization problem (1.46) can then be rewritten as

$$\max_{\alpha,A,\boldsymbol{b}} z, \tag{2.1a}$$

subject to the constraints

$$\alpha_{ik} \geq 0, \tag{2.1b}$$

$$\beta_{ik}(A,\boldsymbol{b}) \geq 0, \tag{2.1c}$$

$$\sum_{k=0}^{i-1} \alpha_{ik} = 1, \tag{2.1d}$$

$$\alpha_{ik} - z\beta_{ik}(A,\boldsymbol{b}) \geq 0, \tag{2.1e}$$

$$t_{qr}(A,\boldsymbol{b}) = \gamma_{qr}, \tag{2.1f}$$

where, as usual, t_{qr} denote the left-hand side of the order conditions up to order-p. Some of the GAMS input files that implement (2.1) are shown in Appendix A.

2.1.2 Generating GAMS Input with Maple

The order condition expressions grow with both p and s and entering them directly quickly becomes tedious and error prone. The proprietary computer algebra system `maple` was used to generate the GAMS input file using the worksheet in Appendix B.1. Basically this involves expanding the order conditions and other constraints in (2.1) and formatting them in the GAMS language.

2.2 Optimal SSP Schemes

The remainder of this chapter presents some optimal strong-stability-preserving Runge-Kutta schemes. Table 2.1 shows the optimal CFL coefficients for s-stage, order-p SSPRK schemes. Note that [GS98] prove there is no SSP (4,4) scheme.

	$s = 1$	$s = 2$	$s = 3$	$s = 4$	$s = 5$	$s = 6$	$s = 7$	$s = 8$
$p = 1$	1	2	3	4	5	6	7	8
$p = 2$		1	2	3	4	5	6	7
$p = 3$			1	2	2.651	3.518	$\boxed{4.288}$	$\boxed{5.107}$
$p = 4$				n/a[a]	1.508	$\boxed{2.295}$	$\boxed{3.321}$	$\boxed{4.146}$

[a] There is no SSP (4,4) scheme.

Table 2.1: Optimal CFL coefficients for s-stage, order-p SSPRK schemes. BARON was not run to completion on boxed entries and thus these represent feasible but not necessarily optimal SSP schemes.

2.2.1 Optimal First- and Second-Order SSP Schemes

The optimal first- and second-order SSP schemes have simple closed-form representations which depend only on the number of stages s. The following results are proven by [GS98], [SR02], and others:

1. The optimal first-order SSP schemes for $s = 1, 2, 3, \ldots$ are

$$\alpha_{ik} = \begin{cases} 1 & k = i - 1, \\ 0 & \text{otherwise.} \end{cases}, \qquad i = 1, \ldots, s, \tag{2.2a}$$

$$\beta_{ik} = \begin{cases} \frac{1}{s} & k = i - 1, \\ 0 & \text{otherwise.} \end{cases}, \qquad i = 1, \ldots, s. \tag{2.2b}$$

   That is, α consists of 1's down its diagonal and β consists of $\frac{1}{s}$ down its diagonal and they have a CFL coefficient of s.

2. The optimal second-order SSP schemes for $s = 2, 3, 4, \ldots$ are

$$\alpha_{ik} = \begin{cases} 1 & k = i-1, \\ 0 & \text{otherwise.} \end{cases}, \qquad i = 1, \ldots, s-1, \tag{2.3a}$$

$$\alpha_{sk} = \begin{cases} \frac{1}{s} & k = 0, \\ \frac{s-1}{s} & k = s-1, \\ 0 & \text{otherwise.} \end{cases}, \tag{2.3b}$$

$$\beta_{ik} = \begin{cases} \frac{1}{s-1} & k = i-1, \\ 0 & \text{otherwise.} \end{cases}, \qquad i = 1, \ldots, s-1, \tag{2.3c}$$

$$\beta_{sk} = \begin{cases} \frac{1}{s} & k = s-1, \\ 0 & \text{otherwise.} \end{cases}. \tag{2.3d}$$

The optimal SSP (s,2) schemes have CFL coefficients of $s - 1$.

2.2.2 Optimal Third-Order SSP Schemes

The optimal SSP (3,3), SSP (4,3), SSP (5,3) and SSP (6,3) schemes are shown in Tables 2.2, 2.3 and 2.4. For these schemes, BARON ran to completion and thus was used to guarantee optimality.

2.2.3 Optimal Fourth-Order SSP Schemes

As noted in [GS98], there is no 4-stage, order-4 strong-stability-preserving Runge-Kutta scheme. For five stages, BARON ran to completion and Table 2.5 shows the optimal SSP (5,4) scheme. For six or more stages, BARON was not able to prove optimality within a reasonable amount of time (24 hours of computation on a Athlon MP 1200). It does however, readily find feasible schemes; for example, Table 2.6 shows a feasible but not proven optimal SSP (6,4) scheme.

$$
\begin{array}{c|ccc}
0 & & & \\
1 & 1 & & \\
1/2 & 1/4 & 1/4 & \\
\hline
 & 1/6 & 1/6 & 2/3
\end{array}
$$

$$
\alpha = \begin{bmatrix} 1 & & \\ 3/4 & 1/4 & \\ 1/3 & 0 & 2/3 \end{bmatrix}
$$

$$
\beta = \begin{bmatrix} 1 & & \\ 0 & 1/4 & \\ 0 & 0 & 2/3 \end{bmatrix}
$$

(a) SSP (3,3)

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

$$
\alpha = \begin{bmatrix} 1 & & & \\ 0 & 1 & & \\ 2/3 & 0 & 1/3 & \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
\beta = \begin{bmatrix} 1/2 & & & \\ 0 & 1/2 & & \\ 0 & 0 & 1/6 & \\ 0 & 0 & 0 & 1/2 \end{bmatrix}
$$

(b) SSP (4,3)

Table 2.2: The optimal SSP (3,3) and SSP (4,3) schemes in Butcher tableau and α–β notation. The CFL coefficients are 1 and 2 respectively.

$$
\begin{array}{c|ccccc}
0 & & & & & \\
0.37727 & 0.37727 & & & & \\
0.75454 & 0.37727 & 0.37727 & & & \\
0.72899 & 0.24300 & 0.24300 & 0.24300 & & \\
0.69923 & 0.15359 & 0.15359 & 0.15359 & 0.23846 & \\
\hline
 & 0.20673 & 0.20673 & 0.11710 & 0.18180 & 0.28763
\end{array}
$$

$$
\alpha = \begin{bmatrix} 1 & & & & \\ 0 & 1 & & & \\ 0.35591 & 0 & 0.64409 & & \\ 0.36793 & 0 & 0 & 0.63207 & \\ 0 & 0 & 0.23759 & 0 & 0.76241 \end{bmatrix}
$$

$$
\beta = \begin{bmatrix} 0.37727 & & & & \\ 0 & 0.37727 & & & \\ 0 & 0 & 0.24300 & & \\ 0 & 0 & 0 & 0.23846 & \\ 0 & 0 & 0 & 0 & 0.28763 \end{bmatrix}
$$

Table 2.3: The optimal SSP (5,3) scheme in Butcher tableau and α–β notation. This scheme has a CFL coefficient of 2.65063.

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
0.28422 & 0.28422 & & & & & \\
0.56844 & 0.28422 & 0.28422 & & & & \\
0.69213 & 0.23071 & 0.23071 & 0.23071 & & & \\
0.56776 & 0.13416 & 0.13416 & 0.13416 & 0.16528 & & \\
0.85198 & 0.13416 & 0.13416 & 0.13416 & 0.16528 & 0.28422 & \\
\hline
 & 0.17016 & 0.17016 & 0.10198 & 0.12563 & 0.21604 & 0.21604
\end{array}
$$

$$
\alpha = \begin{bmatrix}
1 & & & & & \\
0 & 1 & & & & \\
0.18828 & 0 & 0.81172 & & & \\
0.41849 & 0 & 0 & 0.58151 & & \\
0 & 0 & 0 & 0 & 1 & \\
0 & 0.23989 & 0 & 0 & 0 & 0.76011
\end{bmatrix}
$$

$$
\beta = \begin{bmatrix}
0.28422 & & & & & \\
0 & 0.28422 & & & & \\
0 & 0 & 0.23071 & & & \\
0 & 0 & 0 & 0.16528 & & \\
0 & 0 & 0 & 0 & 0.28422 & \\
0 & 0.06818 & 0 & 0 & 0 & 0.21604
\end{bmatrix}
$$

Table 2.4: The optimal SSP (6,3) scheme in Butcher tableau and α–β notation. This scheme has a CFL coefficient of 3.51839.

$$\begin{array}{c|ccccc}
0 & & & & & \\
0.39175 & 0.39175 & & & & \\
0.58608 & 0.21767 & 0.36841 & & & \\
0.47454 & 0.082692 & 0.13996 & 0.25189 & & \\
0.93501 & 0.067966 & 0.11503 & 0.20703 & 0.54497 & \\
\hline
 & 0.14681 & 0.24848 & 0.10426 & 0.27444 & 0.22601
\end{array}$$

$$\alpha = \begin{bmatrix}
1 & & & & \\
0.44437 & 0.55563 & & & \\
0.6201 & 0 & 0.3799 & & \\
0.17808 & 0 & 0 & 0.82192 & \\
0.0081647 & 0 & 0.51723 & 0.13374 & 0.34086
\end{bmatrix}$$

$$\beta = \begin{bmatrix}
0.39175 & & & & \\
0 & 0.36841 & & & \\
0 & 0 & 0.25189 & & \\
0 & 0 & 0 & 0.54497 & \\
0 & 0 & 0 & 0.088679 & 0.22601
\end{bmatrix}$$

Table 2.5: The optimal SSP (5,4) scheme in Butcher tableau and α–β notation. This scheme has a CFL coefficient of 1.50818.

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
0.35530 & 0.35530 & & & & & \\
0.60227 & 0.27049 & 0.33179 & & & & \\
0.46975 & 0.1224 & 0.15014 & 0.19721 & & & \\
0.56481 & 0.076343 & 0.093643 & 0.123 & 0.27182 & & \\
1.00063 & 0.076343 & 0.093643 & 0.123 & 0.27182 & 0.43582 & \\
\hline
 & 0.15225 & 0.18675 & 0.15554 & 0.13485 & 0.2162 & 0.15442
\end{array}
$$

$$
\alpha = \begin{bmatrix}
1 & & & & & \\
0.2387 & 0.7613 & & & & \\
0.54749 & 0 & 0.45251 & & & \\
0.37629 & 0 & 0 & 0.62371 & & \\
0 & 0 & 0 & 0 & 1 & \\
0.13024 & 0.15681 & 0.21688 & 0 & 0 & 0.49608
\end{bmatrix}
$$

$$
\beta = \begin{bmatrix}
0.3553 & & & & & \\
0 & 0.33179 & & & & \\
0 & 0 & 0.19721 & & & \\
0 & 0 & 0 & 0.27182 & & \\
0 & 0 & 0 & 0 & 0.43582 & \\
0 & 0.068342 & 0.094518 & 0 & 0 & 0.15442
\end{bmatrix}
$$

Table 2.6: A SSP (6,4) scheme in Butcher tableau and α–β notation. This scheme has a CFL coefficient of 2.29455. This scheme has not been proven optimal.

Chapter 3

Fourth-Order Runge-Kutta Methods with Embedded SSP Pairs

In this chapter we look for fourth-order Runge-Kutta schemes with embedded strong-stability-preserving Runge-Kutta pairs. We begin with the formulation of an optimization problem for finding such pairs. The optimization software GAMS/BARON is then used to compute solutions to this problem. This chapter then closes with some comments about using this technique for order-5 and higher.

3.1 Finding Lower-Order Pairs with BARON

We wish to find RK (s,p) schemes with embedded SSP (s^*,p^*) schemes where $s^* \leq s$ and $p^* \leq p$. The optimization problem (2.1) for the SSP (s^*,p^*) can be augmented as follows

$$\max_{\alpha, A, \boldsymbol{b}, \hat{\boldsymbol{b}}} z, \tag{3.1a}$$

subject to the constraints

$$\alpha_{ik} \geq 0, \qquad i = 1, \dots, s^*,\ k = 1, \dots, i-1, \tag{3.1b}$$

$$\beta_{ik}(A, \hat{\boldsymbol{b}}) \geq 0, \qquad i = 1, \dots, s^*,\ k = 1, \dots, i-1, \tag{3.1c}$$

$$\sum_{k=0}^{i-1} \alpha_{ik} = 1, \qquad i = 1, \dots, s^*, \tag{3.1d}$$

$$\alpha_{ik} - z\beta_{ik}(A, \hat{\boldsymbol{b}}) \geq 0, \qquad i = 1, \dots, s^*,\ k = 1, \dots, i-1, \tag{3.1e}$$

$$t_{qr}(A, \hat{\boldsymbol{b}}) = \gamma_{qr}, \qquad \text{(up to order-}p^*\text{)}, \tag{3.1f}$$

$$t_{qr}(A, \boldsymbol{b}) = \gamma_{qr}, \qquad \text{(up to order-}p\text{)}, \tag{3.1g}$$

where, as before, t_{qr} and γ_{qr} denote the left- and right-hand side of the order conditions. Note that we are only optimizing z, the CFL coefficient of the SSP scheme; in particular, the RK method only has to be feasible.

At first glance, it seems strange that we specify s^*; after all, in most tradional embedded Runge-Kutta methods, both schemes have access to all of the stages. However, the SSP condition imposes additional constraints upon the coefficients of all stages up to the last one used by the SSP scheme (namely stage s^*). For example, (3.1c) specifies that each β coefficient used by the SSP scheme is non-negative which implies the corresponding A coefficients must also be non-negative.[1] However, non-negative A coefficients is *not* a requirement for non-SSP Runge-Kutta methods. Using too many or all of the stages for the SSP scheme could (theoretically) make it impossible to satisfy the necessary order conditions for the Runge-Kutta scheme. Although in practice this was not observed, increasing s^* did occasionally have an adverse effect on the resulting schemes, e.g., the RK (5,4) / SSP (4,1) versus the RK (5,4) / SSP (5,1) methods in Table 3.4.

An example GAMS input file that implements (3.1) for RK (5,4) with embedded SSP (3,3) is shown in Appendix A.2 and others can be found in Appendix C. CPU time for each of the computations in this chapter was limited to 8 hours on a Athlon MP 1200 processor with 512MB of RAM.

[1] This follows trivially from the recursive relationship (1.11) between Butcher tableaux and α–β notation.

3.2 4-Stage Methods

For $s = p = 4$, the CFL coefficients for the possible combinations of s^* and p^* are shown in Table 3.1. Note that it is not possible to embed a third-order RK scheme in a RK (4,4)

	$s^* = 1$	$s^* = 2$	$s^* = 3$	$s^* = 4$
$p^* = 1$	1	2	2	$\boxed{0.957}$ $\lceil 2.654 \rceil$
$p^* = 2$		1	1	$\boxed{0.957}$ $\lceil 2.229 \rceil$
$p^* = 3$			n/a[a]	n/a[a]
$p^* = 4$				n/a[b]

[a]It is not possible to embed a 3$^{\text{rd}}$-order RK scheme in an 4$^{\text{th}}$-order RK scheme.
[b]There is no SSP (4,4) scheme.

Table 3.1: CFL coefficients of SSP (s^*,p^*) schemes embedded in order-4 linearly stable RK schemes. Boxed entries correspond to methods which are feasible but not proven optimal ($\lceil\cdot\rceil$ denotes proven upper bounds)

scheme regardless of strong-stability properties (see [HNW93]) and there is no SSP (4,4) scheme (as proven in [GS98]). For many of the calculations, the alloted time was not sufficient to guarantee optimality for the given values of s, p, s^* and p^*; in these cases, both the best value found and the upper bound are shown.

Tables 3.2 and 3.3 show the Butcher tableaux for the particular schemes with $p^* = 1$ and $p^* = 2$ respectively. The upper and lower bounds for the A and $\boldsymbol{b}$ coefficients were chosen to be 10 and -10 respectively. In some cases (like Table 3.2a), these values were actually chosen by BARON; barring a rather unlikely coincidence, this would seem to indicate the presence of at least one free parameter in the solution that could be used, for example, to minimize the magnitude of the A and $\boldsymbol{b}$ coefficients. All 4-stage, order-4 methods have the same linear stability region (shown in Figure 1.2d) and therefore all of these embedded methods have the same linear stability region for the RK (4,4) scheme.

3.3 5-Stage Methods

For $s = 5$, $p = 4$, the CFL coefficients for the possible combinations of s^* and p^* are shown in Table 3.4. Again note that there is no SSP (4,4) scheme and that for some of the calculations, the alloted time was not sufficient to for BARON to run to complete and

0				
1/2	1/2			
0	10	-10		
1	-0.45000	1.50000	-0.050000	
$\hat{b}$	1/2	1/2		
b	.17500	0.66667	-0.0083333	0.16667

(a) RK (4,4), SSP (2,1), $C = 2$, $T = 1.7$s

0				
1/2	1/2			
1/2	1/4	1/4		
1	0	-1	2	
$\hat{b}$	1/4	1/4	1/2	
b	1/6	0	2/3	1/6

(b) RK (4,4), SSP (3,1), $C = 2$, $T = 911$s

0				
1/2	1/2			
0	10	-10		
1	-0.45000	1.50000	-0.050000	
$\hat{b}$	1/2	1/2		
b	.17500	0.66667	-0.0083333	0.16667

(c) RK (4,4), SSP (4,1), $C = 0.957$

Table 3.2: RK (4,4) schemes with embedded SSP (s^*,1) pairs. C is the CFL coefficient of the SSP scheme and T is the computation time.

