

INTEGRATING-FACTOR-BASED 2-ADDITIVE
RUNGE-KUTTA METHODS FOR
ADVECTION-REACTION-DIFFUSION EQUATIONS

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Andrew Kroshko

©Andrew Kroshko, May 2011. All rights reserved.

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

ABSTRACT

There are three distinct processes that are predominant in models of flowing media with interacting components: advection, reaction, and diffusion. Collectively, these processes are typically modelled with partial differential equations (PDEs) known as advection-reaction-diffusion (ARD) equations.

To solve most PDEs in practice, approximation methods known as numerical methods are used. The method of lines is used to approximate PDEs with systems of ordinary differential equations (ODEs) by a process known as semi-discretization. ODEs are more readily analysed and benefit from well-developed numerical methods and software. Each term of an ODE that corresponds to one of the processes of an ARD equation benefits from particular mathematical properties in a numerical method. These properties are often mutually exclusive for many basic numerical methods.

A limitation to the widespread use of more complex numerical methods is that the development of the appropriate software to provide comparisons to existing numerical methods is not straightforward. Scientific and numerical software is often inflexible, motivating the development of a class of software known as problem-solving environments (PSEs). Many existing PSEs such as MATLAB have solvers for ODEs and PDEs but lack specific features, beyond a scripting language, to readily experiment with novel or existing solution methods. The PSE developed during the course of this thesis solves ODEs known as initial-value problems, where only the initial state is fully known. The PSE is used to assess the performance of new numerical methods for ODEs that integrate each term of a semi-discretized ARD equation. This PSE is part of the PSE `pythODE` that uses object-oriented and software-engineering techniques to allow implementations of many existing and novel solution methods for ODEs with minimal effort spent on code modification and integration.

The new numerical methods use a commutator-free exponential Runge–Kutta (CFERK) method to solve the advection term of an ARD equation. A matrix exponential is used as the exponential function, but CFERK methods can use other numerical methods that model the flowing medium. The reaction term is solved separately using an explicit Runge–Kutta method because solving it along with the diffusion term can result in stepsize restrictions and hence inefficiency. The diffusion term is solved using a Runge–Kutta–Chebyshev method that takes advantage of the spatially symmetric nature of the diffusion process to avoid stepsize restrictions from a property known as stiffness. The resulting methods, known as integrating-factor-based 2-additive-Runge–Kutta methods, are shown to be able to find higher-accuracy solutions in less computational time than competing methods for certain challenging semi-discretized ARD equations. This demonstrates the practical viability both of using CFERK methods for advection and a 3-splitting in general.

ACKNOWLEDGEMENTS

I offer special thanks to my supervisor Dr. Raymond J. Spiteri for his vision and for giving me the opportunity to pursue this project. His instruction, guidance, patience, and financial support made this thesis possible. I wish to thank the faculty in the Department of Computer Science and the Department of Mathematics and Statistics for providing me with further instruction and guidance. I am particularly grateful to the members of the Numerical Simulation Laboratory for their support, helpful discussions, and friendship during the course of this thesis.

I wish to thank my parents for cultivating my interest in research and demonstrating to me all that could be accomplished with hard work. My father, Paul Kroshko, for introducing me to computer programming and giving me my first taste of numerical analysis at a young age. My mother, Joan Kroshko, for cultivating my lifelong passion for mathematics and the natural sciences. Without their endless support, patience, and encouragement I could never have taken this path. As well, I wish to thank my sister, Jeanette Kroshko, for her indispensable support as I pursued my Masters degree. Lastly, I wish to thank my brother, Thomas Kroshko, for his day to day support and friendship while finishing this thesis.

I wish to thank the friends and other family members who have given me their unwavering support over the years. Finally, I wish to thank the members of running community in Saskatoon for giving me an additional outlet for my energies while writing this thesis.

To my parents.

CONTENTS

Permission to Use	i
Abstract	ii
Acknowledgements	iii
Contents	v
List of Tables	vii
List of Figures	viii
List of Abbreviations	x
1 Introduction	1
1.1 Structure of the thesis	5
2 Background	6
2.1 ODE theory	6
2.1.1 Numerical methods for ODEs	7
2.1.2 Stiffness and stability	9
2.2 Runge–Kutta methods	11
2.2.1 Additive Runge–Kutta methods	13
2.3 Other IVP methods	15
2.4 Order conditions	16
2.4.1 Order conditions for Runge–Kutta methods	19
2.4.2 Order conditions for additive Runge–Kutta methods	21
2.5 Error control	22
2.5.1 Step-doubling error estimation	23
2.5.2 Error estimation via embedded methods	23
2.5.3 Step size selection	25
2.6 Dense output	26
2.7 Advection-reaction-diffusion equations	27
2.8 The method of lines	29
3 A problem-solving environment for the numerical solution of IVPs	32
3.1 IVP software implementation	34
3.2 Problem-solving environments	36
3.3 Architecture and design of <code>pythODE</code>	38
3.3.1 The <code>Solution</code> object	39
3.3.2 The <code>Solver</code> class and control flow	42
3.3.3 <code>SolverModule</code> objects	44
3.3.4 Statistics and error checking	44
3.3.5 Testing	46
4 Integrating-factor-based 2-additive-Runge–Kutta methods for ARD equations	48
4.1 Semi-Lagrangian exponential integrators for advection	49
4.1.1 Exponential Lie group methods	50
4.1.2 Commutator-free Lie group exponential methods	52
4.1.3 Order conditions for CFERK methods	54

4.1.4	Krylov subspace approximations to the matrix exponential	56
4.1.5	Error control for CFERK methods	56
4.2	Stabilized ERK methods for the diffusion term	57
4.3	Integrating-factor-based 2-ARK methods	60
4.3.1	Operator integrating factor splitting	60
4.4	Format of IF-2-ARK methods	61
4.4.1	Order conditions	62
4.5	The Burgers equation with Brusselator reaction terms	63
4.6	Proposed second-order IF-2-ARK methods	66
4.6.1	Constituent methods	66
4.6.2	10-Stage IF-2-ARK methods	67
4.7	Experimental results	68
4.7.1	Performance comparisons	70
5	Contributions and future work	80
5.1	Contributions	80
5.2	The <code>pythODE</code> PSE	81
5.3	Future work	82
5.3.1	Improvements to the IF-2-ARK methods	82
5.3.2	More complex ARD systems	83
5.3.3	High-order variants	83
5.3.4	Semi-Lagrangian methods	84
5.3.5	Further development of <code>pythODE</code>	84
	References	86
	A Examples related to the derivation of order conditions	92
	B Representative Behaviour of the Burgers, Brusselator, and Burgers-Brusselator Equations	102
	C Details on the IF-2-ARK methods	110
C.1	Derivation of IF-2-ARK methods	110
C.2	Tableaux of constituent methods used to propose IF-2-ARK methods	112

LIST OF TABLES

2.1	Total number of trees associated with derivatives of up to tenth order.	19
2.2	Total number of order conditions for RK methods up to tenth order.	20
2.3	Minimum number of stages $s_{\min}(p)$ required for ERK Methods of order p	21
2.4	Total number of order and coupling conditions for N -ARK methods of up to $N = 4$ and fifth order.	22
4.1	Coefficients used for the Brusselator and their steady state eigenvalues.	65
4.2	Maximum eigenvalues and stable stepsizes for grid size $n_{\text{grid}} = 100$ and diffusion values ϵ using the 10-stage ROCK2.	66

LIST OF FIGURES

2.1	The region (shaded) of absolute stability for FE.	10
2.2	The region (shaded) of absolute stability for BE.	10
2.3	The region (shaded) of absolute stability for RADAU5.	13
3.1	Flowchart of a linearly implicit Runge–Kutta solver.	34
3.2	Solution object with Solver object and two SolverModule objects	41
4.1	Stability region of 10-stage ROCK2 (dark shading) and the embedded method (light shading and dashed boundary).	59
4.2	Stability region of 14-stage ROCK4 (dark shading) and the embedded method (light shading and dashed boundary).	60
4.3	Eigenvalues of the first-order upwind finite-difference operator (2.47) with $n_{\text{grid}} = 100, x \in [0, 1]$	65
4.4	Eigenvalues of the second-order central finite-difference operator (2.45) with $n_{\text{grid}} = 100, x \in [0, 1]$	65
4.5	The region of absolute stability region for the 10-stage ERK method sharing b and c with the ROCK2 method (C.2).	67
4.6	Convergence of the trial reference solutions in number of digits of accuracy as compared to the reference solution with a tolerance of 10^{-10} for $\alpha = 0.2, \beta = 0.5$ (4.29a) (top), $\alpha = 2, \beta = 5$ (4.29b) (middle), and $\alpha = 1, \beta = 3$ (4.29c) (bottom).	71
4.7	Computational time compared to the RMS error of the solution with $\epsilon = 0.005$ (top), $\epsilon = 0.002$ (middle), $\epsilon = 0.001$ (bottom), and for the Burgers equation with $n_{\text{grid}} = 100$	74
4.8	Computational time compared to the RMS error of the solution with $\alpha = 0.5, \beta = 0.2$ (4.29a) (top), $\alpha = 2, \beta = 5$ (4.29b) (middle), $\alpha = 1, \beta = 3$ (4.29c) (bottom) as reaction coefficients for the Brusselator equation (4.24) with $\epsilon = 0.001$ and $n_{\text{grid}} = 100$	75
4.9	Computational time compared to the RMS error of the solution with $\alpha = 0.5, \beta = 0.2$ (4.29a) (top), $\alpha = 2, \beta = 5$ (4.29b) (middle), $\alpha = 1, \beta = 3$ (4.29c) (bottom) as reaction coefficients for the Burgers–Brusselator equation (4.25) with $\epsilon = 0.001$ and $n_{\text{grid}} = 100$	76
4.10	Computational time compared to the RMS error of the solution with $\epsilon = 0.002$ (top), $\epsilon = 0.001$ (middle), and $\epsilon = 0.0005$ (bottom) for the Burgers–Brusselator equation (4.25) with Brusselator reaction coefficients $\alpha = 0.5, \beta = 0.2$ (4.29a) and $n_{\text{grid}} = 100$	77
4.11	Computational time compared to the RMS error of the solution with $\epsilon = 0.002$ (top), $\epsilon = 0.001$ (middle), and $\epsilon = 0.0005$ (bottom) for the Burgers–Brusselator equation (4.25) with Brusselator reaction coefficients $\alpha = 2, \beta = 5$ (4.29b) and $n_{\text{grid}} = 100$	78
4.12	Computational time compared to the RMS error of the solution with $\epsilon = 0.002$ (top), $\epsilon = 0.001$ (middle), and $\epsilon = 0.0005$ (bottom) for the Burgers–Brusselator equation (4.25) with Brusselator reaction coefficients $\alpha = 1, \beta = 3$ (4.29c) and $n_{\text{grid}} = 100$	79
B.1	Solutions to the Burgers equation (4.5) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.005, 0.002, 0.001\}$ (left to right), and $n_{\text{grid}} = 100$	103
B.2	Solutions to the Brusselator equation (4.24) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 0.5, \beta = 0.2$ (4.29a) that have asymptotically decaying behaviour at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$	104
B.3	Solutions to the Brusselator equation (4.24) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 2.0, \beta = 5.0$ (4.29b) that are stable and oscillatory at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$	105

B.4	Solutions to the Brusselator equation (4.24) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 1.0$, $\beta = 3.0$ (4.29c) that are unstable at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$	106
B.5	Solutions to the Burgers–Brusselator equation (4.25) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 0.5$, $\beta = 0.2$ (4.29a) that have asymptotically decaying behaviour at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$	107
B.6	Solutions to the Burgers–Brusselator equation (4.25) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 2.0$, $\beta = 5.0$ (4.29b) that are stable and oscillatory at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$	108
B.7	Solutions to the Burgers–Brusselator equation (4.25) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 1.0$, $\beta = 3.0$ (4.29c) that are unstable at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$	109

LIST OF ABBREVIATIONS

ARD	Advection-Reaction-Diffusion
ARK	Additive Runge–Kutta
BDF	Backwards Differentiation Formula
BE	Backward Euler
CFERK	Commutator-Free Exponential Runge–Kutta
CFL	Courant–Friedrichs–Lewy
DIRK	Diagonally Implicit Runge–Kutta
ERK	Explicit Runge–Kutta
FE	Forward Euler
FSAL	First Same as Last
GUI	Graphical User Interface
IF-2-ARK	Integrating-Factor-Based 2-Additive-Runge–Kutta
IMEX	Implicit-Explicit Runge–Kutta
IRK	Implicit Runge–Kutta
IVP	Initial-Value Problem
MOL	Method of Lines
NDF	Numerical Differentiation Formula
ODE	Ordinary Differential Equation
OIF	Operator Integrating factor
PDE	Partial Differential Equation
PSE	Problem-Solving Environment
RHS	Right-Hand Side
RKC	Runge–Kutta–Chebyshev
RK	Runge–Kutta
SDIRK	Singly Diagonally Implicit Runge–Kutta

CHAPTER 1

INTRODUCTION

Many physical processes are modelled in terms of *differential equations*, i.e., equations for an unknown function in terms of its derivatives. Examples of differential equations are found in many of the mathematical models that are derived from fields such as physics, engineering, chemistry, biology, and the earth sciences [51, p.1]. Many specific physical processes in these fields are modelled as a flowing medium consisting of several interacting components. Examples include combustion in a flame [8, p.33], relevant to chemistry and engineering applications, oceanic and atmospheric models [79], relevant to current environmental issues such as climate change, and models of tumour angiogenesis, relevant to medical research [51, p.134].

There are three distinct physical processes that are predominant in many models of flowing media with interacting components: the motion of the flowing medium itself is a process known as *advection*, the interactions of the different species is a process known as *reaction*, and the change of concentrations due to gradients in the solution is a process known as *diffusion*. Collectively, models that use these three processes are known as *advection-reaction-diffusion* models and are expressed mathematically using differential equations [51, p.1]. The type of differential equations used are known as *partial differential equations*, i.e., differential equations for an unknown function of multiple independent variables in terms of its partial derivatives with respect to those variables. In most applications, an analytical solution to the partial differential equation describing a particular model is unavailable; therefore approximation methods known as *numerical methods* are used instead. Numerical methods *discretize* the variables of a differential equation; i.e., each variable is approximated at only a finite number of points within its domain in space and time. A common methodology used to solve partial differential equations is the process known as the *method of lines*, which approximates the partial differential equation by a system of ordinary differential equations, i.e., a differential equation with derivatives in terms of only one independent variable, typically time. Compared to partial differential equations, ordinary differential equations are more readily analysed and benefit from well-developed algorithms and software. This thesis is concerned with numerical methods for solving ordinary differential equations, mainly those derived from the semi-discretization of advection-reaction-diffusion equations by the method of lines.

A significant barrier to solving many advection-reaction-diffusion equations efficiently is that

each distinct process often requires a numerical method with different mathematical properties, and those properties are often mutually exclusive among many basic numerical methods. For example, one methodology for developing more complex numerical methods is known as *splitting*, where a different numerical method is used for each distinct process and the results are periodically merged. The three-component process of advection, reaction, and diffusion usually combines additively in the expression of the differential equation that describes a particular model or the corresponding terms of the ordinary differential equation after semi-discretization [51]. This motivates the development of complex numerical methods that additively combine more basic numerical methods. Crank–Nicolson–Leapfrog [51, p.387] is an example of a numerical method that is commonly used for advection-reaction-diffusion models and splits the vector field into two parts for an efficient solution. Because advection-reaction-diffusion equations model three processes, at least one of the two components of a numerical method that splits the vector field into two parts must be used for two physical processes at once. A potentially better solution is to use a distinct numerical method for all three physical processes of an advection-reaction-diffusion, an example of which is given in [8].

Mathematical analysis can typically give the general properties of a numerical method, such as order of convergence or limits on stability. However, empirical experimentation is required to determine which numerical methods are most practical and efficient for particular classes of ordinary differential equations. A major barrier to the widespread use of more complex numerical algorithms is the effort required to develop software that implements new numerical methods and compares them with existing methods.

To overcome the limitations of using traditional scientific software, software known as problem-solving environments are used [81]; well-known examples of problem-solving environments are MATLAB, MAPLE, and MATHEMATICA. These problem-solving environments implement numerical methods for solving partial differential equations and ordinary differential equations, often with uniform interfaces to allow the user to easily experiment with the various solvers. However, except for the embedded scripting languages, existing problem-solving environments do not have specific features to allow detailed experimentation with the wide range of ordinary differential equation methods that have been proposed and developed. In particular, the scripting language within MATLAB is used extensively in the literature to perform numerical experiments. However, a problem-solving environment that allows the user to design algorithms without extensive code modification is desirable in order to easily experiment with specific algorithms for specific classes of ordinary differential equations, including semi-discretized advection-reaction-diffusion equations.

The problem-solving environment `pythODE` has been developed specifically to experiment with a wide variety of numerical methods for ordinary differential equations, including novel solution methods. Modern object-oriented and software-engineering techniques are used to create a modular

framework that allows the components of an ordinary differential equation solver to be added or interchanged with minimal code modification. Additional features allow for the design of experiments involving many different solver components and their parameters, as well as detailed analysis of those experiments. The part of `pythODE` developed specifically for this thesis solves initial-value problems, which are ordinary differential equations where the state, typically at the initial time, is fully known. The typical solution method for an initial-value problem is to take a finite number of timesteps from the initial time until the final time where a solution is desired. The initial-value problems considered in this thesis are advection-reaction-diffusion equations semi-discretized by the method of lines.

The class of advection-reaction-diffusion equations of interest are *advection-dominated*; i.e., where the behaviour of the advection process is dominant over the diffusion and reaction processes. The problem-solving environment `pythODE` is used to test a composite numerical method that treats the three terms of an advection-reaction-diffusion equation with separate methods. Many numerical methods have stepsize restrictions due to stability or accuracy, as a result of properties such as *stiffness* and the Courant–Friedrichs–Lewy condition, when solving the initial-value problems from the semi-discretization of advection-reaction-diffusion equations. When present, these restrictions potentially require a computational cost many orders of magnitude higher for some classes of numerical methods than for others. Although a general-purpose implicit initial-value problem method typically solves most semi-discretized advection-reaction-diffusion equations accurately, there exist specialized methods considered in this thesis to overcome stepsize restrictions due to stiffness and the Courant–Friedrichs–Lewy condition more efficiently.

All three processes of an advection-reaction-diffusion equation can cause numerical methods to have some form of stepsize restriction after semi-discretization. The advection term is generally not considered stiff; however, stepsize restrictions due to the Courant–Friedrichs–Lewy condition are important. The diffusion term is generally considered stiff because, with most practical spatial grids, the corresponding initial-value problem term is stiff. The reaction term can either be stiff or non-stiff, depending on the specific advection-reaction-diffusion problem. However, the stiffness of the reaction term of the semi-discretized advection-reaction-diffusion is typically problem dependent. In the case of a stiff reaction term, typically there are no specialized methods to solve it more efficiently than a general-purpose implicit initial-value problem method. However, because the reaction term is not typically spatially coupled, i.e., it does not have partial derivatives with respect to the spatial variable, using splitting to solve it with a separate initial-value problem method from the advection and diffusion terms results in many smaller systems of equations. These can often be solved more efficiently than the large coupled system of equations that would result from solving the entire initial-value problem with an implicit initial-value problem method [89]. Even for the non-stiff case, solving the reaction term along with the other terms can interfere with finding the solution

of a semi-discretized advection-reaction-diffusion.

In general, numerical methods known as *Eulerian methods*, which observe the flowing medium from a fixed spatial grid, have many unsatisfactory characteristics when used to solve the advection term, including stepsize restrictions and poor qualitative behaviour. More suitable are *semi-Lagrangian* methods, which follow the fluid particles in the flowing medium and map the solution back to the fixed grid periodically. Semi-Lagrangian methods have enhanced stability and can exhibit superior qualitative behaviour to Eulerian methods [96, p.2207]. A connection has been made between semi-Lagrangian methods and exponential maps, such as the matrix exponential, by using *Lie group* theory [18, 22]. This theory gives a mathematical framework to construct numerical methods for advection-reaction-diffusion equations that incorporate semi-Lagrangian methods, which are specific only to the advection term, along with numerical methods suitable for the semi-discretized reaction and diffusion terms. The numerical methods of interest in this thesis are known as *Runge–Kutta* methods. High accuracy is achieved by approximating the solution at intermediate times and using these approximations to find the numerical solution at the end of the timestep. Numerical methods known as *additive Runge–Kutta* methods [7, 89] combine Runge–Kutta methods, typically two, for different terms of an additive vector field and have commonly been used for semi-discretized advection-reaction-diffusion equations.

A class of numerical methods that use exponential maps for the advection term are known as *commutator-free exponential Runge–Kutta* methods and can be viewed as generalizations of the well-known classic Runge–Kutta methods. When the diffusion process occurs isotropically, i.e., it occurs symmetrically with respect to each spatial dimension, this structure can be exploited by specific Runge–Kutta methods to overcome stiffness in the diffusion term. In the case of many advection-dominated advection-reaction-diffusion problems, the stiff diffusion term can be solved efficiently without requiring the solution of systems of implicit equations by using a class of Runge–Kutta methods known as *Runge–Kutta–Chebyshev* methods.

A simplified advection-dominated advection-reaction-diffusion equation is proposed and is shown to cause numerical difficulties with some existing initial-value problem methods. It is constructed as a combination of the well-studied *Burgers equation* (4.5), which has advection and diffusion, along with the well-studied *Brusselator equation* (4.24), which has diffusion and a nonstiff oscillatory reaction. This makes for an ideal initial-value problem to test 3-additive splitting methods. A new class of methods that we call integrating-factor-based 2-additive-Runge–Kutta methods is developed that combines a commutator-free exponential Runge–Kutta method for advection, a Runge–Kutta–Chebyshev method for diffusion, and a classic explicit Runge–Kutta method for the non-stiff reaction term. Although a semi-Lagrangian integrator could in principle be used for the exponential map of the commutator-free exponential Runge–Kutta method, a method based on the matrix exponential to approximate the exact flow is used instead as a proof of concept.

It is shown through numerical experiments with `pythODE` that integrating-factor-based 2-additive-Runge–Kutta methods can have lower computational cost than other methods such as Runge–Kutta–Chebyshev or 2-additive Runge–Kutta, which are known to solve some advection-reaction-diffusion equations more efficiently than fully implicit numerical methods. The integrating-factor-based 2-additive-Runge–Kutta methods are also shown to find numerical solutions with smaller errors than is feasible with the other methods tested.

1.1 Structure of the thesis

The remainder of the thesis is divided into the following chapters. The theory of numerical methods for the classic Runge–Kutta and additive Runge–Kutta methods, advection-reaction-diffusion equations, and the method of lines is covered in Chapter 2. A survey of existing software for solving semi-discretized advection-reaction-diffusion equations and an overview of the design of the problem-solving environment `pythODE` is covered in Chapter 3. Commutator-free exponential Runge–Kutta methods, Runge–Kutta–Chebyshev methods, and the new integrating-factor-based 2-additive-Runge–Kutta methods that treat each process of an advection-reaction-diffusion equation separately are covered in Chapter 4, with experiments to show that for specific semi-discretized advection-reaction-diffusion equations, these methods can outperform existing methods in terms of both efficiency and accuracy. The contributions and proposed future work based on the results from testing the new integrating-factor-based 2-additive-Runge–Kutta methods as well as additional detail on the current state of development of `pythODE` are covered in Chapter 5.

CHAPTER 2

BACKGROUND

2.1 ODE theory

The material in this section is largely adapted from [15] and [43].

An equation for an unknown function \mathbf{y} in terms of its derivatives is known as a *differential equation*. If the unknown function \mathbf{y} depends on only one independent variable t , i.e., $\mathbf{y} = \mathbf{y}(t)$, then the equation is known as an *ordinary differential equation* (ODE) and has the general form

$$\mathbf{F} \left(t, \mathbf{y}, \frac{d}{dt}\mathbf{y}, \frac{d^2}{dt^2}\mathbf{y}, \dots, \frac{d^M}{dt^M}\mathbf{y} \right) = \mathbf{0}. \quad (2.1)$$

A *solution* of (2.1) is a function $\mathbf{y} = \mathbf{y}(t)$ that satisfies (2.1). This thesis is primarily concerned with ODEs of the form

$$\frac{d}{dt}\mathbf{y}(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad (2.2a)$$

where $\mathbf{y}(t) : \mathbb{R} \rightarrow \mathbb{R}^m$ is a function with independent variable time $t \in \mathbb{R}$ and $\mathbf{f} : \mathbb{R} \times \mathbb{R}^m \rightarrow \mathbb{R}^m$ is typically called the *right-hand side* (RHS) of the ODE. Under suitable assumptions, ODEs of the form (2.1) can be converted to a system of first-order equations of the form (2.2a) for convenience in the use of software and analysis. An initial-value problem (IVP) is an ODE (2.2a) together with additional information about the solution known as the *initial condition*

$$\mathbf{y}(t_0) = \mathbf{y}_0, \quad (2.2b)$$

where $t_0 \in \mathbb{R}$ is known as the *initial time* and $\mathbf{y}_0 \in \mathbb{R}^m$ is known as the *initial value*.

In order to further simplify the analysis of (2.2a), the RHS can be converted into *autonomous form* by defining a new dependent variable, $t = \mathbf{Y}_{m+1}(t)$,

$$\mathbf{Y}(t) = \begin{pmatrix} \mathbf{y}(t) \\ t \end{pmatrix} = \begin{pmatrix} \bar{\mathbf{Y}}(t) \\ \mathbf{Y}_{m+1}(t) \end{pmatrix},$$

where $\bar{\mathbf{Y}}(t)$ represents the first m components of $\mathbf{Y}(t)$. The autonomous form allows the RHS to

be considered without the explicit dependence on t , i.e.,

$$\frac{d}{dt}\mathbf{Y}(t) = \mathbf{F}(\mathbf{Y}(t)),$$

where

$$\mathbf{F}(\mathbf{y}(t)) = \begin{pmatrix} \mathbf{f}(\mathbf{Y}_{m+1}(t), \bar{\mathbf{Y}}(t)) \\ 1 \end{pmatrix}.$$

Without loss of generality, the autonomous form of an ODE (2.2a) can be stated as

$$\frac{d}{dt}\mathbf{y}(t) = \mathbf{f}(\mathbf{y}(t)), \tag{2.3}$$

which is the notation used for the autonomous form of an ODE in this thesis with initial condition (2.2b) in the case of an IVP.

A function $\mathbf{f}(\mathbf{y}(t))$ is said to be *Lipschitz continuous* [15, p.5] on $D \subseteq \mathbb{R}^m$ if for some $R > 0$, a constant L can be chosen such that

$$\|\mathbf{f}(\mathbf{u}) - \mathbf{f}(\mathbf{v})\| \leq L\|\mathbf{u} - \mathbf{v}\| \quad \forall \quad \|\mathbf{u} - \mathbf{v}\| \leq R, \tag{2.4}$$

for any $\mathbf{u}, \mathbf{v} \in D$. If $\mathbf{f}(\mathbf{y})$ is Lipschitz continuous, then if a solution to (2.2) exists at some time, it is guaranteed to be unique [61, p.5]. Lipschitz continuity does not necessarily guarantee a solution exists on all of D [87, p.7].

2.1.1 Numerical methods for ODEs

Analytical methods exist for solving particular classes of ODEs, but these classes do not encompass many problems in practice, specifically large, complex, or nonlinear problems. Problems for which an analytical solution does not exist or is infeasible to use can generally be approximated using *numerical methods*.

Numerical methods for IVPs typically compute a solution sequentially by taking one or more steps in time t from the initial time t_0 to the final time t_f . In order to apply numerical methods effectively in general, it is necessary to estimate the accuracy of the solution that is computed.

The *global error* of the numerical solution of an IVP at time t_{n+1} is defined as

$$\mathbf{e}_{\text{global},n+1} = \mathbf{y}(t_{n+1}) - \mathbf{y}_{n+1},$$

where $\mathbf{y}(t_{n+1})$ is the exact solution at t_{n+1} and \mathbf{y}_{n+1} is the numerical solution at step $n + 1$ with a corresponding time of t_{n+1} . The global error is the net error accumulated during the solution, and in practice it cannot be calculated exactly because the exact solution is not generally known.

However, even in the case where an analytical solution is not known, the exact solution can be shown to exist and be unique [61, p.5].

An alternative to finding global error is to use the concept of *local error* or *local truncation error*, which in fact can be estimated in practice. The *local solution* $\tilde{\mathbf{y}}(t)$ is the exact solution of (2.2a) over one timestep Δt_n given the numerical solution \mathbf{y}_n as the initial condition. The numerical solution can be stated in terms of the local solution as

$$\mathbf{e}_{\text{local}} = \tilde{\mathbf{y}}(t_{n+1}) - \mathbf{y}_{n+1}, \quad (2.5)$$

where $\tilde{\mathbf{y}}(t_{n+1}) : \mathbb{R} \rightarrow \mathbb{R}^m$ is the local solution at t_{n+1} , $\mathbf{y}_{n+1} \in \mathbb{R}^m$ is the numerical solution at t_{n+1} , and $\Delta t_n = t_{n+1} - t_n$ is the stepsize. The asymptotic behaviour of the local error is $\mathcal{O}(\Delta t_n^{p+1})$, where p is known as the *order of convergence* of the numerical method. The asymptotic behaviour of the global error is $\mathcal{O}(\Delta t_n^p)$ because a numerical method takes $\mathcal{O}\left(\frac{t_f - t_0}{\Delta t}\right)$ steps in total over the solution interval and under the assumption $t_f - t_0 = \mathcal{O}(1)$ [86].