0				
1	1			
1/2	3/8	1/8		
1	4.66238	1.22079	-4.88317	0
$\hat{b}$	1/2	1/2		
b	0.16667	0.23493	0.666670	-0.068262

(a) RK (4,4), SSP (2,2), $C = 1$, $T = 41.7$s

0				
1	1			
1/2	3/8	1/8		
1	-6.5	-2.5	10	
$\hat{b}$	1/2	1/2	0	
b	1/6	2/15	2/3	1/30

(b) RK (4,4), SSP (3,2), $C = 1$, $T = 4313$s

0				
1	1			
1/2	3/8	1/8		
1	4.66238	1.22079	-4.88317	0
$\hat{b}$	1/2	1/2		
b	0.16667	0.23493	0.666670	-0.068262

(c) RK (4,4), SSP (2,2), $C = 1$, $T = 41.7$s

Table 3.3: RK (4,4) schemes with embedded SSP (s^*,2) pairs. C is the CFL coefficient of the SSP scheme and T is the computation time.

ensure optimality.

Table 3.5 shows the Butcher tableaux for the embedded methods with $p^* = 3$. These methods are significant because in the WENO context discussed in Chapter 1, they are competitive with the commonly used optimal SSP (3,3) scheme. Consider the RK (5,4) / SSP (5,3) method where the RK scheme has a linear stability radius of about $\rho \approx 1.84$ (see Figure 3.1) and the SSP pair has a CFL coefficient of $C \approx 2.30$. Thus by Section 1.7.1, we would expect that the overall CFL coefficient for the method would be $\min(C, \rho) \approx 1.84$. The effective CFL coefficient for this 5-stage embedded method is thus about $\frac{1.84}{5} = 0.368$ and thus the method is about 10% more computationally efficient then the optimal SSP (3,3) scheme (which has an effective CFL coefficient of $\frac{1}{3}$) and about 20% more efficient then the optimal SSP (5,4) scheme (which has an effective CFL coefficient of about $\frac{1.508}{5} = 0.302$). The embedded method is also potentially more accurate in smooth regions of the domain if used with WENO discretizations as discussed earlier. It is likely possible to optimize the linear stability properties of the RK scheme and further improve these methods.

	$s^* = 1$	$s^* = 2$	$s^* = 3$	$s^* = 4$	$s^* = 5$
$p^* = 1$	1	2	3	3.995	3.624
$p^* = 2$		1	2	2.999	$\boxed{2.943}$ $\lceil 3.796 \rceil$
$p^* = 3$			1.000	2	$\boxed{2.301}$ $\lceil 3.603 \rceil$
$p^* = 4$				n/a[a]	$\boxed{1.508}$ $\lceil 2.965 \rceil$

[a]There is no SSP (4,4) scheme.

Table 3.4: CFL coefficients of SSP (p,s) schemes embedded in linearly stable RK (5,4) schemes. Boxed entries correspond to methods which are feasible but not proven optimal ($\lceil \cdot \rceil$ denotes proven upper bounds).

3.4 Higher-Order Schemes

In theory, this technique should work for $s = 6$ and $p = 5$ as well, however, the nine additional constraints from the order-5 order conditions increase the complexity of the optimization problem (3.1). Unfortunately, BARON was not able to find a feasible solution to any problems with $p = 5$ within several days of computation.

In the next chapter, we simplify the problem by specifying particular SSP schemes and

0					
1	1				
1/2	1/4	1/4			
2.59674	5.94669	-10	6.65005		
-0.021492	0.059473	0.058176	-0.13569	-0.0034480	
$\hat{b}$	1/6	1/6	2/3		
b	5.30033	0.34232	0.14262	-0.0062935	-4.778980

(a) RK (5,4), SSP (3,3), $C = 1$, $T = 28800$s

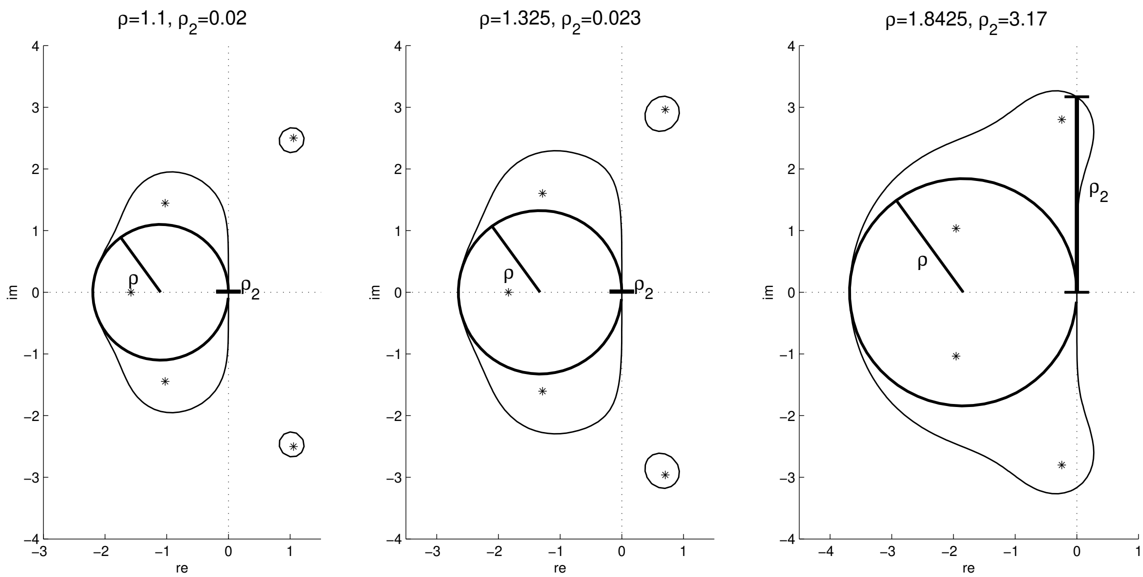
0					
1/2	1/2				
1	1/2	1/2			
1/2	1/6	1/6	1/6		
1	-3.8692	1.3738	-1.7477	5.2431	
$\hat{b}$	1/6	1/6	1/6	1/2	
b	1/6	1/3	0.13488	1/3	0.031788

(b) RK (5,4), SSP (4,3), $C = 2$, $T = 5106$s

0					
0.36717	0.36717				
0.58522	0.26802	0.3172			
0.44156	0.11606	0.13735	0.18816		
0.8464	0.11212	0.13269	0.18178	0.4198	
$\hat{b}$	0.17279	0.094505	0.12947	0.29899	0.30424
b	0.12293	0.31981	-0.15316	0.31887	0.39155

(c) RK (5,4), SSP (5,3), $C = 2.30128$, $T = 64800$s

Table 3.5: RK (5,4) schemes with embedded SSP (s^*,3) pairs. C is the CFL coefficient of the SSP scheme and T is the computation time in BARON.



(a) RK (5,4) / SSP (3,3) (b) RK (5,4) / SSP (4,3) (c) RK (5,4) / SSP (5,3)

Figure 3.1: Linear stability regions for RK (5,4) schemes with embedded third-order SSP pairs.

attempting to satisfy the order conditions using the remaining A and $\boldsymbol{b}$ coefficients

Chapter 4

Fifth-Order Runge-Kutta Methods with Embedded SSP Pairs

In this chapter we look for fifth-order Runge-Kutta schemes with embedded strong-stability-preserving Runge-Kutta pairs. Recall, the motivation is to find a fifth-order linearly stable Runge-Kutta scheme with an embedded third-order SSPRK scheme. The linearly stable scheme could then be used in smooth regions of a WENO spatially discretized problem and the SSPRK scheme used near shocks or other discontinuities where spurious oscillations are more likely to develop. However, the techniques in this chapter, particularly the modified Verner's method in Section 4.2, are not limited to the hyperbolic conservation law application, and could potentially be used to find other types of embedded pairs.

We begin by finding some p-order Runge-Kutta schemes with $p \leq 4$ with embedded SSPRK schemes.

4.1 Specifying an RKSSP Scheme

Instead of solving the complete optimization problem (3.1) for an RK (s,p) scheme with embedded SSPRK (s^*,p^*) scheme, the problem can be simplified by specifying a particular SSPRK scheme and thereby decreasing the number of unknown coefficients.

Recalling the motivation of finding a fifth-order linearly stable scheme with embedded third-order SSP scheme, we concentrate on the problem with $p = 5$ and $p^* = 3$. For example with $s = 6$ and specifying the optimal SSP (3,3) scheme from Section 2.2.2, we have the partially complete Butcher tableau in Table 4.1. Tables 4.2 and 4.3 show two other embedded methods, and of course many others are possible.

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
1 & 1 & & & & & \\
1/2 & 1/4 & 1/4 & & & & \\
c_4 & a_{41} & a_{42} & a_{43} & & & \\
c_5 & a_{51} & a_{52} & a_{53} & a_{54} & & \\
c_6 & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & \\
\hline
\hat{\boldsymbol{b}} & 1/6 & 1/6 & 2/3 & 0 & 0 & 0 \\
\boldsymbol{b} & b_1 & b_2 & b_3 & b_4 & b_5 & b_6
\end{array}
$$

Table 4.1: Butcher tableau for an RK (6,5) scheme with the optimal SSP (3,3) scheme embedded.

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
1/2 & 1/2 & & & & & \\
1 & 1/2 & 1/2 & & & & \\
1/2 & 1/6 & 1/6 & 1/6 & & & \\
c_5 & a_{51} & a_{52} & a_{53} & a_{54} & & \\
c_6 & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & \\
\hline
\hat{\boldsymbol{b}} & 1/6 & 1/6 & 1/6 & 1/2 & 0 & 0 \\
\boldsymbol{b} & b_1 & b_2 & b_3 & b_4 & b_5 & b_6
\end{array}
$$

Table 4.2: Butcher tableau for an RK (6,5) scheme with the optimal SSP (4,3) scheme embedded.

We are left with the problem of finding the remaining coefficients such that the 17 order conditions are satisfied. We can do this by directly looking for a feasible solution or by converting the problem into one of optimization through several techniques.

One intuitive way of formulating an optimization problem is to maximize the linear stability radius ρ; that is, set the objective function to be ρ and specify the order conditions as constraints. Another option is to minimize the sum of the coefficients squared and again specify the order conditions as constraints. Unfortunately, BARON was not able to find any feasible solutions within a reasonable amount of computing time (3 or 4 days) using either of these ideas. Instead we seek a solution directly by solving the order conditions algebraically.

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
0.37727 & 0.37727 & & & & & & \\
0.75454 & 0.37727 & 0.37727 & & & & & \\
0.72899 & 0.24300 & 0.24300 & 0.24300 & & & & \\
0.69923 & 0.15359 & 0.15359 & 0.15359 & 0.23846 & & & \\
c_6 & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & & \\
c_7 & a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & \\
\hline
\hat{\boldsymbol{b}} & 0.20867 & 0.19871 & 0.11710 & 0.18561 & 0.28992 & 0 & 0 \\
\boldsymbol{b} & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7
\end{array}
$$

Table 4.3: Butcher tableau for an RK (7,5) scheme with the optimal SSP (5,3) scheme embedded.

4.2 Modified Verner's Method

In [Ver82], Verner presents a technique of deriving explicit Runge-Kutta methods which he compares to solving difficult jigsaw puzzles. Verner begins with $s = p$ and satisfies as many of the order conditions as possible. Additional stages are then introduced with zero weights and the remaining order conditions satisfied with the help of certain simplifying assumptions (see [HNW93]). Here we follow a modified technique which requires only that we assume p of the s nodes are distinct; for example, for an RK (6,5) method, we have to assume that some set of five of the six $\boldsymbol{c}$ coefficients are distinct. Put another way, one duplicated $\boldsymbol{c}$ coefficient is acceptable.

Consider the problem of embedding the known optimal SSP (3,3) scheme into an unknown RK (6,5) scheme. We begin by satisfying 11 of the 17 order conditions by means of a series of Vandermonde system inversions.

4.2.1 Six Stages with Embedded Optimal SSP (3,3)

The "broad tree" order conditions τ, t_{21}, t_{31}, t_{41}, and t_{51} can be written in the following Vandermonde matrix formulation

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & c_2 & c_3 & c_4 & c_5 & c_6 \\ 0 & c_2^2 & c_3^2 & c_4^2 & c_5^2 & c_6^2 \\ 0 & c_2^3 & c_3^3 & c_4^3 & c_5^3 & c_6^3 \\ 0 & c_2^4 & c_3^4 & c_4^4 & c_5^4 & c_6^4 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{pmatrix} = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \\ 1/5 \end{pmatrix}. \tag{4.1}$$

If the optimal SSP (3,3) scheme from Section 2.2.2 is embedded then $c_2 = 1$ and $c_3 = 1/2$ and the system can be rewritten as

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1/2 & c_4 & c_5 \\ 0 & 1 & 1/4 & c_4^2 & c_5^2 \\ 0 & 1 & 1/8 & c_4^3 & c_5^3 \\ 0 & 1 & 1/16 & c_4^4 & c_5^4 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \\ 1/5 \end{pmatrix} - b_6 \begin{pmatrix} 1 \\ c_6 \\ c_6^2 \\ c_6^3 \\ c_6^4 \end{pmatrix}, \tag{4.2}$$

and, assuming c_4 and c_5 are distinct from each other and from $\{0, 1/2, 1\}$, this system is invertible. The solution of this system uniquely determines b_1, b_2, b_3, b_4, and b_5 in terms of the free parameters c_4, c_5, c_6, and b_6.

Now consider the t_{21}, t_{32}, t_{43}, and t_{56} order conditions, which can be written in the following Vandermonde matrix system of *second-order homogeneous polynomials*

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1/2 & c_4 & c_5 \\ 0 & 1 & 1/4 & c_4^2 & c_5^2 \\ 0 & 1 & 1/8 & c_4^3 & c_5^3 \end{pmatrix} \begin{pmatrix} \sum b_j a_{j1} \\ \sum b_j a_{j2} \\ \sum b_j a_{j3} \\ \sum b_j a_{j4} \\ \sum b_j a_{j5} \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1/6 \\ 1/12 \\ 1/20 \end{pmatrix}, \tag{4.3}$$

where the second-order homogeneous polynomials are defined as

$$\begin{pmatrix} I_{61} \\ I_{62} \\ I_{63} \\ I_{64} \\ I_{65} \end{pmatrix} = \begin{pmatrix} \sum b_j a_{j1} \\ \sum b_j a_{j2} \\ \sum b_j a_{j3} \\ \sum b_j a_{j4} \\ \sum b_j a_{j5} \end{pmatrix}. \tag{4.4}$$

Rewriting the system (4.3) as

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1/2 & c_4 \\ 0 & 1 & 1/4 & c_4^2 \\ 0 & 1 & 1/8 & c_4^3 \end{pmatrix} \begin{pmatrix} I_{61} \\ I_{62} \\ I_{63} \\ I_{64} \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1/6 \\ 1/12 \\ 1/20 \end{pmatrix} - I_{65} \begin{pmatrix} 1 \\ c_5 \\ c_5^2 \\ c_5^3 \end{pmatrix}, \tag{4.5}$$

we invert to find I_{61}, I_{62}, I_{63}, and I_{64} in terms of c_4, c_5, and I_{65}.

Continuing with the t_{32}, t_{44}, and t_{58} order conditions, we can form the Vandermonde system

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1/2 & c_4 \\ 0 & 1 & 1/4 & c_4^2 \end{pmatrix} \begin{pmatrix} \sum b_j a_{jk} a_{k1} \\ \sum b_j a_{jk} a_{k2} \\ \sum b_j a_{jk} a_{k3} \\ \sum b_j a_{jk} a_{k4} \end{pmatrix} = \begin{pmatrix} 1/6 \\ 1/24 \\ 1/60 \end{pmatrix}. \tag{4.6}$$

Defining the third-order homogeneous polynomials as

$$I_{5l} = \sum_{j,k} b_j a_{jk} a_{kl}, \tag{4.7}$$

we invert the system

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1/2 \\ 0 & 1 & 1/4 \end{pmatrix} \begin{pmatrix} I_{51} \\ I_{52} \\ I_{53} \end{pmatrix} = \begin{pmatrix} 1/6 \\ 1/24 \\ 1/60 \end{pmatrix} - I_{54} \begin{pmatrix} 1 \\ c_4 \\ c_4^2 \end{pmatrix}, \tag{4.8}$$

to find I_{51}, I_{52}, and I_{53} in terms of c_4 and I_{54}.

The fourth-order homogenous polynomial equations come from the t_{44} and t_{59} order conditions written as the Vandermonde system

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1/2 \end{pmatrix} \begin{pmatrix} \sum b_j a_{jk} a_{kl} a_{l1} \\ \sum b_j a_{jk} a_{kl} a_{l2} \\ \sum b_j a_{jk} a_{kl} a_{l3} \end{pmatrix} = \begin{pmatrix} 1/24 \\ 1/120 \end{pmatrix}, \tag{4.9}$$

and, defining $I_{4m} = \sum_{j,k,l} b_j a_{jk} a_{kl} a_{lm}$, we invert the system

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} I_{41} \\ I_{42} \end{pmatrix} = \begin{pmatrix} 1/24 \\ 1/120 \end{pmatrix} - I_{43} \begin{pmatrix} 1 \\ 1/2 \end{pmatrix}, \tag{4.10}$$

to find I_{41} and I_{42} in terms of I_{43}.

Finally, we write the t_{59} order condition as

$$\begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} \sum b_j a_{jk} a_{kl} a_{lm} a_{m1} \\ \sum b_j a_{jk} a_{kl} a_{lm} a_{m2} \end{pmatrix} = \begin{pmatrix} 1/120 \end{pmatrix}, \tag{4.11}$$

and the solution for the fifth-order homogenous polynomials is

$$I_{31} = 1/120 - I_{32}. \tag{4.12}$$

The "jigsaw puzzle" is half done. The information gleaned so far can be summarized in the following *homogeneous polynomial tableau*[1]:

$$\begin{array}{|cccccc} I_{41} & I_{42} & \boxed{I_{43}} & & & \\ I_{51} & I_{52} & I_{53} & \boxed{I_{54}} & & \\ I_{61} & I_{62} & I_{63} & I_{64} & \boxed{I_{65}} & \\ \hline b_1 & b_2 & b_3 & b_4 & b_5 & \boxed{b_6} \end{array} \tag{4.13}$$

where the unboxed entries are known from the process above in terms of the boxed entries and the unknown entries of $\boldsymbol{c}$ (namely, c_4, c_5, c_6). Table 4.4 shows the homogeneous polynomial expressions associated with each of these I_{ij}.