The most basic method for finding the numerical solution of an IVP is *Euler's method*, also known as *Forward Euler* (FE). One step of FE is given by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t_n \mathbf{f}(t_n, \mathbf{y}_n), \quad (2.6)$$

representing a constant approximation of \mathbf{f} in (2.2a) over one timestep. The approximate solution given by FE is the union of these approximations over the entire solution interval. Although in theory as $\Delta t_n \rightarrow 0^+$, FE approaches the exact solution, in practice the accuracy is limited by the finite amount of precision associated with computer arithmetic. The asymptotic behaviour of the local truncation error of FE after one step is $\mathcal{O}(\Delta t_n^2)$, and the order of convergence is one. This is generally too low to solve problems efficiently in practice; obtaining a highly accurate solution requires a considerable computational effort because stepsizes must be made small.

FE is also an example of an *explicit method*. A numerical method is said to be explicit if all quantities involved in taking a step are given in terms of known (past) quantities. Otherwise, the method is said to be implicit; i.e., there is at least one quantity given in terms of itself or unknown (future) quantities. The most basic implicit method is known as *Backward Euler* (BE), with one step given by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t_n \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}). \quad (2.7)$$

Finding a solution using this method requires the solution of m simultaneous equations and hence for a given stepsize is more computationally expensive than FE. The asymptotic behaviour of the local truncation error of BE after one step is $\mathcal{O}(\Delta t_n^2)$, and the order of convergence is one. This is the same order of convergence as FE, but in the next subsection we show why this method can

take much larger steps than FE for some problems, making it a useful method in practice.

2.1.2 Stiffness and stability

The material in this subsection is largely adapted from [15, 44, 61].

For some stepsizes that are small enough for a desired accuracy, FE is unstable for some IVPs; i.e., the numerical solution exponentially accumulates error. When a numerical method is constrained to use a stepsize that is excessively small relative to the accuracy required, it is said that the problem is *stiff* in the interval where this occurs. Stiff regions have large Lipschitz constants (2.4) that cause some numerical methods to become unstable, therefore resulting in an exponential accumulation of error and hence a “blow up” of the solution. The stepsize needed to yield a sufficiently accurate solution of stiff problems can be so small that solving the IVP becomes prohibitively expensive.

In order to study the stability of numerical methods, we first consider the linearization of (2.2a) at t_n as a system of linear constant-coefficient ODEs [44, p.15] given by

$$\frac{d}{dt}\hat{\mathbf{y}}(t) = \mathbf{J}_{\mathbf{f}}(\mathbf{y}(t_n))\hat{\mathbf{y}}(t), \quad (2.8)$$

where $\hat{\mathbf{y}}(t)$ is the solution to the linearized system of ODEs and $\mathbf{J}_{\mathbf{f}} \in \mathbb{R}^{m \times m}$ is known as the *Jacobian matrix*, given by

$$\mathbf{J}_{\mathbf{f}} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \cdots & \frac{\partial f_1}{\partial y_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial y_1} & \cdots & \frac{\partial f_m}{\partial y_m} \end{bmatrix}. \quad (2.9)$$

For the purposes of stability analysis, $\mathbf{J}_{\mathbf{f}}$ is assumed to be diagonalizable; i.e., there exists an invertible matrix \mathbf{P} such that $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_m) = \mathbf{P}^{-1}\mathbf{J}_{\mathbf{f}}\mathbf{P}$. The case where $\mathbf{J}_{\mathbf{f}}$ is not diagonalizable is covered in [46] but is beyond the scope of this thesis. A change of variables $\mathbf{u}(t) = \mathbf{P}^{-1}\hat{\mathbf{y}}(t)$ results in

$$\frac{d}{dt}\mathbf{u}(t) = \mathbf{\Lambda}\mathbf{u}(t), \quad (2.10)$$

which is equivalent to (2.8). This change of variables decouples the ODEs and allows (2.3) to be considered as m *scalar test problems* of the form

$$\frac{d}{dt}u(t) = \lambda u(t), \quad (2.11)$$

where $\lambda \in \mathbb{C}$ is the eigenvalue associated with a particular component of the ODE in (2.10).

Decoupling the variables in (2.2a) allows the stability of a numerical method to be studied using (2.11). If $\Re(\lambda) \leq 0$, then the exact solution of the scalar test problem remains bounded as

$t \rightarrow +\infty$. It is therefore desirable that any numerical method applied to (2.11) also produces a bounded solution. Applying FE (2.6) to (2.11) yields

$$u_{n+1} = u_n + \lambda\Delta t_n u_n. \quad (2.12)$$

It is easily seen that (2.12) remains bounded if and only if $|1 + \lambda\Delta t_n| \leq 1$. The values for $\lambda\Delta t_n$ for which a numerical method remains bounded when solving (2.11) define the *region of absolute stability*. Letting $z = \lambda\Delta t$, there exists a function $R(z)$ such that the region of absolute stability in the complex plane is given as $|R(z)| \leq 1$. In the case of FE, this region is defined by $R(z) = 1 + z$, a region that is a circle of radius 1 centred at $(-1, 0)$ as shown in Figure 2.1. Applying BE (2.7) to (2.11) yields

$$u_{n+1} = u_n + \lambda\Delta t_n u_{n+1}. \quad (2.13)$$

It is seen that (2.13) remains bounded if and only if $R(z) = \frac{1}{1-z}$. In this case, the region of absolute stability is the whole complex plane excluding a circle of radius 1 centred at $(1, 0)$ as shown in Figure 2.2. BE has an unbounded region of stability, and it is stable when solving stiff IVPs [44, p.3].

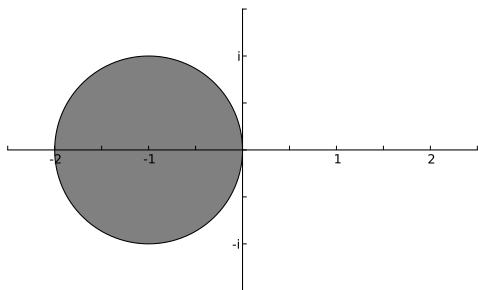


Figure 2.1: The region (shaded) of absolute stability for FE.

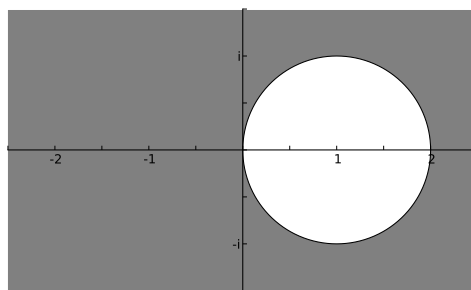


Figure 2.2: The region (shaded) of absolute stability for BE.

Several types of stability are defined based on the geometry of the stability region. Any method for which $|R(z)| \leq 1 \vee \Re(z) \leq 0$ is said to be *absolutely stable*; this property is also known as *A-stability*. A less restrictive form of stability is *A(α)-stability*, which is the region defined by $|R(z)| \leq 1$ if $-\alpha < \pi - \arg(z) < \alpha$, $\alpha \in (0, \pi/2]$. When $\alpha = \pi/2$ this corresponds to A-stability. A stronger form of stability is *L-stability*, which implies A-stability with the additional condition that $|R(z)| \rightarrow 0$ as $\Re(z) \rightarrow -\infty$ [61, p.225]. Although L-stability is effective for many types of IVPs because it rapidly dampens stiff components of the ODE, this property is not always desirable [44, p.44; 61, p.227–231].

Bounded stability regions constrain many numerical methods to a small stepsize when solving stiff problems. If this relatively small step size occurs because the solution is not smooth, this is to

be expected due to accuracy considerations, so the problem is not considered stiff in that interval. In general, numerical methods with unbounded stability regions are able to solve stiff problems more efficiently than those with bounded stability regions [61, p.220].

A good working definition of stiffness is that an IVP is stiff in an interval if it can be solved more efficiently in that interval with an implicit method designed for stiff problems, i.e., a *stiff method*, than with a method not suitable for stiff problems, i.e., a *non-stiff method*, such as an explicit method. There are specific explicit methods that can solve particular classes of stiff IVPs efficiently, but in general many practical problems are stiff and require the development of implicit methods for their efficient solution.

2.2 Runge–Kutta methods

The material in this section is largely adapted from [15] and [43].

Runge–Kutta (RK) methods are a generalization of FE and BE that use multiple function evaluations to obtain higher orders of convergence. The general form of an RK method is

$$\mathbf{k}_i = \mathbf{f} \left(t_n + \Delta t_n c_i, \mathbf{y}_n + \Delta t_n \sum_{j=1}^s a_{ij} \mathbf{k}_j \right), \quad i = 1, 2, \dots, s, \quad (2.14a)$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t_n \sum_{i=1}^s b_i \mathbf{k}_i, \quad (2.14b)$$

where a_{ij} are the elements of a matrix $\mathbf{A} \in \mathbb{R}^{s \times s}$, b_i , and c_i are elements of the vectors $\mathbf{b}, \mathbf{c} \in \mathbb{R}^s$ respectively, with $\mathbf{A}, \mathbf{b}, \mathbf{c}$ forming the *Butcher tableau* that characterizes a specific RK method. The \mathbf{k}_i are the s intermediate *stages* produced by the RK method and represent approximations to the derivative of $\mathbf{y}(t_n + \Delta t_n c_i)$. The Butcher tableau is often displayed graphically as follows

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

or

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array} \quad (2.15)$$

If \mathbf{A} is a strictly lower-triangular matrix, then the method is known as an *explicit Runge–Kutta* (ERK) method; i.e., each stage can be found sequentially without the need to solve systems of

simultaneous equations. The general form of an ERK method is given by

$$\mathbf{k}_i = \mathbf{f} \left(t_n + \Delta t_n c_i, \mathbf{y}_n + \Delta t_n \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j \right), \quad i = 1, 2, \dots, s, \quad (2.16a)$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t_n \sum_{i=1}^s b_i \mathbf{k}_i. \quad (2.16b)$$

An example of an ERK method that is presented in many textbooks, is “the” RK method [43, p.138], which has a Butcher tableau of

$$\begin{array}{c|ccc} 0 & & & \\ 1/2 & 1/2 & & \\ 1/2 & 0 & 1/2 & \\ 1 & 0 & 0 & 1 \\ \hline & 1/6 & 2/6 & 2/6 & 1/6 \end{array} \quad (2.17)$$

This method is fourth-order accurate and requires four evaluations of the RHS per step.

If \mathbf{A} is not strictly lower triangular then the method is called an *implicit Runge-Kutta* (IRK) method. IRK methods require the solution of systems of simultaneous equations because in principle the stages cannot be found explicitly. An example of an IRK method is the 3-stage Radau IIA method of fifth order, commonly known as RADAU5, that has a Butcher tableau of

$$\begin{array}{c|ccc} \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{360} \\ 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array} \quad (2.18)$$

RADAU5 requires the solution of $3m$ simultaneous equations for each step, making the implementation significantly more complicated than for an ERK method. RADAU5 has a stability function $R(z) = \frac{1+2z/5+z^2/20}{1-3z/5+3z^2/20-z^3/60}$, which implies L-stability and gives the unbounded stability region that is shown in Figure 2.3. In general, $R(z)$ for IRK methods is a rational function [43, p.41]. General IRK methods like the Radau IIA family can have arbitrarily high orders of convergence and still possess properties such as L-stability.

IRK methods that have a lower-triangular matrix \mathbf{A} are known as *diagonally implicit Runge-Kutta* (DIRK) methods. These methods allow the stages to be evaluated sequentially and require the solution of only m simultaneous equations for each stage. Therefore their implementation is

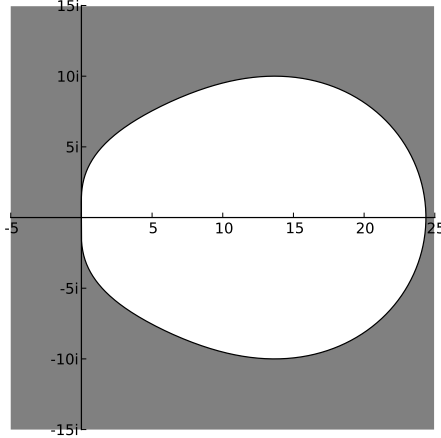


Figure 2.3: The region (shaded) of absolute stability for RADAU5.

simplified over a general IRK method with a full \mathbf{A} matrix. Furthermore, if all of the a_{ii} are equal, the same iteration matrix can be used for each stage when using *Newton's method*, an algorithm that requires the Jacobian (2.9) to solve the systems of simultaneous equations. The resulting methods are known as *singly diagonally implicit Runge–Kutta* (SDIRK) methods. An example of an SDIRK method is the L-stable SDIRK of fourth order [44, p.100], which has a Butcher tableau of

$$\begin{array}{c|cccccc}
 \frac{1}{4} & \frac{1}{4} & & & & \\
 \frac{3}{4} & \frac{1}{2} & \frac{1}{4} & & & \\
 \frac{11}{20} & \frac{17}{50} & -\frac{1}{25} & \frac{1}{4} & & \\
 \frac{1}{2} & \frac{371}{1360} & -\frac{137}{2720} & \frac{15}{544} & \frac{1}{4} & \\
 1 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \\
 \hline
 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4}
 \end{array} \tag{2.19}$$

RK methods applied to extremely stiff problems exhibit *order reduction*: they behave as though they have a lower order of convergence than that implied by the asymptotic behaviour of the local error (2.5). For stiff problems, all RK methods exhibit order reduction, ultimately being reduced to their stage order in the case of extremely stiff problems. The *stage order* is the minimum order of convergence of the method and its constituent stages. The constituent stages of an ERK or SDIRK method have an order of convergence of one, whereas general IRK methods can have higher stage orders. For example, s stage IRK methods of the Radau family have stage order s ; hence RADAU5 (2.18) has a stage order of three [44, p.237; 61, p.282].

2.2.1 Additive Runge–Kutta methods

The material in this subsection is largely adapted from [7], [51], and [55].

Some ODEs have a RHS that can be decomposed additively. An N -additive ODE has an RHS

of the form

$$\mathbf{f}(t, \mathbf{y}) = \sum_{\nu=1}^N \mathbf{f}^{[\nu]}(t, \mathbf{y}). \quad (2.20)$$

In many cases, each $\mathbf{f}^{[\nu]}(t, \mathbf{y})$ has unique characteristics, often arising from distinct physical processes, such as in the case of semi-discretized advection-reaction-diffusion equations that are further described in Section 2.7, that can be more efficiently solved using distinct numerical methods [55].

ARK methods treat the N terms (2.20) with N separate RK methods, allowing the use of an optimal method for each term. The general form of an additive Runge–Kutta (ARK) method is given by

$$\mathbf{k}_i^{[\nu]} = \mathbf{f}^{[\nu]} \left(t_n + \Delta t_n c_i^{[\nu]}, \mathbf{y}_n + \Delta t_n \sum_{\nu=1}^N \sum_{j=1}^s a_{ij}^{[\nu]} \mathbf{k}_j^{[\nu]} \right), \quad i = 1, 2, \dots, s,$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t_n \sum_{\nu=1}^N \sum_{j=1}^s b_j^{[\nu]} \mathbf{k}_j^{[\nu]},$$

where the variables are defined as for standard RK methods (2.14), except that there are N Butcher tableaux and Ns stage values. The Butcher tableaux for ARK methods then consist of N individual tableaux

$$\begin{array}{c|c} \mathbf{c}^{[1]} & \mathbf{A}^{[1]} \\ \hline & \mathbf{b}^{[1]T} \end{array}, \begin{array}{c|c} \mathbf{c}^{[2]} & \mathbf{A}^{[2]} \\ \hline & \mathbf{b}^{[2]T} \end{array}, \dots, \begin{array}{c|c} \mathbf{c}^{[N]} & \mathbf{A}^{[N]} \\ \hline & \mathbf{b}^{[N]T} \end{array}$$

In practical implementations, there are often simplifying assumptions, e.g., $\mathbf{b}^{[\mu]} = \mathbf{b}^{[\nu]}$, $\mathbf{c}^{[\mu]} = \mathbf{c}^{[\nu]}$, $\mu \neq \nu$, $\mu, \nu = 1, 2, \dots, N$. These assumptions reduce the number of remaining order conditions to be satisfied by the remaining free method coefficients required for the method to have a given order; this topic is discussed in Section 2.4.

The ARK methods most commonly used in practice are 2-ARK methods. Methods that are composed of an ERK method and an IRK method are known as *implicit-explicit* (IMEX) Runge–Kutta methods. These methods are commonly designed for 2-additive ODEs, where one term $\mathbf{f}^{[1]}$ is nonlinear or non-stiff and the other term $\mathbf{f}^{[2]}$ is linear or stiff [7]. The general form of a *linearly*

implicit IMEX method that uses an SDIRK method to solve $\mathbf{f}^{[2]}$ is

$$\mathbf{k}_i^{[1]} = \mathbf{f}^{[1]} \left(t_n + \Delta t_n c_i^{[1]}, \mathbf{y}_n + \Delta t_n \sum_{j=1}^{i-1} \left(a_{ij}^{[1]} \mathbf{k}_j^{[1]} + a_{ij}^{[2]} \mathbf{k}_j^{[2]} \right) \right), \quad i = 1, 2, \dots, s, \quad (2.21a)$$

$$\left(\mathbf{I} - \Delta t a_{ii}^{[2]} \mathbf{J} \right) \mathbf{k}_i^{[2]} = \mathbf{f}^{[2]} \left(t_n + \Delta t_n c_i^{[2]}, \mathbf{y}_n + \Delta t_n \sum_{j=1}^{i-1} \left(a_{ij}^{[1]} \mathbf{k}_j^{[1]} + a_{ij}^{[2]} \mathbf{k}_j^{[2]} \right) \right), \quad i = 1, 2, \dots, s, \quad (2.21b)$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t_n \sum_{i=1}^s \left(b_i^{[1]} \mathbf{k}_i^{[1]} + b_i^{[2]} \mathbf{k}_i^{[2]} \right), \quad (2.21c)$$

where a system of linear equations (2.21b) must be solved for each $\mathbf{k}_i^{[2]}$. An advantage of IMEX methods based on SDIRK is that for additive ODEs with a (stiff) linear term, each of the stages can be found sequentially with only one solution of a linear system, whereas a non-additive IRK method would generally require the solution of a nonlinear system.

An example of an IMEX method is IMEX(4,4,3) [7], which is third-order accurate with an L-stable SDIRK method, given by

0	0	0	0	0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0	0
$\frac{2}{3}$	$\frac{11}{18}$	$\frac{1}{18}$	0	0	0	$\frac{2}{3}$	0	$\frac{1}{6}$	$\frac{1}{2}$	0
$\frac{1}{2}$	$\frac{5}{6}$	$-\frac{5}{6}$	$\frac{1}{2}$	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
1	$\frac{1}{4}$	$\frac{7}{4}$	$\frac{3}{4}$	$-\frac{7}{4}$	0	1	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{2}$
	$\frac{1}{4}$	$\frac{7}{4}$	$\frac{3}{4}$	$-\frac{7}{4}$	0		0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{2}$

2.3 Other IVP methods

Multistep methods are a class of IVP methods that use information from multiple previous steps to obtain a high order of convergence. Explicit multistep methods are generally more efficient for high-accuracy solutions when function evaluations are expensive because they only require one function evaluation per step for any order, in contrast to the multiple function evaluations per step required by ERK methods [43, p.427]. Implicit multistep methods have the disadvantage of not being A -stable at orders of convergence higher than two, a consequence known as the *second Dahlquist barrier*. The highest known order of convergence for $A(\alpha)$ -stable implicit multistep methods is

six using BDF-based methods [44, p.247], but in practice the order is limited to five. *General linear methods* are a generalization of both multistep and RK methods, including ARK methods. They use both past values and multiple function evaluations per step to achieve a high order of convergence; they are further described in [15, 16, 43, 44, 53].

2.4 Order conditions

The material in this section is largely adapted from [15, 43, 61].

Numerical methods with high orders of convergence can reduce the computational cost of solving an IVP compared with low-order methods by many orders of magnitude, especially when high accuracy is desired [43, p.130]. High-order methods require careful choices for the values of \mathbf{A} , \mathbf{b} , and \mathbf{c} in the Butcher tableaux. The relationships between these coefficients that determine the order of a method are known as the *order conditions*.

The *Taylor series* [43, p.46] of $\mathbf{y}(t_n + \Delta t)$ satisfying (2.2a) at t_n is given by

$$\mathbf{y}(t_n + \Delta t) = \mathbf{y}(t_n) + \sum_{i=1}^{\infty} \frac{(\Delta t)^i}{i!} \frac{d^{(i-1)}}{dt^{(i-1)}} \mathbf{f}(\mathbf{y}(t_n)). \quad (2.22)$$

The Taylor series of a numerical method at \mathbf{y}_n is defined as the Taylor series of the function that results when the local solution $\tilde{\mathbf{y}}_n(t)$ is substituted into the numerical method. The truncation error (2.5) of a numerical method is equivalent to the terms of (2.22) that have $i > p + 1$, where p is the order of convergence. Thus for a method to have an order of convergence of p , the Taylor series of the numerical method must match the Taylor series of the local solution up to and including term $p + 1$. The order conditions required for an order of convergence of p are derived by matching the derivatives up to and including those of order p of the numerical solution with those of the local solution. Therefore if a numerical method satisfies all order conditions up to order p , then it has an order of convergence of p [43, p.144].

To describe the structure of higher derivatives of $\mathbf{y}(t)$, constructs known as *elementary differentials* are used, a term first introduced by John Butcher [13, 41]. The first derivative of $\mathbf{y}(t)$ is associated with an elementary differential given by

$$\mathbf{G}^1(\mathbf{f}(\mathbf{y}(t))) := \mathbf{f}(\mathbf{y}(t)). \quad (2.23a)$$

The elementary differentials associated with higher-order derivatives form a set of expressions, where the set of elementary differentials associated with derivative q of $\mathbf{y}(t)$ is the set $\mathbf{G}^q(\mathbf{y}(t))$

that contains a representative of each equivalent expression in the set $\bar{\mathbf{G}}^q(\mathbf{y}(t))$ given by

$$\bar{\mathbf{G}}^q(\mathbf{f}(\mathbf{y}(t))) := \left\{ \begin{array}{l} \mathbf{g}^q := \sum_{i=1}^m \left(\sum_{h_1, h_2, \dots, h_\sigma=1}^m \frac{\partial^\sigma f_i(\mathbf{y}(t))}{\partial y_{h_1} \partial y_{h_2} \dots \partial y_{h_\sigma}} (\mathbf{g}^{\delta_1})_{h_1} (\mathbf{g}^{\delta_2})_{h_2} \dots (\mathbf{g}^{\delta_\sigma})_{h_\sigma} \right) \mathbf{e}_i \\ q = 1 + \delta_1 + \delta_2 + \dots + \delta_\sigma \\ \sigma \in 1, 2, \dots, q-1 \\ \delta_j \in 1, 2, \dots, q-1 \\ \mathbf{g}^{\delta_j} \in \mathbf{G}^{\delta_j}(\mathbf{y}(t)) \end{array} \right\}, \quad (2.23b)$$

where $\mathbf{g}^q = \mathbf{g}^q(\mathbf{f}(\mathbf{y}(t))) : \mathbb{R}^m \rightarrow \mathbb{R}^m$ is an elementary differential of order q , y_{h_i} is component h_i of $\mathbf{y}(t)$, $(\mathbf{g}^{\delta_j})_{h_i}$ is component h_i of the elementary differential \mathbf{g}^{δ_j} , and $\mathbf{e}_i \in \mathbb{R}^m, i \in 1, 2, \dots, m$, are the standard basis vectors [43, 145-150; 61, p.157-162].


Derivative q of $\mathbf{y}(t)$ is given in terms of elementary differentials by

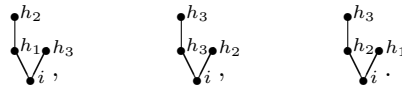
$$\frac{d^q}{dt^q} \mathbf{y}(t) = \sum_{\mathbf{g}^q \in \mathbf{G}^q(\mathbf{f}(\mathbf{y}(t)))} \alpha(\mathbf{g}^q) \mathbf{g}^q, \quad (2.24)$$

where $\alpha(\mathbf{g}^q)$ is a factor expressing the number of times each elementary differential \mathbf{g}^q appears in the expression of the derivative. This representation corresponds to a special case of Faà di Bruno's formula for the Taylor series of the exact solution $\mathbf{y}(t)$ from (2.2a) [43, p.150].

The increasing complexity of these expressions and the requirement to derive values such as $\alpha(\mathbf{g}^q)$ from (2.24) make it advantageous to use a graphical representation in the form of *rooted trees*. A rooted tree has one vertex that is known as the *root*. The vertices are connected via *branches*, and a vertex that does not have branches extending away from the root is a *leaf*. Let the mapping $\psi(v)$ take the vertex v to the next vertex closer to the root of the tree. The number of vertices of a rooted tree is known as its *order*.

The first-order elementary differential $\mathbf{G}^1(\mathbf{f}(\mathbf{y}(t))) \sim \bullet$ is represented as a single vertex. The elementary differential of $\mathbf{g}^q \in \mathbf{G}^q(\mathbf{f}(\mathbf{y}(t)))$ is represented as a rooted tree with σ branches extending from the root and constructed recursively by joining the roots of the trees representing $\mathbf{g}^{\delta_1}, \mathbf{g}^{\delta_2}, \dots, \mathbf{g}^{\delta_\sigma}$ from (2.23b). When $\mathbf{g}^{\delta_i} = \mathbf{G}^1(\mathbf{f}(\mathbf{y}(t)))$, the branch terminates in a leaf. Any rooted tree $\tau \sim \mathbf{g}^q$ has $\alpha(\tau) = \alpha(\mathbf{g}^q)$, which corresponds to the number of topologically unique

labellings of the non-root vertices of τ . For example, the rooted tree  has three topologically unique labellings of the non-root vertices, given by



An example of a rooted tree and the corresponding elementary differential is

$$\begin{array}{c} \bullet^{h_4} \\ \swarrow \downarrow \searrow \\ \bullet^{h_1} \bullet^{h_2} \bullet^{h_3} \\ \downarrow \\ \bullet^i \end{array} \sim \sum_{i=1}^m \left(\sum_{h_1, h_2, h_3=1}^m \frac{\partial^3 f_i(\mathbf{y}(t))}{\partial y_{h_1} \partial y_{h_2} \partial y_{h_3}} f_{h_1}(\mathbf{y}(t)) f_{h_2}(\mathbf{y}(t)) \left(\sum_{h_4=1}^m \frac{\partial f_{h_3}(\mathbf{y}(t))}{\partial y_{h_4}} f_{h_4}(\mathbf{y}(t)) \right) \right) \mathbf{e}_i,$$

that has $\sigma = 3$, $\delta_1, \delta_2 = 1$, and $\delta_3 = 2$. The elementary differential represented by \mathbf{g}^{δ_3} has $\sigma = 1$ and $\delta_1 = 1$. The first four derivatives of the exact solution of (2.2a) in terms of elementary differentials and rooted trees are as follows

$$\bullet \sim \frac{d}{dt} \mathbf{y}(t) = \mathbf{f}(\mathbf{y}(t)), \quad (2.25a)$$

$$\begin{array}{c} \bullet \\ \downarrow \\ \bullet \end{array} \sim \frac{d^2}{dt^2} \mathbf{y}(t) = \sum_{i=1}^m \left(\sum_{h=1}^m \frac{\partial f_i(\mathbf{y}(t))}{\partial y_h} f_h(\mathbf{y}(t)) \right) \mathbf{e}_i, \quad (2.25b)$$

$$\begin{aligned} \begin{array}{c} \bullet \\ \swarrow \downarrow \searrow \\ \bullet \bullet \bullet \end{array} + \begin{array}{c} \bullet \\ \downarrow \\ \bullet \end{array} &\sim \frac{d^3}{dt^3} \mathbf{y}(t) \\ &= \sum_{i=1}^m \left(\sum_{h_1, h_2=1}^m \frac{\partial^2 f_i(\mathbf{y}(t))}{\partial y_{h_1} \partial y_{h_2}} f_{h_1}(\mathbf{y}(t)) f_{h_2}(\mathbf{y}(t)) \right) \mathbf{e}_i \\ &+ \sum_{i=1}^m \left(\sum_{h_1, h_2=1}^m \frac{\partial f_i(\mathbf{y}(t))}{\partial y_{h_1}} \frac{\partial f_{h_1}(\mathbf{y}(t))}{\partial y_{h_2}} f_{h_2}(\mathbf{y}(t)) \right) \mathbf{e}_i, \end{aligned} \quad (2.25c)$$

$$\begin{aligned} \begin{array}{c} \bullet \\ \swarrow \downarrow \searrow \\ \bullet \bullet \bullet \end{array} + 3 \begin{array}{c} \bullet \\ \swarrow \downarrow \searrow \\ \bullet \bullet \bullet \end{array} + \begin{array}{c} \bullet \\ \swarrow \downarrow \searrow \\ \bullet \bullet \bullet \end{array} + \begin{array}{c} \bullet \\ \downarrow \\ \bullet \end{array} &\sim \frac{d^4}{dt^4} \mathbf{y}(t) \\ &= \sum_{i=1}^m \left(\sum_{h_1, h_2, h_3=1}^m \frac{\partial^3 f_i(\mathbf{y}(t))}{\partial y_{h_1} \partial y_{h_2} \partial y_{h_3}} f_{h_1}(\mathbf{y}(t)) f_{h_2}(\mathbf{y}(t)) f_{h_3}(\mathbf{y}(t)) \right) \mathbf{e}_i \\ &+ 3 \sum_{i=1}^m \left(\sum_{h_1, h_2, h_3=1}^m \frac{\partial^2 f_i(\mathbf{y}(t))}{\partial y_{h_1} \partial y_{h_2}} f_{h_1}(\mathbf{y}(t)) \frac{\partial f_{h_2}(\mathbf{y}(t))}{\partial y_{h_3}} f_{h_3}(\mathbf{y}(t)) \right) \mathbf{e}_i \\ &+ \sum_{i=1}^m \left(\sum_{h_1, h_2, h_3=1}^m \frac{\partial f_i(\mathbf{y}(t))}{\partial y_{h_1}} \frac{\partial^2 f_{h_1}(\mathbf{y}(t))}{\partial y_{h_2} \partial y_{h_3}} f_{h_2}(\mathbf{y}(t)) f_{h_3}(\mathbf{y}(t)) \right) \mathbf{e}_i \\ &+ \sum_{i=1}^m \left(\sum_{h_1, h_2, h_3=1}^m \frac{\partial f_i(\mathbf{y}(t))}{\partial y_{h_1}} \frac{\partial f_{h_1}(\mathbf{y}(t))}{\partial y_{h_2}} \frac{\partial f_{h_2}(\mathbf{y}(t))}{\partial y_{h_3}} f_{h_3}(\mathbf{y}(t)) \right) \mathbf{e}_i, \end{aligned} \quad (2.25d)$$

The expression corresponding to the tree $\tau = \begin{array}{c} \bullet \\ \swarrow \downarrow \searrow \\ \bullet \bullet \bullet \end{array}$ in (2.25d) has a coefficient of 3 because there

are three topologically unique labellings of the non-root vertices of τ , therefore in (2.24) $\alpha(\tau) = 3$.

Higher derivatives result in a combinatorial explosion of the number of trees needed for a representation of a particular derivative. Table 2.1 gives the number of topologically unique trees associated with each order of derivative q .

q	1	2	3	4	5	6	7	8	9	10
number of unique trees	1	1	2	4	9	20	48	115	286	719

Table 2.1: Total number of trees associated with derivatives of up to tenth order.

2.4.1 Order conditions for Runge–Kutta methods

The derivatives of the numerical solution from an RK method can be represented in terms of rooted trees in a similar manner to the exact solution. In this thesis it is assumed that the following holds

$$\sum_{j=1}^s a_{ij} = c_i, \quad (2.26)$$

allowing the autonomous form of an ODE (2.3) to be used in the analysis of RK methods [43, p.143].

The derivatives of the numerical solution of an RK method satisfy

$$\begin{aligned} \Phi_{v_1}(\tau) &= \sum_{v_2, v_3, \dots, v_q=1}^s a_{\psi(v_2), v_2} a_{\psi(v_3), v_3} \cdots a_{\psi(v_q), v_q}, \\ \frac{d^q}{dt^q} \tilde{\mathbf{y}}_n(t) &= \sum_{\tau \in T^q} \alpha(\tau) \gamma(\tau) \sum_{i=1}^s b_i \Phi_i(\tau) \mathbf{g}_\tau^q, \end{aligned} \quad (2.27)$$

where v_1, v_2, \dots, v_q are the q vertices of the rooted tree τ with v_1 representing the root, $\gamma(\tau)$ is the product of the number of vertices of τ with the number of vertices of the resulting rooted trees when the roots are removed one by one from τ , T^q is the set of rooted trees of order q , and $a_{\psi(v), v}$ is the value at row $\psi(v)$ and column v in the matrix \mathbf{A} from the Butcher tableau where the vertices are labelled with the set $\{1, 2, \dots, s\}$ based on the summations in (2.27) [43, p.151], and \mathbf{g}_τ^q is the elementary differential from $\mathbf{G}^q(\mathbf{f}(\mathbf{y}_n))$ equivalent to the tree τ .

For example, the fifth order tree $\tau = \begin{array}{c} \bullet^{h_4} \\ \swarrow \downarrow \searrow \\ \bullet^{h_1} \bullet^{h_2} \bullet^{h_3} \\ \downarrow \\ \bullet^i \end{array}$, gives $\Phi_i(\tau)$ from (2.27) as

$$\Phi_i(\tau) = \sum_{h_1, h_2, h_3, h_4=1}^s a_{i, h_1} a_{i, h_2} a_{i, h_3} a_{h_3, h_4} = \sum_{i, h_3=1}^s c_i^2 a_{i, h_3} c_{h_3},$$

This procedure is then repeated for all rooted trees of fifth order to form the fifth-order derivative of the numerical solution from (2.27).

The order conditions for order p are constructed by ensuring the expressions for the derivative and the numerical solution match up to order p . The number of order conditions that need to be satisfied for an order of convergence of p is the same as the number of rooted trees corresponding to that order. The total number of order conditions for an RK method to be of up to tenth order are given in Table 2.2.

p	1	2	3	4	5	6	7	8	9	10
conditions for order p	1	2	4	8	17	37	85	200	486	1205

Table 2.2: Total number of order conditions for RK methods up to tenth order.

Derivative q of (2.27) can be made equal to (2.24) if the additional terms in (2.27) are made equal to one by satisfying the expression

$$\gamma(\tau) \sum_{i=1}^s b_i \Phi_i(\tau) = 1. \quad (2.28)$$

The expressions (2.28) derived from derivative q constitute the order conditions for order q .

The order conditions up to an order of convergence of four follow, where (2.26) is used to simplify them where necessary [43, p.137,145]. The order condition that ensures a RK method has an order of convergence of one is

$$\sum_{i=1}^s b_i = 1. \quad (2.29)$$

The additional order condition that ensures an RK method has an order of convergence of two is

$$2 \sum_{i=1}^s b_i c_i = 1. \quad (2.30)$$

The two additional order conditions that ensure an RK method has an order of convergence of three are

$$3 \sum_{i=1}^s b_i c_i^2 = 1, \quad (2.31a)$$

$$6 \sum_{i,j=1}^s b_i a_{ij} c_j = 1. \quad (2.31b)$$

The four additional order conditions that ensure an RK method has an order of convergence of four

are

$$\begin{aligned}
4 \sum_{i=1}^s b_i c_i^3 &= 1, \\
8 \sum_{i,j=1}^s b_i c_i a_{ij} c_j &= 1, \\
12 \sum_{i,j=1}^s b_i a_{ij} c_j^2 &= 1, \\
24 \sum_{i,j,k=1}^s b_i a_{ij} a_{jk} c_k &= 1.
\end{aligned} \tag{2.32}$$

The maximum attainable order for an RK method is $p = 2s$ using the implicit Gaussian methods [44, p.71]. An ERK method requires $s_{\min}(p) \leq \frac{p^2-2p+4}{2}$ stages to achieve an order of convergence of p . Table 2.3 shows the bounds for the minimum number of stages for ERK methods of various orders [14, 27, 37, 40].

$p \leq 4$	$p = 5, 6$	$p = 7$	$p = 8$
$s_{\min}(p) = p$	$s_{\min}(p) = p + 1$	$s_{\min}(p) = 9$	$s_{\min}(p) = 11$
$p = 9$	$p = 10$	$p = 11, 12$	$p = 13, 14$
$12 \leq s_{\min}(p) \leq 17$	$13 \leq s_{\min}(p) \leq 17$	$p + 3 \leq s_{\min}(p) \leq 25$	$p + 3 \leq s_{\min}(p) \leq 35$

Table 2.3: Minimum number of stages $s_{\min}(p)$ required for ERK Methods of order p .

2.4.2 Order conditions for additive Runge–Kutta methods

The material in this subsection is largely adapted from [7, 51, 55].

The order conditions for an ARK method include the order conditions for the constituent RK methods as well as *coupling conditions*, which are order conditions that involve more than one of the constituent RK methods. They are represented by N -colourings of the rooted trees corresponding to the elementary differentials.

Each of the N colours corresponds to one of the constituent methods. The derivatives of the numerical solution of an ARK method satisfy

$$\Phi_{v_1}(\tau) = \sum_{v_2, v_3, \dots, v_q=1}^s a_{\psi(v_2), v_2}^{[\mu_2]} a_{\psi(v_3), v_3}^{[\mu_3]} \dots a_{\psi(v_q), v_q}^{[\mu_q]}, \tag{2.33a}$$

$$\frac{d^q}{dt^q} \tilde{\mathbf{y}}_n(t) = \sum_{\tau_c \in T^q} \sum_{\tau \in \tau_c} \alpha(\tau) \gamma(\tau) \sum_{i=1}^s \sum_{\mu_1 \in M} b_i^{[\mu_1]} \Phi_i(\tau) \mathbf{g}_\tau^q, \tag{2.33b}$$


where v_1, v_2, \dots, v_q are the q vertices of the coloured rooted tree τ with μ_1 the colour of the root v_1

and $\mu_2, \mu_3, \dots, \mu_p$ the corresponding colours of the rest of the vertices, M is the set of N colours, τ_c is a set of N -coloured trees of the same topology, T^q is the set of all rooted coloured trees of order q , and $a_{\psi(v),v}^{[\mu]}$ is the value at row $\psi(v)$ and column v in the matrix \mathbf{A} from the Butcher tableau of the method corresponding to the colour μ .

It is often assumed that

$$\mathbf{b}^{[\mu_i]} = \mathbf{b}^{[\mu_j]}, \mathbf{c}^{[\mu_i]} = \mathbf{c}^{[\mu_j]}, \quad \forall \mu_i, \mu_j \in N, \quad (2.34)$$

because then only coupling conditions based on colourings of the internal vertices of the rooted trees need to be considered, i.e., those that are not a root or leaf [55]. These coupling conditions are the expressions from the order conditions for standard RK methods (2.28) with a_{ij} in $\Phi(\tau)$ replaced by $a_{ij}^{[\mu_j]}$ as in (2.33a).

For example, the fourth-order coloured tree  gives a coupling condition of

$$12 \sum_{i, h_1, h_2, h_3=1}^s b_i^{[1]} a_{i, h_1}^{[2]} a_{i, h_2}^{[2]} a_{h_1, h_3}^{[1]} = 12 \sum_{i, h_1=1}^s b_i^{[1]} a_{i, h_1}^{[2]} c_i^{[2]} c_{h_1}^{[1]} = 1,$$

where $N = 2$ with \bullet and \circ corresponding to the colours 1 and 2 respectively.

The total number of conditions, both order and coupling, for N -ARK methods up to $N = 4$ and fifth order are given in Table 2.4. The number of order and coupling conditions increases very rapidly with increasing p and N due to the combinatorial explosion in the number of colourings of the associated rooted trees [55].

p	1	2	3	4	5
$N = 2$	2	4	18	60	230
$N = 3$	3	12	51	258	1509
$N = 4$	4	20	112	768	6028

Table 2.4: Total number of order and coupling conditions for N -ARK methods of up to $N = 4$ and fifth order.

2.5 Error control

The material in this section is largely adapted from [15] and [43].

Although the global error reflects the accuracy of the solution to (2.2), in general, it is only

possible to control the local error directly. In practice, the global error can be indirectly controlled by limiting the local error at each step. The solution of an IVP generally requires different stepsizes at various times to limit the error to a certain level. To minimize computational cost, an IVP solver must have a method of estimating the error and predicting the optimal stepsize. Ideally, the estimate should be done with minimal extra cost over that needed to find the numerical solution.

2.5.1 Step-doubling error estimation

Step-doubling error estimation is a method to estimate the local error based on *Richardson extrapolation* [43, p.156]. It can be used with any numerical method for which the order of convergence is known. It requires three steps to be taken from the current time t_n , two regular steps of size Δt_n , and one double step of size $2 \Delta t_n$ for the purposes of error estimation. The error in the first regular step is

$$\mathbf{e}_{n+\frac{1}{2}} = \mathbf{y}(t_n + \Delta t_n) - \mathbf{y}_{n+\frac{1}{2}} = C \Delta t_n^{p+1} + \mathcal{O}(\Delta t_n^{p+2}),$$

where $\mathbf{y}(t_n + \Delta t_n)$ is the local solution at $t_n + \Delta t_n$, $\mathbf{y}_{n+\frac{1}{2}}$ is the numerical solution at the first regular step, and p is the order of convergence of the numerical method. The error after the second regular step, taking into account the error after the first regular step, is

$$\mathbf{e}_{n+1} = \mathbf{y}(t_n + 2 \Delta t_n) - \mathbf{y}_{n+1} = 2 C (\Delta t_n^{p+1}) + \mathcal{O}(\Delta t_n^{p+2}).$$

The error after the double step is

$$\hat{\mathbf{e}}_{n+1} = \mathbf{y}(t_n + 2 \Delta t_n) - \hat{\mathbf{y}}_{n+1} = \hat{C} (2 \Delta t_n)^{p+1} + \mathcal{O}(\Delta t_n^{p+2}),$$

where $\hat{\mathbf{y}}_{n+1}$ is the numerical solution of the double step. If the constants C and \hat{C} are taken to be equal, the estimate to the local error is

$$\mathbf{y}(t_n + 2 \Delta t_n) - \mathbf{y}_{n+1} = \frac{\mathbf{y}_{n+1} - \hat{\mathbf{y}}_{n+1}}{2^p - 1} + \mathcal{O}(\Delta t_n^{p+2}), \quad (2.35)$$

which has an order of convergence of $p + 1$ [43, p.164].

A disadvantage of step-doubling error estimation is that it incurs the expense of 3 steps in order for the numerical solution to advance 2 steps. However, one step used in step-doubling is independent of the other two; therefore parallel computing can be used to mitigate this disadvantage.

2.5.2 Error estimation via embedded methods

Because finding the stage values is the most expensive part of an RK method, the computational cost of finding an error estimate can be minimized by reusing the stage values. *Embedded methods*

use the stage values with additional sets of quadrature weights to produce numerical solutions with differing orders of accuracy. The local error estimate for embedded RK methods is

$$\mathbf{e}_{n+1} = \tilde{\mathbf{y}}(t_n + \Delta t_n) - \mathbf{y}_{n+1} = \hat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1} + \mathcal{O}(\Delta t_n^{p+2}), \quad (2.36)$$

where $\tilde{\mathbf{y}}(t_n + \Delta t_n)$ is the local solution, \mathbf{y}_{n+1} is the numerical solution generated by the lower-order method, $\hat{\mathbf{y}}_{n+1}$ is the numerical solution generated by the higher-order method, \mathbf{e}_{n+1} is the local error estimate, and p is the order of convergence of the numerical solution of the higher-order method. The Butcher tableau for an embedded RK method is written as

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \hat{\mathbf{b}}^T \\ & \mathbf{b}^T \end{array}$$

where $\hat{\mathbf{b}}$ contains the quadrature weights of the embedded method. An RK method can contain more than one embedded method. An example is DOPRI853, an eighth-order method plus third-order and fifth-order embedded methods. Desirable properties cannot be found in a sixth-order embedded method without additional function evaluations, so the fifth-order embedded method estimates the error, and the third-order method is used to correct overestimation of the error; this is described in [43, p.243].

Although embedded methods often have more stage values than regular RK methods for a particular order of convergence, this approach is less costly than step-doubling. When the higher-order method is used to advance the integration, this is known as *local extrapolation*. When the coefficients of the truncation error (2.5) are too large for the higher-order method, using local extrapolation results in the estimated error becoming much smaller than the true error [43, p.178]. When local extrapolation is used and the error coefficients of the higher-order method are optimized, the estimate does not make sense in terms of a local error estimate but still produces a useful stepsize selection scheme [43, p.171]. This is because the local error generally has little in common with the global error, and the value produced is adequate for stepsize selection [43, p.168].

An example of an embedded RK method is Dormand–Prince 5(4) pair, which uses local extrapolation.

ulation with methods of fifth and fourth order. The Butcher tableau is given by

$$\begin{array}{c|ccccccc}
 0 & & & & & & & \\
 1/5 & 1/5 & & & & & & \\
 3/10 & 3/40 & 9/40 & & & & & \\
 4/5 & 44/45 & -56/15 & 32/9 & & & & \\
 8/9 & 19372/6561 & -25360/2187 & 64448/6561 & -212/729 & & & \\
 1 & 9017/3168 & -355/33 & 46732/5247 & 49/176 & -5103/18686 & & \\
 1 & 35/384 & 0 & 500/1113 & 125/192 & -2187/6784 & 11/84 & \\
 \hline
 & 35/384 & 0 & 500/1113 & 125/192 & -1287/6784 & 11/84 & 0 \\
 & 5179/57600 & 0 & 7591/16695 & 393/640 & -92097/339200 & 187/2100 & 1/40
 \end{array} \tag{2.37}$$

A concept known as *First Same As Last* (FSAL) is used to further reduce the number of function evaluations. The sixth stage of (2.37) at t_{n+1} is the fifth-order method used to advance the integration. The fourth-order method is a seven-stage method using the fifth-order method as its final stage. However, because the final stage is an RHS evaluation at t_{n+1} , it can be used as the first stage for the following step. In practice the seven stages of Dormand–Prince 5(4) now have no additional RHS evaluations over the minimum six stages necessary for a fifth-order ERK method [43, p.178]. Dormand–Prince 5(4) has proven to be extremely successful and is used in the MATLAB `ode45` code [88] and the FORTRAN software package DOPRI5 [43, p.475].

2.5.3 Step size selection

Step size selection is used to estimate the optimal step size during the integration such that the estimated local error remains below a user specified tolerance. The acceptable tolerances for the local error estimate for a numerical method are given by

$$\tau_{n+1,i} = \tau_{\text{rel},i} \max(y_{n,i}, y_{n+1,i}) + \tau_{\text{abs},i}, \tag{2.38}$$

where $\tau_{\text{rel},i}$ and $\tau_{\text{abs},i}$ are relative and absolute tolerances respectively for each component i of \mathbf{y} . The overall measure of the error is given by

$$\epsilon_{n+1} = \sqrt{\frac{1}{m} \sum_{i=1}^m \left(\frac{e_{n+1,i}}{\tau_{n+1,i}} \right)^2}, \tag{2.39}$$

where $e_{n+1,i}$ is component i of \mathbf{e}_{n+1} from (2.36). If $\epsilon_{n+1} > 1$, the estimated value of the local error is deemed to be unacceptably large, the step is rejected, and a new step is attempted from t_n . If $\epsilon_{n+1} \leq 1$, the estimated value of the local error is deemed to be acceptable, the step is accepted,

and a new step is attempted from t_{n+1} .

The new optimal stepsize is estimated by

$$\Delta t_{\text{opt}} = \Delta t_n (1/\epsilon_{n+1})^{(1/p+1)}, \quad (2.40)$$

where p is the minimum order of convergence of the methods used. Heuristics are often applied to the optimal stepsize to minimize the probability the next step will be rejected. A common way of doing this is

$$\Delta t_{\text{new}} = \alpha \min(a_{\text{max}} \Delta t_n, \max(\Delta t_{\text{opt}}, a_{\text{min}} \Delta t_n)),$$

where a_{max} and a_{min} are values representing the maximum amount the stepsize can increase and decrease respectively, and α is a safety factor. These values are generally different based on whether the step has been accepted or rejected, for example, to avoid a stepsize increase after a step is rejected. Typical values are $\alpha = 0.9$, $a_{\text{min}} = 0.2$, and $a_{\text{max}} = 5.0$ for an accepted step and $a_{\text{max}} = 1.0$ for a rejected step.

Control-theoretic stepsize selection

Principles from control theory can be used for stepsize selection; an example is the proportional-integral (PI) stepsize controller of [39]. This error controller is designed specifically for the stepsize changes that occur when stiff problems are solved with IRK methods. Changes in stepsize that are too large can result in large numbers of rejected steps; this stepsize controller uses control-theoretic techniques to avoid this situation. The estimate for the optimal stepsize after an accepted step, other than the first accepted step, is

$$\Delta t_{\text{opt}} = \frac{\Delta t_n}{\Delta t_{n-1}} \left(\frac{\epsilon_n}{\epsilon_{n+1}^2} \right)^{\frac{1}{p+1}},$$

where Δt_{n-1} is the stepsize of most recent accepted step and ϵ_n is the measure of the error (2.39) calculated from that step. After a rejected step, the optimal stepsize is calculated from (2.40).

2.6 Dense output

Strictly speaking, the numerical solution of (2.2) only exists at discrete points on the solution interval $[t_0, t_f]$. For many applications it is desirable to estimate the solution at arbitrary points; this is accomplished using *dense output* formulae. The estimates are often of a lower order of accuracy than the main numerical method.

A dense output formula for an RK method takes the form

$$\mathbf{y}_n(\theta) = \mathbf{y}_n + \Delta t_n \sum_{i=1}^{s^*} b_i(\theta) \mathbf{k}_i,$$

where there are $s^* - s$ additional stages over the regular RK method, the \mathbf{k}_i are the stages from the regular RK method plus any additional stages, $\theta = \frac{t-t_n}{\Delta t_n} \in [0, 1]$ indicates the relative time when a solution is desired, and $b_i(\theta)$ are the quadrature weights. The order conditions for the dense output satisfy

$$\sum_{j=1}^{s^*} b_j(\theta) \Phi_j(\tau) = \frac{\theta^{(|\tau|)}}{\gamma(\tau)},$$

that uses the notation of (2.28) and must satisfy the order conditions associated with all trees τ up to order p^* for the dense output to have an order of convergence of p^* .

An example of a fourth-order dense output formula for Dormand–Prince 5(4) (2.37) is

$$\begin{aligned} b_1(\theta) &= \theta^2(3 - 2\theta) b_1 + \theta(\theta - 1)^2 - \theta^2(\theta - 1)^2 5 (2558722523 - 31403016 \theta)/11282082432, \\ b_3(\theta) &= \theta^2(3 - 2\theta) b_3 + \theta^2(\theta - 1)^2 100 (88272551 - 15701508 \theta)/32700410799, \\ b_4(\theta) &= \theta^2(3 - 2\theta) b_4 - \theta^2(\theta - 1)^2 25 (443332067 - 3143016 \theta)/1880347072, \\ b_5(\theta) &= \theta^2(3 - 2\theta) b_5 + \theta^2(\theta - 1)^2 32805 (23143187 - 3489224 \theta)/199316789632, \\ b_6(\theta) &= \theta^2(3 - 2\theta) b_6 - \theta^2(\theta - 1)^2 55 (29972135 - 707673 \theta)/822651844, \\ b_7(\theta) &= \theta^2(\theta - 1) + \theta^2(\theta - 1)^2 10 (741447 - 829305 \theta)/29380423. \end{aligned}$$

This method is designed to minimize the coefficients of the truncation error of the dense output estimate. The coefficients of the truncation error (2.5) are found by evaluating the derivatives greater than p or p^* for the main method and dense output formula respectively, using the elementary differentials (2.23), and comparing them to those of the exact solution.

2.7 Advection-reaction-diffusion equations

The material in this section is adapted largely from [51].

A partial differential equation (PDE) is an equation for an unknown function of at least two independent variables involving the partial derivatives with respect to those variables. This thesis is primarily concerned with IVP methods that are part of the integration process for a class of PDEs known as advection-reaction-diffusion (ARD) equations. ARD equations typically represent a physical process that models the time evolution of a chemical or biological system in a flowing medium. The distinct chemical or biological components within the model are known as the *species*.

These systems are often modelled by looking at the behaviour of a small cell within a larger system.

The *advection* term of the PDE models the mass balance of the cell due to a flowing medium. The conservation of mass requires that the net change in concentration in a cell to be the sum of the net inflow and outflow. Thus, the advection term satisfies

$$\frac{\partial}{\partial t} \mathbf{u}(t, \mathbf{x}) + \frac{\partial}{\partial \mathbf{x}} (\mathcal{A}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})) \mathbf{u}(t, \mathbf{x})) = \mathbf{0}, \quad (2.41)$$

where $\mathbf{u}(t, \mathbf{x}) \in \mathbb{R} \times \mathbb{R}^d \rightarrow \mathbb{R}^q$ is the vector of concentrations of the q species at time t and a spatial position of $\mathbf{x} \in \mathbb{R}^d$ and $\mathcal{A}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})) \in \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^q \rightarrow \mathbb{R}^q$ is known as the advection coefficient.

The *diffusion* term of the PDE models the change in $\mathbf{u}(t, \mathbf{x})$ due to gradients in the solution and the fluxes across cell boundaries. The diffusion term satisfies

$$\frac{\partial}{\partial t} \mathbf{u}(t, \mathbf{x}) = \frac{\partial}{\partial \mathbf{x}} \left(\mathcal{D}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})) \frac{\partial}{\partial \mathbf{x}} \mathbf{u}(t, \mathbf{x}) \right), \quad (2.42)$$

where $\mathcal{D}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})) \in \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^q \rightarrow \mathbb{R}^q$ is the diffusion coefficient.

The *reaction* term of the PDE models the change in $\mathbf{u}(t, \mathbf{x})$ due to sources, sinks, and chemical reactions. The reaction term satisfies

$$\frac{\partial}{\partial t} \mathbf{u}(t, \mathbf{x}) = \mathbf{r}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})), \quad (2.43)$$

where $\mathbf{r}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})) \in \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^q \rightarrow \mathbb{R}^q$ models the reaction terms in one cell.

The general form of ARD equations modelling the net effect of the three processes is therefore given by

$$\frac{\partial}{\partial t} \mathbf{u}(t, \mathbf{x}) + \frac{\partial}{\partial \mathbf{x}} (\mathcal{A}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})) \mathbf{u}(t, \mathbf{x})) = \frac{\partial}{\partial \mathbf{x}} \left(\mathcal{D}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})) \frac{\partial}{\partial \mathbf{x}} \mathbf{u}(t, \mathbf{x}) \right) + \mathbf{r}(t, \mathbf{x}, \mathbf{u}(t, \mathbf{x})). \quad (2.44)$$

The *boundary conditions* are given by

$$\mathcal{B}_0(t, \mathbf{x}(t), \mathbf{u}(t, \mathbf{x})) \mathbf{u}(t, \mathbf{x}) + \mathcal{B}_1(t, \mathbf{x}(t), \mathbf{u}(t, \mathbf{x})) \frac{\partial}{\partial \mathbf{x}} \mathbf{u}(t, \mathbf{x}) = \gamma(t, \mathbf{x}), \quad \mathbf{x} \in \Gamma,$$

where $\mathcal{B}_0(t, \mathbf{x}(t), \mathbf{u}(t, \mathbf{x}))$, $\mathcal{B}_1(t, \mathbf{x}(t), \mathbf{u}(t, \mathbf{x})) \in \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^q \rightarrow \mathbb{R}^q$ are coefficients, Γ is a set representing the spatial boundary, and $\gamma(t, \mathbf{x})$ is an arbitrary function of the spatial variables. The initial condition is given by

$$\mathbf{u}(0, \mathbf{x}) = \mathbf{u}_0(\mathbf{x}).$$

2.8 The method of lines

The method of lines (MOL) is the process of semi-discretizing PDEs and leaving a single continuous dependent variable, typically time, resulting in a coupled system of ODEs (2.2a) of dimension $m = q \times n_{\text{grid}}$ with $\mathbf{y}(t) = \{\mathbf{u}_i(t)\}_{i=1}^{n_{\text{grid}}} \in \mathbb{R}^m$, where $\mathbf{u}_i(t) \in \mathbf{u}(t, \mathbf{x}_i), i = 1, 2, \dots, n_{\text{grid}}$ are the concentrations at time t for the spatial locations \mathbf{x}_i corresponding to the n_{grid} grid points. *Fully discrete* approximations can now be obtained by integrating in time with a suitable timestepping method.

In the case of a semi-discretized system of ARD equations, the advection terms tend to have predominantly imaginary eigenvalues, whereas the reaction and diffusion terms tend to have predominantly real eigenvalues, with those from the diffusion terms tending to be relatively large and those from the reaction terms tending to be problem dependent. In this application, different numerical methods can be optimal for integrating each term of (2.44) [51].

The methods used for semi-discretization in this thesis are *finite-difference methods* on a grid that is uniform in each spatial dimension with a spacing given by $\Delta \mathbf{x}$. The methods in the following examples use only one spatial dimension, one species, and *periodic boundary conditions*, i.e., in calculations \mathbf{u}_0 and $\mathbf{u}_{n_{\text{grid}}+1}$ are identified with \mathbf{u}_m and \mathbf{u}_1 respectively.

An example of *central differences* being used to semi-discretize the second-order partial derivative associated with a constant coefficient diffusion term (2.42) is given by

$$\frac{\partial}{\partial x} \left(\mathcal{D} \frac{\partial}{\partial x} \mathbf{u}(t, x) \right) \approx \mathcal{D} \frac{u_{i-1}(t) - 2u_i(t) + u_{i+1}(t)}{(\Delta x)^2} = \frac{\mathcal{D}}{(\Delta x)^2} \mathbf{D}_2^0 \mathbf{y}(t), \quad (2.45a)$$

$$\mathbf{D}_2^0 = \begin{bmatrix} -2 & 1 & & & & & 1 \\ 1 & -2 & 1 & & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \ddots & \ddots & \\ & & & & & 1 & -2 & 1 \\ 1 & & & & & & 1 & -2 \end{bmatrix}, \quad (2.45b)$$

where $u_i(t)$ corresponds to spatial point i and time t , $\mathbf{y}(t) : \mathbb{R} \rightarrow \mathbb{R}^m$ is the solution of the system of ODEs (2.2a) at time t from the semi-discretization, $\mathcal{D} \in \mathbb{R}$ is the constant diffusion coefficient, and there is an order of convergence of two with respect to Δx .

An example of central differences being used to semi-discretize the first-order spatial derivative

associated with a constant coefficient advection term (2.41) is given by

$$\mathcal{A} \frac{\partial}{\partial x} u(t, x_i) \approx \mathcal{A} \frac{u_{i-1}(t) - u_{i+1}(t)}{2 \Delta x} = \frac{\mathcal{A}}{2 \Delta x} \mathbf{D}_1^0 \mathbf{y}(t),$$

$$\mathbf{D}_1^0 = \begin{bmatrix} 0 & -1 & & & & & & & 1 \\ 1 & 0 & -1 & & & & & & \\ & \ddots & \ddots & \ddots & & & & & \\ & & & \ddots & \ddots & \ddots & & & \\ & & & & \ddots & \ddots & \ddots & & \\ & & & & & 1 & 0 & -1 & \\ -1 & & & & & & 1 & 0 & \end{bmatrix},$$

where $0 \geq \mathcal{A} \in \mathbb{R}$ is the constant advection coefficient and there is an order of convergence of two with respect to Δx . Central differencing introduces *artificial dispersion* from the different Fourier modes traveling at different speeds, leading to oscillations not present in the exact solution of the underlying PDE.