We then proceed with a back-substitution algorithm starting in the bottom-right corner and working column-by-column to the left. Beginning in the fifth column, we find

$$a_{65} = \frac{I_{65}}{b_6}. \tag{4.14a}$$

We move to the fourth column and calculate

$$a_{54} = \frac{I_{54}}{I_{65}}, \tag{4.14b}$$

$$a_{64} = \frac{I_{64} - b_5 a_{54}}{b6}. \tag{4.14c}$$

Working down the third column, we find

$$a_{43} = \frac{I_{43}}{I_{54}}, \tag{4.14d}$$

$$a_{53} = \frac{I_{53} - b_5 a_{54} a_{43} - b_6 a_{64} a_{43}}{b_6 a_{65}}, \tag{4.14e}$$

$$a_{63} = \frac{I_{63} - b_4 a_{43} - b_5 a_{53}}{b_6}. \tag{4.14f}$$

[1]Note this is *not* a Butcher tableau.

$I_{21} = b_6 a_{65} a_{54} a_{43} a_{32} a_{21}$					
$I_{31} = \sum b_j a_{jk} a_{kl} a_{lm} a_{m1}$	$I_{32} = b_6 a_{65} a_{54} a_{43} a_{32}$				
$I_{41} = \sum b_j a_{jk} a_{kl} a_{l1}$	$I_{42} = \sum b_j a_{jk} a_{kl} a_{l2}$	$I_{43} = b_6 a_{65} a_{54} a_{43}$			
$I_{51} = \sum b_j a_{jk} a_{k1}$	$I_{52} = \sum b_j a_{jk} a_{k2}$	$I_{53} = \sum b_j a_{jk} a_{k3}$	$I_{54} = b_6 a_{65} a_{54}$		
$I_{61} = \sum b_j a_{j1}$	$I_{62} = \sum b_j a_{j2}$	$I_{63} = \sum b_j a_{j3}$	$I_{64} = \sum b_j a_{j4}$	$I_{65} = b_6 a_{65}$	
b_1	b_2	b_3	b_4	b_5	b_6

Table 4.4: Expressions in the homogeneous polynomial tableau for $s = 6$, $p = 5$.

$I_{21} = b_7 a_{76} a_{65} a_{54} a_{43} a_{32} a_{21}$						
$I_{31} = \sum b_j a_{jk} a_{kl} a_{lm} a_{mn} a_{n1}$	$I_{32} = b_7 a_{76} a_{65} a_{54} a_{43} a_{32}$					
$I_{41} = \sum b_j a_{jk} a_{kl} a_{lm} a_{m1}$	$I_{42} = \sum b_j a_{jk} a_{kl} a_{lm} a_{m2}$	$I_{43} = b_7 a_{76} a_{65} a_{54} a_{43}$				
$I_{51} = \sum b_j a_{jk} a_{kl} a_{l1}$	$I_{52} = \sum b_j a_{jk} a_{kl} a_{l2}$	$I_{53} = \sum b_j a_{jk} a_{kl} a_{l3}$	$I_{54} = b_7 a_{76} a_{65} a_{54}$			
$I_{61} = \sum b_j a_{jk} a_{k1}$	$I_{62} = \sum b_j a_{jk} a_{k2}$	$I_{63} = \sum b_j a_{jk} a_{k3}$	$I_{64} = \sum b_j a_{jk} a_{k4}$	$I_{65} = b_7 a_{76} a_{65}$		
$I_{71} = \sum b_j a_{j1}$	$I_{72} = \sum b_j a_{j2}$	$I_{73} = \sum b_j a_{j3}$	$I_{74} = \sum b_j a_{j4}$	$I_{75} = \sum b_j a_{j5}$	$I_{76} = b_7 a_{76}$	
b_1	b_2	b_3	b_4	b_5	b_6	b_7

Table 4.5: Expressions in the homogeneous polynomial tableau for $s = 7$, $p = 5$.

$I_{21} = b_8 a_{87} a_{76} a_{65} a_{54} a_{43} a_{32} a_{21}$							
$I_{31} = \sum b_j a_{jk} a_{kl} a_{lm} a_{mn} a_{no} a_{o1}$	$I_{32} = b_8 a_{87} a_{76} a_{65} a_{54} a_{43} a_{32}$						
$I_{41} = \sum b_j a_{jk} a_{kl} a_{lm} a_{mn} a_{n1}$	$I_{42} = \sum b_j a_{jk} a_{kl} a_{lm} a_{mn} a_{n2}$	$I_{43} = b_8 a_{87} a_{76} a_{65} a_{54} a_{43}$					
$I_{51} = \sum b_j a_{jk} a_{kl} a_{lm} a_{m1}$	$I_{52} = \sum b_j a_{jk} a_{kl} a_{lm} a_{m2}$	$I_{53} = \sum b_j a_{jk} a_{kl} a_{lm} a_{m3}$	$I_{54} = b_8 a_{87} a_{76} a_{65} a_{54}$				
$I_{61} = \sum b_j a_{jk} a_{kl} a_{l1}$	$I_{62} = \sum b_j a_{jk} a_{kl} a_{l2}$	$I_{63} = \sum b_j a_{jk} a_{kl} a_{l3}$	$I_{64} = \sum b_j a_{jk} a_{kl} a_{l4}$	$I_{65} = b_8 a_{87} a_{76} a_{65}$			
$I_{71} = \sum b_j a_{jk} a_{k1}$	$I_{72} = \sum b_j a_{jk} a_{k2}$	$I_{73} = \sum b_j a_{jk} a_{k3}$	$I_{74} = \sum b_j a_{jk} a_{k4}$	$I_{75} = \sum b_j a_{jk} a_{k5}$	$I_{76} = b_8 a_{87} a_{76}$		
$I_{81} = \sum b_j a_{j1}$	$I_{82} = \sum b_j a_{j2}$	$I_{83} = \sum b_j a_{j3}$	$I_{84} = \sum b_j a_{j4}$	$I_{85} = \sum b_j a_{j5}$	$I_{86} = \sum b_j a_{j6}$	$I_{87} = b_8 a_{87}$	
b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8

Table 4.6: Expressions in the homogeneous polynomial tableau for $s = 8$, $p = 5$.

Continuing in this fashion we can find expressions for each a_{ij} which depend on c_4, c_5, c_6, b_6, I_{65}, I_{54}, and I_{43}. In particular

$$a_{31} = \frac{1 - a_{21} - 120 I_{32} + 60 a_{21} I_{43}}{120 I_{43}} \tag{4.15a}$$

$$a_{32} = \frac{I_{32}}{I_{43}}. \tag{4.15b}$$

However, by our choice of the embedded SSP (3,3) scheme, these expressions for a_{31} and a_{32} should both equal $\frac{1}{4}$ and of course $a_{21} = c_2 = 1$. Solving (4.15), we find that for the method to contain the embedded optimal SSP (3,3) scheme[2] we must have

$$I_{32} = \frac{I_{43}}{4}. \tag{4.16}$$

Recall from Section 1.1.5 that the linear stability region for a RK (6,5) method is determined by one-contour of the expansion factor

$$\phi = 1 + z + \frac{z^2}{2!} + \cdots + \frac{z^5}{5!} + tt_6^{(6)} z^6, \tag{4.17}$$

where $tt_6^{(6)}$ is the "tall tree" of order 6, specifically,

$$tt_6^{(6)} = b_6 a_{65} a_{54} a_{43} a_{32} a_{21}. \tag{4.18}$$

Note that $tt_6 = I_{21} = I_{32} a_{21}$ and thus we have

$$tt_6^{(6)} = \frac{I_{43}}{4}. \tag{4.19}$$

The linear stability properties of our RK (6,5) scheme are therefore determined completely by our choice of I_{43}.

The formation and solution of the homogeneous polynomial equations has satisfied the order conditions τ, t_{21}, t_{31}, t_{32}, t_{41}, t_{43}, t_{44}, t_{51}, t_{56}, t_{59}. The remaining order conditions, t_{42}, t_{52}, t_{53}, t_{54}, t_{55}, and t_{57} define six equations that the a_{ij}, $\boldsymbol{c}$ and $\boldsymbol{b}$ must satisfy. Thus we have a system of six nonlinear equation in seven variables: c_4, c_5, c_6, I_{65}, I_{54}, I_{43} and b_6. Using `maple`, we can compute four solutions to this system shown in Tables 4.7, 4.8, 4.9, and 4.10.

Notice that each of the solutions contain free parameters and thus each corresponds to a family of embedded RK (6,5) / SSP (3,3) methods. We will analyze each of these families to find a "good" RK (6,5) / SSP (3,3) method based on the following approximately ranked criteria:

[2] At first glance this seems counterintuitive because we began by embedding the optimal SSP (3,3) scheme; however, until this point we had only used the node values $\boldsymbol{c}$ and not the particular a_{ij} values of SSP (3,3) scheme.

$$
\begin{aligned}
&\boxed{c_6},\\
c_4 &= 3/5,\\
I_{43} &= -1/75,\\
I_{54} &= -5/36,\\
c_5 &= \frac{3}{10}\frac{10c_6-7}{5c_6-3},\\
I_{65} &= -\frac{100}{27}\frac{(5c_6-3)^4}{(10c_6-7)(20c_6-9)(5c_6-6)},\\
b_6 &= -\frac{1}{4}\frac{1}{(50c_6^2-60c_6+21)(c_6-1)(2c_6-1)c_6(5c_6-3)},
\end{aligned}
$$

Table 4.7: The first `maple` solution with free parameter c_6.

$$
\begin{aligned}
&\boxed{I_{43}}, \quad \boxed{b_6},\\
c_6 &= 1,\\
c_5 &= 3/5-\sqrt{6}/10 \quad \text{or} \quad 3/5+\sqrt{6}/10,\\
c_4 &= 6/5-c_5 \quad \text{(the other root)},\\
I_{54} &= \frac{16}{45}c_5-\frac{1}{5},\\
I_{65} &= -\frac{5}{9}c_5+\frac{19}{36}
\end{aligned}
$$

Table 4.8: The second `maple` solution with free paremeters I_{43} and b_6.

$$\boxed{c_5},\quad \boxed{I_{43}},\quad \boxed{I_{65}},$$

$$c_6 = 1,$$

$$I_{54} = -60\frac{I_{65}^2 c_5^2\left(4c_5^2 - 4c_5 + 1\right)}{(5c_5-1)\left(120I_{65}c_5^3 - 60I_{65}c_5^2 - 1\right)\left(120I_{65}c_5^3 - 120I_{65}c_5^2 - 1 + 30I_{65}c_5\right)},$$

$$c_4 = \frac{1}{60}\frac{120I_{65}c_5^3 - 60I_{65}c_5^2 - 1}{I_{65}c_5\left(2c_5-1\right)},$$

$$b_6 = -\frac{I_{65}\left(5184I_{65}^2c_5^2 - 38016I_{65}^2c_5^3 + 104256I_{65}^2c_5^4 - 144I_{65}c_5 - 126720c_5^5I_{65}^2 + 528I_{65}c_5^2 - 480I_{65}c_5^3 + 1 + 57600c_5^6I_{65}^2\right)}{(c_5-1)\left(60I_{65}\left(2c_5^3 - 3c_5^2 + c_5 - 1\right)\right)\left(144I_{65}^2\left(100c_5^6 - 150c_5^5 + 60c_5^4 + 35c_5^3 - 38c_5^2 + 9c_5\right) - 12I_{65}\left(20c_5^3 + 15c_5^2 - c_5 - 2\right) + 1\right)},$$

Table 4.9: The third `maple` solution with free parameters c_5, I_{43} and I_{65}.

$$\boxed{I_{65}},\quad \boxed{c_5},$$

$$c_6 = 1,$$

$$c_4 = \frac{60(2c_5^3 - c_5^2)I_{65} - 1}{60(2c_5-1)c_5I_{65}},$$

$$I_{54} = \frac{-60(2c_5-1)^2c_5^2I_{65}^2}{(5c_5-1)\left(30(2c_5-1)^2c_5I_{65} - 1\right)\left(60(2c_5-1)c_5^2I_{65} - 1\right)},$$

$$b_6 = \frac{-576c_5^2(2c_5-1)^2(5c_5-3)^2I_{65}^3 + 48c_5(2c_5-1)(5c_5-3)I_{65}^2 - I_{65}}{\left(8640c_5^2(c_5-1)(50c_5^4 - 50c_5^3 + 5c_5^2 + 20c_5 - 9)(2c_5-1)^2I_{65}^2 - 144c5(2c_5-1)(150c_5^4 - 225c_5^3 + 85c_5^2 + 25c_5 - 19)I_{65}^2 + 24(15c_5^3 - 15c_5^2 + 3c_5 + 1)I_{65} - 1\right)(c_5-1)},$$

$$I_{43} = -\frac{60(2c_5^3 - 3c_5^2 + c_5)I_{65} - 1}{900(5c_5-1)(2c_5-1)c_5I_{65}}$$

Table 4.10: The fourth `maple` solution with free parameters c_5 and I_{65}.

1. Linear stability properties (large ρ and ρ_2 values).

2. Small magnitude A and $\boldsymbol{b}$ coefficients. Large magnitude A and $\boldsymbol{b}$ values can contribute to dangerous build up of rounding errors.

3. $b_2 = 0$. Without this property, the internal stages may be restricted to first-order (see [Ver82]).

Of course, other properties could also be used to evaluate the solutions.

The first family of solutions is shown in Table 4.7 and has one free parameter c_6. The linear stability properties of this family of schemes are fixed because I_{43} is fixed. In particular $tt_6^{(6)} = \frac{I_{43}}{4} \approx -.0033$, and examining Figure 1.4 this corresponds to a linear stability radius of $\rho \approx 1.25$ and a negligible linear stability imaginary axis inclusion ($\rho_2 \approx 0.024$).

The second solution shown in Table 4.8 has two free parameters I_{43} and b_6. Additionally, c_4 and c_5 are the roots of the quadratic $10x^2 - 12x + 3$. Here we choose $c_5 = 3/5 - \sqrt{6}/10$. Recalling from Section 1.1.5 that $tt_6^{(6)} \approx \frac{292}{100000}$ maximizes $\min(\rho, \rho_2)$, we choose $I_{43} = 4\frac{292}{100000}$. The resulting one-parameter family of methods is shown in Table 4.11a. By choosing a value for the parameter b_6, say $b_6 = \frac{1}{2}$, we obtain the embedded RK (6,5) / SSP (3,3) method shown in Table 4.11b. This method has optimal linear stability properties (in the sense that $\min(\rho, \rho_2)$ is maximized) and has no overly large coefficients. However, $b_2 \neq 0$ and thus the method cannot have high stage order.

The third `maple` solution is shown in Table 4.9 with free parameters c_5, I_{43}, and I_{65}. Again we optimize the linear stability properties by choosing $I_{43} = 4\frac{292}{100000}$. Next we chose $c_5 = \frac{2}{3}$ and I_{65} such that $b_2 = 0$; the resulting embedded RK (6,5) / SSP (3,3) method is shown in Table 4.12. This method has optimal linear stability properties, no overly large coefficients, and $b_2 = 0$.

The fourth `maple` solution is shown in Table 4.10 with free parameters c_5 and I_{65}. In this case, I_{43} is known in terms of the free parameters and thus to optimize the linear stability properties, we solve the equation

$$I_{43} = 4\frac{292}{100000}, \tag{4.20}$$

and find an expression for I_{65} in terms of c_5. The remaining parameter c_5 is chosen so that $b_2 = 0$ and no coefficients are overly large. Table 4.13 shows the resulting embedded RK (6,5) / SSP (3,3) method.

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
1 & 1 & & & & & \\
\frac{1}{2} & \frac{1}{4} & \frac{1}{4} & & & & \\
\frac{3}{5}+\frac{\sqrt{6}}{10} & \frac{6258}{15625}+\frac{1063}{15625}\sqrt{6} & \frac{6891}{31250}+\frac{2751}{31250}\sqrt{6} & -\frac{657}{31250}-\frac{876}{15625}\sqrt{6} & & & \\
\frac{3}{5}-\frac{\sqrt{6}}{10} & \frac{296}{3125}\sqrt{6}-\frac{807}{6250} & \frac{27}{6250}\sqrt{6}+\frac{54}{3125} & \frac{618}{3125}\sqrt{6}-\frac{231}{6250} & \frac{468}{625}-\frac{248}{625}\sqrt{6} & & \\
1 & 1+\frac{-\frac{2593}{45000}\sqrt{6}-\frac{157}{11250}}{b_6} & \frac{-\frac{177}{5000}\sqrt{6}-\frac{172}{1875}}{b_6} & \frac{-\frac{469}{7500}\sqrt{6}-\frac{31}{1875}}{b_6} & \frac{-\frac{13}{180}+\frac{\sqrt{6}}{10}}{b_6} & \frac{\frac{7}{36}+\frac{\sqrt{6}}{18}}{b_6} & \\
\hline
\hat{\boldsymbol{b}} & \frac{1}{6} & \frac{1}{6} & \frac{2}{3} & 0 & 0 & 0 \\
\boldsymbol{b} & \frac{1}{9} & -b_6 & 0 & \frac{4}{9}-\frac{\sqrt{6}}{36} & \frac{4}{9}+\frac{\sqrt{6}}{36} & b_6
\end{array}
$$

(a) One parameter family of methods

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
1 & 1 & & & & & \\
1/2 & 1/4 & 1/4 & & & & \\
0.84495 & 0.56717 & 0.43615 & -0.15835 & & & \\
0.35505 & 0.10290 & 0.027862 & 0.44746 & -0.22315 & & \\
1 & 0.68981 & -0.35689 & -0.33943 & 0.34546 & 0.66105 & \\
\hline
\hat{\boldsymbol{b}} & \frac{1}{6} & \frac{1}{6} & \frac{2}{3} & 0 & 0 & 0 \\
\boldsymbol{b} & 0.11111 & -1/2 & 0 & 0.37639 & 0.51250 & 1/2
\end{array}
$$

(b) $b_6 = \frac{1}{2}$ (Rounded to five decimal places)

Table 4.11: An embedded RK (6,5) / SSP (3,3) method obtaned from the second `maple` solution. This method has $z = 1$, $\rho \approx 1.4$ and $\rho_2 \approx 1.4$.