Finite differences for the advection term benefits from *upwind-biasing* for stabilization [77]. A first-order upwind discretization can also be used for the first-order spatial derivative associated with the advection term. An example of first-order upwind discretization with constant advection coefficients is given by

$$\mathcal{A} \frac{\partial}{\partial x} u(t, x_i) \approx \mathcal{A} \frac{u_{i-1}(t) - u_i(t)}{\Delta x} = \frac{\mathcal{A}}{\Delta x} \mathbf{D}_1^- \mathbf{y}(t), \quad (2.47a)$$

$$\mathbf{D}_1^- = \begin{bmatrix} -1 & & & & & & & & 1 \\ 1 & -1 & & & & & & & \\ & \ddots & \ddots & & & & & & \\ & & & \ddots & \ddots & & & & \\ & & & & \ddots & \ddots & & & \\ & & & & & 1 & -1 & & \\ & & & & & & 1 & -1 & \end{bmatrix}, \quad (2.47b)$$

which has an order of convergence of one with respect to Δx . Despite the lower order of convergence, upwind-biased schemes are often more satisfactory in practice because they do not introduce dispersion; however they can introduce *artificial diffusion* [51, p.56].

To illustrate the MOL applied to an ARD equation, consider the constant-coefficient ARD equation with one spatial dimension and one species given by

$$\frac{d}{dt} u(t, x) + \mathcal{A} \frac{\partial}{\partial x} u(t, x) = \mathcal{D} \frac{\partial^2}{\partial x^2} u(t, x) + r(u(t, x)),$$

where $\mathcal{A}, \mathcal{D} \in \mathbb{R}$ are the constant advection and diffusion coefficients. The advection and diffusion term are semi-discretized with upwind (2.47) and central (2.45) discretizations respectively,

therefore obtaining

$$\frac{d}{dt}\mathbf{y}(t) - \frac{\mathcal{A}}{\Delta x}\mathbf{D}_1^-\mathbf{y}(t) = \frac{\mathcal{D}}{(\Delta x)^2}\mathbf{D}_2^0\mathbf{y}(t) + \mathbf{r}(\mathbf{y}(t)),$$

where $\mathbf{y}(t) : \mathbb{R} \rightarrow \mathbb{R}^m$ is the spatial discretization of $u(x, t)$, $\mathbf{r}(\mathbf{y}(t)) = \{r(u_i)\}_{i=1}^{n_{\text{grid}}} : \mathbb{R}^m \rightarrow \mathbb{R}^m$ is the reaction term, and the resulting system can be trivially converted into a system of ODEs of the form (2.2a).

CHAPTER 3

A PROBLEM-SOLVING ENVIRONMENT FOR THE NUMERICAL SOLUTION OF IVPs

Numerical integration of IVPs using FE (2.6) dates back to the foundation of differential calculus. The first numerical methods with an order of convergence higher than one were the multistep methods published in 1883 by Adams [43, p.356]. RK methods were developed at the beginning of the twentieth century by Runge, Heun, and Kutta [43, p.134]. The development of computers allowed these calculations to take place electronically, with FORTRAN becoming the first dominant language for numerical computing.

The methods most commonly used for solving IVPs are FE (2.6) and “the” fourth-order ERK method (2.17). These methods are so ubiquitous that they are often the only RK methods known to many practitioners [61, p.183]. Although FE is commonly used because of its simple implementation, it is generally inefficient due to its low order of convergence; therefore fourth-order ERK methods are more suitable for non-stiff problems. Because the size of the stability region generally scales sublinearly with increasing order of convergence, the low cost of each step using FE generally means less computational cost overall in comparison to ERK methods when solving stiff problems [44, p.58; 61, p.221]. ERK methods are often used to solve stiff problems by many practitioners due to their simplified implementation even if the computational cost is greatly increased. An exception where a fourth-order ERK can be more efficient for stiff problems is when eigenvalues from (2.9) have significant imaginary parts, such as for the semi-discretized advection equation (2.41). This is because four-stage, fourth-order ERK methods have stability regions that cover a significant portion of the imaginary axis, unlike methods with $s \in \{1, 2\}$ stages [51, p.170] and $p = s$.

When used alone, RK methods lack error control such as that described in Section 2.5. Although any integration method of known order can use step-doubling error control from Section 2.5.1, if no error control is provided then a *constant stepsize* might as well be used. However, the efficient solution of IVPs requires different stepsizes at different times to limit local error; use of a constant stepsize restricts the stepsize to the smallest value required in the solution interval. Sophisticated non-stiff IVP solvers use error control. The FORTRAN software package DOPRI5 [43, p.475] and

the MATLAB function `ode45` [88] both use the explicit Dormand–Prince 5(4) scheme (2.37). The FORTRAN software package DOPRI853 uses the eighth-order DOPRI853 ERK method described in Section 2.5.2, and it is used extensively for high-precision computations due to its high order of convergence [43, p.243]. Multistep methods are popular for non-stiff problems with costly evaluations of the RHS. The MATLAB solver `ode113` is a representative example of a non-stiff multistep solver and uses methods of orders 1 to 13 [88].

When the MOL is applied to ARD equations, the resulting IVP is often stiff due to fine grid sizes and the inherent properties of the reaction and diffusion terms [51, p.64, 144]. Stiff solvers are implicit in most cases and therefore require the solution of systems of simultaneous equations. This makes them expensive to execute, but the stable steps are generally much larger than for an explicit method when solving a stiff problem [61, p.189]. Due to the larger stepsizes, a stiff solver can solve stiff problems with many orders of magnitude less computational cost than a non-stiff solver [44, p.143]. For moderately stiff problems, it is still practical to use explicit methods [33, p.36–39] when computational cost is not of utmost importance because non-stiff methods are less costly per step. Runge–Kutta–Chebyshev methods are an example of explicit methods designed for moderately stiff problems. They have an extended stability region for problems when the eigenvalues from (2.10) are mostly large, real, and negative such as the semi-discretized diffusion equation (2.42) [44, p.31; 51, p.419]. Runge–Kutta–Chebyshev methods are further discussed in Chapter 4. Even relatively small ODE systems can be extremely stiff and require a stiff solver in practice [85].

Multistep methods have commonly been used for stiff IVP solvers, with *variable-order* BDFs (Backwards Differentiation Formulae) being popular [88]. These methods are $A(\alpha)$ -stable, with higher-order methods having smaller stability regions. Methods with orders of convergence between 1 and 5 are used, with the lower-order methods being used to start the integration and accommodate IVPs that need a larger stability region. Commonly used software packages implementing BDF methods include LSODE [80] and VODE [12]. LSODE can detect if a problem is stiff and apply BDF methods rather than high-order explicit multistep methods when appropriate. The MATLAB function `ode15s` [88] implements an extension of BDF methods, known as NDFs (Numerical Differentiation Formulae).

General IRK methods can have many desirable properties such as A -stability and often retain these properties at arbitrarily high orders. Stability when the eigenvalues of (2.9) are imaginary is important for ARD equations; therefore stiff solvers using IRK methods are often preferred over other stiff solvers such as the $A(\alpha)$ -stable multistep solvers. The FORTRAN package RADAU5 uses the Radau IIA method of order five (2.18) [44, p.568]. The FORTRAN package RADAUP [44, p.574], which allows the user to select an order of five, nine, or thirteen, provides higher-order methods while still allowing lower-order methods because they are more efficient at crude tolerances. The FORTRAN

package SDIRK4 implements the fourth-order SDIRK method (2.19). The software package RODAS and the `ode23s` [88] function from MATLAB implement *Rosenbrock methods* [44, p.102], i.e., *linearly implicit Runge–Kutta* methods, which only require the solution of a linear system of equations, even for problems with a nonlinear RHS [44, p.102]. The FORTRAN package IRKC is designed for stiff reaction-diffusion problems. It uses an IMEX method with a Runge–Kutta–Chebyshev method as the explicit method for the diffusion term and an implicit method for the reaction term. The software package `odeToJava` uses the IMEX methods of [55] for solving stiff problems and Dormand–Prince 5(4) (2.37) for solving non-stiff problems. The MATLAB functions `ode23t` and `ode23tb` are also used for stiff problems and are described in [88].

3.1 IVP software implementation

The material in this section is largely adapted from [87].

This section outlines how the methods from Chapter 2 are used to construct IVP software. A flowchart showing the sequence of the components from a linearly implicit RK IVP solver is shown in Figure 3.1.

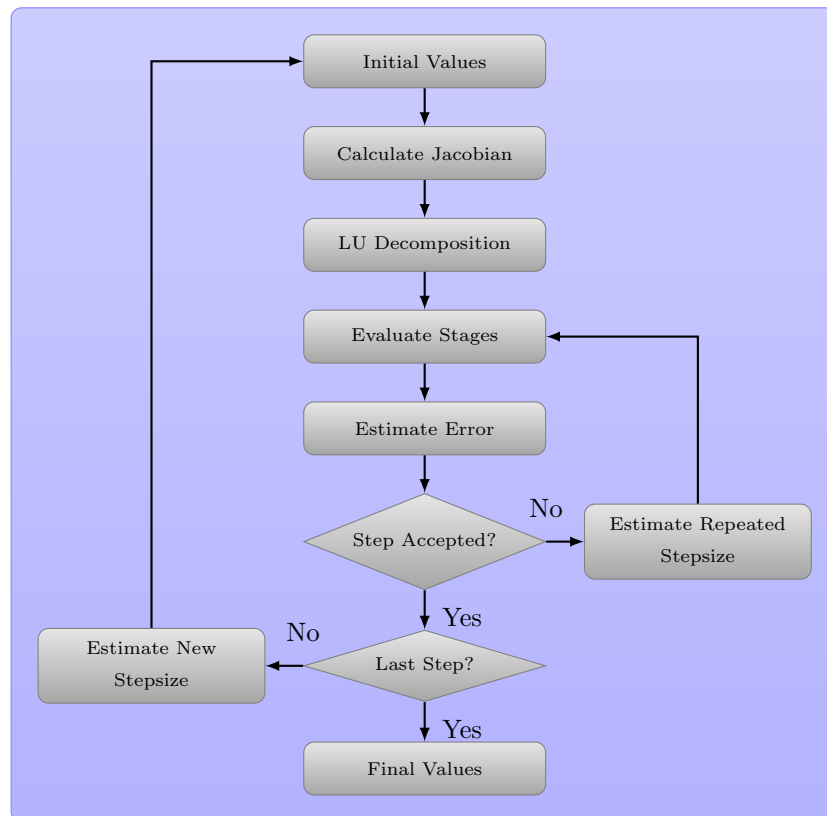


Figure 3.1: Flowchart of a linearly implicit Runge–Kutta solver.

IVP solvers are initialized with user-supplied initial time and initial values. In the case of a constant-stepsize solver, the initial stepsize is used for the whole integration, often requiring

considerable experimentation to ensure the solution is reliable. Although an initial stepsize is often provided for variable-stepsize solvers, there are algorithms to estimate an optimal value [43, p.169]. An improper estimate is not generally costly in this case because in practice the number of rejected or undersized steps at the start is small compared to the total number of steps.

One-step methods can start the integration immediately, but multistep methods require simpler methods to build up enough past solution values. This can be done by repeatedly using a one-step method or by using a variable-order multistep method. As the first-order method in variable-order multistep software, either FE (2.6) is used for non-stiff solvers or a one-step implicit method such as BE (2.7) is used for stiff solvers. The second step can now be computed with a two-step method if the solver finds this appropriate. This is continued until the solver has enough past steps for all of its methods, after which time it can choose whichever method it deems appropriate. Other methods that require past quantities, such as the PI stepsize controller described in Section 2.5.3, have their own methods of handling the beginning of the solution.

RK methods require the evaluation of the stages (2.14a) before finding the solution at the end of the step using the quadrature weights (2.14b). ERK methods and DIRK methods do this sequentially, with DIRK methods requiring the solution of simultaneous equations for each stage. General IRK methods such as RADAU5 (2.18) require the evaluation of all of the stages at once. Solving the systems of simultaneous nonlinear equations requires Newton's method because many other methods such as functional iteration are not robust enough for this application; this is because achieving convergence with functional iteration requires a restriction on the stepsize that is similar to the one for stability [44, p.119]. If the Jacobian (2.9) is evaluated at each step and stage, this maximizes the stability of the numerical method. Due to the expense of Jacobian evaluations, however, the Jacobian is often *frozen*; i.e., it is used beyond the time and solution values for which it was evaluated. This is an effective tradeoff when the computational savings from the reduction in stability from a frozen Jacobian offsets the computational cost of repeatedly evaluating and factoring it. Additional information on the solution of systems of simultaneous equations can be found in [54, 97].

Error estimation for each step, as described in Section 2.5, takes place after the integration has been attempted for that step. This estimate is then used in conjunction with stepsize selection like that from Section 2.5.3 to estimate the optimal stepsize. Additional information on stepsize selection for multistep methods can be found in [43]. If a step is rejected due to a level of error higher than the specified tolerances, the next step then proceeds using the same initial time and solution values used in the current step with the estimated optimal stepsize.

If the estimated error is within the specified tolerances, then the step is accepted, and the next step uses the time and solution values from the current step as its initial solution values. After a step has been accepted, dense output formulae may be applied in order to estimate the solution at

off-step times contained within the current step. At this point the solution values may be written to a data structure or file, if desired.

At the beginning of each new step, a check is made to see if that step is expected to reach the final time. Accordingly, the last step in the solution may be adjusted if appropriate. It is safely assumed that shortening the last step decreases the estimated error, as can easily be seen from the asymptotic behaviour of the truncation error (2.5). In many implementations, the last step may also be stretched by a small factor in order to reach the final time, thus avoiding an extra step.

3.2 Problem-solving environments

Many methods for solving IVPs (2.2) are implemented only in certain software packages and in a particular language. Some of the software packages described in this chapter are the only available implementations of the constituent methods. Porting each IVP to a different software package can make experimenting with a wide range of solution methods a substantial development effort. An example of where methods are compared can be found in [33, p.37], where a comparison is made of the computational cost of various software packages on a target set of problems. Experimentation with a method unavailable in existing software packages is done in [93], where the RADAU5 method was modified to use a digital filter, one that is more sophisticated than the control-theoretic method described in Section 2.5.3, for stepsize control. Typically this type of modification requires a significant development effort, limiting the type and scope of experimentation that has been done.

The FORTRAN solvers described in this chapter are used as library functions and require the user to write a driver for a particular IVP. Many of these software packages have been in wide use for a substantial amount of time; for instance, a common version of LSODE dates back to 1981. A reason for this is because the exact solutions to scientific computing problems are not generally known, even if the methods used are theoretically sound, and incorrect solutions due to bugs or poor implementation are not always obvious. Therefore it can take a substantial amount of time for a software package to become trusted within the community. Reasons for the popularity of FORTRAN are the suitability when computational cost is important, domain-specific syntax for manipulating floating-point arrays, an extensive selection of high-performance numerical libraries, and the most commonly available implementations of many scientific problems, e.g., the code FUNWAVE for simulating shallow water equations using the fully nonlinear Boussinesq model that is further described in [59].

These types of issues with numerical software have led to development of problem-solving environments (PSEs). Rice and Boisvert have given the following description of a PSE [81]:

A PSE is a computer system that provides all the computational facilities necessary to solve a target class of problems efficiently. The facilities include advanced solution methods, automatic or semi-automatic selection of solution methods, and ways to easily

incorporate novel solution methods. They also include facilities to check the formulation of the problem posed, to automatically (or semi-automatically) select computing devices, to view or assess the correctness of solutions, and to manage the overall computational process. Moreover, PSEs use the terminology of the target class of problems, therefore users can solve them without specialized knowledge of the underlying computer hardware, software, or algorithms. In principle, PSEs provide a framework that is all things to all people; they solve simple or complex problems, support both rapid prototyping and detailed analysis, and can be used both in introductory education or at the frontiers of science.

MATLAB is an example of a PSE for scientific computing problems. It has a scripting language with special features for matrix operations and includes an extensive library of mathematical functions, an interactive command prompt for easy prototyping, graphical user interfaces (GUIs) for tasks such as plotting data, and toolboxes for special functionality such as signal processing. The MATLAB IVP solvers described in this chapter can be interchanged easily, a significant advantage given that many problems are more amenable to certain integration methods [61, p.226-231]. Despite the ease of scripting MATLAB, it is still not straightforward to experiment with substantially different IVP methods beyond those that are already implemented.

Other PSE-like approaches include `odeToJava`, which has a similar interfaces for each of its different IVP solvers, allows arbitrary Butcher tableaux, and includes several options for error control and input/output. `VFGEN` is a front end that allows a common input file to be used for for 11 different ODE solvers, including the FORTRAN ones mentioned in this chapter, as well as other applications such as automatic differentiation or generating L^AT_EX code [104]. These PSE-like approaches make it easier to experiment with different methods, but they still tie the user to existing solvers and methodologies.

Due to the restrictions on available software, it can be necessary to compare across different software packages and programming languages, as is done in [33, p.37]. Although this type of comparison is useful for finding the best software package for a target problem class, it only indirectly compares the underlying methods. Many published comparisons between numerical methods are not rigorous but rather are a “mathematical demonstration” [93, p.245] to show the theoretical properties and practical viability of the methods being tested. Code comparisons such as those in [33, p.37] are not performed *ceteris paribus*, i.e., testing only one aspect of a method at a time with all other things being equal. Differences in implementation and non-uniformities in performance across programming environments mean that testing across software packages introduces many additional variables and therefore cannot be performed *ceteris paribus*. To rigorously evaluate numerical methods empirically requires a large number of precisely controlled experiments [93, p.245], a requirement that is often impractical within existing software packages. A PSE is the ideal environment for this type of experimentation between different numerical methods.

3.3 Architecture and design of pythODE

The pythODE PSE is designed to be modular and based on accepted software engineering principles rather than being optimized for performance. The component of pythODE developed specifically for this thesis solves IVPs, another component solves boundary-value problems, where information about the solution is given at times other than the initial time. In order to facilitate experimentation with IVP methods, pythODE allows the different components of an IVP solver to be integrated without modifying the numerical code, therefore easily allowing a user to perform a wide range of experimentation. The object-oriented architecture of pythODE offers an abstraction of common algorithms for IVPs that include Runge–Kutta methods as well as the multistep and general linear methods from Section 2.3, which together constitute a large family of methods that have been the subject of active research for over 100 years [43, p.134]. Additional numerical methods for IVPs have been identified that fit within the general architecture of pythODE. If a front end such as a graphical user interface were developed, no programming would be strictly required to construct new solvers from existing components.

The pythODE PSE is also a platform to facilitate computational experimentation. For example, pythODE could be used to compare stepsize selection methods, such as the control-theoretic method described in Section 2.5.3 and the more sophisticated digital filter methods described in [92, 93]. In [93], a more sophisticated digital filter method was only used with the integration methods in RADAU5 and those in another package, DASPK. Despite using only two integration methods, this experiment likely required a substantial development effort on the part of the authors. In a modular software environment, these digital filter methods could be compared rigorously with another stepsize selection method using a wider range of integration methods. Each constituent method would only have to be written once but then could be reused in many combinations. The only component that would change within the solver would be the integration module, therefore experimentation can easily be performed *ceteris paribus*. In the case where the user is only interested in the solution to an IVP, computational experimentation is still necessary because even well-used methods can exhibit unexpected behaviour [17, 91]. To have confidence in the solution to an IVP, it is generally necessary to solve it with various solver parameters such as stepsize when constant or error tolerance (2.38).

The pythODE PSE could also be used as a classroom tool due to the ease of comparing and demonstrating different methods for solving IVPs. This would provide more flexibility than other PSEs commonly used for this purpose such as MATLAB, MAPLE, and MATHEMATICA. An example of this usage is changing a Butcher tableau to easily demonstrate the differences between the members of a family of methods.

The pythODE PSE provides testing and statistics components to facilitate the development of

new modules. An important component is a test suite that can help to verify the functionality of the whole package. It can detect abnormal behaviour by verifying standard methods against a test suite of IVPs, allowing continued development while keeping `pythODE` trustworthy. Furthermore, although existing software packages provide statistics on the solution process, they tend to be basic and differ between software packages. An important feature of `pythODE` is that it provides extensive statistics about all IVP methods and incorporates them without code modification.

PYTHON is the language of choice for the `pythODE` PSE. It provides access to a wide variety of language paradigms including imperative, object-oriented, and functional programming. The overhead from the modularity and the relative slowness of PYTHON compared to languages such as FORTRAN and C [82] make `pythODE` unsuitable if performance is of utmost importance. Despite this, the ease with which it allows testing of new methods means it can provide information for the development of high-performance IVP solvers. PYTHON includes features such as operator overloading, which allows the programmer to use a more domain-specific notation when coding mathematical expressions [82]. It also includes a large standard library, giving access to a large amount of functionality. C and FORTRAN code can be easily called from PYTHON, thus allowing access to a large amount of existing software. However, mixing languages may interfere with performing tests *ceteris paribus* because C and FORTRAN generally execute the same operations significantly faster than does PYTHON.

The main numerical libraries used in `pythODE` are NUMPY and SCIPY. NUMPY provides multidimensional arrays of arbitrary data types, including real and complex floating-point numbers as well as basic linear algebra operations. SCIPY uses NUMPY for fundamental operations and adds functionality such as sparse linear algebra, special functions, and statistics. The library SYMPY allows symbolic manipulation of mathematical expressions. Symbolic manipulation is generally too inefficient to be used during the numerical solution process, but it is used to derive the coefficients of some numerical methods. The MATPLOTLIB library provides plotting and visualization of data in NUMPY arrays. Together these libraries add a domain-specific functionality for PYTHON comparable to commercial PSEs such as MATLAB, MAPLE, and MATHEMATICA.

3.3.1 The Solution object

Distributing the behaviour of software amongst objects results in many connections between those objects. This can reduce the re-usability of those objects because in the worst case, every object knows about every other object. With so many interconnections, these objects cannot act without the support of other objects, making the system behave as if it were monolithic. The degree to which components of a system rely on other components is known as *coupling*. A high degree of coupling can make it difficult to change the behaviour of the system in any significant way [38, p.273]. The IVP software packages described in this chapter are generally monolithic and have a high degree of

coupling amongst the various components; therefore they are not readily modified.

A *mediator pattern* is an object that encapsulates how a set of other objects interact, keeping them from referring to each other explicitly. Rather than interact directly with each other, the objects now interact indirectly through the mediator. When the objects no longer have to refer to each other explicitly, the system is known as *loosely coupled*. A loosely coupled system allows the interaction of an object to be varied independently without affecting the other objects [38, p.273].

In `pythODE`, an object known as `Solution`, an example of which is shown in Figure 3.2, acts as a mediator between the different components of an IVP solver. This type of abstraction is less common in scientific computing than other types of computing. `Solution` holds all current and past solution points, their stage values, as well as any other information required, for example, stepsize, or information from the error controller. The numerical solution is stored in `NUMPY` arrays on a per-step basis, but other types of information, such as strings to record information such as errors or reasons for step rejections or integers are also accommodated. This allows single-step methods such as RK, as well as multistep methods and general linear methods [15, 16, 43, 53], to be implemented in the same framework. Past solution values may still be required for single-step methods because error control such as the PI error controller from Section 2.5.3 or FSAL use past information. If applicable, higher derivatives can be stored in order to allow the implementation of methods such as Runge–Kutta–Nyström [43, p.284] and Taylor-series methods [28]. When each component can be modified completely independently of the others, this readily allows changing the behaviour of the solver. Easy modification allows the large number of carefully controlled experiments needed to make empirical claims [93]. Even components that are functionally dependent, such as error estimation and stepsize selection, still integrate well into this framework.

A new construct called *subsolutions* are provided for the `Solution` objects in order to accommodate composite methods such as step-doubling and extrapolation methods [43, p.129; 44, p.131]. These are IVP solutions that branch off from the main solution at a particular step but use solution values from the main solution prior to this step. This behaviour is transparent to solver components and allows the same modules to operate on both the main solution and its subsolutions. An example of where subsolutions are applied is in the context of step-doubling error control. The regular steps operate on the main solution because they are used to advance the solution, and a subsolution is created for the double-sized step. In this case the module used for integration does not have to know about step-doubling error control.

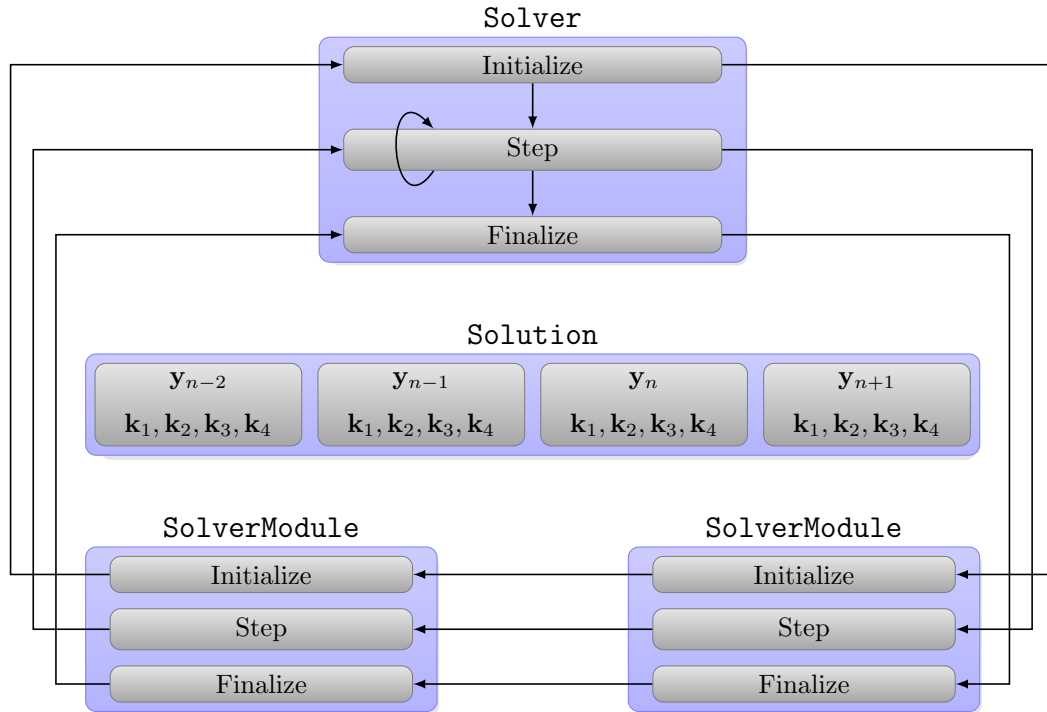


Figure 3.2: Solution object with Solver object and two SolverModule objects

Solution has a compact and intuitive syntax that allows its interface to resemble mathematical notation. To extend the dynamic language paradigm used in PYTHON, stage and solution points do not have to be explicitly allocated. They are allocated by referencing; therefore no prior knowledge of the structure of a particular **Solution** object is necessary.

The following is an example of accessing the current value of **Solution** and storing it in the variable **z1**: the first zero indicates that this is the current step, the second zero indicates that this is derivative zero, i.e., the solution itself.

```
z1 = example_solution[0][0]
```

The following is an example of setting the second derivative of the solution point two steps prior to the current solution point to the variable **z2**.

```
example_solution[-2][2] = z2
```

The following is an example of accessing the first derivative of stage four of the solution point prior to the current solution point and storing it in the variable **z3**.

```
z3 = example_solution[-1]['stage'][1][4]
```

The use of the key **'stage'** gives an example of how metadata can be stored and accessed on a per-step and per-stage basis. Arbitrary keys can be used to store data without prior declaration.

The method **step** for **Solution** objects shifts the solution points so the current solution point becomes the previous solution point to the current solution point, the previous solution point to the

current solution point is shifted to a point two points previous to the current solution point, etc. This accomplishes stepping in time of the IVP solver, and `step` can be called with an argument to step more than one solution point. The following is an example of stepping the `Solution` object `example_solution` two points forward.

```
example_solution.step(2)
```

This keeps the previously computed values but allows them to be referenced differently as each subsequent timesteps requires. In the general case this is *stateful*, i.e., information about what has previously occurred is maintained.

For IVPs that require many steps or are of a significant size, it is generally not desirable to store all past solution points due to memory limitations. Therefore `Solution` includes methods to remove old values. It is unknown how many previous solution points each solver module may need; therefore this must be declared for each module. The method `keep` is called at the initialization of each solver module, then the maximum value from all calls to `keep` is used as the number of steps to save. After each step, the unneeded solution values are removed, thus saving resources. If more past values than are required for the integration are needed, it is advisable to use a module to write the solution values to a data structure or a file.

3.3.2 The Solver class and control flow

The class in `pythODE` that coordinates program flow and stepping is `Solver`. `Solver` is subclassed for different types of stepping such as with constant stepsize (no error estimation), with step-doubling error estimation (2.35), and with embedded error estimation (2.36). In this case, these subclasses can be behavioural subtypes and avoid violating the *Liskov substitution principle*, which states that any property true for a type (or class) must also be true for its subtypes (or subclasses) [63]. The Liskov substitution principle can easily be violated when using object-oriented techniques in the design of mathematical software because there exist many analogous cases of the classic circle-ellipse problem from [67] when implementing mathematical structures using object-oriented design. The circle-ellipse problem is used to show that although a circle is a type of ellipse, in general, a circle that is represented as a subtype of an ellipse may not be a behavioural subtype; i.e., a property provable for a type of objects should be true for its subtypes. For example, an ellipse type may contain a method to stretch one of the principle axes but not the other. A circle subtype would therefore not be able to satisfy both its own invariant, that both axes are equal, and at the same time satisfy the behavioural requirements of the method to stretch one axis inherited from the ellipse type. Failure to respect the Liskov substitution principle is known to cause difficulty in the overall implementation, interoperability, and maintainability of complex object-oriented systems [67], such as `pythODE`.

`Solver` is divided into three main parts, which do not consist of any numerical code but rather coordinate the other parts of the program. The interface for `Solver` is shown in Listing 3.1 and includes the following methods.