The RK (6,5) / SSP (3,3) methods in Tables 4.11, 4.12, and 4.13 all have $z = 1$ and $\rho = 1.4$ and thus they all the effective CFL coefficient

$$\frac{\min(z, \rho)}{s} = \frac{1}{6}, \tag{4.21}$$

and thus incur about twice the computational expense of the optimal SSP (3,3) scheme. However, using the WENO discretization idea, the method should be able to produce a more accurate fifth-order answer in smooth regions of the domain.

0						
1	1					
1/2	1/4	1/4				
1/5	2046/15625	−454/15625	1533/15625			
2/3	−739/5625	511/5625	−566/16875	20/27		
1	11822/21875	−6928/21875	−4269/21875	−4/7	54/35	
$\hat{\boldsymbol{b}}$	1/6	1/6	2/3	0	0	0
$\boldsymbol{b}$	1/24	0	0	125/336	27/56	5/48

(a) Exact

0						
1	1					
1/2	1/4	1/4				
.20000	0.13094	−0.029056	0.098112			
.66667	−0.13138	0.090844	−0.033541	0.74074		
1	0.54043	−0.31671	−0.19515	−0.57143	1.54286	
$\hat{\boldsymbol{b}}$	1/6	1/6	2/3	0	0	0
$\boldsymbol{b}$	0.041667	0	0	0.37202	0.48214	0.10417

(b) Rounded to five decimal places

Table 4.12: An embedded RK (6,5) / SSP (3,3) method obtained from the third `maple` solution. This method has $z = 1$, $\rho \approx 1.4$ and $\rho_2 \approx 1.4$.

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
1 & 1 & & & & & \\
1/2 & 1/4 & 1/4 & & & & \\
\frac{3093}{5000} - \frac{1}{5000}\omega & -\frac{133061}{976562500}\omega + \frac{403437673}{976562500} & -\frac{1435113}{7812500000}\omega + \frac{1836344509}{7812500000} & -\frac{231033393}{7812500000} + \frac{937101}{7812500000}\omega & & & \\
\frac{2783}{4380} + \frac{1}{4380}\omega & -\frac{262067215723}{888417573750000}\omega - \frac{216640215414989}{888417573750000} & -\frac{207982693}{4376441250000}\omega - \frac{139465003199}{4376441250000} & -\frac{427734676}{820582734375}\omega - \frac{164599620368}{820582734375} & \frac{1481472168480461}{1332626360625000} + \frac{1455325261177}{1332626360625000}\omega & & \\
1 & \frac{398687721063}{113116675000} - \frac{519005209}{113116675000}\omega & \frac{965133}{193750} - \frac{422}{96875}\omega & -\frac{438}{96875}\omega + \frac{287766}{96875} & -\frac{49326946366513}{38058333425000} + \frac{232132067059}{38058333425000}\omega & -\frac{49499360244}{5391909701} + \frac{39718278}{5391909701}\omega & \\
\hline
\hat{\boldsymbol{b}} & 1/6 & 1/6 & 2/3 & 0 & 0 & 0 \\
\boldsymbol{b} & \frac{1619195}{14011872} + \frac{31}{14011872}\omega & 0 & 0 & \frac{1441907950799}{3097309942944} + \frac{208054643}{3097309942944}\omega & \frac{778955284829}{1577916284112} - \frac{35622103}{1577916284112}\omega & -\frac{49507}{662256} - \frac{31}{662256}\omega
\end{array}
$$

(a) Exact ($\omega = \sqrt{1446649}$)

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
1 & 1 & & & & & \\
1/2 & 1/4 & 1/4 & & & & \\
0.37805 & 0.24924 & 0.014110 & 0.11470 & & & \\
0.91000 & -0.59864 & -0.089027 & -0.82754 & 2.42520 & & \\
1 & -1.99400 & -0.25808 & -2.46757 & 6.04004 & -0.32039 & \\
\hline
\hat{\boldsymbol{b}} & 1/6 & 1/6 & 2/3 & 0 & 0 & 0 \\
\boldsymbol{b} & 0.11822 & 0 & 0 & 0.54633 & .46651 & -0.13106
\end{array}
$$

(b) Rounded to five decimal places

Table 4.13: An embedded RK (6,5) / SSP (3,3) method obtained from the fourth `maple` solution. This method has $z = 1$, $\rho \approx 1.4$ and $\rho_2 \approx 1.4$.

4.2.2 Seven Stages with Embedded SSP (5,3)

Assume that we have embedded a 5-stage, third-order SSP method such as the optimal SSP (5,3) scheme in Section 2.2.2. Suppose, as is the case for the optimal SSP (5,3) scheme, that the values for $0, c_2, c_3, c_4, c_5$ are distinct.

The "broad tree" order conditions τ, t_{21}, t_{31}, t_{41}, and t_{51} can be written in the Vandermonde matrix formulation

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ 0 & c_2^2 & c_3^2 & c_4^2 & c_5^2 & c_6^2 & c_7^2 \\ 0 & c_2^3 & c_3^3 & c_4^3 & c_5^3 & c_6^3 & c_7^3 \\ 0 & c_2^4 & c_3^4 & c_4^4 & c_5^4 & c_6^4 & c_7^4 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \\ 1/5 \end{pmatrix}. \tag{4.22}$$

By the distinctness of c_2, c_3, c_4 and c_5, the system

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & c_2 & c_3 & c_4 & c_5 \\ 0 & c_2^2 & c_3^2 & c_4^2 & c_5^2 \\ 0 & c_2^3 & c_3^3 & c_4^3 & c_5^3 \\ 0 & c_2^4 & c_3^4 & c_4^4 & c_5^4 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \\ 1/4 \\ 1/5 \end{pmatrix} - b_6 \begin{pmatrix} 1 \\ c_6 \\ c_6^2 \\ c_6^3 \\ c_6^4 \end{pmatrix} - b_7 \begin{pmatrix} 1 \\ c_7 \\ c_7^2 \\ c_7^3 \\ c_7^4 \end{pmatrix}, \tag{4.23}$$

is invertible and the solution uniquely determines b_1, b_2, b_3, b_4, and b_5 in terms of the free parameters c_6, c_7, b_6, and b_7.

The t_{21}, t_{32}, t_{43}, and t_{56} order conditions can be written in the Vandermonde matrix system

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & c_2 & c_3 & c_4 & c_5 & c_6 \\ 0 & c_2^2 & c_3^2 & c_4^2 & c_5^2 & c_6^2 \\ 0 & c_2^3 & c_3^3 & c_4^3 & c_5^3 & c_6^3 \end{pmatrix} \begin{pmatrix} \sum b_j a_{j1} \\ \sum b_j a_{j2} \\ \sum b_j a_{j3} \\ \sum b_j a_{j4} \\ \sum b_j a_{j5} \\ \sum b_j a_{j6} \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1/6 \\ 1/12 \\ 1/20 \end{pmatrix}. \tag{4.24}$$

The second-order homogeneous polynomials are now defined as

$$I_{7k} = \sum_j b_j a_{jk}, \tag{4.25}$$

and rewriting system (4.24), we invert

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & c_2 & c_3 & c_4 \\ 0 & c_2^2 & c_3^2 & c_4^2 \\ 0 & c_2^3 & c_3^3 & c_4^3 \end{pmatrix} \begin{pmatrix} I_{71} \\ I_{72} \\ I_{73} \\ I_{74} \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1/6 \\ 1/12 \\ 1/20 \end{pmatrix} - I_{75} \begin{pmatrix} 1 \\ c_5 \\ c_5^2 \\ c_5^3 \end{pmatrix} - I_{76} \begin{pmatrix} 1 \\ c_6 \\ c_6^2 \\ c_6^3 \end{pmatrix}, \tag{4.26}$$

to we find I_{71}, I_{72}, I_{73}, and I_{74} in terms of c_6, I_{75} and I_{76}.

The t_{32}, t_{44}, and t_{58} order conditions form the Vandermonde system

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & c_2 & c_3 & c_4 & c_5 \\ 0 & c_2^2 & c_3^2 & c_4^2 & c_5^2 \end{pmatrix} \begin{pmatrix} \sum b_j a_{jk} a_{k1} \\ \sum b_j a_{jk} a_{k2} \\ \sum b_j a_{jk} a_{k3} \\ \sum b_j a_{jk} a_{k4} \\ \sum b_j a_{jk} a_{k5} \end{pmatrix} = \begin{pmatrix} 1/6 \\ 1/24 \\ 1/60 \end{pmatrix}. \tag{4.27}$$

We define the third-order homogeneous polynomials as $I_{6l} = \sum_{j,k} b_j a_{jk} a_{kl}$ and invert the system

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & c_2 & c_3 \\ 0 & c_2^2 & c_3^2 \end{pmatrix} \begin{pmatrix} I_{61} \\ I_{62} \\ I_{63} \end{pmatrix} = \begin{pmatrix} 1/6 \\ 1/24 \\ 1/60 \end{pmatrix} - I_{64} \begin{pmatrix} 1 \\ c_4 \\ c_4^2 \end{pmatrix} - I_{65} \begin{pmatrix} 1 \\ c_5 \\ c_5^2 \end{pmatrix}, \tag{4.28}$$

to find I_{61}, I_{62}, and I_{63} in terms of I_{64} and I_{65}.

The t_{44} and t_{59} order conditions form the Vandermonde system

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & c_2 & c_3 & c_4 \end{pmatrix} \begin{pmatrix} \sum b_j a_{jk} a_{kl} a_{l1} \\ \sum b_j a_{jk} a_{kl} a_{l2} \\ \sum b_j a_{jk} a_{kl} a_{l3} \\ \sum b_j a_{jk} a_{kl} a_{l4} \end{pmatrix} = \begin{pmatrix} 1/24 \\ 1/120 \end{pmatrix}, \tag{4.29}$$

and we invert the system

$$\begin{pmatrix} 1 & 1 \\ 0 & c_2 \end{pmatrix} \begin{pmatrix} I_{51} \\ I_{52} \end{pmatrix} = \begin{pmatrix} 1/24 \\ 1/120 \end{pmatrix} - I_{53} \begin{pmatrix} 1 \\ c_3 \end{pmatrix} - I_{54} \begin{pmatrix} 1 \\ c_4 \end{pmatrix}, \tag{4.30}$$

to find I_{51} and I_{52} in terms of I_{53} and I_{54}.

Finally, we write the t_{59} order condition as

$$\begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \sum b_j a_{jk} a_{kl} a_{lm} a_{m1} \\ \sum b_j a_{jk} a_{kl} a_{lm} a_{m2} \\ \sum b_j a_{jk} a_{kl} a_{lm} a_{m3} \end{pmatrix} = \begin{pmatrix} 1/120 \end{pmatrix}, \tag{4.31}$$

and find

$$I_{41} = 1/120 - I_{42} - I_{43}. \tag{4.32}$$

The homogeneous polynomial tableau for the RK (7,5) scheme is thus

$$\begin{array}{|ccccccc}
I_{41} & \boxed{I_{42}} & \boxed{I_{43}} & & & & \\
I_{51} & I_{52} & \boxed{I_{53}} & \boxed{I_{54}} & & & \\
I_{61} & I_{62} & I_{63} & \boxed{I_{64}} & \boxed{I_{65}} & & \\
I_{71} & I_{72} & I_{73} & I_{74} & \boxed{I_{75}} & \boxed{I_{76}} & \\
\hline
b_1 & b_2 & b_3 & b_4 & b_5 & \boxed{b_6} & \boxed{b_7}
\end{array} \tag{4.33}$$

where the unboxed entries have been determined above in terms of the boxed entries and c_6 and c_7. Table 4.5 shows the homogeneous polynomials associated with each of these I_{ij}.

Using the back-substitution algorithm, we find

$$a_{76} = \frac{I_{76}}{b_7}, \tag{4.34a}$$

$$a_{65} = \frac{I_{65}}{I_{76}}, \tag{4.34b}$$

$$a_{75} = \frac{I_{75} - b_6 a_{65}}{b_7}, \tag{4.34c}$$

$$a_{54} = \frac{I_{54}}{I_{65}}, \tag{4.34d}$$

$$a_{64} = \frac{I_{64} - b_6 a_{65} a_{54} - b_7 a_{75} a_{54}}{b_7 a_{76}}, \tag{4.34e}$$

$$a_{74} = \frac{I_{74} - b_5 a_{54} - b_6 a_{64}}{b_7}, \tag{4.34f}$$

$$a_{43} = \frac{I_{43}}{I_{54}}, \tag{4.34g}$$

$$\ddots$$

Continuing in this fashion we can find expressions for each a_{ij}, $i \geq 4$ which depend on c_6, c_7, b_6, b_7, I_{76}, I_{75}, I_{65}, I_{64}, I_{54}, I_{53}, I_{43}, and I_{42}. Of course, these expressions also depend on $c_2 = a_{21}$, c_3, c_4, c_5, a_{31}, and a_{32} but these have already been specified by our choice of the SSP (5,3) scheme. Indeed this choice also specifies values for the $a_{4j}, 1 \leq j \leq 3$ and $a_{5j}, 1 \leq j \leq 4$ expressions: this gives a system of seven equations which must be satisfied for the RK (7,5) scheme to contain the specified SSP (5,3) scheme.

Perhaps surprisingly, these seven equations depend only on I_{42}, I_{43}, I_{53}, I_{54}, I_{64}, and I_{65}; specifically they are *independent* of I_{76}, I_{75}, c_6, c_7, b_6, and b_7. In fact, `maple` can find a solution to these equations with one free parameter, that is we can solve for any five of them to be linear in terms of the sixth. Suppose we choose I_{65} to be the free parameter. We will use this free parameter to maximize the linear stability region of the RK (7,5) scheme.

Maximizing the Linear Stability Region

Recall that the linear stability region for an RK (7,5) method is determined by one-contour of the expansion factor

$$\phi = 1 + \frac{z}{2} + \frac{z^2}{6} + \cdots + \frac{z^5}{120} + tt_6^{(7)} z^6 + tt_7^{(7)} z^7, \tag{4.35}$$

where $tt_6^{(7)}$ and $tt_7^{(7)}$ are the "tall trees" of order 6 and 7, specifically:

$$\begin{aligned} tt_6^{(7)} = b_6 a_{65} a_{54} a_{43} a_{32} c_2 + b_7 \big(& a_{75} a_{54} a_{43} a_{32} c_2 + a_{76} a_{64} a_{43} a_{32} c_2 \\ & + a_{76} a_{65} a_{53} a_{32} c_2 + a_{76} a_{65} a_{54} a_{42} c_2 + a_{76} a_{65} a_{54} a_{43} c_3 \big), \end{aligned} \tag{4.36}$$

$$tt_7^{(7)} = b_7 a_{76} a_{65} a_{54} a_{43} a_{32} a_{21}. \tag{4.37}$$

These two expressions are dependent only on I_{42}, I_{43}, I_{53}, I_{54}, I_{64}, and I_{65} and thus using our solution from above, only on I_{65}. For example, using the optimal SSP (5,3) scheme, the tall trees become

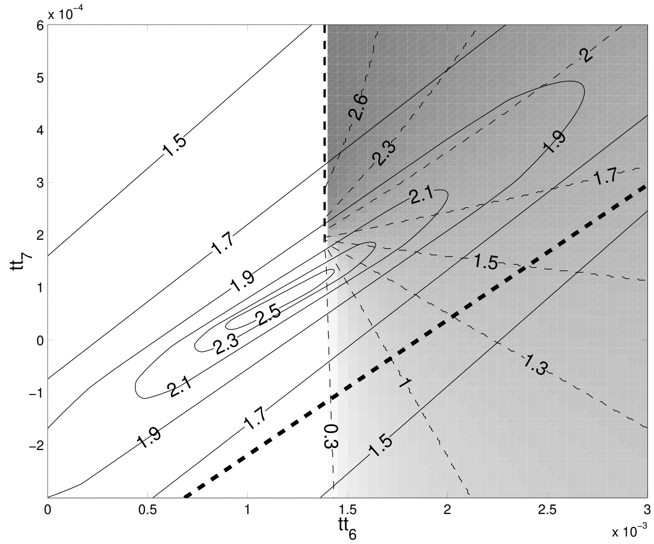
$$tt_6^{(7)} \approx .102 \times 10^{-1} I_{65} + .185 \times 10^{-2}, \tag{4.38}$$

$$tt_7^{(7)} \approx .825 \times 10^{-2} I_{65}, \tag{4.39}$$

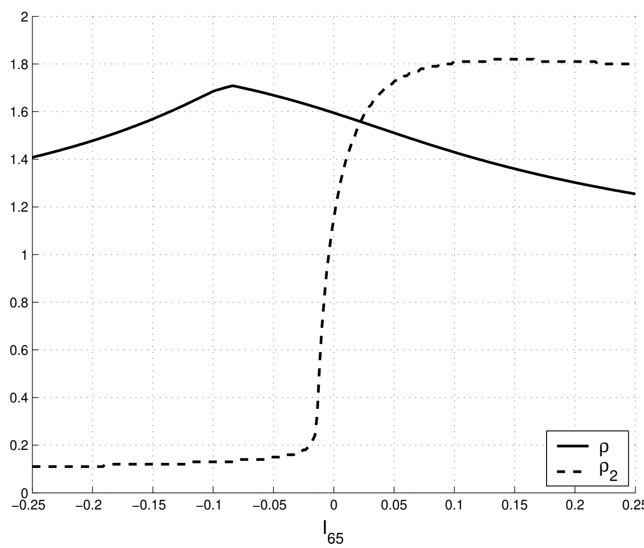
which define a straight line through the $tt_6^{(7)}$-$tt_7^{(7)}$ space shown in Figure 4.1a. By examining the cross-section of $tt_6^{(7)}$–$tt_7^{(7)}$ space defined by this line (see Figure 4.1b), we can choose I_{65} such that the resulting RK (75) scheme has linear stability properties that are in some sense *optimal*. A possible choice is to maximize $\min(\rho, \rho_2)$,i.e., maximize the minimum of the linear stability radius and the linear stability imaginary axis inclusion. In this case, by choosing $I_{65} = .02215$, we find $\rho \approx \rho_2 \approx 1.56$.