- The `initialize` method runs prior to the first step of the IVP solver.
- The `step` method runs for the first step and every subsequent step of the solution of the IVP.
- The `finalize` method runs when the solver is finished solving the IVP.
- The `reset` method returns the solver to its initial state.
- The `run` method starts the integration after all objects have been initialized.

`Solver` also coordinates information that must be passed between different components of an IVP solver but is not supported using `Solution`. An example of where information must be shared between different components is when the error controller requires the order of convergence of the integration method. A more complex example is when a component wants to force a rejected timestep, for instance, in the case of an error. This is done by communicating directly with `Solver`. Ambiguity is avoided by developing standard nomenclature and interfaces for these types of data and actions.

Listing 3.1: The interface for `Solver` objects.

```
class Solver:
    def initialize(self, solver, solution):
        # code that runs before solving the IVP
        pass

    def step(self, solver, solution):
        # code that runs for each timestep
        pass

    def finalize(self, solver, solution):
        # code that runs after solving the IVP
        pass

    def reset(self):
        # reset the solver back to the initial state
        pass

    def run(self):
        # run the solver to start the timestepping
        pass
```

3.3.3 SolverModule objects

The primary location of the numerical code of an IVP solver in `pythODE` is within `SolverModule` objects. These objects interact with each other through the `Solution` object and are coordinated with the `Solver` object. `SolverModule` objects have the same `initialize`, `step`, and `finalize` methods from `Solver`, as shown in Listing 3.2. The particular method from a `SolverModule` object is run from the analogous methods in `Solver`. An example of a `Solver` object coordinating two `SolverModule` objects is shown in Figure 3.2.

The interface for a `SolverModule` is shown in Listing 3.2. An example of functionality appropriate for the `initialize` method is pre-processing applied to the initial values or writing the header of an output file. Examples of functionality appropriate for the `step` method are the integration itself, error control, and output to a file or data structure. Examples of functionality appropriate for the `finalize` method are post-processing of the final solution and closing output files.

In general, each of the methods in a `SolverModule` includes code to read from the `Solution` object at the beginning of its execution and to write to the `Solution` object when it is finished executing.

Listing 3.2: The interface for `SolverModule` objects.

```
class SolverModule:
    def initialize(self, solver, solution):
        # code that runs before solving the IVP
        # e.g., opening files for output, inputting data from a file, etc.
        pass

    def step(self, solver, solution):
        # code that runs for each step
        # e.g., integration, error control, etc.
        pass

    def finalize(self, solver, solution):
        # code that runs after solving the IVP
        # e.g., closing files, output final values, etc.
        pass
```

3.3.4 Statistics and error checking

Decorator patterns are used to dynamically and transparently attach additional responsibilities to objects without inheritance or code modification. Inheritance requires a new class for each combination of extensions and can become complex. Decorators add responsibilities by enclosing an object and performing operations before and after forwarding requests to the decorated object, therefore depending only on the interface rather than the implementation of the object. This allows

code reuse because functionality can be in a single place when it would not be otherwise possible when using behavioural subtyping [38, p.175].

Decorators are also the name of a feature in the core functionality of PYTHON that can be used to implement decorator patterns. In `pythODE`, decorators are used for tasks such as gathering statistics and error checking. As an example, an important metric to determine the relative performance of IVP methods is the total number of RHS evaluations during the solution. Unlike computational time, the number of RHS evaluations is independent of many low-level implementational details or the programming platform. IVP software often allows counting the number of RHS evaluations. The counters for RHS evaluations are typically scattered throughout the program; for instance in the RADAUP solver there are six calls to incrementing the global RHS evaluation counter. In `pythODE`, a decorator can be added to the RHS itself to count the number of evaluations without any code modification of the solver components themselves. An example of a decorator being used for this is shown in Listing 3.3.

Listing 3.3: Applying a decorator to count evaluations of an already existing function.

```
# the global variable for counting
count = 0

def count_rhs_evaluations(f):
    # add the decorator to the new function
    def count_f(f):
        return f(t,f)
    # this function is not seen externally
    def decorator(*args,**kwds):

        # operations before forwarding the arguments
        # i.e., counting the function calls with a global variable
        count += 1

        # call the decorated function
        result = f(*args,**kwds)

        # operations after forwarding the arguments

        # return the result from the decorated function
        return result
    # return the new function
    return decorator

# decorating the RHS
@count_rhs_evaluations
def f(t,y):
    # define the RHS here and return
    pass
```

Using decorators to implement this functionality means the numerical code is less cluttered and

the function counter only needs to be in one location. If modules are added or code is modified so RHS evaluations occur in additional locations, no additional development is required to record this statistic accurately. An example of where this may occur is that some dense output formulae require additional RHS evaluations for each evaluation of the solution at off-step times.

Another example of the usage of decorators is checking for exceptional floating point values such as *infinity* (Inf) or *not a number* (NaN). These can interfere with many floating point operations and need to be identified and handled. If they are not properly handled the solver may be forced to terminate when appropriate corrective action could be taken instead. Using decorators, the solver can perform and respond to these checks from only one location. Sometimes there is also problem-specific error checking that needs to be performed. For instance, some ARD equations should not produce negative values in the numerical solution because such values are non-physical and may result in instability. A decorator could check for negative values and take an appropriate action such as rejecting the current step, even if the local error (2.5) itself is within tolerances (2.38).

Other examples of where decorators could be used include counting accepted and rejected steps and recording the optimal parameters of heuristics that could have been used to avoid a rejected step. The eigenvalues of the Jacobian matrix could be determined at arbitrary solution or stage values in order to help diagnose problematic IVPs [61, p.224]. This is a computationally expensive statistic to record, but using decorators allows it to be added or removed easily.

3.3.5 Testing

Testing is a critical part of developing any software package. PSEs like `pythODE` are under constant development as new numerical methods are proposed and analyzed. Therefore verification must occur on a regular basis to ensure errors are not introduced. A standard method for testing IVPs is to compare the numerical solution from a particular software package to a *reference solution*, which is a numerical solution to a particular IVP that has been solved to high accuracy with proven numerical software. Sets of test problems and software for generating reference solutions can be found in [35, 50, 68]. The `pythODE` PSE has an automatic test suite to check against a large number of problems with reference solutions. This test suite is run periodically to find errors that may have been introduced, allowing both continued development and trust in the solver.

Another method of testing IVP software is to ensure the integration methods match the theoretical order of convergence from the asymptotic behaviour of (2.5). This is necessary because with a small enough stepsize, an “incorrect” method can still produce an accurate solution. An empirical method of calculating the order of convergence is to find the error with a constant-stepsize solver. The stepsize is then halved, and the integration is then repeated until the asymptotic behaviour can be observed. Testing the order of convergence can find errors in the implementation, such as incorrect coefficients in the Butcher tableau, that may be missed with accuracy tests alone.

The testing code includes methods to generate reference solutions and compare different parameters for solvers and IVPs. Typically a reference solution is found by solving the IVP with decreasing error tolerances (2.38) until successive solutions change by less than a desired amount. The solution with the smallest tolerances is then taken to be the reference solution. The code to compare sets of parameters includes methods for extracting the dependent variables for each experiment and comparing them against the independent variables. This allows the testing code to be used for computational experiments as well.

CHAPTER 4

INTEGRATING-FACTOR-BASED 2-ADDITIVE-RUNGE–KUTTA METHODS FOR ARD EQUATIONS

Optimal methods for solving the ODE systems (2.2a) from the semi-discretization of ARD equations (2.44) are dependent on the properties of the underlying ARD system. This can be illustrated by examining the error bound of a uniformly semi-discretized ARD equation, given by

$$\|\mathbf{u}_{\Delta\mathbf{x}}(t_n) - \mathbf{y}_n\| \leq \|\mathbf{u}_{\Delta\mathbf{x}}(t_n) - \mathbf{y}(t_n)\| + \|\mathbf{y}(t_n) - \mathbf{y}_n\| \leq C_{\Delta\mathbf{x}} |\Delta\mathbf{x}|^{p(\Delta\mathbf{x})} + C_{\Delta t} \Delta t^{p(\Delta t)}, \quad (4.1)$$

where $\mathbf{u}_{\Delta\mathbf{x}}(t_n)$ is the exact solution of the PDE (2.44) restricted to the grid used for semi-discretization, $\mathbf{y}(t_n)$ is the exact solution of the semi-discretized ARD system, \mathbf{y}_n is the numerical solution of the semi-discretized ARD system, $p(\Delta\mathbf{x})$ is the order of convergence in space with an associated constant $C_{\Delta\mathbf{x}}$, and $p(\Delta t)$ is the order of convergence in time with an associated constant $C_{\Delta t}$. The semi-discrete system forms a family of ODE systems parametrized by $\Delta\mathbf{x}$; therefore it requires additional analysis in comparison to the non-parametrized ODE system (2.2a). The overall order of convergence in time of the IVP (2.2a) resulting from semi-discretization of the ARD system as $\Delta t \rightarrow 0^+$ with $\Delta\mathbf{x}$ fixed is different from the case when $\Delta t \rightarrow 0$, $|\Delta\mathbf{x}| \rightarrow 0^+$ simultaneously because the constant $C_{\Delta t}$ in (4.1) is generally dependent on the choice of $\Delta\mathbf{x}$ [51, p.95].

The behaviour of the error bound (4.1) motivates the development of ODE solution methods based on knowledge of the underlying ARD equations (2.44) as well as the semi-discretization methods used for them. In this chapter we introduce IVP methods that are optimal for each of the three terms of a semi-discretized ARD equation,

$$\frac{d}{dt}\mathbf{y}(t) - \mathcal{A}(t, \mathbf{y}(t)) \cdot \mathbf{y}(t) = \mathcal{D}(t, \mathbf{y}(t)) \cdot \mathbf{y}(t) + \mathbf{r}(t, \mathbf{y}(t)), \quad (4.2)$$

$\mathcal{A}(t, \mathbf{y}(t)) : \mathbb{R} \times \mathbb{R}^m \rightarrow \mathbb{R}^{m \times m}$ is the advection operator, $\mathcal{D}(t, \mathbf{y}(t)) : \mathbb{R} \times \mathbb{R}^m \rightarrow \mathbb{R}^{m \times m}$ is the diffusion operator, and $\mathbf{r}(t, \mathbf{y}(t)) : \mathbb{R} \times \mathbb{R}^m \rightarrow \mathbb{R}^m$ is the reaction term composed of n_{grid} decoupled vector functions of size q each. The size of the ODE system is m where $m = q \times n_{\text{grid}}$, q is the number of species, and n_{grid} is the number of spatial grid points.

The dependence of the error terms of the semi-discretized system (4.1) on $\Delta\mathbf{x}$ leads to stepsize

restrictions based on the size of the spatial grid when using an explicit method to solve (4.2). For certain problems, the stepsize restrictions are not severe enough to require stiff methods for the semi-discretized system, but this is problem dependent [51, p.149].

ARD equations are often solved to only moderate accuracy in time because the primary source of error is due to the relatively coarse accuracy in space. Therefore higher-order ODE methods do not always give higher accuracy; i.e., the extra computational work to achieve high order does not give a more accurate solution to the underlying PDE. Higher-order ODE methods for semi-discretized PDEs are advantageous for transient phenomena and fluid mechanics applications [51, p.440]. Improvements in modelling physical phenomena by ARD systems also make finding high-accuracy solutions more important. The increase in computational resources due to Moore’s law makes high accuracy more practical and thus higher-order algorithms more important [45, 84].

4.1 Semi-Lagrangian exponential integrators for advection

A well-known condition that restricts Δt due to stability for explicit methods when solving the semi-discretized advection equation (2.41) is the *Courant–Friedrichs–Lewy* (CFL) condition [29, 30], with the one-dimensional constant-coefficient case given by

$$\nu = \frac{\Delta t |a|}{\Delta x} \leq C,$$

where ν is known as the *Courant number*, a is the advection coefficient, and C is a positive constant dependent on the particular advection problem but independent of Δt and Δx .

The *mathematical domain of dependence* of a solution point $\mathbf{u}(t, \mathbf{x})$ of a PDE is the set of all points (t, \mathbf{x}) such that $\mathbf{u}(t, \mathbf{x})$ is dependent on the data at those points. The *numerical domain of dependence* of a numerical solution point $\mathbf{u}_{\Delta \mathbf{x}}(t_n, \mathbf{x}_i)$ is the set of all points (t_j, \mathbf{x}_k) such that $\mathbf{u}_{\Delta \mathbf{x}}(t_n, \mathbf{x}_i)$ is dependent on the data at those points. The CFL condition reflects that the mathematical domain of dependence must lie within the numerical domain of dependence as a necessary condition for stability. Otherwise points of the numerical solution do not have the necessary knowledge of previous solution points for a stable solution [34, p.422; 51, p.102,149].

Methods such as finite differences that model advection by using a fixed grid to observe the flowing medium are known as *Eulerian methods*. Eulerian methods typically suffer from numerical issues such as artificial diffusion, artificial dispersion, and stepsize restrictions on explicit methods due to the CFL condition. *Lagrangian methods* model advection as fluid particles that are followed through space rather than being observed from a fixed grid as with Eulerian methods. They do not have stability restrictions, such as the CFL condition, when using explicit timestepping methods and have reduced artificial dispersion. A disadvantage of Lagrangian methods is that the grid typically becomes highly irregular during timestepping [32, p.936; 96, p.2207].

Semi-Lagrangian methods, also known as *Eulerian–Lagrangian methods*, observe the flowing medium from a Lagrangian perspective but model different fluid particles at each timestep. This approach achieves the stability of Lagrangian schemes but maintains the regular grid of Eulerian schemes [96, p.2207]. The first-order upwind scheme (2.47) is in fact equivalent to the simplest semi-Lagrangian methods in one dimension [96, p.2212]; this explains the excellent qualitative behaviour in comparison to the second-order central difference scheme as shown in [51, p.54].

Characteristics are curves along which the solution of a PDE reduces to the solution of an ODE. In the case of the advection equation (2.41) characteristics satisfy IVPs of the form

$$\frac{dX(\tau)}{d\tau} = \mathcal{A}(\tau, X(\tau)), \quad X(t_{n+1}) = \mathbf{x}_i, \quad (4.3)$$

where $X(\tau) : \mathbb{R} \rightarrow \mathbb{R}^q$ is the characteristic that maps spatial positions to time $\tau \in \mathbb{R}$ [22, p.141; 72, p.86]. Semi-Lagrangian methods follow characteristics from a grid point \mathbf{x}_i at t_{n+1} to a spatial location at t_n , generally an off-grid point. Interpolation between grid points is used to find the concentration at the spatial location for t_n corresponding to the characteristic [105, p.660]. This interpolated value is then used to find numerical solution for the grid point \mathbf{x}_i at t_{n+1} . Characteristic equations (4.3) do not generally have analytical solutions, especially for problems with multiple spatial dimensions and variable coefficients. Instead, characteristics are found by interpolation, e.g., using finite elements [8, 9, 77], or solved using IVP methods such as RK [6, 22, 103]. Semi-Lagrangian methods can nearly eliminate artificial dispersion [22, p.157] but exhibit artificial diffusion, have difficulty with boundary conditions, have a high computational cost relative to other methods, and do not conserve mass [6, 103, 105]. *Semi-Lagrangian localized adjoint methods* are an improvement of semi-Lagrangian methods that conserve mass and are able to handle general boundary conditions [36, 103]. Solving for each characteristic and performing the associated interpolation can be done independently; therefore semi-Lagrangian methods are suitable for parallel computing [66].

4.1.1 Exponential Lie group methods

Exponential methods for solving ODEs use an *exponential map*, such as the matrix exponential of the Jacobian (2.9), to advance the integration. Certain exponential integrators are formally explicit but avoid stepsize restrictions due to stability when solving stiff IVPs because exponential maps can follow rapidly changing solution curves better than linear approximations can [76]. In practice, although exponential methods are not a new idea [62, 78], they are not mainstream methods because they have been considered to be less efficient than IRK and BDF methods due to the high cost of computing the required functions such as matrix exponentials. The development of lower-cost algorithms to compute, in particular, matrix exponentials, such as the *Krylov subspace approximations*

described in Section 4.1.4 below, has brought renewed interest to exponential methods [71, p.1].

A family of exponential methods known as *exponential Lie group methods* can preserve the qualitative behaviour of an ODE system, such as conserving energy or angular momentum when simulating rigid bodies or other problems from classical mechanics [24; 42, p.127,169; 74]. A Lie group G is a smooth differentiable manifold \mathcal{M} that is closed under a group operation $G \times G \rightarrow G$. A *Lie algebra* \mathfrak{g} is the tangent space at the identity of G and is closed under an operation known as the *Lie bracket* $[\cdot, \cdot] : \mathfrak{g} \times \mathfrak{g} \rightarrow \mathfrak{g}$. Lie groups that are known as *matrix Lie groups*, where the group elements are finite-dimensional vector spaces, are a common class. The Lie algebra of a matrix Lie group is a vector space, with a Lie bracket that is the matrix commutator $[A, B] := AB - BA$. An exponential map $\exp : \mathfrak{g} \rightarrow G$, that is a matrix exponential in the case of matrix Lie groups, takes the members of the Lie algebra to the members of the Lie group [76]. Matrix Lie group differential equations that remain on \mathcal{M} have the form

$$\frac{d}{dt} \mathbf{Y}(t) = \mathbf{J}(\mathbf{Y}(t)) \cdot \mathbf{Y}(t), \quad (4.4a)$$

$$\mathbf{Y}(0) = \mathbf{Y}_0, \quad (4.4b)$$

where $\mathbf{Y}(t) \in G$ and $\mathbf{J}(\mathbf{Y}(t)) \in \mathfrak{g}$ are locally Lipschitz continuous to ensure existence and uniqueness [42, p.118]. In the context of differential equations defined on Lie groups, G is the space of solutions and the group operation \times maps the whole solution space to itself [52, p.15].

One of the simplest equations containing a nonlinear advection term is the Burgers equation. When coupled with diffusion, it is used to simulate fluid flow containing *shocks*, which are sharp transitions in a fluid flow. The one-dimensional Burgers equation with diffusion is given by

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \epsilon \frac{\partial^2 u}{\partial x^2}, \quad (4.5)$$

where $u = u(t, x) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ is a concentration and $\epsilon \in \mathbb{R}$ is the diffusion coefficient. The Lie group operation for the Burgers equation can be defined from the following transformations:

$$\text{Spatial translation } \mathbf{G}_1 : (x, t, u) \longrightarrow (x + \kappa_1, t, u), \quad (4.6a)$$

$$\text{Time translation } \mathbf{G}_2 : (x, t, u) \longrightarrow (x, t + \kappa_2, u), \quad (4.6b)$$

$$\text{Projection } \mathbf{G}_3 : (x, t, u) \longrightarrow \left(\frac{x}{1 - \kappa_3 t}, \frac{t}{1 - \kappa_3 t}, (1 - \kappa_3 t)u + \kappa_3 x \right), \quad (4.6c)$$

$$\text{Scale transformation } \mathbf{G}_4 : (x, t, u) \longrightarrow (xe^{\kappa_4}, te^{2\kappa_4}, ue^{-\kappa_4}), \quad (4.6d)$$

$$\text{Galilean boost } \mathbf{G}_5 : (x, t, u) \longrightarrow (x + \kappa_5 t, t, u + \kappa_5), \quad (4.6e)$$

where $\kappa_1, \kappa_2, \dots, \kappa_5$ are parameters for the transformations and $\mathcal{M} \sim \mathbb{R}^3$. Most classical numerical schemes, which include Eulerian methods, respect (4.6a), (4.6b), and (4.6d). In general (4.6c) and

(4.6e) are not respected [25]. Many aspects of qualitative behaviour are generally not preserved by the IVP methods or the finite difference methods of Chapter 2 because the only possible action of these methods is to evolve using linear approximations, which generally do not reflect these types of properties of the exact solution [60].

This motivates a search for numerical methods that better preserve qualitative behaviour such as the Lie group structure. Numerical methods for (4.4) can preserve qualitative behaviour by ensuring the numerical solution stays on \mathcal{M} . This is usually accomplished by solving (4.4) in the context of \mathfrak{g} , then mapping the result back to G by using the exponential map in the solution formula [20]. For many practical problems such as the Burgers equation (4.5), the manifold is $\mathcal{M} \sim \mathbb{R}^m$, but Lie group methods are still useful because they can evolve in the same qualitative manner as the exact solution [60]. The most basic exponential Lie group method for (4.4), considered analogous to FE, is given by

$$\mathbf{Y}_{n+1} = \exp(\mathbf{J}(\mathbf{Y}_n)) \cdot \mathbf{Y}_n, \quad (4.7)$$

where $\mathbf{Y}_{n+1}, \mathbf{Y}_n \in G$ and $\mathbf{J}(\mathbf{y}_n) \in \mathfrak{g}$. In [18, 22] it is shown that by taking the exact flow of the advection term (2.41) from \mathbf{Y}_n using characteristics (4.3) and solving with (4.7), a semi-Lagrangian integrator is obtained. However, although there does not yet appear to be a rigorous justification for the connection to Lie group theory, published results show excellent qualitative behaviour of high-order exponential Lie group methods that use the exact flow of the advection term (2.41) of the underlying PDE compared to the corresponding Eulerian methods, which use the matrix exponential of $\mathcal{A}(t, \mathbf{y}(t))$ from (4.2), in [18, 22].

4.1.2 Commutator-free Lie group exponential methods

Higher-order exponential Lie group methods typically require a relatively large number of exponential calculations, e.g., fifteen exponentials in the case of a fourth-order *Crouch–Grossman method*, each taking approximately $25m^3$ floating-point operations when solving an ODE on a matrix Lie group and the exponential calculation used is a matrix exponential [24, 31]. The formula for explicit Crouch–Grossman methods for ODEs with the form (4.4) is given by

$$\begin{aligned} \mathbf{Y}_i &= \exp(\Delta t_n a_{i,i-1} \mathbf{K}_i) \cdots \exp(\Delta t_n a_{i,2} \mathbf{K}_i) \cdot \exp(\Delta t_n a_{i,1} \mathbf{K}_i) \cdot \mathbf{Y}_n, \quad i = 1, 2, \dots, s, \\ \mathbf{K}_i &= \mathbf{J}(\mathbf{Y}_i), \quad i = 1, 2, \dots, s, \\ \mathbf{Y}_{n+1} &= \exp(\Delta t_n b_s \mathbf{K}_s) \cdots \exp(\Delta t_n b_2 \mathbf{K}_2) \cdot \exp(\Delta t_n b_1 \mathbf{K}_1) \cdot \mathbf{Y}_n, \end{aligned}$$

where $a_{i,j}$ and b_i correspond to the \mathbf{A} and \mathbf{b} values of a classic RK Butcher tableau (2.15) respectively, and \mathbf{Y}_i is the solution at stage \mathbf{K}_i [24]. To obtain an order of convergence greater

than three, additional order conditions are required for Crouch–Grossman methods in comparison to the classic RK methods [24, 31, 76]. To reduce the number of matrix exponentials required, *Munthe-Kaas methods* use commutators, each taking approximately $4m^3$ floating-point operations when solving an ODE on a matrix Lie Group and the commutator operation used is a matrix commutator [24, 73]. The formula for explicit Munthe-Kaas methods for ODEs with the form (4.4) is given by

$$\begin{aligned} \mathbf{Y}_i &= \exp(\tilde{\mathbf{Y}}_i) \cdot \mathbf{Y}_n, \quad \tilde{\mathbf{Y}}_i = \Delta t_n \sum_{j=1}^{i-1} a_{i,j} \mathbf{K}_j, \quad i = 1, 2, \dots, s, \\ \mathbf{K}_i &= \pi(\tilde{\mathbf{K}}_i) = \tilde{\mathbf{K}}_i + \pi_1[\tilde{\mathbf{Y}}_i, \tilde{\mathbf{K}}_i] + \pi_2[\tilde{\mathbf{Y}}_i, [\tilde{\mathbf{Y}}_i, \tilde{\mathbf{K}}_i]] + \dots, \quad \tilde{\mathbf{K}}_i = \sum_{j=1}^{i-1} \mathbf{J}(\mathbf{Y}_j), \quad i = 1, 2, \dots, s, \\ \mathbf{Y}_{n+1} &= \exp\left(\Delta t_n \sum_{i=1}^s b_i \mathbf{K}_s\right) \cdot \mathbf{Y}_n, \end{aligned}$$

where $a_{i,j}$ and b_i correspond to the \mathbf{A} and \mathbf{b} values of a classic RK Butcher tableau (2.15) respectively, \mathbf{Y}_i is the solution at stage \mathbf{K}_i , and $\pi(\mathbf{Z})$ is a sufficiently high-order polynomial approximation to $\mathbf{Z}/(\exp(\mathbf{Z}) - 1)$ with coefficients π_1, π_2, \dots [24]. If $\pi(\mathbf{Z})$ is a sufficiently high-order approximation, then explicit Munthe-Kaas methods will have the same order of convergence as the classic RK method with the same \mathbf{A} and \mathbf{b} coefficients, with no additional order conditions [24, 73]. Lie group methods that use commutators have stepsize restrictions due to stability when solving stiff differential equations [24]. This motivates the search for *commutator-free* Lie group methods that reduce the numbers of exponentials compared to Crouch–Grossman methods or similar methods [24, 75].

Commutator-free exponential Runge–Kutta (CFERK) methods, which can be configured to use a minimal number of exponentials, take the form

$$\mathbf{Y}_i = \exp\left(\Delta t_n \sum_{j=1}^{i-1} \alpha_{i,j}^{[J]} \mathbf{K}_j\right) \cdot \dots \cdot \exp\left(\Delta t_n \sum_{j=1}^{i-1} \alpha_{i,j}^{[2]} \mathbf{K}_j\right) \cdot \exp\left(\Delta t_n \sum_{j=1}^{i-1} \alpha_{i,j}^{[1]} \mathbf{K}_j\right) \cdot \mathbf{Y}_n, \quad (4.8a)$$

$$\mathbf{K}_i = \mathbf{J}(\mathbf{Y}_i), \quad i = 1, 2, \dots, s, \quad (4.8b)$$

$$\mathbf{Y}_{n+1} = \exp\left(\Delta t_n \sum_{i=1}^s \beta_i^{[K]} \mathbf{K}_i\right) \cdot \dots \cdot \exp\left(\Delta t_n \sum_{i=1}^s \beta_i^{[2]} \mathbf{K}_i\right) \cdot \exp\left(\Delta t_n \sum_{i=1}^s \beta_i^{[1]} \mathbf{K}_i\right) \cdot \mathbf{Y}_n, \quad (4.8c)$$

where $\sum_{k=1}^J \alpha_{i,j}^{[k]} = a_{i,j}$ and $\sum_{j=1}^K \beta_i^{[j]} = b_i$ correspond to the \mathbf{A} and \mathbf{b} values of the classic RK Butcher tableau (2.15) respectively, and \mathbf{Y}_i is the solution at stage \mathbf{K}_i [18, p.9; 22, p.141; 24, 75]. The Crouch–Grossman methods from are a special case of (4.8) with $J = s$ and $\alpha_{i,j}^{[k]} = \delta_{jk} a_{ij}$ where a_{ij} are the \mathbf{A} values for Crouch–Grossman methods and δ_{jk} is the Kronecker delta [24]. Crouch–Grossman, Munthe-Kaas, and CFERK methods in principle could use an implicit Butcher tableau, which would require the solution of systems of implicit equations to find the stage values [24]. It can easily be seen that the exponential methods described in this section can be reduced to classic

RK methods if the matrix exponential is approximated by the translation $\exp(t\mathbf{Y})\mathbf{z} \approx t\mathbf{Y} + \mathbf{z}$ [75].

4.1.3 Order conditions for CFERK methods

The classic RK order conditions from Section 2.4.1 form a subset of the order conditions for CFERK methods. Additional order conditions arise because of more method coefficients, and when finding higher derivatives of (4.4), the evaluation of elementary differentials (2.23) involves matrix multiplication, which is not commutative. Evaluating higher derivatives involves more Lie group theory than that which is presented here, but it can be found in [19, 22, 24, 75, 76].

Similar to the classic RK methods, visualizing the order conditions of CFERK methods can be accomplished using rooted trees. This is done by visualizing methods of order p using rooted trees of order $p+1$, but there is no precise analogy with the use of rooted trees to derive the methods from Chapter 2. In addition, the rooted trees for CFERK are known as *ordered*; i.e., the specific branch to which a subtree is attached while constructing the rooted tree is important [75, 76]. For example, two rooted trees that correspond to different elementary differentials for CFERK methods, but not to classic RK methods, are given by

$$\begin{array}{c} \bullet \\ | \\ \bullet \\ / \quad \backslash \\ \bullet \quad \bullet \\ | \quad | \\ i \quad i \end{array}, \quad \begin{array}{c} \bullet \\ | \\ \bullet \\ | \quad \backslash \\ \bullet \quad \bullet \\ | \quad | \\ i \quad i \end{array}, \quad (4.9)$$

where the ordering is because the subtree $\begin{array}{c} \bullet \\ | \\ \bullet \\ | \quad \backslash \\ \bullet \quad \bullet \\ | \quad | \\ i \quad i \end{array}$ that terminates with the vertex labelled h_2 can be attached to either of the vertices labelled with h_1 or h_3 on the subtree $\begin{array}{c} \bullet \\ / \quad \backslash \\ \bullet \quad \bullet \\ | \quad | \\ i \quad i \end{array}$.