Solving the Remaining Order Conditions

Six order conditions, namely t_{42}, t_{52}, t_{53}, t_{54}, t_{55}, and t_{57}, have not yet been satisfied by the solution of the homogeneous polynomial equations above.



(a) Linear stability disc radius (solid contours) and imaginary axis inclusion (shading and dotted contours).



(b) Cross-section along dashed line in (a). Linear stability disc radius (solid line) and imaginary axis inclusion (dash-dot line).

Figure 4.1: Matching the coefficients of the RK (75) scheme with the optimal SSP (53) scheme restricts the values of $tt_6^{(7)}$ and $tt_7^{(7)}$ to a line parameterized by I_{65} (dashed line in (a)). A judicious choice of I_{65} will result in a scheme with optimal linear stability properties.

Solving these equations with `maple` results in the Butcher tableau shown in Table 4.14. This method has an effective CFL coefficient of

$$\frac{\min(z,\rho)}{s} = \frac{1.56}{7} = 0.222, \tag{4.40}$$

and is thus about 50% more computationally expensive then the optimal SSP (3,3) scheme. However, using the WENO discretization idea, the method should be able to produce a more accurate fifth-order answer in smooth regions of the domain.

There is another family of solutions with one free parameter but the magnitudes of its coefficients are unreasonably large regardless of the parameter. A typical Butcher tableau is shown in Table 4.15.

0							
0.37727	0.37727						
0.75454	0.37727	0.37727					
0.72899	0.24300	0.24300	0.24300				
0.69923	0.15359	0.15359	0.15359	0.23846			
0.88817	0.11302	1.49947	0.13475	−1.06421	0.20515		
1	−0.51211	3.91736	−0.04705	−0.21862	−1.64544	−0.49413	
$\hat{\boldsymbol{b}}$	0.20673	0.20673	0.11710	0.18180	0.28763	0	0
$\boldsymbol{b}$	0.12210	0.49290	−0.23202	−1.98395	1.85394	0.96554	−0.21851

Table 4.14: An embedded RK (7,5) / SSP (5,3) scheme with $z \approx 2.65$, $\rho \approx 1.56$ and $\rho_2 \approx 1.56$.

0							
0.37727	0.37727						
0.75454	0.37727	0.37727					
0.72899	0.24300	0.24300	0.24300				
0.69923	0.15359	0.15359	0.15359	0.23846			
0.15811	−2.69551	2.56572	−10.79002	10.38573	0.69219		
1	−0.28039	3.44396	−22.22852	38.01971	−18.39261	0.43783	
$\hat{\boldsymbol{b}}$	0.20673	0.20673	0.11710	0.18180	0.28763	0	0
$\boldsymbol{b}$	0.10015	0.57233	−3.25595	7.85429	−4.38191	0.03801	0.07309

Table 4.15: A poor embedded RK (7,5) / SSP (5,3) scheme with $z \approx 2.65$, $\rho \approx 1.56$ and $\rho_2 \approx 1.56$. The coefficients of this scheme are unreasonably large.

Chapter 5

Concluding Remarks

In this thesis, we have successfully constructed the first high-order Runge-Kutta methods with embedded strong-stability-preserving pairs. Specifically, we have found families of 6-stage fifth-order Runge-Kutta methods with embedded third-order SSP pairs and one 7-stage fifth-order Runge-Kutta method with an embedded third-order SSP pair.

Although the original motivation for such methods came from the solution of hyperbolic conservation laws whose solutions contain both smooth and non-smooth regions, the techniques we developed and to some extent, the methods themselves, are more general than that. In particular, the modified Verner's technique in Chapter 4 can be used to embed a given Runge-Kutta method in a larger method under very general assumptions. For example, this technique could be used to build error control pairs for existing Runge-Kutta schemes.

Additionally, Chapters 2 and 3 showed that GAMS/BARON is a viable software package for constructing strong-stability-preserving Runge-Kutta schemes and lower-order embedded methods. For lower-order schemes, BARON can find global optima and even on higher-order methods, BARON is often quick to locate feasible solutions which are likely close to optimal.

It remains to be investigated how effective the new embedded methods will be for the the time evolution of hyperbolic conservation laws. The question of automatic stepsize control for strong-stability-preserving methods is also worthy of future research.

Appendix A

GAMS/BARON Codes

A.1 Example Optimal SSP Input Files

ssp33.gms

```
$ eolcom #

positive variables
    k10
    k20, k21
    b1, b2, b3
    r21
    r31, r32
    ;

# objective cannot be declared positive
variables
    z
    ;

k10.up = 1       ;
k20.up = 1       ; k21.up = 1       ;
b1.up = 1       ; b2.up = 1       ; b3.up = 1       ;
r21.up = 1;
r31.up = 1; r32.up = 1;
z.up = 4       ;
z.lo = -100       ;

# initial guess for minos run (not required if baron is run first)
#k10.l = .5     ;
#k20.l = .333333; k21.l = .333333;
#b1.l = .333333; b2.l = .333333; b3.l = .333333;
#r21.l = .5     ;
#r31.l = .333333; r32.l = .333333;
#z.l = -48       ;
```

```
equations
    zc10
    zc20, zc21
    zc30, zc31, zc32
    ar20, ar30
    bp20
    bp30, bp31
    t11
    t21
    t31, t32
    ;

# SSP conditions
# conditions arising from dummy var z:
zc10 .. 1-z*k10 =G= 0;
zc20 .. 1-r21-z*(k20-r21*k10) =G= 0;
zc21 .. r21-z*k21 =G= 0;
zc30 .. 1-r31-r32-z*(b1-r31*k10-r32*k20) =G= 0;
zc31 .. r31-z*(b2-r32*k21) =G= 0;
zc32 .. r32-z*b3 =G= 0;
# each row of alpha must sum to 1:
ar20 .. 1-r21 =G= 0;
ar30 .. 1-r31-r32 =G= 0;
# each beta must be >= 0:
bp20 .. k20-r21*k10 =G= 0;
bp30 .. b1-r31*k10-r32*k20 =G= 0;
bp31 .. b2-r32*k21 =G= 0;

# Order Conditions
t11 .. b1+b2+b3 =E= 1;
t21 .. 2*( b2*k10+b3*(k20+k21) ) =E= 1;
t31 .. 3*( b2*k10*k10+b3*(k20+k21)*(k20+k21) ) =E= 1;
t32 .. 6*( b3*k21*k10 ) =E= 1;

# decimals only affects the display command and cannot be > 8
option decimals = 8;

# BARON run:
model m /all/;
option nlp = baron;
m.optfile = 1;
m.workspace = 500;
solve m maximizing z using nlp;

# MINOS run:
model m2 /all/;
option nlp = minos;
option sysout = on;
m2.optfile = 1;
solve m2 maximizing z using nlp;

variables
```

```
    r20, r30
    beta10
    beta20, beta21
    beta30, beta31, beta32
    ;
r20.l = 1 - r21.l;
r30.l = 1 - r31.l - r32.l;
beta10.l = k10.l;
beta20.l = k20.l - r21.l*k10.l;
beta21.l = k21.l;
beta30.l = b1.l - r31.l*k10.l - r32.l*k20.l;
beta31.l = b2.l - r32.l*k21.l;
beta32.l = b3.l;
file out / ssp33.coeff /;
put out;
out.nd=15;
out.nr=0;
out.nz=0;
out.nw=22;

put z.lo, z.l, z.up/;
put '# A matrix:'//;
put k10.l/;
put k20.l, k21.l/;
put '# b vector:'/;
put b1.l, b2.l, b3.l/;
put /'# alpha matrix:'/;
put 1/;
put r20.l, r21.l/;
put r30.l, r31.l, r32.l/;
put '# beta matrix:'/;
put beta10.l/;
put beta20.l, beta21.l/;
put beta30.l, beta31.l, beta32.l/;

putclose out;
```

A.2 Example Embedded RK/SSP Input File

rk54_ssp33.gms

```
$ eolcom #

positive variables
    k10
    k20, k21
    bh1, bh2, bh3
    r21
    r31, r32
    ;
```

```
variables
    k30, k31, k32
    k40, k41, k42, k43
    b1, b2, b3, b4, b5

variables
    z
    ;

k10.up = 5;
k20.up = 5; k21.up = 5;
bh1.up = 5; bh2.up = 5; bh3.up = 5;
r21.up = 1;
r31.up = 1; r32.up = 1;
z.up = 4;
z.lo = -100;

k30.up = 10; k31.up = 10; k32.up = 10;
k40.up = 10; k41.up = 10; k42.up = 10; k43.up = 10;
b1.up = 10; b2.up = 10; b3.up = 10; b4.up = 10; b5.up = 10;
k30.lo = -10; k31.lo = -10; k32.lo = -10;
k40.lo = -10; k41.lo = -10; k42.lo = -10; k43.lo = -10;
b1.lo = -10; b2.lo = -10; b3.lo = -10; b4.lo = -10; b5.lo = -10;

# initial guess for minos run (not required if baron is run first)
#k10.l = .5     ;
#k20.l = .333333; k21.l = .333333;
#bh1.l = .333333; bh2.l = .333333; bh3.l = .333333;
#r21.l = .5     ;
#r31.l = .333333; r32.l = .333333;
#z.l = 2.5     ;

equations
    zc10
    zc20, zc21
    zc30, zc31, zc32
    ar20, ar30
    bp20
    bp30, bp31
    th11
    th21
    th31, th32
#   c4up, c4lo
#   c5up, c5lo
    t11
    t21
    t31, t32
    t41, t42, t43, t44
    ;


# SSP conditions
# conditions arising from dummy var z:
```

```
zc10 .. 1-z*k10 =G= 0;
zc20 .. 1-r21-z*(k20-r21*k10) =G= 0;
zc21 .. r21-z*k21 =G= 0;
zc30 .. 1-r31-r32-z*(bh1-r31*k10-r32*k20) =G= 0;
zc31 .. r31-z*(bh2-r32*k21) =G= 0;
zc32 .. r32-z*bh3 =G= 0;
# each row of alpha must sum to 1:
ar20 .. 1-r21 =G= 0;
ar30 .. 1-r31-r32 =G= 0;
# each beta must be >= 0:
bp20 .. k20-r21*k10 =G= 0;
bp30 .. bh1-r31*k10-r32*k20 =G= 0;
bp31 .. bh2-r32*k21 =G= 0;

# Order Conditions for SSP scheme
th11 .. bh1+bh2+bh3 =E= 1;
th21 .. 2*( bh2*k10+bh3*(k20+k21) ) =E= 1;
th31 .. 3*( bh2*k10*k10+bh3*(k20+k21)*(k20+k21) ) =E= 1;
th32 .. 6*( bh3*k21*k10 ) =E= 1;

# each c_i should be in [0,1]
#c4up .. k30 + k31 + k32 =L= 1;
#c4lo .. k30 + k31 + k32 =G= 0;
#c5up .. k40 + k41 + k42 + k43 =L= 1;
#c5lo .. k40 + k41 + k42 + k43 =G= 0;

# Order Conditions for RK scheme
t11 .. b1+b2+b3+b4+b5 =E= 1;
t21 .. 2*( b2*k10+b3*(k20+k21)+b4*(k30+k31+k32)+b5*(k40+k41+k42+k43) ) =E= 1;
t31 .. 3*(b2*k10*k10+b3*(k20+k21)*(k20+k21)+b4*(k30+k31+k32)*(k30+k31+k32) +
b5*(k40+k41+k42+k43)*(k40+k41+k42+k43)) =E= 1;
t32 .. 6*(b3*k21*k10+b4*k31*k10+b4*k32*(k20+k21)+b5*k41*k10+b5*k42*(k20+k21) +
b5*k43*(k30+k31+k32)) =E= 1;
t41 .. 4*(b2*k10*k10*k10+b3*(k20+k21)*(k20+k21)*(k20+k21) +
b4*(k30+k31+k32)*(k30+k31+k32)*(k30+k31+k32) +
b5*(k40+k41+k42+k43)*(k40+k41+k42+k43)*(k40+k41+k42+k43)) =E= 1;
t42 .. 8*( b3*k21*k10*(k20+k21)+b4*k31*k10*(k30+k31+k32) +
b4*k32*(k20+k21)*(k30+k31+k32)+b5*k41*k10*(k40+k41+k42+k43) +
b5*k42*(k20+k21)*(k40+k41+k42+k43) +
        b5*k43*(k30+k31+k32)*(k40+k41+k42+k43) ) =E= 1;
t43 .. 12*( b3*k21*k10*k10+b4*k31*k10*k10+b4*k32*(k20+k21)*(k20+k21) +
b5*k41*k10*k10+b5*k42*(k20+k21)*(k20+k21) +
        b5*k43*(k30+k31+k32)*(k30+k31+k32) ) =E= 1;
t44 .. 24*( b4*k32*k21*k10+b5*k42*k21*k10+b5*k43*k31*k10 +
        b5*k43*k32*(k20+k21) ) =E= 1;


# BARON run:
model m /all/;
option nlp = baron;
m.optfile = 1;
m.workspace = 400;
solve m maximizing z using nlp;
```

```
# MINOS run:
#model m2 /all/;
#option nlp = minos;
#option sysout = on;
#m2.optfile = 1;
#solve m2 maximizing z using nlp;

variables
    r20, r30
    beta10
    beta20, beta21
    beta30, beta31, beta32
    ;
r20.l = 1 - r21.l;
r30.l = 1 - r31.l - r32.l;
beta10.l = k10.l;
beta20.l = k20.l - r21.l*k10.l;
beta21.l = k21.l;
beta30.l = bh1.l - r31.l*k10.l - r32.l*k20.l;
beta31.l = bh2.l - r32.l*k21.l;
beta32.l = bh3.l;
file out / rk54_ssp33.coeff /;
put out;
out.nd=15;
out.nr=0;
out.nz=0;
out.nw=22;

put z.lo, z.l, z.up/;
put '# A matrix:'//;
put k10.l/;
put k20.l, k21.l/;
put k30.l, k31.l, k32.l/;
put k40.l, k41.l, k42.l, k43.l/;
put '# bh vector:'/;
put bh1.l, bh2.l, bh3.l/;
put '# b vector:'/;
put b1.l, b2.l, b3.l, b4.l, b5.l/;
put /'# alpha matrix:'/;
put 1/;
put r20.l, r21.l/;
put r30.l, r31.l, r32.l/;
put '# beta matrix:'/;
put beta10.l/;
put beta20.l, beta21.l/;
put beta30.l, beta31.l, beta32.l/;

putclose out;
```

Appendix B

Maple Worksheets

B.1 generate_gams_ssp.mws

```
# Notes:
#     the resulting .gms file needs to the all of the "^" replaced with
# "**" or alternatively powers could be expanded
> restart:
> with(LinearAlgebra):
# Number of stages and order.  Note only s <= 8, p <= 5 is supported
# without making changes below
> s := 7; p := 5;

                                s := 7


                                p := 5

# Upper bound on each k_ij, upper and lower bounds on z
> KUP := 1; ZUP := 4; ZLO := 1;

                               KUP := 1


                               ZUP := 4


                               ZLO := 1

# Size of workspace:
> WORKSPACE := 500;

                           WORKSPACE := 500

# The output filename: (will be overwritten if exists)
> GAMS_FILENAME := sprintf("ssp%d%d.gms", s, p);

                      GAMS_FILENAME := "ssp75.gms"

# Filename that GAMS should store the coefficients in:
> COEF_FILENAME := sprintf("ssp%d%d.coeff", s, p);

                     COEF_FILENAME := "ssp75.coeff"

# Shouldn't need to change anything past here
> fd := fopen(GAMS_FILENAME, WRITE);

                               fd := 0

# Header
```

```
> fprintf(fd, "$ eolcom #\n"):
> #fprintf(fd, "$ inlinecom /* */\n"):
> fprintf(fd, "\n"):
>
# Optionally put some comments at the top of the GAMS file:
> # fprintf(fd,"# SSP65 example\n\n"):
>
# Define A, b, alpha, beta
# Need to make this bigger if you want to support s > 8:
> A := Matrix([[0,0,0,0,0,0,0,0], [k10,0,0,0,0,0,0,0],
> [k20,k21,0,0,0,0,0,0], [k30,k31,k32,0,0,0,0,0],
> [k40,k41,k42,k43,0,0,0,0], [k50,k51,k52,k53,k54,0,0,0],
> [k60,k61,k62,k63,k64,k65,0,0], [k70,k71,k72,k73,k74,k75,k76,0]],
> readonly=true);
                [ 0     0     0     0     0     0     0    0]
                [                                           ]
                [k10    0     0     0     0     0     0    0]
                [                                           ]
                [k20   k21    0     0     0     0     0    0]
                [                                           ]
                [k30   k31   k32    0     0     0     0    0]
           A := [                                           ]
                [k40   k41   k42   k43    0     0     0    0]
                [                                           ]
                [k50   k51   k52   k53   k54    0     0    0]
                [                                           ]
                [k60   k61   k62   k63   k64   k65    0    0]
                [                                           ]
                [k70   k71   k72   k73   k74   k75   k76   0]

> b := Vector([b1,b2,b3,b4,b5,b6,b7,b8], readonly=true);
> #b := Vector([bh1,bh2,bh3,bh4,bh5,bh6,bh7,bh8], readonly=true);
                                    [b1]
                                    [  ]
                                    [b2]
                                    [  ]
                                    [b3]
                                    [  ]
                                    [b4]
                               b := [  ]
                                    [b5]
                                    [  ]
                                    [b6]
                                    [  ]
                                    [b7]
                                    [  ]
                                    [b8]

# First column will not be used for optimization:
> alpha := Matrix([[1,0,0,0,0,0,0,0], [r20,r21,0,0,0,0,0,0],
> [r30,r31,r32,0,0,0,0,0], [r40,r41,r42,r43,0,0,0,0],
> [r50,r51,r52,r53,r54,0,0,0], [r60,r61,r62,r63,r64,r65,0,0],
> [r70,r71,r72,r73,r74,r75,r76,0], [r80,r81,r82,r83,r84,r85,r86,r87]],
> readonly=true);
                [ 1     0     0     0     0     0     0     0 ]
                [                                             ]
                [r20   r21    0     0     0     0     0     0 ]
                [                                             ]
                [r30   r31   r32    0     0     0     0     0 ]
                [                                             ]
                [r40   r41   r42   r43    0     0     0     0 ]
       alpha := [                                             ]
                [r50   r51   r52   r53   r54    0     0     0 ]
                [                                             ]
                [r60   r61   r62   r63   r64   r65    0     0 ]
                [                                             ]
                [r70   r71   r72   r73   r74   r75   r76    0 ]
                [                                             ]
                [r80   r81   r82   r83   r84   r85   r86   r87]

> c := Vector(s):
> for j from 1 to s do
>   for k from 1 to s do
```

```
>     c[j] := c[j] + A[j,k];
>   end;
> end;
> VectorOptions(c, readonly=true);
> c := c;
                    [                0               ]
                    [                                ]
                    [               k10              ]
                    [                                ]
                    [            k20 + k21           ]
                    [                                ]
               c := [         k30 + k31 + k32        ]
                    [                                ]
                    [      k40 + k41 + k42 + k43     ]
                    [                                ]
                    [   k50 + k51 + k52 + k53 + k54  ]
                    [                                ]
                    [k60 + k61 + k62 + k63 + k64 + k65]

> beta := Matrix(s,s):
> for i from 1 to (s-1) do
>   for k from 1 to i do
>     beta[i,k] := A[i+1,k] - sum('alpha[i,j+1]*A[j+1,k]','j'=k..i-1):
>   end do:
> end do:
> for k from 1 to s do
>   beta[s,k] := b[k] - sum('alpha[i,j+1]*A[j+1,k]','j'=k..i-1):
> end do:
> MatrixOptions(beta, readonly=true);
> beta := beta;

  beta :=

        [k10 , 0 , 0 , 0 , 0 , 0 , 0]

        [k20 - r21 k10 , k21 , 0 , 0 , 0 , 0 , 0]

        [k30 - r31 k10 - r32 k20 , k31 - r32 k21 , k32 , 0 , 0 , 0 ,

        0]

        [k40 - r41 k10 - r42 k20 - r43 k30 , k41 - r42 k21 - r43 k31

        , k42 - r43 k32 , k43 , 0 , 0 , 0]

        [k50 - r51 k10 - r52 k20 - r53 k30 - r54 k40 ,

        k51 - r52 k21 - r53 k31 - r54 k41 , k52 - r53 k32 - r54 k42 ,

        k53 - r54 k43 , k54 , 0 , 0]

        [k60 - r61 k10 - r62 k20 - r63 k30 - r64 k40 - r65 k50 ,

        k61 - r62 k21 - r63 k31 - r64 k41 - r65 k51 ,

        k62 - r63 k32 - r64 k42 - r65 k52 , k63 - r64 k43 - r65 k53 ,

        k64 - r65 k54 , k65 , 0]

        [b1 - r71 k10 - r72 k20 - r73 k30 - r74 k40 - r75 k50

         - r76 k60 ,

        b2 - r72 k21 - r73 k31 - r74 k41 - r75 k51 - r76 k61 ,

        b3 - r73 k32 - r74 k42 - r75 k52 - r76 k62 ,

        b4 - r74 k43 - r75 k53 - r76 k63 , b5 - r75 k54 - r76 k64 ,

        b6 - r76 k65 , b7]

#
>
# Variables & Bounds
# Variables
```

```
> fprintf(fd, "positive variables\n"):
> for j from 2 to s do
>   fprintf(fd, "    "):
>   for k from 1 to (j-2) do
>     printf(     "%s, ", convert(A[j,k],string)):
>     fprintf(fd, "%s, ", convert(A[j,k],string)):
>   end do:
>   printf(     "%s\n", convert(A[j,j-1],string)):
>   fprintf(fd, "%s\n", convert(A[j,j-1],string)):
> end do:
k10
k20, k21
k30, k31, k32
k40, k41, k42, k43
k50, k51, k52, k53, k54
k60, k61, k62, k63, k64, k65
> fprintf(fd, "    "):
> for j from 1 to (s-1) do
>   printf(     "%s, ", convert(b[j],string)):
>   fprintf(fd, "%s, ", convert(b[j],string)):
> end do:
> printf(     "%s\n", convert(b[s],string)):
> fprintf(fd, "%s\n", convert(b[s],string)):
>
b1, b2, b3, b4, b5, b6,
b7
> for j from 2 to s do
>   fprintf(fd, "    "):
>   for k from 2 to (j-1) do
>     printf(     "%s, ", convert(alpha[j,k],string)):
>     fprintf(fd, "%s, ", convert(alpha[j,k],string)):
>   end do:
>   printf(     "%s\n", convert(alpha[j,j],string)):
>   fprintf(fd, "%s\n", convert(alpha[j,j],string)):
> end do:
r21
r31, r32
r41, r42, r43
r51, r52, r53, r54
r61, r62, r63, r64, r65
r71, r72, r73, r74, r75, r76
> fprintf(fd, "    ;\n\n"):
> fprintf(fd, "# objective cannot be declared positive\n"):
> fprintf(fd, "variables\n    z\n    ;\n\n"):
# Bounds
# Upper bounds on matrix A
> for j from 2 to s do
>   for k from 1 to (j-2) do
>     printf(     "%s.up = %g; ", convert(A[j,k],string), KUP):
>     fprintf(fd, "%s.up = %g; ", convert(A[j,k],string), KUP):
>   end do:
>   printf(     "%s.up = %g;\n", convert(A[j,j-1],string), KUP):
>   fprintf(fd, "%s.up = %g;\n", convert(A[j,j-1],string), KUP):
> end do:
k10.up = 1       ;
k20.up = 1       ; k21.up = 1       ;
k30.up = 1       ; k31.up = 1       ; k32.up = 1       ;
k40.up = 1       ; k41.up = 1       ; k42.up = 1       ; k43.up = 1
   ;
k50.up = 1       ; k51.up = 1       ; k52.up = 1       ; k53.up = 1
   ; k54.up = 1       ;
k60.up = 1       ; k61.up = 1       ; k62.up = 1       ; k63.up = 1
   ; k64.up = 1       ; k65.up = 1       ;
> for j from 1 to s do
>   printf(     "%s.up = %g; ", convert(b[j],string), KUP):
>   fprintf(fd, "%s.up = %g; ", convert(b[j],string), KUP):
> end do:
> printf(     "\n"):
> fprintf(fd, "\n"):
b1.up = 1       ; b2.up = 1       ; b3.up = 1       ; b4.up = 1
; b5.up = 1       ; b6.up = 1       ; b7.up = 1       ;

> for j from 2 to s do
>   for k from 2 to (j-1) do
>     printf(     "%s.up = 1; ", convert(alpha[j,k],string)):
>     fprintf(fd, "%s.up = 1; ", convert(alpha[j,k],string)):
```

```
>   end do:
>   printf(     "%s.up = 1;\n", convert(alpha[j,j],string)):
>   fprintf(fd, "%s.up = 1;\n", convert(alpha[j,j],string)):
> end do:
r21.up = 1;
r31.up = 1; r32.up = 1;
r41.up = 1; r42.up = 1; r43.up = 1;
r51.up = 1; r52.up = 1; r53.up = 1; r54.up = 1;
r61.up = 1; r62.up = 1; r63.up = 1; r64.up = 1; r65.up = 1;
r71.up = 1; r72.up = 1; r73.up = 1; r74.up = 1; r75.up = 1; r76.up =
1;
> fprintf(fd, "z.up = %g;\n", ZUP):
> fprintf(fd, "z.lo = %g;\n", ZLO):
> fprintf(fd, "\n"):
>
> fprintf(fd, "# initial guess for minos run (not required if baron is
> run first)\n"):
> for j from 2 to s do
>   printf(     "#"):
>   fprintf(fd, "#"):
>   for k from 1 to (j-2) do
>     printf(     "%s.l = %g; ", convert(A[j,k],string), 1/j):
>     fprintf(fd, "%s.l = %g; ", convert(A[j,k],string), 1/j):
>   end do:
>   printf(     "%s.l = %g;\n", convert(A[j,j-1],string), 1/j):
>   fprintf(fd, "%s.l = %g;\n", convert(A[j,j-1],string), 1/j):
> end do:
#k10.l = .5     ;
#k20.l = .333333; k21.l = .333333;
#k30.l = .25    ; k31.l = .25    ; k32.l = .25    ;
#k40.l = .2     ; k41.l = .2     ; k42.l = .2     ; k43.l = .2     ;
#k50.l = .166667; k51.l = .166667; k52.l = .166667; k53.l = .166667;
k54.l = .166667;
#k60.l = .142857; k61.l = .142857; k62.l = .142857; k63.l = .142857;
k64.l = .142857; k65.l = .142857;
> printf(     "#"):
> fprintf(fd, "#"):
> for j from 1 to s do
>   printf(     "%s.l = %g; ", convert(b[j],string), 1/s):
>   fprintf(fd, "%s.l = %g; ", convert(b[j],string), 1/s):
> end do:
> printf(     "\n"):
> fprintf(fd, "\n"):
#
b1.l = .142857; b2.l = .142857; b3.l = .142857; b4.l = .142857; b5.l =
.142857; b6.l = .142857; b7.l = .142857;

> for j from 2 to s do
> printf(     "#"):
> fprintf(fd, "#"):
>   for k from 2 to (j-1) do
>     printf(     "%s.l = %g; ", convert(alpha[j,k],string), 1/j):
>     fprintf(fd, "%s.l = %g; ", convert(alpha[j,k],string), 1/j):
>   end do:
>   printf(     "%s.l = %g;\n", convert(alpha[j,j],string), 1/j):
>   fprintf(fd, "%s.l = %g;\n", convert(alpha[j,j],string), 1/j):
> end do:
#r21.l = .5     ;
#r31.l = .333333; r32.l = .333333;
#r41.l = .25    ; r42.l = .25    ; r43.l = .25    ;
#r51.l = .2     ; r52.l = .2     ; r53.l = .2     ; r54.l = .2     ;
#r61.l = .166667; r62.l = .166667; r63.l = .166667; r64.l = .166667;
r65.l = .166667;
#r71.l = .142857; r72.l = .142857; r73.l = .142857; r74.l = .142857;
r75.l = .142857; r76.l = .142857;
> printf(     "#z.l = %g;\n", (ZUP+ZLO)/2):
> fprintf(fd, "#z.l = %g;\n", (ZUP+ZLO)/2):
#z.l = 2.5     ;
> fprintf(fd, "\n"):
>
# Equation Header
> fprintf(fd, "equations\n"):
> for j from 1 to s do
>   fprintf(fd, "    ");
>   for k from 1 to (j-1) do
>     printf(     "zc%d%d, ", j, k-1);
```

```
>     fprintf(fd, "zc%d%d, ", j, k-1);
>   end do:
>   printf(     "zc%d%d\n", j, j-1);
>   fprintf(fd, "zc%d%d\n", j, j-1);
> end do:
zc10
zc20, zc21
zc30, zc31, zc32
zc40, zc41, zc42, zc43
zc50, zc51, zc52, zc53, zc54
zc60, zc61, zc62, zc63, zc64, zc65
zc70, zc71, zc72, zc73, zc74, zc75, zc76
> fprintf(fd, "    "):
> for j from 2 to (s-1) do
>   printf(     "ar%d%d, ", j, 0):
>   fprintf(fd, "ar%d%d, ", j, 0):
> end do:
> printf(     "ar%d%d\n", s, 0):
> fprintf(fd, "ar%d%d\n", s, 0):
>
ar20, ar30, ar40, ar50, ar60,
ar70
> for j from 2 to s do
>   fprintf(fd, "    "):
>   for k from 1 to (j-2) do
>     printf(     "bp%d%d, ", j, k-1):
>     fprintf(fd, "bp%d%d, ", j, k-1):
>   end do:
>   printf(     "bp%d%d\n", j, j-2):
>   fprintf(fd, "bp%d%d\n", j, j-2):
> end do:
bp20
bp30, bp31
bp40, bp41, bp42
bp50, bp51, bp52, bp53
bp60, bp61, bp62, bp63, bp64
bp70, bp71, bp72, bp73, bp74, bp75
# The number of order conditions: (pg 147 of Hairer):
> cardTq := [1,1,2,4,9,20,48,115,286,719];

           cardTq := [1, 1, 2, 4, 9, 20, 48, 115, 286, 719]

> for j from 1 to p do
>   fprintf(fd, "    "):
>   for k from 1 to (cardTq[j]-1) do
>     printf(     "t%d%d, ", j, k):
>     fprintf(fd, "t%d%d, ", j, k):
>   end do:
>   printf(     "t%d%d\n", j, cardTq[j]):
>   fprintf(fd, "t%d%d\n", j, cardTq[j]):
> end do:
t11
t21
t31, t32
t41, t42, t43, t44
t51, t52, t53, t54, t55, t56, t57, t58, t59
> fprintf(fd, "    ;\n\n"):
>
>
# SSP Conditions
> fprintf(fd, "# SSP conditions\n"):
> fprintf(fd, "# conditions arising from dummy var z:\n"):
> printf(     "zc%d%d .. %s =G= 0;\n", 1, 0, convert(alpha[1,1] -
> z*beta[1,1],string)):
> fprintf(fd, "zc%d%d .. %s =G= 0;\n", 1, 0, convert(alpha[1,1] -
> z*beta[1,1],string)):
> for j from 2 to s do
>   printf(     "zc%d%d .. %s =G= 0;\n", j, 0,
> convert((1-sum('alpha[j,k]','k'=2..j)) - z*beta[j,1],string)):
>   fprintf(fd, "zc%d%d .. %s =G= 0;\n", j, 0,
> convert((1-sum('alpha[j,k]','k'=2..j)) - z*beta[j,1],string)):
>   for k from 2 to j do
>     printf(     "zc%d%d .. %s =G= 0;\n", j, k-1, convert(alpha[j,k] -
> z*beta[j,k],string)):
>     fprintf(fd, "zc%d%d .. %s =G= 0;\n", j, k-1, convert(alpha[j,k] -
> z*beta[j,k],string)):
```

```
>   end do:
> end do:
zc10 .. 1-z*k10 =G= 0;
zc20 .. 1-r21-z*(k20-r21*k10) =G= 0;
zc21 .. r21-z*k21 =G= 0;
zc30 .. 1-r31-r32-z*(k30-r31*k10-r32*k20) =G= 0;
zc31 .. r31-z*(k31-r32*k21) =G= 0;
zc32 .. r32-z*k32 =G= 0;
zc40 .. 1-r41-r42-r43-z*(k40-r41*k10-r42*k20-r43*k30) =G= 0;
zc41 .. r41-z*(k41-r42*k21-r43*k31) =G= 0;
zc42 .. r42-z*(k42-r43*k32) =G= 0;
zc43 .. r43-z*k43 =G= 0;
zc50 .. 1-r51-r52-r53-r54-z*(k50-r51*k10-r52*k20-r53*k30-r54*k40) =G=
0;
zc51 .. r51-z*(k51-r52*k21-r53*k31-r54*k41) =G= 0;
zc52 .. r52-z*(k52-r53*k32-r54*k42) =G= 0;
zc53 .. r53-z*(k53-r54*k43) =G= 0;
zc54 .. r54-z*k54 =G= 0;
zc60 ..
1-r61-r62-r63-r64-r65-z*(k60-r61*k10-r62*k20-r63*k30-r64*k40-r65*k50)
=G= 0;
zc61 .. r61-z*(k61-r62*k21-r63*k31-r64*k41-r65*k51) =G= 0;
zc62 .. r62-z*(k62-r63*k32-r64*k42-r65*k52) =G= 0;
zc63 .. r63-z*(k63-r64*k43-r65*k53) =G= 0;
zc64 .. r64-z*(k64-r65*k54) =G= 0;
zc65 .. r65-z*k65 =G= 0;
zc70 ..
1-r71-r72-r73-r74-r75-r76-z*(b1-r71*k10-r72*k20-r73*k30-r74*k40-r75*k5
0-r76*k60) =G= 0;
zc71 .. r71-z*(b2-r72*k21-r73*k31-r74*k41-r75*k51-r76*k61) =G= 0;
zc72 .. r72-z*(b3-r73*k32-r74*k42-r75*k52-r76*k62) =G= 0;
zc73 .. r73-z*(b4-r74*k43-r75*k53-r76*k63) =G= 0;
zc74 .. r74-z*(b5-r75*k54-r76*k64) =G= 0;
zc75 .. r75-z*(b6-r76*k65) =G= 0;
zc76 .. r76-z*b7 =G= 0;
>
> fprintf(fd, "# each row of alpha must sum to 1:\n"):
> for j from 2 to s do
>   printf(    "ar%d0 .. %s =G= 0;\n", j, convert(1 -
> sum('alpha[j,k]','k'=2..j), string)):
>   fprintf(fd,"ar%d0 .. %s =G= 0;\n", j, convert(1 -
> sum('alpha[j,k]','k'=2..j), string)):
> end do:
ar20 .. 1-r21 =G= 0;
ar30 .. 1-r31-r32 =G= 0;
ar40 .. 1-r41-r42-r43 =G= 0;
ar50 .. 1-r51-r52-r53-r54 =G= 0;
ar60 .. 1-r61-r62-r63-r64-r65 =G= 0;
ar70 .. 1-r71-r72-r73-r74-r75-r76 =G= 0;
> fprintf(fd, "# each beta must be >= 0:\n"):
> for j from 2 to s do
>   for k from 1 to j-1 do
>     printf(    "bp%d%d .. %s =G= 0;\n", j, k-1, convert(beta[j,k],
> string)):
>     fprintf(fd,"bp%d%d .. %s =G= 0;\n", j, k-1, convert(beta[j,k],
> string)):
>   end do:
> end do:
bp20 .. k20-r21*k10 =G= 0;
bp30 .. k30-r31*k10-r32*k20 =G= 0;
bp31 .. k31-r32*k21 =G= 0;
bp40 .. k40-r41*k10-r42*k20-r43*k30 =G= 0;
bp41 .. k41-r42*k21-r43*k31 =G= 0;
bp42 .. k42-r43*k32 =G= 0;
bp50 .. k50-r51*k10-r52*k20-r53*k30-r54*k40 =G= 0;
bp51 .. k51-r52*k21-r53*k31-r54*k41 =G= 0;
bp52 .. k52-r53*k32-r54*k42 =G= 0;
bp53 .. k53-r54*k43 =G= 0;
bp60 .. k60-r61*k10-r62*k20-r63*k30-r64*k40-r65*k50 =G= 0;
bp61 .. k61-r62*k21-r63*k31-r64*k41-r65*k51 =G= 0;
bp62 .. k62-r63*k32-r64*k42-r65*k52 =G= 0;
bp63 .. k63-r64*k43-r65*k53 =G= 0;
bp64 .. k64-r65*k54 =G= 0;
bp70 .. b1-r71*k10-r72*k20-r73*k30-r74*k40-r75*k50-r76*k60 =G= 0;
bp71 .. b2-r72*k21-r73*k31-r74*k41-r75*k51-r76*k61 =G= 0;
bp72 .. b3-r73*k32-r74*k42-r75*k52-r76*k62 =G= 0;
```

```
bp73 .. b4-r74*k43-r75*k53-r76*k63 =G= 0;
bp74 .. b5-r75*k54-r76*k64 =G= 0;
bp75 .. b6-r76*k65 =G= 0;
> fprintf(fd, "\n"):
#
>
# Order Conditions
> fprintf(fd, "# Order Conditions\n"):
>
# OC1
> if (p >= 1) then
>   # this is called tau but easier for the scripts if we call it t11
>   sum1 := sum('b[j]', 'j'=1..s):
>   printf(     "t11 .. %s =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t11 .. %s =E= 1;\n", convert(sum1,string)):
> fi:
t11 .. b1+b2+b3+b4+b5+b6+b7 =E= 1;
>
# OC2
> if (p >= 2) then
>   sum1 := sum('b[j]*c[j]', 'j'=1..s):
>   printf(     "t21 .. 2*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t21 .. 2*( %s ) =E= 1;\n", convert(sum1,string)):
> fi:
t21 .. 2*(
b2*k10+b3*(k20+k21)+b4*(k30+k31+k32)+b5*(k40+k41+k42+k43)+b6*(k50+k51+
k52+k53+k54)+b7*(k60+k61+k62+k63+k64+k65) ) =E= 1;
>