There are no additional order conditions for second-order CFERK methods (4.8) compared to classic RK methods, and thus any second-order classic RK Butcher tableau (2.15) can be used. Beginning with third-order CFERK methods, additional order conditions and at least one additional exponential evaluation are required. The augmented Butcher tableau for a third-order explicit CFERK method is given by

$$\begin{array}{c|ccc|cc} c_1 & 0 & 0 & \cdots & 0 & 0 \\ c_2 & a_{2,1} & 0 & \cdots & 0 & 0 \\ c_3 & a_{3,1} & a_{3,2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & 0 \\ \hline & \beta_1^{[1]} & \beta_2^{[1]} & \cdots & \beta_{s-1}^{[1]} & \beta_s^{[1]} \\ & \beta_1^{[2]} & \beta_2^{[2]} & \cdots & \beta_{s-1}^{[2]} & \beta_s^{[2]} \end{array} := \quad (4.10)$$

where the parameters from (4.8) are $J = 1, K = 2$. The additional order condition is given by

$$\sum_{i=1}^s \beta_i^{[1]} c_i + \frac{1}{2} \beta_i^{[2]} = \frac{1}{3}, \quad (4.11)$$

that is derived from the two different ordered trees given in (4.9) [24]. To obtain fourth order, at least one additional exponential is required over third order [24], the augmented Butcher tableau for a fourth-order CFERK method is given by

$$\frac{\mathbf{c}}{\mathbf{b}^T} \left| \begin{array}{c} \mathbf{A} \\ \mathbf{b}^T \end{array} \right| := \begin{array}{c|cccc|cccc} c_1 & 0 & 0 & \cdots & 0 & 0 & c_1 & 0 & 0 & \cdots & 0 & 0 \\ c_2 & a_{2,1}^{[1]} & 0 & \cdots & 0 & 0 & c_2 & a_{2,1}^{[2]} & 0 & \cdots & 0 & 0 \\ c_3 & a_{3,1}^{[1]} & a_{3,2}^{[1]} & \cdots & 0 & 0 & c_3 & a_{3,1}^{[2]} & a_{3,2}^{[2]} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_s & a_{s,1}^{[1]} & a_{s,2}^{[1]} & \cdots & a_{s,s-1}^{[1]} & 0 & c_s & a_{s,1}^{[2]} & a_{s,2}^{[2]} & \cdots & a_{s,s-1}^{[2]} & 0 \\ \hline & \beta_1^{[1]} & \beta_2^{[1]} & \cdots & \beta_{s-1}^{[1]} & \beta_s^{[1]} & & \beta_1^{[1]} & \beta_2^{[1]} & \cdots & \beta_{s-1}^{[1]} & \beta_s^{[1]} \\ & \beta_1^{[2]} & \beta_2^{[2]} & \cdots & \beta_{s-1}^{[2]} & \beta_s^{[2]} & & \beta_1^{[2]} & \beta_2^{[2]} & \cdots & \beta_{s-1}^{[2]} & \beta_s^{[2]} \end{array} \quad (4.12)$$

where the parameters from (4.8) are $J = 2, K = 2$. It appears (4.12) requires s additional exponentials in comparison to a classic non-augmented Butcher tableau; however practical methods are constructed so the exponentials are reused wherever possible. Four additional order conditions are given by

$$\sum_{i=1}^s \frac{1}{6} \beta_i^{[1]} c_i + \frac{1}{18} \beta_i^{[2]} = \frac{1}{24}, \quad (4.13a)$$

$$\sum_{i=1}^s \frac{1}{4} \beta_i^{[1]} c_i^2 + \frac{1}{12} \beta_i^{[2]} = \frac{1}{24}, \quad (4.13b)$$

$$\sum_{i=1}^s \frac{1}{2} \beta_i^{[1]} a_{ij} c_i + \frac{1}{12} \beta_i^{[2]} = \frac{1}{24}, \quad (4.13c)$$

which result from different orderings of the subtrees for the set of fifth-order rooted trees. A further order condition

$$\frac{1}{2} \left(\sum_{i,j=1}^2 b_i c_i \alpha_{i,j}^{[1]} c_j + \sum_{i,j,k=1}^s b_i a_{i,j} c_j \alpha_{j,k}^{[2]} \right) = \frac{1}{24}, \quad (4.13d)$$

is required because of the coupling between the two tableaux of $\alpha_{i,j}^{[k]}$ for this method [24, 76].

4.1.4 Krylov subspace approximations to the matrix exponential

Krylov subspace approximations to the matrix exponential are *iterative methods* that approximate the action of the matrix exponential as

$$\exp(t\mathbf{A}) \cdot \mathbf{v} \approx p_{\eta-1}(t \cdot \mathbf{A}) \cdot \mathbf{v}, \quad (4.14)$$

where $t \in \mathbb{R}$, $\mathbf{A} \in \mathbb{R}^{m \times m}$, $\mathbf{v} \in \mathbb{R}^m$, and $p_{\eta-1}(t \cdot \mathbf{A})$ is a matrix polynomial of degree $\eta - 1$. The approximation (4.14) is an element of the Krylov subspace K_η of dimension η [83] defined by

$$K_\eta := \text{span}(\mathbf{v}, \mathbf{A} \cdot \mathbf{v}, \mathbf{A}^2 \cdot \mathbf{v}, \dots, \mathbf{A}^{\eta-1} \cdot \mathbf{v}).$$

Krylov subspace approximations of a matrix function $\rho(\mathbf{A})$ use successive evaluations of $\rho(\cdot)$ on smaller matrices than \mathbf{A} [83]. The iterations resulting from Krylov subspace approximations exhibit *superlinear convergence* under appropriate conditions and have been shown for some stiff IVPs to be a viable alternative to solving the same IVP with IRK methods that use Newton's method to solve the systems of simultaneous nonlinear equations at each step [47, 48].

The stopping criterion for the iterations of the Krylov subspace approximations is given by $\|\mathbf{d}\|_{\text{tol}} < 1$, with the same norm used by the integrator

$$\|\mathbf{d}\|_{\text{tol}} = \sqrt{\frac{1}{m} \sum_{i=1}^m (d_i / \tau_{n,i})^2},$$

where \mathbf{d} is an estimation of the residual internal to the Krylov approximation method and $\tau_{n,i}$ are the tolerances for the local error estimate given in (2.38). This stopping criterion avoids excessive computational cost for approximating the action of the matrix exponential. More details on the algorithm for Krylov subspace approximations can be found in [47].

4.1.5 Error control for CFERK methods

In [24], CFERK methods are used to solve a PDE derived from heat conduction using the step-doubling error estimation from Section 2.5.1 and the stepsize control from Section 2.5.3. We are unaware of embedded error estimation being used for CFERK methods or how many additional exponential or RHS evaluations would be required. The MATLAB software package **EXP4** described in [47] uses embedded error estimation with a fourth-order exponential main method and a third-order exponential embedded method; however it is not a Lie group method, despite its use of the matrix exponential of the Jacobian. The stepsize controller in **EXP4** uses the predictive stepsize controller described in Section 2.5.3. Krylov subspace methods, for example, use a variable number of iterations to evaluate the matrix exponential. Therefore the cost and accuracy of evaluating the

matrix exponential should be taken into account when selecting stepsize for error control. EXP4 is an example of a software package that uses this type of stepsize selection strategy [47].

4.2 Stabilized ERK methods for the diffusion term

The semi-discretized diffusion operator $\mathcal{D}(t, \mathbf{y}(t))$ of an ARD equation (4.2) typically gives a Jacobian (2.9) that is negative semi-definite and symmetric; therefore the eigenvalues are real and non-positive [51, p.64]. ODEs with these types of eigenvalues normally require a stiff solver. *Stabilized* ERK methods have an extended stability region to accommodate the eigenvalues from an ODE system that is moderately stiff. Stabilized ERK methods can achieve better computational efficiency on some large systems compared to an IRK solver such as RADAU5 [3]. Their implementation is simplified over an IRK method because it is not necessary to evaluate the Jacobian or solve systems of equations when using an ERK method. The large stability regions are achieved by additional stages relative to the number required to achieve the order of convergence of the method. Stabilized ERK methods designed for the diffusion term have stability regions that contain a relatively large interval of the negative real axis [94, p.317; 101, p.2].

When solving semi-discretized diffusion problems with explicit methods, there are stepsize restrictions due to the CFL condition. Diffusion problems have a mathematical domain of dependence encompassing the whole problem domain; in the one-dimensional constant-coefficient case this leads to stepsize restrictions of the form

$$\nu = \frac{\Delta t |\epsilon|}{(\Delta x)^2} \leq C, \quad (4.15)$$

where ϵ is the diffusion coefficient, C is a positive constant that is problem dependent but independent of Δt and Δx . Conditions such as (4.15) can lead to complications in advection-diffusion equations because each term may impose significantly different timestep restrictions.

Runge–Kutta–Chebyshev (RKC) methods, named because the stability polynomial of the prototypical method is based on *Chebyshev polynomials*, are an example of stabilized ERK methods. The Chebyshev polynomials of the first kind are given recursively by

$$\begin{aligned} T_0(z) &= 1, \\ T_1(z) &= z, \\ T_j(z) &= 2zT_{j-1}(z) - T_{j-2}(z), \\ T_j(z) &= \cos(j \arccos(z)), \end{aligned}$$

where $T_j(z)$ is the Chebyshev polynomial of the first kind of degree j . Optimal first-order RKC

methods use a shifted Chebyshev polynomial as the stability polynomial, which is given by

$$R_s(z) = T_s(1 + z/s^2), \quad (4.16)$$

where the method has a stability interval of $l_s = [-0.821842s^2, 0]$ along the negative real axis when using s stages.

The stability polynomials of RKC methods with maximal stability interval l_s have a stability region of width zero at some points within l_s . The stability polynomials of practical methods are *dampened*; i.e., a dampening parameter η is introduced such that $|R_s(z)| \leq \eta \leq 1$ instead of $|R_s(z)| \leq 1$ in the case of the undampened stability polynomials. This reduces l_s slightly but ensures eigenvalues that lie a relatively small distance from the negative real axis fall within the stability region [44]. Stabilized ERK methods such as RKC methods must ensure the large number of stages do not excessively accumulate round-off error [94, p.317]. It can be seen from Section 2.7 that the dimension of the ODE system (4.2) can become extremely large in the case of fine meshes, multiple spatial dimensions, and many species; therefore another requirement for practical stabilized ERK methods is to use a minimal amount of storage.

The methods from the FORTRAN codes ROCK2 [4] and ROCK4 [2] have an order of convergence of two and four, respectively, and possess embedded methods. The stability polynomials of these methods approximate closely the optimal stability polynomials with the maximum possible l_s . The stability polynomials do not have an analytical form [2, 4] but can be stated as

$$R_s(z) = w_p(z)P_{s-p}(z), \quad (4.17)$$

where $w_p(z)$ is a polynomial of degree p having only complex zeros, $P_{s-p}(z)$ is a polynomial of degree $s - p$, and p is the order of convergence of the resulting method. The stability polynomials are *orthogonal polynomials* satisfying

$$\int_0^{l_s} R_i(z)R_j(z)W(z)dz = 0,$$

where $R_i(z), R_j(z)$ are two orthogonal stability polynomials and $W(z) = \frac{w_p(z)^2}{\sqrt{1-z^2}}$ is a weighting function. A well-known result is that orthogonal polynomials possess a three-term recursion relation [4, p.13]; the recursion for (4.17) is

$$P_i(z) = (\mu_i z - \nu_i)P_{i-1}(z) - \kappa_i P_{i-2}(z), \quad (4.18)$$

where μ_i, ν_i, κ_i are recursion parameters [2]. The recursion parameters of (4.18) are found by solving a system of nonlinear equations as described in [1, 2, 4]. These types of stability polynomials exist

for an arbitrarily large even order of convergence [1]. The three-term recursion leads to a recursive formula for methods based on stability polynomials of (4.17) with $p = 2$ given by

$$\mathbf{Y}_0 = \mathbf{y}_n, \tag{4.19a}$$

$$\mathbf{Y}_1 = \mathbf{Y}_0 + \Delta t \mu_1 \mathbf{f}(\mathbf{Y}_0), \tag{4.19b}$$

$$\mathbf{Y}_j = \Delta t \mu_j \mathbf{f}(\mathbf{Y}_{j-1}) - \nu_j \mathbf{Y}_{j-1} - \kappa_j \mathbf{Y}_{j-2}, \quad j = 2, 3, \dots, s-2, \tag{4.19c}$$

and is followed by a finishing procedure derived from $w_p(z)$ for the final values given by

$$\mathbf{Y}_{s-1} = \mathbf{Y}_{s-2} + \Delta t \sigma \mathbf{f}(\mathbf{Y}_{s-2}), \tag{4.20a}$$

$$\mathbf{Y}_s^* = \mathbf{Y}_{s-1} + \Delta t \sigma \mathbf{f}(\mathbf{Y}_{s-1}), \tag{4.20b}$$

$$\mathbf{Y}_s = \mathbf{Y}_s^* - \Delta t \sigma \left(1 - \frac{\tau}{\sigma^2}\right) (\mathbf{f}(\mathbf{Y}_{s-1}) - \mathbf{f}(\mathbf{Y}_{s-2})), \tag{4.20c}$$

where \mathbf{Y}_s is the result \mathbf{y}_{n+1} and \mathbf{Y}_s^* is a first-order embedded method that can be used to produce an estimate for $\hat{\mathbf{y}}_{n+1}$ in the error controllers described in Section 2.5. A damping factor represented by $\eta = 0.95$ shortens the stability interval only marginally from the near optimal $l_s \approx 0.82s^2$ to $l_s \approx 0.81s^2$ in the case of $p = 2$ and a moderately sized s [4]. The stability region for ROCK2 with 10 stages is shown in Figure 4.1.

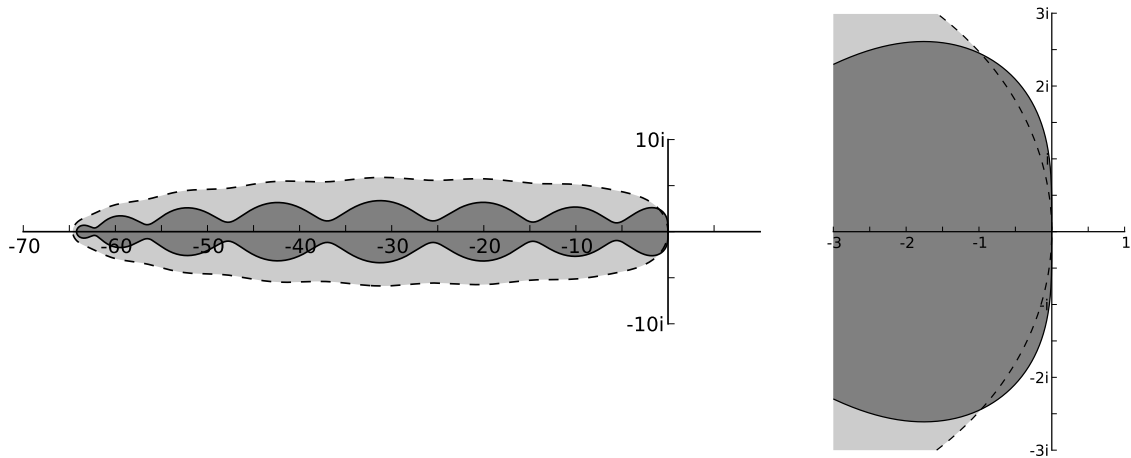


Figure 4.1: Stability region of 10-stage ROCK2 (dark shading) and the embedded method (light shading and dashed boundary).

In [2] the coefficients for ROCK4 with $p = 4$ and a different finishing procedure are derived by *composing* RK methods with stability polynomials of P_{s-4} and $w_4(z)$ respectively. The stability region for ROCK4 is shown in Figure 4.2. Methods that use a three-term recursion (4.19) can be converted to a conventional Butcher tableau (2.15) for the purposes of analysis; in practical

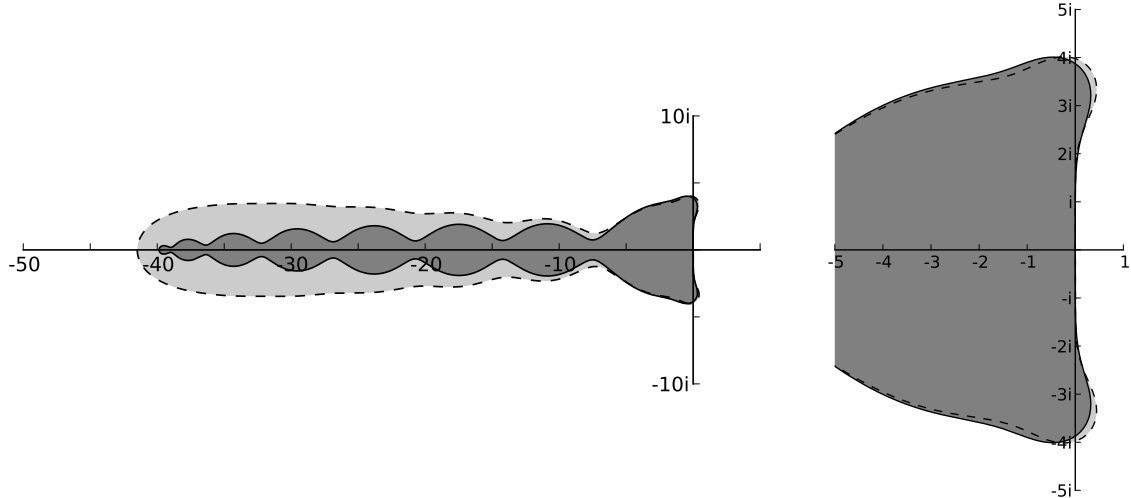


Figure 4.2: Stability region of 14-stage ROCK4 (dark shading) and the embedded method (light shading and dashed boundary).

implementations these representations have increased computational and storage requirements.

4.3 Integrating-factor-based 2-ARK methods

In this section the general form of integrating-factor-based 2-ARK (IF-2-ARK) methods based on those from [18, 22] are derived. These new methods use a CFERK method for advection and RK methods for reaction and diffusion.

4.3.1 Operator integrating factor splitting

ARK methods are a type of splitting method where the vector field of an ODE system is additively decomposed, then each part of the vector field is integrated separately. These individual solutions are then combined to give an integration of the complete system. Splitting is mainly used to improve the feasibility of integrating a system compared to a single integration method for the entire system. For example, this situation occurs in the context of semi-discretized ARD equations (4.2) if the reaction term is extremely stiff but the diffusion term is only modestly stiff. A single stiff integration method applied to (4.2) generally leads to extremely large systems of simultaneous equations that must be solved using Newton’s method. The cost of solving large systems of equations is extremely high; however, using a splitting method to solve the reaction terms separately avoids spatial coupling and results in many relatively small systems of equations that are generally not as costly to solve [69, 89, 101].

Operator splitting is a type of splitting where the operators applied to the vector field, such as the advection, reaction, and diffusion operators from (4.2), are isolated so they can be integrated

in time separately [51, 69]. *Operator integrating factor* (OIF) splitting [65] is a type of operator splitting that is used to generate numerical methods for systems of ODEs by finding an *integrating factor*, a function that is chosen to facilitate the solution of an ODE [5]. In [18, 22] OIF splitting, using integrating factors based on $\mathcal{A}(t, \mathbf{y}(t))$ from (4.2), generate CFERK-based matrix-matrix multiplications, and matrix-vector multiplications. OIF splitting reformulates (4.2) in the form

$$\mathbf{y}(t) = \mathcal{W}(t) \cdot \mathbf{z}(t), \quad (4.21a)$$

$$\frac{d}{dt} \mathcal{W}(t) = \mathcal{A}(t, \mathcal{W}(t) \cdot \mathbf{z}(t)) \cdot \mathcal{W}(t), \quad (4.21b)$$

$$\frac{d}{dt} \mathbf{z}(t) = \mathcal{W}^{-1}(t) \cdot \mathcal{D}(t, \mathcal{W}(t) \cdot \mathbf{z}(t)) \cdot \mathcal{W}(t) \cdot \mathbf{z}(t) + \mathcal{W}^{-1}(t) \cdot \mathbf{r}(t, \mathcal{W}(t) \cdot \mathbf{z}(t)), \quad (4.21c)$$

$$\mathcal{W}(t_0) = \mathbb{I}, \quad \mathbf{z}(t_0) = \mathbf{y}_0, \quad (4.21d)$$

where $\mathcal{W}(t) \in \mathbb{R}^{m \times m}$ is the integrating factor, $\mathbf{z}(t) \in \mathbb{R}^m$ is an auxiliary variable, and \mathbb{I} is the identity matrix [18, 22, 65]. A full derivation of (4.21) is given in Appendix C. The splitting (4.21) effectively decouples the advection term from the reaction and diffusion terms of (4.2). Numerical schemes known as *transport-diffusion* schemes [77], where advection of the flowing medium is modeled before applying diffusion, can be viewed in the context of OIF splitting [65]. The CFERK-based methods presented in this section, as well as those in [18, 19, 22], can be viewed as higher-order generalizations of the transport-diffusion algorithm.

4.4 Format of IF-2-ARK methods

The integrating factor $\mathcal{W}(t)$ from (4.21) corresponds to $\exp(\mathcal{A}(\mathbf{y}(t)))$ when $\mathcal{A}(\cdot)$ is not explicitly dependent on time [65], thus providing motivation to use a CFERK method to integrate $\mathcal{W}(t)$. The variable $\mathbf{z}(t)$, corresponding to a reaction-diffusion problem, is integrated with a 2-ARK method such as the IMEX methods (2.21) in [7, 55, 101]. Timestep restrictions from the diffusion term (4.15) are still present in advection-dominated problems where $\|\mathcal{D}(\mathbf{y}(t))\| \ll \|\mathcal{A}(\mathbf{y}(t))\|$ [22]. Using a stabilized ERK method from Section 4.2 for a moderately stiff diffusion term requires only that the solution of small systems of non-spatially coupled equations for the reaction term.

The formula for IF-2-ARK methods up to fourth order used to solve (4.2), when the coefficients are not explicitly dependent on t , is derived in Appendix C by eliminating the auxiliary variables

and given by

$$\rho_i = \exp \left(\Delta t_n \sum_{j=1}^{i-1} \alpha_{i,j}^{[2]} \mathcal{A}(\mathbf{Y}_j) \right) \cdot \exp \left(\Delta t_n \sum_{j=1}^{i-1} \alpha_{i,j}^{[1]} \mathcal{A}(\mathbf{Y}_j) \right), \quad i = 1, 2, \dots, s, \quad (4.22a)$$

$$\rho_{i,j} := \rho_i \cdot \rho_j^{-1}, \quad (4.22b)$$

$$\mathbf{Y}_i = \rho_i \cdot \mathbf{y}_n + \Delta t_n \sum_{j=1}^s (\mathbf{A}_{\mathcal{D}})_{i,j} \rho_{i,j} \cdot \mathcal{D}(\mathbf{Y}_j) \cdot \mathbf{Y}_j + \Delta t_n \sum_{j=1}^s (\mathbf{A}_{\mathbf{r}})_{i,j} \rho_{i,j} \cdot \mathbf{r}(\mathbf{Y}_j), \quad i = 1, 2, \dots, s, \quad (4.22c)$$

$$\rho_{n+1} = \exp \left(\Delta t_n \sum_{i=1}^s \beta_i^{[2]} \mathcal{A}(\mathbf{Y}_i) \right) \cdot \exp \left(\Delta t_n \sum_{i=1}^s \beta_i^{[1]} \mathcal{A}(\mathbf{Y}_i) \right), \quad (4.22d)$$

$$\rho_{n+1,i} := \rho_{n+1} \cdot \rho_i^{-1}, \quad (4.22e)$$

$$\mathbf{y}_{n+1} = \rho_{n+1} \cdot \mathbf{y}_n + \Delta t_n \sum_{i=1}^s (\mathbf{b}_{\mathcal{D}})_i \rho_{n+1,i} \cdot \mathcal{D}(t, \mathbf{Y}_i) \cdot \mathbf{Y}_i + \Delta t_n \sum_{i=1}^s (\mathbf{b}_{\mathbf{r}})_i \rho_{n+1,i} \cdot \mathbf{r}(\mathbf{Y}_i), \quad (4.22f)$$

where s is the number of stages, $(\mathbf{A}_{\mathcal{A}})_{i,j} = \alpha_{i,j}^{[1]} + \alpha_{i,j}^{[2]}$, $(\mathbf{b}_{\mathcal{A}})_i = \beta_i^{[1]} + \beta_i^{[2]}$, and $\mathbf{c}_{\mathcal{A}}$ correspond to the tableau of the CFERK (4.8) used to integrate the advection term. $\mathbf{A}_{\mathcal{D}}$, $\mathbf{b}_{\mathcal{D}}$, and $\mathbf{c}_{\mathcal{D}}$ correspond to the tableau of the RK method (2.14) used to integrate the diffusion term, and $\mathbf{A}_{\mathbf{r}}$, $\mathbf{b}_{\mathbf{r}}$, and $\mathbf{c}_{\mathbf{r}}$ correspond to the tableau of the RK method (2.14) used to integrate the reaction term. If a non-additive RK method is used to solve for $\mathbf{z}(t)$, the formula (4.22) corresponds to the methods given in [18, 19, 22]. The tableau of (4.22) is given by

$$\frac{\mathbf{c}_{\mathcal{A}}}{\left| \begin{array}{c|c} \mathbf{A}_{\mathcal{A}} & \\ \hline \mathbf{b}_{\mathcal{A}}^T & \end{array} \right.}, \quad \frac{\mathbf{c}_{\mathcal{D}}}{\left| \begin{array}{c|c} \mathbf{A}_{\mathcal{D}} & \\ \hline \mathbf{b}_{\mathcal{D}}^T & \end{array} \right.}, \quad \frac{\mathbf{c}_{\mathbf{r}}}{\left| \begin{array}{c|c} \mathbf{A}_{\mathbf{r}} & \\ \hline \mathbf{b}_{\mathbf{r}}^T & \end{array} \right.} \quad (4.23)$$

Eigenvalues of the reaction terms of semi-discretized ARD equations (4.2) do not in general follow the regular patterns of those from the advection and diffusion terms. Therefore the stability requirements for the reaction term may be different from those of the other terms. For instance, a general-purpose method for the reaction term may require a stability region that contains eigenvalues with both significant complex and negative real parts.

In other cases, a method that has significant stability along the imaginary axis may be most suitable, especially if extra stages are available because of the RKC method for the diffusion term. Optimization of enhanced stability ERK methods is discussed in [56, 57, 58, 100].

4.4.1 Order conditions

In addition to the order conditions (2.28), (4.11), and (4.13), the constituent methods must satisfy coupling conditions similar to those of the ARK methods from Section 2.4.2. The coupling conditions for classic ARK methods from Section 2.4.2 form a subset of those for the IF-2-ARK

methods derived in this chapter. Up to third order, no additional coupling conditions are required for integrating-factor-based methods. Exponential ARK methods of fourth order and above require additional coupling conditions [18, 21, 22].

If the RKC methods from Section 4.2 are used for the diffusion term of (4.2) and condition (2.34) holds, the values of \mathbf{b} and \mathbf{c} of (4.23) as well as the number of stages s are fixed. The number of stages for the RKC method is typically higher than necessary for the other constituent methods of (4.22); however, as a proof of concept, the other methods are constructed to have the same number of stages as in the RKC method.

For production software that incorporates exponential ARK methods, it would be advantageous to use a recursive form (4.18) like in [101] to reduce the storage requirements and computational cost. Further analysis is required to determine if it is possible to find higher-order methods that minimize RHS evaluations and computational work for the matrix exponentials while using the recursive formula (4.18).

4.5 The Burgers equation with Brusselator reaction terms

The simplicity and the well-studied, but complex behaviour make the Burgers equation (4.5) important for testing numerical methods for a flowing medium [49]. When $\epsilon \ll 1$ the shocks become more prominent as can be seen in Figure B.1 and spatial discretization results in a more difficult IVP to solve. The spatially discretized Burgers equation is commonly used to test IVP solvers [2, 7, 101, 102].

The Brusselator equation [98, 99] describes an oscillatory chemical system coupled with diffusion. The Brusselator equation is the only known chemical scheme with just two reactants that can possess self-sustaining oscillations [98]. Many of the nonlinear oscillations exhibited by chemical systems can be readily studied using the Brusselator equation [99]. There is no known analytical solution to the Brusselator equation [90], making it interesting for numerical studies. The one-dimensional form of the Brusselator reaction with a diffusion term for spatial coupling is given by

$$\frac{\partial u}{\partial t} = \epsilon \frac{\partial^2 u}{\partial x^2} + \alpha + u^2 v - (\beta + 1)u \quad (4.24a)$$

$$\frac{\partial v}{\partial t} = \epsilon \frac{\partial^2 v}{\partial x^2} + \beta u - u^2 v, \quad (4.24b)$$

where $u = u(t, x), v = v(t, x) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ are the concentrations of the two chemical species, $\alpha, \beta \in \mathbb{R}$ are reaction coefficients, and $\epsilon \in \mathbb{R}$ is the diffusion coefficient. Like the Burgers equation (4.5), the spatially discretized Brusselator equation (4.24) is commonly used to test IVP solvers [4, 10, 39; 44, p.148].

The IVP used to test the new IF-2-ARK methods that are introduced in Section 4.6 is a semi-discretized ARD equation combining the terms of the Brusselator and Burgers equations. We have not found any instances of this equation in the literature, but combining a basic nonlinear advection term and a basic oscillatory reaction term with diffusion is a logical method to construct a model problem for studying numerical methods for ARD equations. The *Burgers–Brusselator* equation is thus defined by

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \epsilon \frac{\partial^2 u}{\partial x^2} + \alpha + u^2 v - (\beta + 1)u, \quad (4.25a)$$

$$\frac{\partial v}{\partial t} + v \frac{\partial v}{\partial x} = \epsilon \frac{\partial^2 v}{\partial x^2} + \beta u - u^2 v, \quad (4.25b)$$

where the variables are defined as in the Brusselator equation (4.24). The Burgers equation can be semi-discretized with the upwind operator (2.47) in the one-dimensional case because, if the initial condition is positive, the solution and therefore the advection coefficient remains positive. The central differencing operator (2.45) is used to semi-discretize the diffusion term.