# OC3
> if (p >= 3) then
>   sum1 := sum('b[j]*c[j]^2', 'j'=1..s):
>   printf(     "t31 .. 3*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t31 .. 3*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       sum1 := sum1 + b[j]*A[j,k]*c[k];
>     end:
>   end:
>   printf(     "t32 .. 6*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t32 .. 6*( %s ) =E= 1;\n", convert(sum1,string)):
> fi:
t31 .. 3*(
b2*k10^2+b3*(k20+k21)^2+b4*(k30+k31+k32)^2+b5*(k40+k41+k42+k43)^2+b6*(
k50+k51+k52+k53+k54)^2+b7*(k60+k61+k62+k63+k64+k65)^2 ) =E= 1;
t32 .. 6*(
b3*k21*k10+b4*k31*k10+b4*k32*(k20+k21)+b5*k41*k10+b5*k42*(k20+k21)+b5*
k43*(k30+k31+k32)+b6*k51*k10+b6*k52*(k20+k21)+b6*k53*(k30+k31+k32)+b6*
k54*(k40+k41+k42+k43)+b7*k61*k10+b7*k62*(k20+k21)+b7*k63*(k30+k31+k32)
+b7*k64*(k40+k41+k42+k43)+b7*k65*(k50+k51+k52+k53+k54) ) =E= 1;
# OC4
> if (p >= 4) then
>   sum1 := sum('b[j]*c[j]^3', 'j'=1..s):
>   printf(     "t41 .. 4*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t41 .. 4*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       sum1 := sum1 + b[j]*A[j,k]*c[k]*c[j];
>     end:
>   end:
>   printf(     "t42 .. 8*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t42 .. 8*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       sum1 := sum1 + b[j]*A[j,k]*c[k]^2;
>     end:
>   end:
>   printf(     "t43 .. 12*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t43 .. 12*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
```

```
>   for j from 1 to s do
>     for k from 1 to s do
>       for l from 1 to s do
>         sum1 := sum1 + b[j]*A[j,k]*A[k,l]*c[l];
>       end:
>     end:
>   end:
>   printf(     "t44 .. 24*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t44 .. 24*( %s ) =E= 1;\n", convert(sum1,string)):
> fi:
t41 .. 4*(
b2*k10^3+b3*(k20+k21)^3+b4*(k30+k31+k32)^3+b5*(k40+k41+k42+k43)^3+b6*(
k50+k51+k52+k53+k54)^3+b7*(k60+k61+k62+k63+k64+k65)^3 ) =E= 1;
t42 .. 8*(
b3*k21*k10*(k20+k21)+b4*k31*k10*(k30+k31+k32)+b4*k32*(k20+k21)*(k30+k3
1+k32)+b5*k41*k10*(k40+k41+k42+k43)+b5*k42*(k20+k21)*(k40+k41+k42+k43)
+b5*k43*(k30+k31+k32)*(k40+k41+k42+k43)+b6*k51*k10*(k50+k51+k52+k53+k5
4)+b6*k52*(k20+k21)*(k50+k51+k52+k53+k54)+b6*k53*(k30+k31+k32)*(k50+k5
1+k52+k53+k54)+b6*k54*(k40+k41+k42+k43)*(k50+k51+k52+k53+k54)+b7*k61*k
10*(k60+k61+k62+k63+k64+k65)+b7*k62*(k20+k21)*(k60+k61+k62+k63+k64+k65
)+b7*k63*(k30+k31+k32)*(k60+k61+k62+k63+k64+k65)+b7*k64*(k40+k41+k42+k
43)*(k60+k61+k62+k63+k64+k65)+b7*k65*(k50+k51+k52+k53+k54)*(k60+k61+k6
2+k63+k64+k65) ) =E= 1;
t43 .. 12*(
b3*k21*k10^2+b4*k31*k10^2+b4*k32*(k20+k21)^2+b5*k41*k10^2+b5*k42*(k20+
k21)^2+b5*k43*(k30+k31+k32)^2+b6*k51*k10^2+b6*k52*(k20+k21)^2+b6*k53*(
k30+k31+k32)^2+b6*k54*(k40+k41+k42+k43)^2+b7*k61*k10^2+b7*k62*(k20+k21
)^2+b7*k63*(k30+k31+k32)^2+b7*k64*(k40+k41+k42+k43)^2+b7*k65*(k50+k51+
k52+k53+k54)^2 ) =E= 1;
t44 .. 24*(
b4*k32*k21*k10+b5*k42*k21*k10+b5*k43*k31*k10+b5*k43*k32*(k20+k21)+b6*k
52*k21*k10+b6*k53*k31*k10+b6*k53*k32*(k20+k21)+b6*k54*k41*k10+b6*k54*k
42*(k20+k21)+b6*k54*k43*(k30+k31+k32)+b7*k62*k21*k10+b7*k63*k31*k10+b7
*k63*k32*(k20+k21)+b7*k64*k41*k10+b7*k64*k42*(k20+k21)+b7*k64*k43*(k30
+k31+k32)+b7*k65*k51*k10+b7*k65*k52*(k20+k21)+b7*k65*k53*(k30+k31+k32)
+b7*k65*k54*(k40+k41+k42+k43) ) =E= 1;
>
# OC5
> if (p >= 5) then
>   sum1 := 0:
>   for j from 1 to s do
>     sum1 := sum1 + b[j]*c[j]^4;
>   end:
>   printf(     "t51 .. 5*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t51 .. 5*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       sum1 := sum1 + b[j]*A[j,k]*c[k]*c[j]^2;
>     end:
>   end:
>   printf(     "t52 .. 10*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t52 .. 10*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       sum1 := sum1 + b[j]*A[j,k]*c[k]^2*c[j];
>     end:
>   end:
>   printf(     "t53 .. 15*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t53 .. 15*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       for l from 1 to s do
>         sum1 := sum1 + b[j]*A[j,k]*A[k,l]*c[l]*c[j];
>       end:
>     end:
>   end:
>   printf(     "t54 .. 30*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t54 .. 30*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
```

```
>   for j from 1 to s do
>     for k from 1 to s do
>       for m from 1 to s do
>         sum1 := sum1 + b[j]*A[j,k]*c[k]*A[j,m]*c[m];
>       end:
>     end:
>   end:
>   printf(     "t55 .. 20*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t55 .. 20*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       sum1 := sum1 + b[j]*A[j,k]*c[k]^3;
>     end:
>   end:
>   printf(     "t56 .. 20*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t56 .. 20*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       for l from 1 to s do
>         sum1 := sum1 + b[j]*A[j,k]*A[k,l]*c[l]*c[k];
>       end:
>     end:
>   end:
>   printf(     "t57 .. 40*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t57 .. 40*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       for l from 1 to s do
>         sum1 := sum1 + b[j]*A[j,k]*A[k,l]*c[l]^2;
>       end:
>     end:
>   end:
>   printf(     "t58 .. 60*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t58 .. 60*( %s ) =E= 1;\n", convert(sum1,string)):
>
>   sum1 := 0:
>   for j from 1 to s do
>     for k from 1 to s do
>       for l from 1 to s do
>         for m from 1 to s do
>           sum1 := sum1 + b[j]*A[j,k]*A[k,l]*A[l,m]*c[m];
>         end:
>       end:
>     end:
>   end:
>   printf(     "t59 .. 120*( %s ) =E= 1;\n", convert(sum1,string)):
>   fprintf(fd, "t59 .. 120*( %s ) =E= 1;\n", convert(sum1,string)):
> fi:
t51 .. 5*(
b2*k10^4+b3*(k20+k21)^4+b4*(k30+k31+k32)^4+b5*(k40+k41+k42+k43)^4+b6*(
k50+k51+k52+k53+k54)^4+b7*(k60+k61+k62+k63+k64+k65)^4 ) =E= 1;
t52 .. 10*(
b3*k21*k10*(k20+k21)^2+b4*k31*k10*(k30+k31+k32)^2+b4*k32*(k20+k21)*(k3
0+k31+k32)^2+b5*k41*k10*(k40+k41+k42+k43)^2+b5*k42*(k20+k21)*(k40+k41+
k42+k43)^2+b5*k43*(k30+k31+k32)*(k40+k41+k42+k43)^2+b6*k51*k10*(k50+k5
1+k52+k53+k54)^2+b6*k52*(k20+k21)*(k50+k51+k52+k53+k54)^2+b6*k53*(k30+
k31+k32)*(k50+k51+k52+k53+k54)^2+b6*k54*(k40+k41+k42+k43)*(k50+k51+k52
+k53+k54)^2+b7*k61*k10*(k60+k61+k62+k63+k64+k65)^2+b7*k62*(k20+k21)*(k
60+k61+k62+k63+k64+k65)^2+b7*k63*(k30+k31+k32)*(k60+k61+k62+k63+k64+k6
5)^2+b7*k64*(k40+k41+k42+k43)*(k60+k61+k62+k63+k64+k65)^2+b7*k65*(k50+
k51+k52+k53+k54)*(k60+k61+k62+k63+k64+k65)^2 ) =E= 1;
t53 .. 15*(
b3*k21*k10^2*(k20+k21)+b4*k31*k10^2*(k30+k31+k32)+b4*k32*(k20+k21)^2*(
k30+k31+k32)+b5*k41*k10^2*(k40+k41+k42+k43)+b5*k42*(k20+k21)^2*(k40+k4
1+k42+k43)+b5*k43*(k30+k31+k32)^2*(k40+k41+k42+k43)+b6*k51*k10^2*(k50+
k51+k52+k53+k54)+b6*k52*(k20+k21)^2*(k50+k51+k52+k53+k54)+b6*k53*(k30+
k31+k32)^2*(k50+k51+k52+k53+k54)+b6*k54*(k40+k41+k42+k43)^2*(k50+k51+k
52+k53+k54)+b7*k61*k10^2*(k60+k61+k62+k63+k64+k65)+b7*k62*(k20+k21)^2*
(k60+k61+k62+k63+k64+k65)+b7*k63*(k30+k31+k32)^2*(k60+k61+k62+k63+k64+
k65)+b7*k64*(k40+k41+k42+k43)^2*(k60+k61+k62+k63+k64+k65)+b7*k65*(k50+
```

```
k51+k52+k53+k54)^2*(k60+k61+k62+k63+k64+k65) ) =E= 1;
t54 .. 30*(
b4*k32*k21*k10*(k30+k31+k32)+b5*k42*k21*k10*(k40+k41+k42+k43)+b5*k43*k
31*k10*(k40+k41+k42+k43)+b5*k43*k32*(k20+k21)*(k40+k41+k42+k43)+b6*k52
*k21*k10*(k50+k51+k52+k53+k54)+b6*k53*k31*k10*(k50+k51+k52+k53+k54)+b6
*k53*k32*(k20+k21)*(k50+k51+k52+k53+k54)+b6*k54*k41*k10*(k50+k51+k52+k
53+k54)+b6*k54*k42*(k20+k21)*(k50+k51+k52+k53+k54)+b6*k54*k43*(k30+k31
+k32)*(k50+k51+k52+k53+k54)+b7*k62*k21*k10*(k60+k61+k62+k63+k64+k65)+b
7*k63*k31*k10*(k60+k61+k62+k63+k64+k65)+b7*k63*k32*(k20+k21)*(k60+k61+
k62+k63+k64+k65)+b7*k64*k41*k10*(k60+k61+k62+k63+k64+k65)+b7*k64*k42*(
k20+k21)*(k60+k61+k62+k63+k64+k65)+b7*k64*k43*(k30+k31+k32)*(k60+k61+k
62+k63+k64+k65)+b7*k65*k51*k10*(k60+k61+k62+k63+k64+k65)+b7*k65*k52*(k
20+k21)*(k60+k61+k62+k63+k64+k65)+b7*k65*k53*(k30+k31+k32)*(k60+k61+k6
2+k63+k64+k65)+b7*k65*k54*(k40+k41+k42+k43)*(k60+k61+k62+k63+k64+k65)
) =E= 1;
t55 .. 20*(
b5*k41^2*k10^2+2*b5*k41*k10*k42*(k20+k21)+2*b5*k41*k10*k43*(k30+k31+k3
2)+b4*k31^2*k10^2+2*b4*k31*k10*k32*(k20+k21)+b4*k32^2*(k20+k21)^2+b3*k
21^2*k10^2+b5*k43^2*(k30+k31+k32)^2+b7*k62^2*(k20+k21)^2+2*b7*k62*(k20
+k21)*k63*(k30+k31+k32)+2*b7*k62*(k20+k21)*k64*(k40+k41+k42+k43)+2*b7*
k62*(k20+k21)*k65*(k50+k51+k52+k53+k54)+b7*k63^2*(k30+k31+k32)^2+2*b7*
k61*k10*k65*(k50+k51+k52+k53+k54)+b6*k54^2*(k40+k41+k42+k43)^2+b6*k53^
2*(k30+k31+k32)^2+2*b6*k53*(k30+k31+k32)*k54*(k40+k41+k42+k43)+b7*k61^
2*k10^2+2*b7*k61*k10*k62*(k20+k21)+2*b7*k61*k10*k63*(k30+k31+k32)+2*b7
*k61*k10*k64*(k40+k41+k42+k43)+b6*k51^2*k10^2+2*b6*k51*k10*k52*(k20+k2
1)+2*b6*k51*k10*k53*(k30+k31+k32)+2*b6*k51*k10*k54*(k40+k41+k42+k43)+b
6*k52^2*(k20+k21)^2+2*b6*k52*(k20+k21)*k53*(k30+k31+k32)+2*b6*k52*(k20
+k21)*k54*(k40+k41+k42+k43)+b5*k42^2*(k20+k21)^2+2*b5*k42*(k20+k21)*k4
3*(k30+k31+k32)+b7*k65^2*(k50+k51+k52+k53+k54)^2+2*b7*k64*(k40+k41+k42
+k43)*k65*(k50+k51+k52+k53+k54)+2*b7*k63*(k30+k31+k32)*k64*(k40+k41+k4
2+k43)+2*b7*k63*(k30+k31+k32)*k65*(k50+k51+k52+k53+k54)+b7*k64^2*(k40+
k41+k42+k43)^2 ) =E= 1;
t56 .. 20*(
b3*k21*k10^3+b4*k31*k10^3+b4*k32*(k20+k21)^3+b5*k41*k10^3+b5*k42*(k20+
k21)^3+b5*k43*(k30+k31+k32)^3+b6*k51*k10^3+b6*k52*(k20+k21)^3+b6*k53*(
k30+k31+k32)^3+b6*k54*(k40+k41+k42+k43)^3+b7*k61*k10^3+b7*k62*(k20+k21
)^3+b7*k63*(k30+k31+k32)^3+b7*k64*(k40+k41+k42+k43)^3+b7*k65*(k50+k51+
k52+k53+k54)^3 ) =E= 1;
t57 .. 40*(
b4*k32*k21*k10*(k20+k21)+b5*k42*k21*k10*(k20+k21)+b5*k43*k31*k10*(k30+
k31+k32)+b5*k43*k32*(k20+k21)*(k30+k31+k32)+b6*k52*k21*k10*(k20+k21)+b
6*k53*k31*k10*(k30+k31+k32)+b6*k53*k32*(k20+k21)*(k30+k31+k32)+b6*k54*
k41*k10*(k40+k41+k42+k43)+b6*k54*k42*(k20+k21)*(k40+k41+k42+k43)+b6*k5
4*k43*(k30+k31+k32)*(k40+k41+k42+k43)+b7*k62*k21*k10*(k20+k21)+b7*k63*
k31*k10*(k30+k31+k32)+b7*k63*k32*(k20+k21)*(k30+k31+k32)+b7*k64*k41*k1
0*(k40+k41+k42+k43)+b7*k64*k42*(k20+k21)*(k40+k41+k42+k43)+b7*k64*k43*
(k30+k31+k32)*(k40+k41+k42+k43)+b7*k65*k51*k10*(k50+k51+k52+k53+k54)+b
7*k65*k52*(k20+k21)*(k50+k51+k52+k53+k54)+b7*k65*k53*(k30+k31+k32)*(k5
0+k51+k52+k53+k54)+b7*k65*k54*(k40+k41+k42+k43)*(k50+k51+k52+k53+k54)
) =E= 1;
t58 .. 60*(
b4*k32*k21*k10^2+b5*k42*k21*k10^2+b5*k43*k31*k10^2+b5*k43*k32*(k20+k21
)^2+b6*k52*k21*k10^2+b6*k53*k31*k10^2+b6*k53*k32*(k20+k21)^2+b6*k54*k4
1*k10^2+b6*k54*k42*(k20+k21)^2+b6*k54*k43*(k30+k31+k32)^2+b7*k62*k21*k
10^2+b7*k63*k31*k10^2+b7*k63*k32*(k20+k21)^2+b7*k64*k41*k10^2+b7*k64*k
42*(k20+k21)^2+b7*k64*k43*(k30+k31+k32)^2+b7*k65*k51*k10^2+b7*k65*k52*
(k20+k21)^2+b7*k65*k53*(k30+k31+k32)^2+b7*k65*k54*(k40+k41+k42+k43)^2
) =E= 1;
t59 .. 120*(
b5*k43*k32*k21*k10+b6*k53*k32*k21*k10+b6*k54*k42*k21*k10+b6*k54*k43*k3
1*k10+b6*k54*k43*k32*(k20+k21)+b7*k63*k32*k21*k10+b7*k64*k42*k21*k10+b
7*k64*k43*k31*k10+b7*k64*k43*k32*(k20+k21)+b7*k65*k52*k21*k10+b7*k65*k
53*k31*k10+b7*k65*k53*k32*(k20+k21)+b7*k65*k54*k41*k10+b7*k65*k54*k42*
(k20+k21)+b7*k65*k54*k43*(k30+k31+k32) ) =E= 1;
> fprintf(fd, "\n"):
>
# Model setup, BARON call
> fprintf(fd, "# only affects the display command and cannot be > 8\n"):
> fprintf(fd, "option decimals = 8;\n\n"):
>
> fprintf(fd, "# BARON run:\n"):
> fprintf(fd, "model m /all/;\n"):
> fprintf(fd, "option nlp = baron;\n"):
> fprintf(fd, "m.optfile = 1;\n"):
> fprintf(fd, "m.workspace = %d;\n", WORKSPACE):
> fprintf(fd, "solve m maximizing z using nlp;\n\n"):
```

```
>
>
> fprintf(fd, "# MINOS run:\n"):
> fprintf(fd, "model m2 /all/;\n"):
> fprintf(fd, "option nlp = minos;\n"):
> fprintf(fd, "option sysout = on;\n"):
> fprintf(fd, "m2.optfile = 1;\n"):
> fprintf(fd, "solve m2 maximizing z using nlp;\n\n"):
>
> fprintf(fd, "variables\n"):
> fprintf(fd, "    "):
> for j from 2 to (s-1) do
>   printf(     "%s, ", convert(alpha[j,1],string)):
>   fprintf(fd, "%s, ", convert(alpha[j,1],string)):
> end do:
> printf(     "%s\n", convert(alpha[j,1],string)):
> fprintf(fd, "%s\n", convert(alpha[j,1],string)):
r20, r30, r40, r50, r60,
r70
> for j from 1 to s do
>   fprintf(fd, "    "):
>   for k from 1 to (j-1) do
>     printf(     "beta%d%d, ", j, k-1):
>     fprintf(fd, "beta%d%d, ", j, k-1):
>   end do:
>   printf(     "beta%d%d\n", j, j-1):
>   fprintf(fd, "beta%d%d\n", j, j-1):
> end do:
beta10
beta20, beta21
beta30, beta31, beta32
beta40, beta41, beta42, beta43
beta50, beta51, beta52, beta53, beta54
beta60, beta61, beta62, beta63, beta64, beta65
beta70, beta71, beta72, beta73, beta74, beta75, beta76
> fprintf(fd, "    ;\n"):
> for j from 2 to s do
>   printf(     "%s.l = 1 - ", convert(alpha[j,1], string)):
>   fprintf(fd, "%s.l = 1 - ", convert(alpha[j,1], string)):
>   for k from 2 to (j-1) do
>     printf(     "%s.l - ", convert(alpha[j,k], string)):
>     fprintf(fd, "%s.l - ", convert(alpha[j,k], string)):
>   end do:
>   printf(     "%s.l;\n", convert(alpha[j,j], string)):
>   fprintf(fd, "%s.l;\n", convert(alpha[j,j], string)):
>   #printf(     "%s.l = %s;\n", convert(alpha[j,1], string),
> convert(1-sum('alpha[j,k]', 'k'=2..j), string)):
>   #fprintf(fd, "%s.l = %s;\n", convert(alpha[j,1], string),
> convert(1-sum('alpha[j,k]', 'k'=2..j), string)):
> end do:
>
r20.l = 1 - r21.l;
r30.l = 1 - r31.l - r32.l;
r40.l = 1 - r41.l - r42.l - r43.l;
r50.l = 1 - r51.l - r52.l - r53.l - r54.l;
r60.l = 1 - r61.l - r62.l - r63.l - r64.l - r65.l;
r70.l = 1 - r71.l - r72.l - r73.l - r74.l - r75.l - r76.l;
> for i from 1 to s do
>   for k from 1 to i do
>     if (i = s) then
>       printf(     "beta%d%d.l = %s.l", i, k-1, b[k]):
>       fprintf(fd, "beta%d%d.l = %s.l", i, k-1, b[k]):
>     else
>       printf(     "beta%d%d.l = %s.l", i, k-1, A[i+1,k]):
>       fprintf(fd, "beta%d%d.l = %s.l", i, k-1, A[i+1,k]):
>     fi:
>     for j from k to i-1 do
>       printf(     " - %s.l*%s.l", alpha[i,j+1], A[j+1,k]):
>       fprintf(fd, " - %s.l*%s.l", alpha[i,j+1], A[j+1,k]):
>     end do:
>     printf(     ";\n"):
>     fprintf(fd, ";\n"):
>   end do:
> end do:
beta10.l = k10.l;
beta20.l = k20.l - r21.l*k10.l;
```

```
beta21.l = k21.l;
beta30.l = k30.l - r31.l*k10.l - r32.l*k20.l;
beta31.l = k31.l - r32.l*k21.l;
beta32.l = k32.l;
beta40.l = k40.l - r41.l*k10.l - r42.l*k20.l - r43.l*k30.l;
beta41.l = k41.l - r42.l*k21.l - r43.l*k31.l;
beta42.l = k42.l - r43.l*k32.l;
beta43.l = k43.l;
beta50.l = k50.l - r51.l*k10.l - r52.l*k20.l - r53.l*k30.l -
r54.l*k40.l;
beta51.l = k51.l - r52.l*k21.l - r53.l*k31.l - r54.l*k41.l;
beta52.l = k52.l - r53.l*k32.l - r54.l*k42.l;
beta53.l = k53.l - r54.l*k43.l;
beta54.l = k54.l;
beta60.l = k60.l - r61.l*k10.l - r62.l*k20.l - r63.l*k30.l -
r64.l*k40.l - r65.l*k50.l;
beta61.l = k61.l - r62.l*k21.l - r63.l*k31.l - r64.l*k41.l -
r65.l*k51.l;
beta62.l = k62.l - r63.l*k32.l - r64.l*k42.l - r65.l*k52.l;
beta63.l = k63.l - r64.l*k43.l - r65.l*k53.l;
beta64.l = k64.l - r65.l*k54.l;
beta65.l = k65.l;
beta70.l = b1.l - r71.l*k10.l - r72.l*k20.l - r73.l*k30.l -
r74.l*k40.l - r75.l*k50.l - r76.l*k60.l;
beta71.l = b2.l - r72.l*k21.l - r73.l*k31.l - r74.l*k41.l -
r75.l*k51.l - r76.l*k61.l;
beta72.l = b3.l - r73.l*k32.l - r74.l*k42.l - r75.l*k52.l -
r76.l*k62.l;
beta73.l = b4.l - r74.l*k43.l - r75.l*k53.l - r76.l*k63.l;
beta74.l = b5.l - r75.l*k54.l - r76.l*k64.l;
beta75.l = b6.l - r76.l*k65.l;
beta76.l = b7.l;
> ## unfortunately, this will not work! (the expression won't have .l's)
> #for j from 1 to s do
> #  for k from 1 to j do
> #    printf(     "beta%d%d.l = %s;\n", j, k-1,
> convert(beta[j,k],string)):
> #    fprintf(fd, "beta%d%d.l = %s;\n", j, k-1,
> convert(beta[j,k],string)):
> #  end do:
> #end do:
>
>
> fprintf(fd, "file out / %s /;\n", COEF_FILENAME):
> fprintf(fd, "put out;\n"):
> fprintf(fd, "out.nd=15;\nout.nr=0;\nout.nz=0;\nout.nw=22;\n\n"):
> fprintf(fd, "put z.lo, z.l, z.up/;\n"):
>
>
>
> fprintf(fd, "put '# A matrix:'//;\n"):
> for j from 2 to s do
>   printf(     "put "):
>   fprintf(fd, "put "):
>   for k from 1 to (j-2) do
>     printf(     "%s.l, ", convert(A[j,k],string)):
>     fprintf(fd, "%s.l, ", convert(A[j,k],string)):
>   end do:
>   printf(     "%s.l/;\n", convert(A[j,j-1],string)):
>   fprintf(fd, "%s.l/;\n", convert(A[j,j-1],string)):
> end do:
put k10.l/;
put k20.l, k21.l/;
put k30.l, k31.l, k32.l/;
put k40.l, k41.l, k42.l, k43.l/;
put k50.l, k51.l, k52.l, k53.l, k54.l/;
put k60.l, k61.l, k62.l, k63.l, k64.l, k65.l/;
> fprintf(fd, "put '# b vector:'/;\n"):
> printf(     "put "):
> fprintf(fd, "put "):
> for j from 1 to (s-1) do
>   printf(     "%s.l, ", convert(b[j],string)):
>   fprintf(fd, "%s.l, ", convert(b[j],string)):
> end do:
> printf(     "%s.l/;\n", convert(b[s],string)):
> fprintf(fd, "%s.l/;\n", convert(b[s],string)):
```

```
>
put
b1.l, b2.l, b3.l, b4.l, b5.l, b6.l,
b7.l/;
> fprintf(fd, "put /'# alpha matrix:'/;\n"):
> printf(     "put 1/;\n"):
> fprintf(fd, "put 1/;\n"):
> for j from 2 to s do
>   printf(     "put "):
>   fprintf(fd, "put "):
>   for k from 1 to (j-1) do
>     printf(     "%s.l, ", convert(alpha[j,k],string)):
>     fprintf(fd, "%s.l, ", convert(alpha[j,k],string)):
>   end do:
>   printf(     "%s.l/;\n", convert(alpha[j,j],string)):
>   fprintf(fd, "%s.l/;\n", convert(alpha[j,j],string)):
> end do:
put 1/;
put r20.l, r21.l/;
put r30.l, r31.l, r32.l/;
put r40.l, r41.l, r42.l, r43.l/;
put r50.l, r51.l, r52.l, r53.l, r54.l/;
put r60.l, r61.l, r62.l, r63.l, r64.l, r65.l/;
put r70.l, r71.l, r72.l, r73.l, r74.l, r75.l, r76.l/;
> fprintf(fd, "put '# beta matrix:'/;\n"):
> for j from 1 to s do
>   printf(     "put "):
>   fprintf(fd, "put "):
>   for k from 1 to (j-1) do
>     printf(     "beta%d%d.l, ", j, k-1):
>     fprintf(fd, "beta%d%d.l, ", j, k-1):
>   end do:
>   printf(     "beta%d%d.l/;\n", j, j-1):
>   fprintf(fd, "beta%d%d.l/;\n", j, j-1):
> end do:
put beta10.l/;
put beta20.l, beta21.l/;
put beta30.l, beta31.l, beta32.l/;
put beta40.l, beta41.l, beta42.l, beta43.l/;
put beta50.l, beta51.l, beta52.l, beta53.l, beta54.l/;
put beta60.l, beta61.l, beta62.l, beta63.l, beta64.l, beta65.l/;
put beta70.l, beta71.l, beta72.l, beta73.l, beta74.l, beta75.l,
beta76.l/;
> fprintf(fd, "\n"):
>
>
> fprintf(fd, "putclose out;\n"):
>
> fclose(fd);
>
```