The *diffusion-free* Brusselator equation is the Brusselator equation (4.24) with the spatial coupling from diffusion removed by setting $\epsilon = 0$. The Jacobian of the scalar diffusion-free Brusselator reaction is given by

$$\begin{bmatrix} 2uv - \beta + 1 & u^2 \\ \beta - 2uv & -u^2 \end{bmatrix},$$

where $u, v \in \mathbb{R}$ correspond to the values of one of the spatially decoupled reaction systems. The *diffusion-free* Brusselator equation contains a *steady state* found by solving the linear system that results from setting $\frac{\partial u}{\partial t} = \frac{\partial v}{\partial t} = 0$. The steady state of the diffusion-free Brusselator is then given by

$$\lim_{t \rightarrow \infty} u = \alpha, \quad \lim_{t \rightarrow \infty} v = \frac{\beta}{\alpha}.$$

The Jacobian of the diffusion-free Brusselator at the steady state is

$$\begin{bmatrix} \beta - 1 & \alpha^2 \\ -\beta & -\alpha^2 \end{bmatrix},$$

with a characteristic polynomial of $\lambda^2 + (1 - \beta + \alpha^2)\lambda + \alpha^2\beta = 0$ [98]. A set of coefficients for the diffusion-free Brusselator, each of which has eigenvalues at the the steady state belonging to different regions of the complex plane, is given in Table 4.1. These coefficients each represent a particular type of dynamical behaviour of the diffusion-free Brusselator reaction and are further described in [98]. When the eigenvalues at the steady state have positive real components, the steady

state is unstable and self-sustaining oscillations exist for the diffusion-free Brusselator. When the eigenvalues at the steady state have negative real components, the steady state is stable [98].

α	β	λ	Steady state stable?
0.5	0.2	$-\frac{21}{40} \pm \frac{\sqrt{41}}{40}$	yes
1	1	$-0.5 \pm \frac{\sqrt{3}}{2}i$	yes
2	5	$\pm 2i$	yes
1	3	$0.5 \pm \sqrt{0.5}i$	no
1	5	$1.5 \pm \frac{\sqrt{5}}{2}$	no

Table 4.1: Coefficients used for the Brusselator and their steady state eigenvalues.

The eigenvalues of the one-dimensional first-order upwind finite-difference operator $\frac{1}{\Delta x} \mathbf{D}_1^-$ from (2.47) are $\lambda_k = \frac{\cos(2\pi k \Delta x) - 1}{\Delta x} - \frac{i \sin(2\pi k \Delta x)}{\Delta x}$, $k = 1, 2, \dots, n_{\text{grid}}$ [51, p.56], the eigenvalues for $n_{\text{grid}} = 100$ in a spatial domain of $x \in [0, 1]$ are depicted in Figure 4.3. The eigenvalues of the one-dimensional second-order central finite-difference operator $\frac{1}{\Delta x^2} \mathbf{D}_2^0$ from (2.45) are $\lambda_k = -\frac{4 \sin^2(\pi k \Delta x)}{\Delta x^2}$, $k = 1, \dots, n_{\text{grid}}$ [51, p.63]; the eigenvalues for $n_{\text{grid}} = 100$ in the complex plane are shown in Figure 4.4. In practical problems, the magnitude of the advection coefficient $\mathcal{A}(\mathbf{y}(t))$ and diffusion coefficient $\mathcal{D}(\mathbf{y}(t))$ are not generally constant with respect to the spatial variable. In this case, the structure of the eigenvalues does not follow the patterns in Figures 4.3 and 4.4, but the assumption that the eigenvalues from advection have large imaginary parts and that the eigenvalues from diffusion remain close to the real axis still holds in general. An example of an advection-diffusion equation where the advection coefficient is not constant with respect to the spatial variable is the Burgers equation (4.5), and an example of an ARD equation where the diffusion coefficient is not constant with respect to the spatial variable is found in [55].

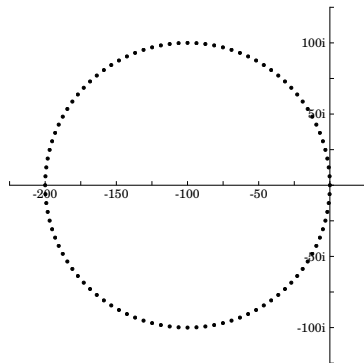


Figure 4.3: Eigenvalues of the first-order upwind finite-difference operator (2.47) with $n_{\text{grid}} = 100$, $x \in [0, 1]$.

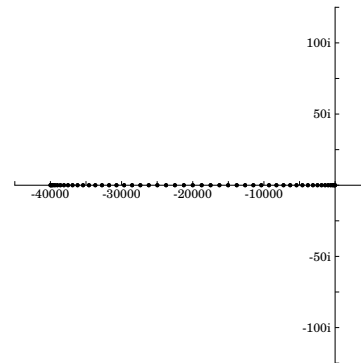


Figure 4.4: Eigenvalues of the second-order central finite-difference operator (2.45) with $n_{\text{grid}} = 100$, $x \in [0, 1]$.

4.6 Proposed second-order IF-2-ARK methods

In this section IF-2-ARK methods with an order of convergence of two are proposed. The proposed methods are based on the general form of IF-2-ARK methods in Section 4.3.

4.6.1 Constituent methods

For the purposes of analysis and ease of implementation, the ROCK2 methods from Section 4.2 have been converted into a conventional Butcher tableau. The number of stages is fixed at 10, with the Butcher tableau given in (C.3) and the stability region shown in Figure 4.1.

ROCK2 is an ERK method (2.16) with a large but bounded stability region; therefore fine enough spatial grids lead to stepsize restrictions. Using the second-order central difference operator (2.45), the maximum eigenvalues and maximum stable stepsize for $n_{\text{grid}} = 100$ with the 10-stage ROCK2 method are shown in Table 4.2. The Jacobian of the diffusion operator is constant; therefore the stepsize restrictions for (4.5), (4.24), and (4.25) can be estimated in this manner.

ϵ	$\max(\lambda)$	$\max(\Delta t)$
0.005	200	$\frac{64}{200}$
0.002	80	$\frac{64}{80}$
0.001	40	$\frac{64}{40}$
0.0005	20	$\frac{64}{20}$

Table 4.2: Maximum eigenvalues and stable stepsizes for grid size $n_{\text{grid}} = 100$ and diffusion values ϵ using the 10-stage ROCK2.

The eigenvalues of the reaction term are harder to analyse, and for the Brusselator coefficients (4.29b), the eigenvalues are generally close to the imaginary axis. It can be seen from Figure 4.1 that ROCK2 does not have good stability near the imaginary axis. Therefore an ERK method (2.16) with a stability region that captures a large interval of the imaginary axis but shares the \mathbf{b} and \mathbf{c} values with the 10-stage ROCK2 (C.3) method is derived. The stability polynomial of $R(z) = 1 + z + \frac{z^2}{2} + \frac{3z^3}{16} + \frac{z^4}{32} + \frac{z^5}{128}$ and a stability region shown in Figure 4.5 is chosen; this is the stability polynomial with the largest stability interval for the imaginary axis for a five-stage second-order ERK method with free \mathbf{b} and \mathbf{c} coefficients [100]. To find the \mathbf{A}_r that gives the chosen stability polynomial, a tableau with 8 free coefficients is constructed. The terms of the stability polynomial as a function of the Butcher tableau are given in [44, p.16], and the resulting nonlinear system is solved for the free coefficients. The tableau of this enhanced stability ERK method is given in (C.2). Further analysis may yield a larger stability region or a smaller leading error term for the method (C.2).

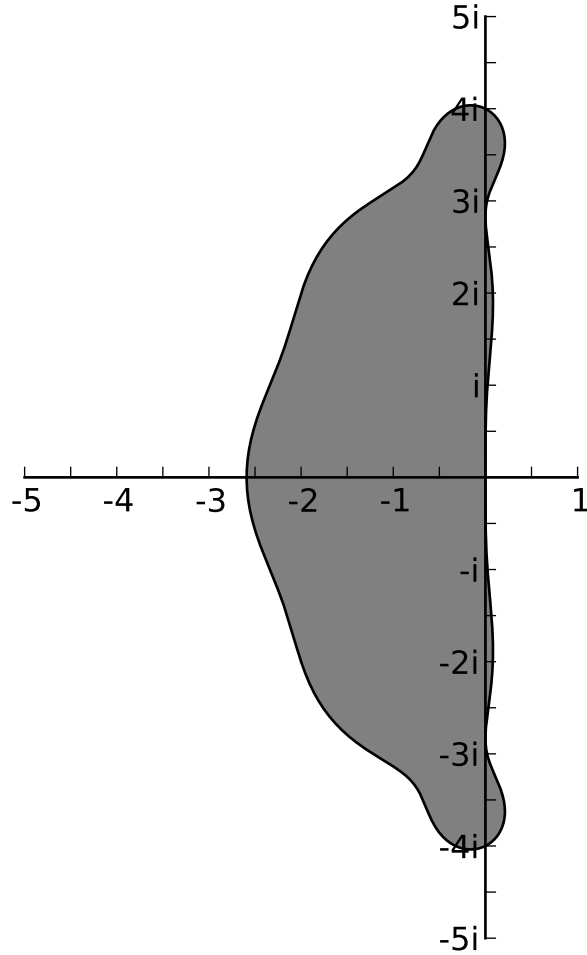


Figure 4.5: The region of absolute stability region for the 10-stage ERK method sharing \mathbf{b} and \mathbf{c} with the ROCK2 method (C.2).

4.6.2 10-Stage IF-2-ARK methods

An IF-2-ARK method (4.22) that uses the tableaux (4.23) to efficiently solve specific advection-dominated semi-discretized ARD problems is proposed. The ROCK2 tableau (C.3) is used for both the CFERK method (4.8) for the advection operator $\mathbf{A}_{\mathcal{A}}$ and the ERK method (2.16) for the diffusion operator $\mathbf{A}_{\mathcal{D}}$. Using the ROCK2 tableau (C.3) for a CFERK method does not take advantage of the extended stability region but is used because it is second order and has the same \mathbf{b} and \mathbf{c} coefficients. The ERK method (C.2) with enhanced imaginary stability is used as the tableau $\mathbf{A}_{\mathcal{R}}$ for the reaction terms. The combination of methods is referred to by a triple of Butcher tableaux with advection first, reaction second, and diffusion third. The method just described is referred to as (C.3,C.2,C.3) in this notation.

The method (C.3,C.2,C.3) is inefficient because all methods have 10 stages, and that is more than the minimum necessary for a second-order CFERK method or an ERK method with the stability

region in Figure 4.5. A practical IF-2-ARK method should take full advantage of the extra stages or use a strategy to save computational cost for the constituent methods where the extra stages are unnecessary.

A strategy to reduce the cost of the exponentials is to not enforce the condition (2.34) but still achieve second order by satisfying the coupling conditions. The tableaux (C.1) satisfy the second-order coupling conditions when used for the CFERK method along with the 10-stage ERK method (C.2) for reaction and ROCK2 (C.3) for diffusion. The resulting method (C.1,C.2,C.3) reduces the number of exponential evaluations per step from 31 to 5 by introducing stages that are redundant. The redundant stages occur because exponential evaluations are only required for the first and last stage of the CFERK. The intermediate expression $\rho_i \cdot \rho_j^{-1}$ from (4.22) evaluates to the identity matrix for stages $s = 1, 2, \dots, 9$ for a 10-stage method; therefore these quantities do not have to be computed. Likewise, ρ_i evaluates to the identity matrix for these stages because the matrix exponential of the zero matrix is the identity matrix and does not have to be computed.

4.7 Experimental results

In order to measure the accuracy of numerical solutions generated from the numerical methods being tested, an approximation to the error is made from a comparison to a reference solution. The measure used is the root-mean-square (RMS) error e_{RMS} of each numerical solution point generated by the method being tested with respect to the reference solution and given by

$$e_{\text{RMS}} = \sqrt{\frac{1}{n_{\text{steps}} m} \sum_{n=1}^{n_{\text{steps}}} \sum_{i=1}^m (y_{n,i} - y_{\text{ref},i}(t_n))^2}, \quad (4.26)$$

where m is the size of the ODE system, $y_{n,i}$ is component i of the solution at step n of the numerical solution of the method being tested, n_{steps} is the number of steps taken by that numerical solution, and $y_{\text{ref},i}(t_n)$ is component i of the reference solution at step n corresponding to time t_n .

The reference solutions are generated using the ARK5(4) method from [55], which is an eight-stage IMEX method with third-order dense output. The reference solutions are stored in files that include the stage values at each step. Therefore a dense reference solution that is third-order accurate can be produced for any time in the domain; this is sufficient to test second-order methods. ARK5(4) is used as a linearly implicit RK method by using a splitting of the RHS into linear and nonlinear parts $\mathbf{f} = \mathbf{f}^{[1]} + \mathbf{f}^{[2]}$, where

$$\mathbf{f}^{[1]}(\mathbf{y}(t)) = \mathbf{f}(\mathbf{y}(t)) - \mathbf{f}^{[2]}(\mathbf{y}(t)), \quad (4.27a)$$

$$\mathbf{f}^{[2]}(\mathbf{y}(t)) = \mathbf{J}_{\mathbf{f}}(\mathbf{y}(t)) \cdot \mathbf{y}(t), \quad (4.27b)$$

where $\mathbf{f}(\mathbf{y}(t))$ is the RHS of the ODE (2.2a), $\mathbf{J}_f(\mathbf{y}(t))$ is the Jacobian (2.9) of the ODE, and $\mathbf{f}^{[1]}(\mathbf{y}(t)), \mathbf{f}^{[2]}(\mathbf{y}(t))$ correspond to those in an IMEX method (2.21) [26]. The ideal IVP method for generating reference solutions for stiff problems is a fully implicit RK method, such as the Radau IIA methods, due to the excellent behaviour on most stiff problems, low long-term storage requirements for the reference solution from taking large steps and having a relatively low number of stages, and dense output with an order of convergence equal to that of the main method.

The spatial and temporal domains, initial condition, boundary conditions, final time, and diffusion parameters used experimentally for the Burgers (4.5), Brusselator (4.24), and Burgers–Brusselator (4.25) equations are given by

$$x \in [0, 1], \tag{4.28a}$$

$$t \in [0, 10], \tag{4.28b}$$

$$u(0, x) = v(0, x) = 1 + \sin(2\pi x), \tag{4.28c}$$

$$u(t, 0) = u(t, 1), v(t, 0) = v(t, 1), \tag{4.28d}$$

$$\epsilon = \{0.005, 0.002, 0.001, 0.0005\}, \tag{4.28e}$$

which correspond to the same spatial domain (4.28a), and time domain (4.28b), but an initial condition (4.28c) that is a modification of the Brusselator (4.24) IVP in [44, p.6]. Periodic boundary conditions (4.28d) are used because this generated interesting behaviour with the Burgers–Brusselator equation (4.25). The sets of reaction coefficients for the Brusselator equation (4.24) and Burgers–Brusselator equation (4.25) used are

$$\alpha = 0.2, \beta = 0.5, \tag{4.29a}$$

$$\alpha = 2, \beta = 5, \tag{4.29b}$$

$$\alpha = 1, \beta = 3. \tag{4.29c}$$

We are unaware of any analytical solutions to the Burgers–Brusselator equation (4.25). As an alternative to an analytical solution, the reference solutions to the semi-discretized Burgers (4.5), Brusselator (4.24), and Burgers–Brusselator (4.25) equations are found with $n_{\text{grid}} = 100$ grid points. A reference solution to the IVP is generated by solving it numerically with successively finer error tolerances until the difference between successive solutions falls below a chosen level. The tolerances (2.38) used by the step controller for generating reference solutions are $\tau_{\text{abs}} = \tau_{\text{rel}} = \{10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}\}$, with 10^{-10} being taken as the tolerance for the reference solution and the other tolerances used to judge the accuracy of the reference solution. The digits of accuracy of the solutions with respect to $\tau_{\text{abs}} = \tau_{\text{rel}} = 10^{-10}$ for various diffusion and Brusselator reaction

coefficients are shown in Figure 4.6. It can be seen that the Brusselator reaction coefficients (4.29b) are more difficult to find high-accuracy solutions for than the other reaction coefficients, with a mean number of digits of accuracy of approximately 7 compared to 9 to 10 for the other Brusselator reaction parameters (4.29a) (4.29c).

Sharp transitions that are characteristic of the Burgers equation (4.5) are shown in Figure B.1 at $t = 2.0$. The Brusselator reaction coefficients (4.29a) have negative real eigenvalues at the steady state, causing any oscillations to rapidly decay, as shown in Figure B.2. The Brusselator reaction coefficients (4.29b) have complex eigenvalues at the steady state that tend to cause any oscillations to be self-sustaining, as can be seen in Figure B.3. The Brusselator reaction coefficients (4.29c) have eigenvalues with positive real parts at the steady state that tend to amplify oscillations. When used for the Brusselator equation (4.24) the amplifying effect of the Brusselator reaction coefficients (4.29c) can offset the dampening effect of diffusion, leading to long-term oscillations, as shown in Figure B.4 and the example in [44, p.6].

The Brusselator reaction coefficients (4.29b) produce long-term oscillations and include sharp transitions, which are characteristic of the Burgers equation (4.5) but not of the Brusselator equation (4.24), when used with the Burgers–Brusselator equation (4.25), as shown in Figure B.6. Conversely, the coefficients (4.29c) that produce long-term oscillations in the Brusselator equation (4.24) do not produce these oscillations when used in the Burgers–Brusselator equation (4.25), as also shown in Figure B.7. The strong dampening effect of the negative real eigenvalues of the reaction term with Brusselator reaction coefficients (4.29a) can be seen in Figure B.5. The Brusselator reaction coefficients (4.29b) are most interesting when used with the Burgers–Brusselator equation (4.25) because they give a stepsize restriction despite being non-stiff. Moreover, there is a lack of a stepsize restriction of similar magnitude in the Burgers equation (4.5) or Brusselator equation (4.24) with the corresponding set of Brusselator reaction or diffusion coefficients.

4.7.1 Performance comparisons

To show that new numerical methods perform well in practice, the most effective measure is a comparison of computational work, such as the time to perform a computation for a given measure of the accuracy, such as the RMS error (4.26), relative to a reference solution.

We are unaware of any studies comparing the computational cost of the CFERK methods [24] or the integrating-factor-based methods that incorporate CFERK methods [18, 22] to mainstream IVP methods. The matrix exponential using Krylov subspace methods has a computational cost of $\mathcal{O}(n_{\text{grid}}^2)$ for a grid size of n_{grid} , in comparison to a cost of $\mathcal{O}(n_{\text{grid}}^3)$ for semi-Lagrangian methods, when used for the advection flow calculation [24]. Therefore the IF-2-ARK methods (C.3,C.2,C.3) and (C.1,C.2,C.3) are used with the matrix exponential of $\mathcal{A}(t, \mathbf{y}(t))$ for the advection

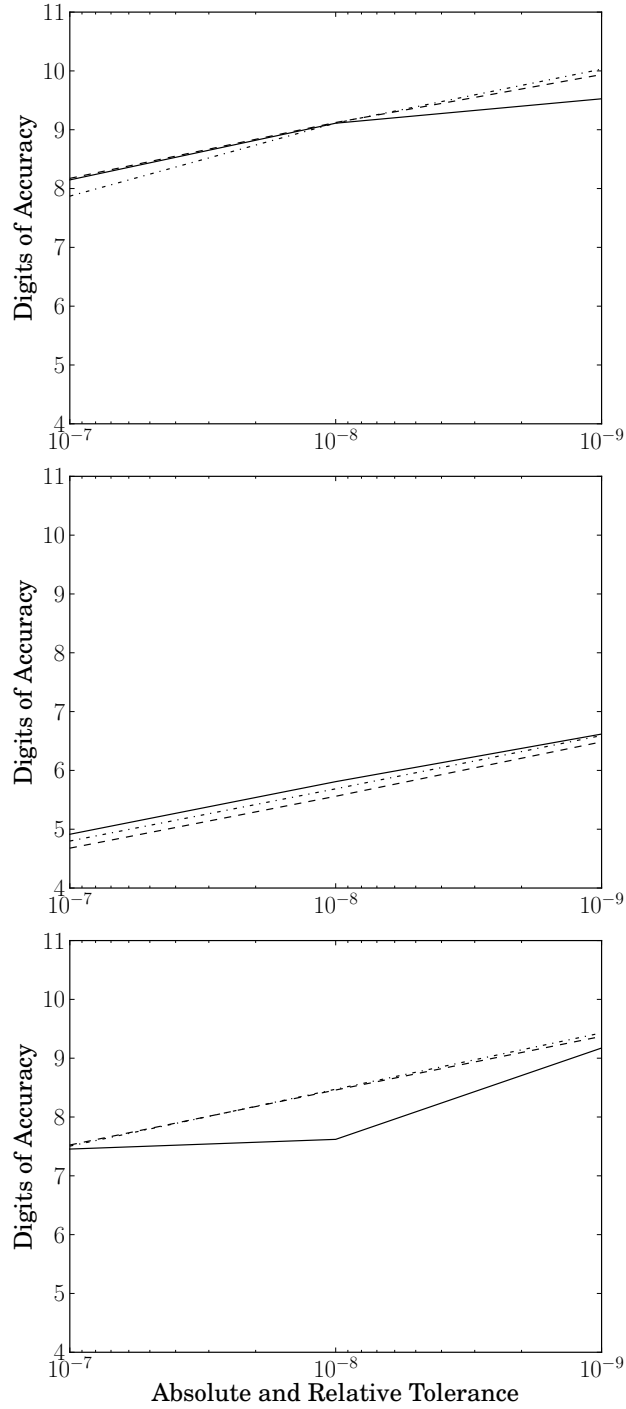


Figure 4.6: Convergence of the trial reference solutions in number of digits of accuracy as compared to the reference solution with a tolerance of 10^{-10} for $\alpha = 0.2, \beta = 0.5$ (4.29a) (top), $\alpha = 2, \beta = 5$ (4.29b) (middle), and $\alpha = 1, \beta = 3$ (4.29c) (bottom). $\epsilon = 0.002$ (---), $\epsilon = 0.001$ (-·-), $\epsilon = 0.0005$ (—).

flow calculation as an initial experiment in order to find if CFERK-based methods can have a lower computational cost than mainstream IVP methods.

The IF-2-ARK methods (C.3,C.2,C.3) and (C.1,C.2,C.3) are compared to IMEX(2,3,2) (C.4), that is used as a linearly implicit RK method with the splitting (4.27). They are also compared to the 10-stage ROCK2 method (C.3), which is an optimal ERK method (2.16) in the sense that the main source of stiffness for the advection-dominated Burgers–Brusselator equation (4.25) with the nonstiff Brusselator reaction coefficients (4.29) is the diffusion term. The IF-2-ARK method with a reduced number of exponentials (C.1,C.2,C.3) described in Section 4.6 is also used as a comparison. Both IF-2-ARK methods (C.3,C.2,C.3) and (C.1,C.2,C.3) are tested with the PI stepsize controller from Section 2.5.3 because this strategy was found to be successful with EXP4 from [47, p.1565].

The tolerances (2.38) used are $\tau_{\text{rel}} = \tau_{\text{abs}} = 10^{(-2-\frac{i}{2})}$, $i = \{0, 1, 2, \dots, 10\}$. The maximum number of steps allowed to find a solution is 300000; any computations requiring more steps are deemed infeasible. Step-doubling error estimation from Section 2.5.1 in conjunction with the conventional stepsize control (2.40) from Section 2.5.3 is used for all methods to ensure a uniform comparison. Where indicated, the methods (C.3,C.2,C.3) and (C.1,C.2,C.3) are also tested using PI instead of the conventional stepsize control (2.40) from Section 2.5.3. Other ERK methods including (C.2) were tried individually, but they all took a much greater number of steps than the 10-stage ROCK2 method (C.3) and were not able to reasonably find a solution for most sets of parameters.

It can be seen from Figure 4.7 that when solving the Burgers equation (4.5), as ϵ decreases from 0.005 to 0.001, the advantage of IMEX(2,3,2) (C.4) over ROCK2 (C.3) increases. The Burgers equation (4.5) generally becomes more difficult numerically as advection dominates [49], and this favours a method with a stability region covering the imaginary axis such as the explicit method from IMEX(2,3,2) (C.4) [7].

It can be seen from Figure 4.8 that IMEX(2,3,2) (C.4) generally has a slight advantage over ROCK2 (C.3) when solving the Brusselator equation (4.24). The Brusselator reaction coefficients (4.29b) and (4.29c) that have complex eigenvalues at the steady state result in these methods finding a less-accurate solution than for the case of the reaction coefficients with only negative real eigenvalues (4.29a). IMEX(2,3,2) (C.4) and ROCK2 (C.3) perform equally well for these parameters.

It can be seen from Figure 4.9 that the Burgers–Brusselator equation (4.25) is a more difficult problem numerically, and that IMEX(2,3,2) (C.4) is the most efficient in nearly all cases, showing the smoothest graph and the most stable behaviour. ROCK2 (C.3) is no longer competitive with IMEX(2,3,2) (C.4) in terms of computational cost and has more difficulty giving a high-accuracy solution. For this problem, the IF-2-ARK methods (C.3,C.2,C.3) and (C.1,C.2,C.3) are not competitive to either ROCK2 or IMEX (C.4) in terms of computational cost. However, the IF-2-ARK method (C.3,C.2,C.3) easily gives high-accuracy solutions. The IF-2-ARK methods (C.1,C.2,C.3)

with only 5 exponential evaluations per step has a poor performance in comparison to the one with 31 exponential evaluations per step (C.3,C.2,C.3) when finding high-accuracy solutions, but it has a reduced computational cost for moderate accuracies. The PI step controller performs no better than conventional step control (2.40) for the IF-2-ARK methods in all cases.

It can be further seen from Figure 4.9 that the Brusselator reaction coefficients (4.29b) are the most costly of the tested Brusselator reaction coefficients (4.29) when finding a solution of a given accuracy. These reaction coefficients (4.29b) are the ones that have complex eigenvalues at the Brusselator steady state and self-sustaining oscillations for the Burgers–Brusselator equation (4.25). In this case, the computational cost is most similar between the IF-2-ARK and non-exponential methods. This motivates a comparison of the IF-2-ARK methods (C.3,C.2,C.3) to IMEX(2,3,2) (C.4) at higher precisions. The 10-stage ROCK2 (C.3) method is not included because it has difficulty producing high-accuracy solutions.

A comparison of the IF-2-ARK method (C.3,C.2,C.3) and IMEX(2,3,2) (C.4) is shown in Figures 4.10, 4.11, and 4.12 for the three sets of Brusselator reaction coefficients tested. The diffusion coefficients used are $\epsilon = 0.002, 0.001, 0.0005$ and the error tolerances are $\tau_{\text{rel}} = \tau_{\text{abs}} = 10^{(-4-\frac{i}{2})}$, $i = \{0, 1, 2, \dots, 12\}$. It can be seen from Figure 4.11 that the IF-2-ARK method (C.3,C.2,C.3) outperforms IMEX(2,3,2) (C.4) for $\epsilon = 0.0005$ and the Brusselator reaction coefficients (4.29b). It can also be seen that with fine tolerances and all of the Brusselator reaction coefficients tested, the IF-2-ARK method (C.3,C.2,C.3) is able to give solutions with accuracies that are not feasible with IMEX(2,3,2) (C.4).

In summary, the Burgers–Brusselator (4.25) is introduced as a simple test problem with a non-constant advection coefficient and an oscillatory reaction term. It is a combination of two classical equations, the Burgers equation (4.5) and the Brusselator equation (4.24). The combined coefficients create behaviour not observed in the constituent equations. The combined system is an ARD equation that is usually solved efficiently by implicit-explicit splitting. Two new 3-additive splittings are proposed, combining 2-ARK methods with CFERK methods to handle advection, and it is found that when finding high-accuracy solutions in the regime where the problem is hardest to solve, one of the new 3-additive splittings (C.3,C.2,C.3) outperforms standard solvers. This is a promising result in that high-accuracy solutions are expected to become more important as the demand for increased model fidelity rises. This is the first study that we know of that shows CFERK methods can be computationally efficient in comparison to existing methods. It also gives definite future directions for research into improved time-integration methods for ARD equations.

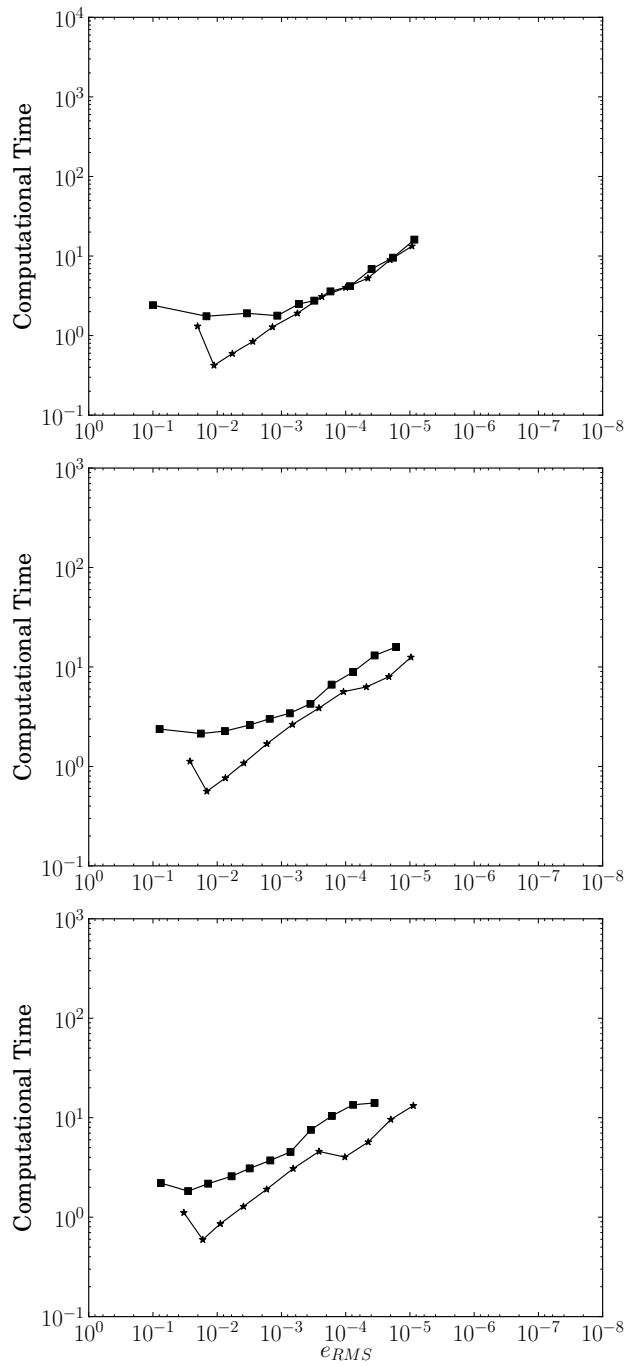


Figure 4.7: Computational time compared to the RMS error of the solution with $\epsilon = 0.005$ (top), $\epsilon = 0.002$ (middle), $\epsilon = 0.001$ (bottom), and for the Burgers equation with $n_{\text{grid}} = 100$. IMEX (2,3,2) (C.4) ★, ROCK2 (C.3) ■.