Appendix C

Additional Source Code

The author's website at http://www.math.sfu.ca/∼cbm/ contains additional source code, errata lists and machine readable code for the various Butcher tableaux and α–β notation matrices contained in this thesis.

Bibliography

[BF01] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole, seventh edition, 2001.

[GAM01] Gams–the solver manuals. GAMS Development Corporation, Washington, 2001.

[GL85] Jonathan B. Goodman and Randall J. LeVeque. On the accuracy of stable schemes for 2D scalar conservation laws. *Math. Comp.*, 45(171):15–21, 1985.

[GS98] Sigal Gottlieb and Chi-Wang Shu. Total variation diminishing Runge-Kutta schemes. *Math. Comp.*, 67(221):73–85, 1998.

[GST01] Sigal Gottlieb, Chi-Wang Shu, and Eitan Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM Rev.*, 43(1):89–112 (electronic), 2001.

[Hea97] Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill, New York, 1997.

[HEOC87] Ami Harten, Björn Engquist, Stanley Osher, and Sukumar R. Chakravarthy. Uniformly high-order accurate essentially nonoscillatory schemes. III. *J. Comput. Phys.*, 71(2):231–303, 1987.

[HNW93] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I: Nonstiff problems*, volume 8 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 1993.

[HW91] E. Hairer and G. Wanner. *Solving ordinary differential equations II: Stiff and differential-algebraic problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1991.

[Jia95] Guang-Shan Jiang. *Algorithm Analysis and Efficient Computation of Conservation Laws*. PhD thesis, Brown University, May 1995.

[Lan98] Culbert B. Laney. *Computational gasdynamics*. Cambridge University Press, Cambridge, 1998.

[OF03] Stanley Osher and Ronald Fedkiw. *Level set methods and dynamic implicit surfaces*, volume 153 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2003.

[QS02] Jianxian Qiu and Chi-Wang Shu. On the construction, comparison, and local characteristic decomposition for high-order central WENO schemes. *J. Comput. Phys.*, 183(1):187–209, 2002.

[RS02a] Steven J. Ruuth and Raymond J. Spiteri. High-order strong-stability-preserving Runge-Kutta methods with downwind-biased spatial discretizations. Unpublished manuscript (revised), 2002.

[RS02b] Steven J. Ruuth and Raymond J. Spiteri. Two barriers on strong-stability-preserving time discretization methods. In *Proceedings of the Fifth International Conference on Spectral and High Order Methods (ICOSAHOM-01) (Uppsala)*, volume 17, pages 211–220, 2002.

[Ruu03] Steven J. Ruuth. Global optimization of high-order strong-stability-preserving Runge-Kutta methods. submitted for publication, July 2003.

[Set99] J. A. Sethian. *Level set methods and fast marching methods: Evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*, volume 3 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, second edition, 1999.

[Shu88] Chi-Wang Shu. Total-variation-diminishing time discretizations. *SIAM J. Sci. Statist. Comput.*, 9(6):1073–1084, 1988.

[Shu97] Chi-Wang Shu. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. Technical Report NASA CR-97-206253 ICASE Report No. 97-65, Institute for Computer Applications in Science and Engineering, November 1997.

[SO88] Chi-Wang Shu and Stanley Osher. Efficient implementation of essentially nonoscillatory shock-capturing schemes. *J. Comput. Phys.*, 77(2):439–471, 1988.

[SR02] Raymond J. Spiteri and Steven J. Ruuth. A new class of optimal high-order strong-stability-preserving time discretization methods. *SIAM J. Numer. Anal.*, 40(2):469–491, 2002.

[Str92] Walter A. Strauss. *Partial differential equations: An introduction*. John Wiley & Sons Inc., New York, 1992.

[SvL85] P. Sonneveld and B. van Leer. A minimax problem along the imaginary axis. *Nieuw Arch. Wisk. (4)*, 3(1):19–22, 1985.

[vdM90] Roeland P. van der Marel. Stability radius of polynomials occurring in the numerical solution of initial value problems. *BIT*, 30(3):516–528, 1990.

[Ver82] J. H. Verner. Deriving explicit methods or how to solve hard jigsaw puzzles. A contribution to the CMS seminar on numerical analysis (unpublished manuscript), July 1982.

[Ver03] J. H. Verner. A contrast of conventional and two-step Runge-Kutta methods. Centre for Scientific Computing and Computing Science seminar, May 2003.

Index