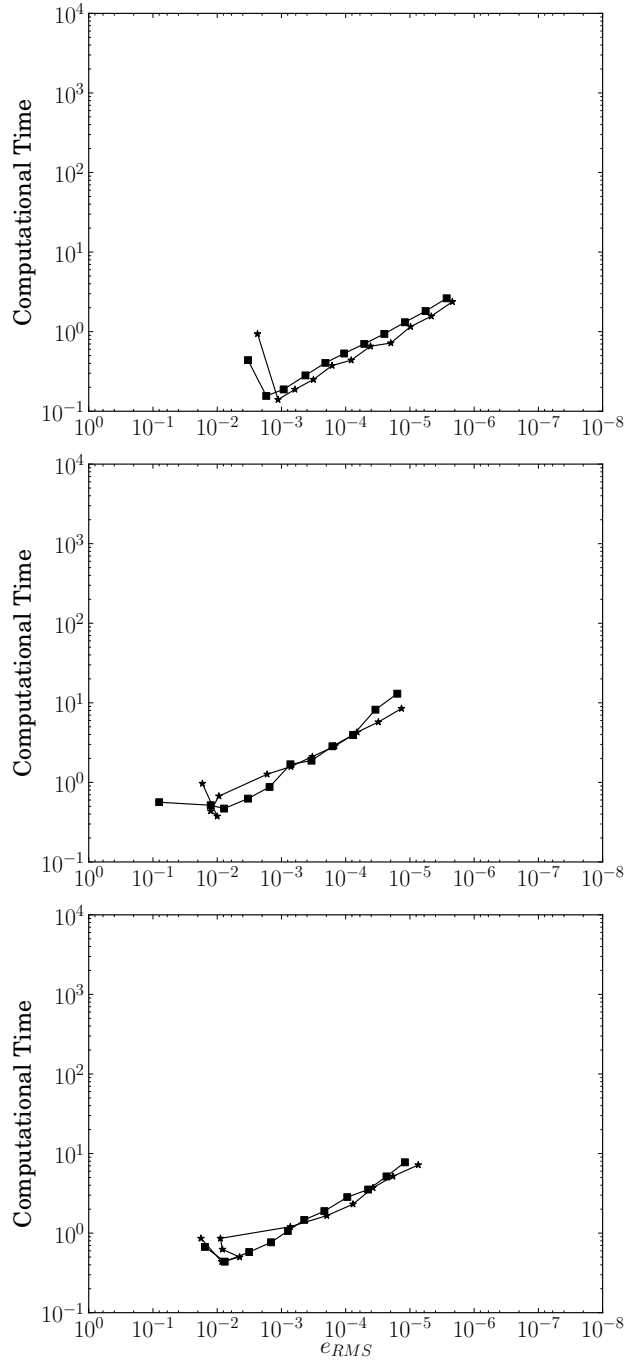


Figure 4.8: Computational time compared to the RMS error of the solution with $\alpha = 0.5, \beta = 0.2$ (4.29a) (top), $\alpha = 2, \beta = 5$ (4.29b) (middle), $\alpha = 1, \beta = 3$ (4.29c) (bottom) as reaction coefficients for the Brusselator equation (4.24) with $\epsilon = 0.001$ and $n_{\text{grid}} = 100$. IMEX (2,3,2) (C.4) \star , ROCK2 (C.3) \blacksquare .

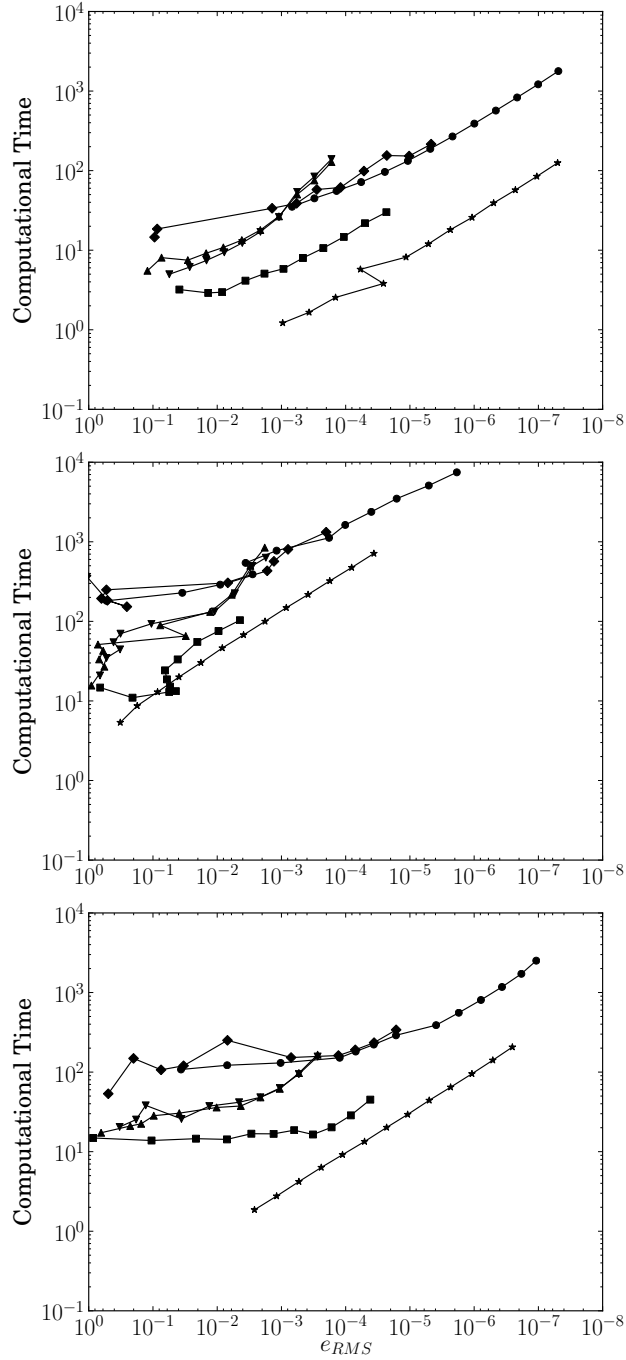


Figure 4.9: Computational time compared to the RMS error of the solution with $\alpha = 0.5, \beta = 0.2$ (4.29a) (top), $\alpha = 2, \beta = 5$ (4.29b) (middle), $\alpha = 1, \beta = 3$ (4.29c) (bottom) as reaction coefficients for the Burgers–Brusselator equation (4.25) with $\epsilon = 0.001$ and $n_{\text{grid}} = 100$. IMEX (2,3,2) (C.4) ★, ROCK2 (C.3) ■, IF-2-ARK (C.3,C.2,C.3) ●, IF-2-ARK (C.3,C.2,C.3) and PI step control ◆, IF-2-ARK (C.1,C.2,C.3) with reduced exponentials ▲, IF-2-ARK (C.1,C.2,C.3) with reduced exponentials and PI step control ▼.

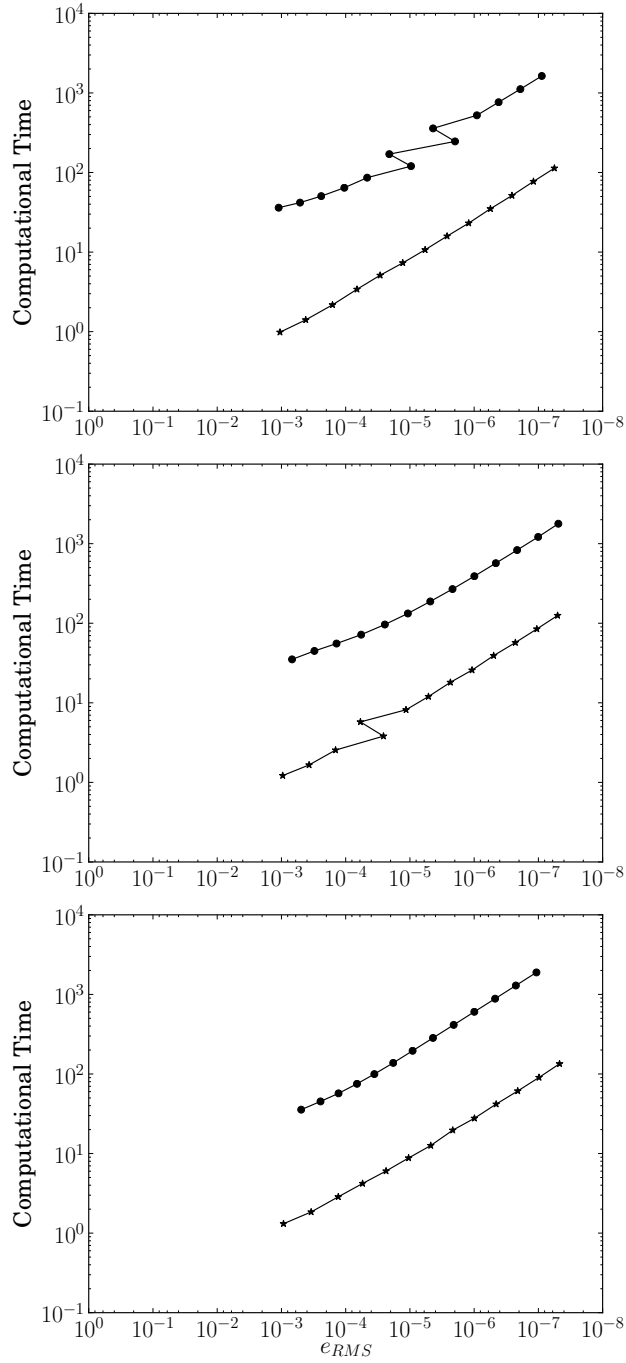


Figure 4.10: Computational time compared to the RMS error of the solution with $\epsilon = 0.002$ (top), $\epsilon = 0.001$ (middle), and $\epsilon = 0.0005$ (bottom) for the Burgers–Brusselator equation (4.25) with Brusselator reaction coefficients $\alpha = 0.5, \beta = 0.2$ (4.29a) and $n_{\text{grid}} = 100$. IMEX (2,3,2) (C.4) ★, IF-2-ARK (C.3,C.2,C.3) ●.

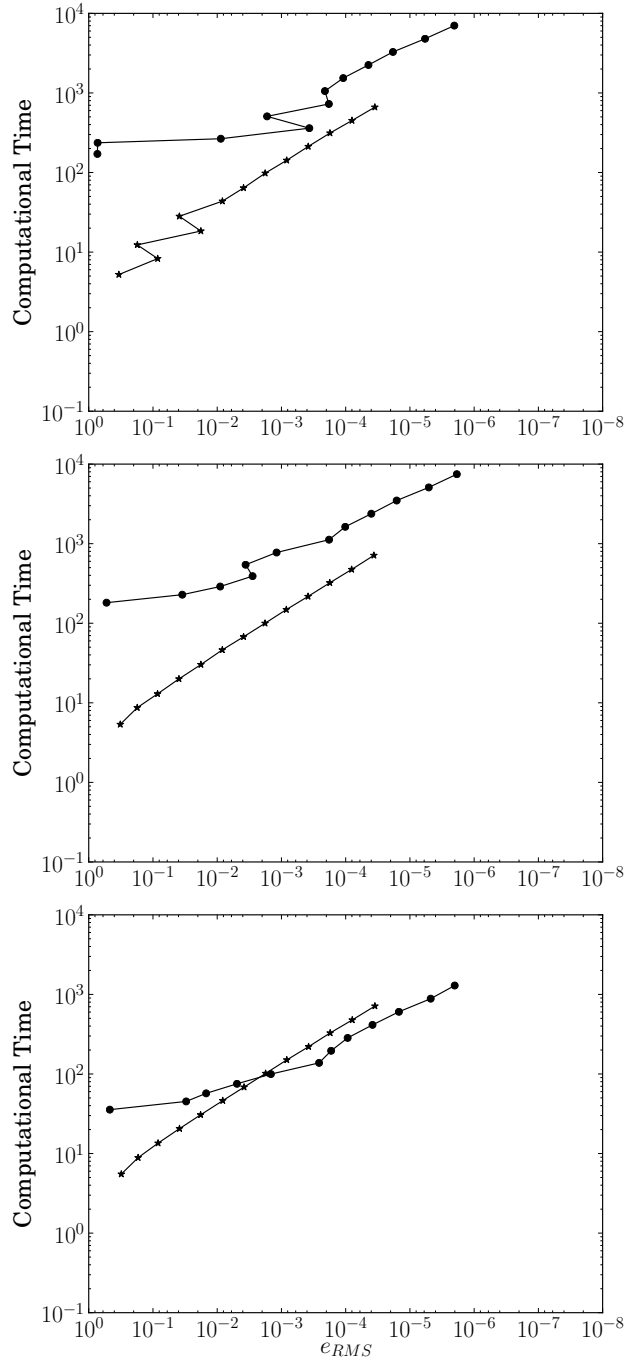


Figure 4.11: Computational time compared to the RMS error of the solution with $\epsilon = 0.002$ (top), $\epsilon = 0.001$ (middle), and $\epsilon = 0.0005$ (bottom) for the Burgers–Brusselator equation (4.25) with Brusselator reaction coefficients $\alpha = 2, \beta = 5$ (4.29b) and $n_{\text{grid}} = 100$. IMEX (2,3,2) (C.4) ★, IF-2-ARK (C.3,C.2,C.3) ●.

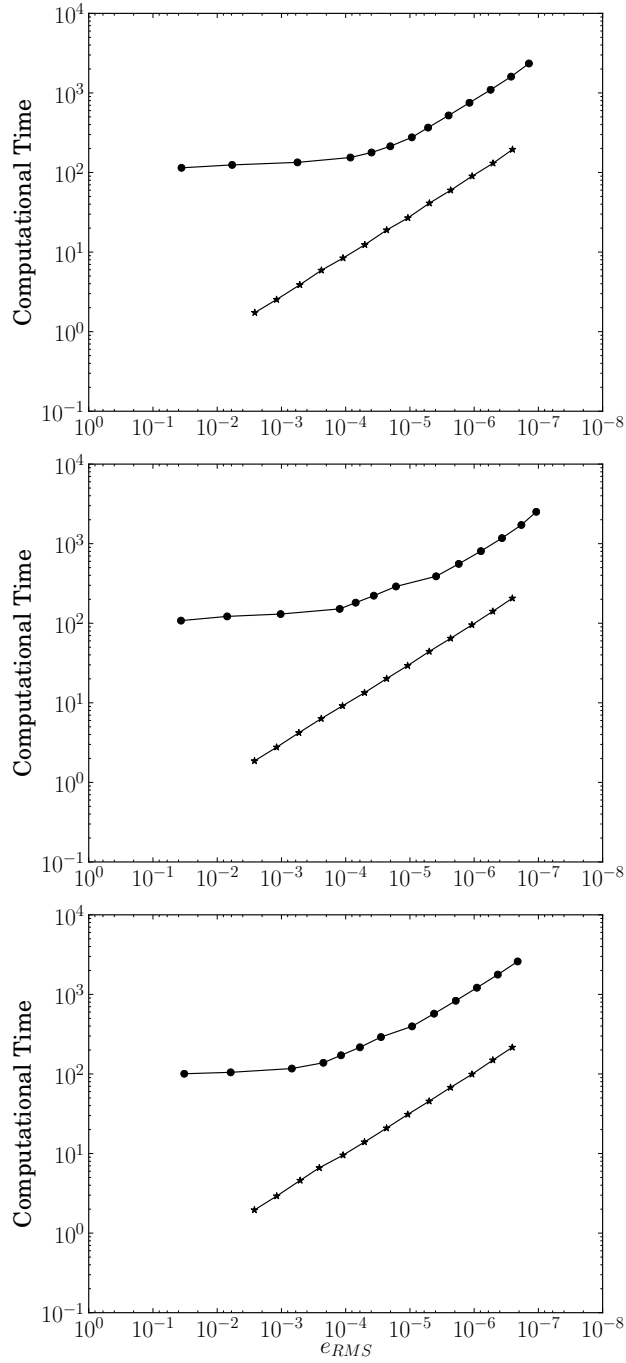


Figure 4.12: Computational time compared to the RMS error of the solution with $\epsilon = 0.002$ (top), $\epsilon = 0.001$ (middle), and $\epsilon = 0.0005$ (bottom) for the Burgers–Brusselator equation (4.25) with Brusselator reaction coefficients $\alpha = 1, \beta = 3$ (4.29c) and $n_{\text{grid}} = 100$. IMEX (2,3,2) (C.4) ★, IF-2-ARK (C.3,C.2,C.3) ●.

CHAPTER 5

CONTRIBUTIONS AND FUTURE WORK

5.1 Contributions

The Burgers–Brusselator equation (4.25) is introduced as a simple test problem with non-constant advection coefficients. This is a contribution because many problems used to test CFERK and other exponential methods generally have constant advection coefficients, e.g., [18, 48]. The Burgers–Brusselator equation (4.25) also has an oscillatory reaction term. The combined coefficients create behaviour not in the constituent equations that leads to interesting behaviour and numerical difficulties with certain values of coefficients, as shown in Figures B.6 and 4.9, respectively.

The results presented in this thesis are the first known study of the performance of the integrating-factor-based methods incorporating CFERK methods such as those in [18, 22] and CFERK methods in general [24]. This is also the first study to show the practicality of CFERK methods. The methods introduced, called IF-2-ARK methods, show the potential viability of 3-additive splitting for certain ARD systems that cause difficulty with more conventional methods when high-accuracy solutions are desired. Further study is required to refine these types of methods to determine whether they can be competitive with existing methods for IVPs in practice.

Figure 4.11 shows that the IF-2-ARK method (C.3,C.2,C.3) introduced in Section 4.6 can have a lower computational cost than other second-order methods for a certain combination of diffusion and reaction parameters when solving the Burgers–Brusselator equation (4.25). The particular coefficients are $\epsilon = 0.0005$, leading to dominance of the advection term, and the Brusselator reaction coefficients (4.29b) that are more costly to solve than the other tested reaction coefficients in (4.29). The high cost to solve the Burgers–Brusselator equation (4.25) with the Brusselator reaction coefficients (4.29b) is shown in Figure 4.9. Figure 4.11 shows that (C.3,C.2,C.3) can solve the Burgers–Brusselator equation (4.25) with a lower computational cost than IMEX (2,3,2) (C.4) as advection becomes increasingly dominant and high accuracy is desired.

The IF-2-ARK method can provide solutions with accuracies that are infeasible with the other methods tested. In addition, the IF-2-ARK methods proposed (C.3,C.2,C.3) and (C.1,C.2,C.3) use an RKC method for the diffusion term; therefore they can easily be used to solve problems with nonlinear diffusion.

5.2 The `pythODE` PSE

The IVP component of the `pythODE` PSE developed during the course of this thesis allows the construction of many numerical methods including RK, multistep, and general linear methods as well as the exponential and additive variants of these methods. The particular abstraction using object-oriented programming with the `Solution` objects, `Solver` objects, and `SolverModule` objects described in Chapter 3 can also incorporate other families of numerical methods. This allows experimentation with many different numerical methods in a single software package. Examples in this thesis include using a number of RK methods, along with additive and exponential variants, that share code for the definition of the IVP and step control strategies, and to perform and analyse the numerical experiments.

The development work on the part of the `pythODE` PSE for solving IVPs is currently at an advanced stage. The general architecture and feature set described in Chapter 3 have been implemented in `pythODE`. However, many features are incomplete, and further testing of the software for stability and usability is required.

The features for large-scale testing and experimentation described in Section 3.3.5 are used to create an extensive test suite that includes IVPs published in [44] and [68] as well as the Luo–Rudy cardiac cell model from [64]. The test suite is executed regularly, and the report generated is stored in a version-control system. The same testing and experimental features were used for many of the preliminary experiments that led up to the results in Chapter 4; those results required many independent aspects from many of the components of an IVP solver to be tested against each other in order to obtain an overview of the behaviour of the new IF-2-ARK methods in comparison to other methods.

The PSE `pythODE` has a large selection of RK and ARK methods with which to solve IVPs. A hindrance is that at present the stiff IVP solvers used in `pythODE` are linearly implicit. If efficiency measurements are not important, the nonlinear systems of equations required by fully implicit numerical methods can be solved through the PSE `pythNon` [95]. In the case of methods such as RADAU5 (2.18) or SDIRK4 (2.19), an efficient implementation suitable for comparing CPU times between fully implicit numerical methods requires special treatment of the resulting nonlinear systems of equations. For example, RADAU5 (2.18) makes use of complex arithmetic, which is well supported by PYTHON, NUMPY, and SCIPY using operator overloading. The methods used as comparisons in the experiments in Chapter 4 are considered the most effective that exist in the current implementation in `pythODE`.

It is possible to perform experiments with large numbers of controlled parameters using `pythODE`; however, a significant limitation is the lack of more advanced features to record the observed values in long-term storage, perform analysis, and visualize the results. An example of some of the more

advanced features is that the figures of the stability regions of RK methods in this thesis, such as Figures 2.3, 4.1, and 4.2, were generated with `pythODE`. In general, for observed values other than stepsize, CPU time, and accuracy, it is necessary to do a significant amount of development to record a particular observed value of interest and to manually analyse the results from the report generated. This process will be automated in the future with formats such as XML [11] or software such as a relational database used to record experimental results for future analysis. This will make conducting numerical experiments more efficient and enhance reproducibility, which has been increasingly identified as a limitation to progress in scientific computing [70].

5.3 Future work

5.3.1 Improvements to the IF-2-ARK methods

The IF-2-ARK method (C.3,C.2,C.3) has a high computational cost for each time step due to the large number of exponential calculations, i.e., 31 in the case of the 10-stage method presented. The primary source of computational cost for the IF-2-ARK methods (4.22) is the need to compute $\rho_{ij} \cdot \mathcal{D}(\mathbf{Y}_i)$ and $\rho_{ij} \cdot \mathbf{r}(\mathbf{Y}_i)$, which transport the derivative of the constituent methods for diffusion and reaction. The method (C.3,C.2,C.3) derived by not enforcing the condition (2.34) to introduce redundant stages and reduce the number of exponential calculations to 5 per step did not produce enough of a reduction in computational cost at moderate accuracies and was not suitable for high-accuracy solutions.

Another strategy that may reduce the cost of the exponentials is to use a constant advection coefficient when evaluating the stages of a CFERK method. It is shown in [48] that with a constant Jacobian matrix there are savings of up to 80–90% in computational cost from reusing information from previous matrix exponential evaluations (4.14) for multiple but similar vectors \mathbf{v} . It is also possible to use a linearly implicit or fully implicit method for the diffusion term rather than an RKC method, thus avoiding the high number of stages associated with using an RKC method as a constituent of a IF-2-ARK method. Further study may identify other potential strategies to reduce the cost of the exponentials.

A variable number of stages for the RKC constituent method of a IF-2-ARK method can ensure the method is optimal for the characteristics of the diffusion term at each step. This strategy is used in the RKC, IRKC, and ROCK2 codes where the spectral radius of the Jacobian is estimated at each step. This type of adaptive strategy is essential for nonlinear diffusion, where the spectral radius of the Jacobian of the diffusion term is unlikely to be predictable in advance. It is essential to ensure the variable number of stages RKC method does not incur excessive computational cost for the additional exponential evaluations that is required.

A further strategy to reduce computational cost is to develop an embedded method for the

IF-2-ARK methods. We do not know of any CFERK methods that have embedded methods. It is therefore unknown if this is a viable strategy in comparison to step-doubling error control. For instance, it was found that the PI step controller was suitable for the non-exponential methods but not for the CFERK methods. An embedded method would likely reduce computational cost of the methods by 33% by eliminating the error control step. With an embedded method, more promising step controllers such as a variant of PI step control could be studied.

Once optimal IF-2-ARK methods have been developed, they should be compared to fully implicit RK methods for the purpose of generating high-accuracy solutions. Fully implicit RK methods typically have an extremely high cost per step; however, the problem-specific nature of IF-2-ARK methods may offer some savings.

The reaction terms of an ARD equation are spatially decoupled; therefore, they are suitable for parallelism. The independent nature of the computations for each grid point makes this type of computation suitable for accelerated platforms such as graphics processing units, which are becoming more common for high-performance computing.

5.3.2 More complex ARD systems

Many ARD systems of interest are more complex than the Burgers–Brusselator system (4.25). It is possible that the IF-2-ARK methods may be more suitable on these more complex ARD systems than the Burgers–Brusselator system (4.25), which was used for this proof-of-concept study. The Burgers–Brusselator system (4.25) is suitable for this type of study because it is one of the simplest ARD systems with nonlinear advection and an oscillatory reaction term. Examples of more complex ARD systems are found in [8, p.33; 51, p.18-22,134; 79]. Further study may identify the characteristics of ARD systems where 3-additive splitting is advantageous.

5.3.3 High-order variants

Satisfactorily solving the sets of simultaneous nonlinear algebraic equations derived from order conditions and coupling conditions has proven to be a hindrance in the design of practical RK and ARK methods [43, p.175; 55]. Most methods of sufficient complexity, such the high-order ARK methods in [55], DOPR54 (2.37), or SDIRK4 (2.19), use simplifying assumptions similar to (2.34) to reduce the complexity of the simultaneous nonlinear algebraic equations that must be solved. The simplifying assumptions do not necessarily imply the numerical method is not optimal because in many cases it is not necessary to use the full number of degrees of freedom provided by the order conditions and coupling conditions to achieve the desired properties in a numerical method.

Table 2.4 shows that for second-order IF-2-ARK methods, the coupling conditions are still relatively small in number even if (2.34) is not satisfied, with six coupling conditions for a second-order method. A third-order IF-2-ARK method is required to satisfy 12 standard RK order conditions

in total for the three constituent methods, 39 standard ARK coupling conditions, and the additional order condition (4.11) for a third-order CFERK method. A fourth-order IF-2-ARK method is required to satisfy 24 standard RK order conditions in total for the three methods, 234 standard ARK coupling conditions, the four additional order conditions (4.13) for the fourth-order CFERK method, and a minimum of four additional coupling conditions given in [21].

High-order IRK methods are commonly used to find high-accuracy solutions to difficult stiff IVPs. Therefore, it is essential to develop high-order IF-2-ARK methods and compare the efficiency with existing high-order methods such as the IMEX methods ARK5(4) [55] and fully implicit RK such as RADAU5 (2.18).

5.3.4 Semi-Lagrangian methods

The feasibility of using a matrix exponential in a CFERK method (4.8) for advection flow calculations has been shown in this thesis. However, semi-Lagrangian methods have a computational cost of $\mathcal{O}(n_{\text{grid}}^3)$ for grid size n_{grid} , compared to $\mathcal{O}(n_{\text{grid}}^2)$ for Krylov subspace methods [24]. Semi-Lagrangian methods better capture the qualitative behaviour of the underlying model. Therefore, due to improvements in physical models, they may be better suited for finding high-accuracy solutions. It may also be possible to use a coarser grid with higher-order semi-Lagrangian methods to achieve high accuracy. Semi-Lagrangian methods can capture the behaviour of the underlying PDE on coarser grids better than Eulerian methods can, as can be seen in [22]. Memory usage can be a significant limitation to the accuracy of simulations in a high-performance computing environment, coarse grids can reduce memory usage in some cases.

There are various methodologies of developing semi-Lagrangian methods [96]. However, the method of characteristics described in Section 4.1 is suitable for parallelism because it converts the advection term into many non-stiff IVPs, and the interpolation is done independently as well. Therefore, even if the overall computational cost of the semi-Lagrangian methods is higher, they may be may yield a solution in less time on parallel computers.

In [23] semi-Lagrangian methods are implemented using integrating-factor-based multistep methods that solve differential-algebraic equations (DAEs), which are generally more complex to solve than ODEs. DAEs arise from the semi-discretization of problems similar to ARD equations that incorporate conditions such as incompressibility for the flowing medium. These methods are suitable for study in `pythODE`, and it remains to be seen if they are suitable for practical problems.

5.3.5 Further development of `pythODE`

The experimental capabilities of `pythODE` allow the design of experiments to determine the optimal values of the independent aspects of the solution process such as the numerical methods, heuristics, and method parameters, all of which may have complex interactions with each other. The mea-

sured CPU time using `pythODE` does not correspond exactly with an eventual high-performance implementation using a language such as FORTRAN. However, these comparisons give a reasonable estimate to motivate future development. An example of this type of comparison is in [48], where methods are compared in MATLAB, which has many of the same non-uniformities relative to a high-performance computing environment as PYTHON does.

Many other numerical studies other than those performed in this thesis are suitable for `pythODE`. With further use, the applicability of results from the PSE to an HPC environment will become apparent. In particular, numerical experiments such as those in [43] and [44] can be easily replicated. Ultimately a PSE like `pythODE` could generate code for a high-performance computing environment as well.

High-priority development tasks for `pythODE` are to further automate and reduce the development cost of conducting numerical experiments through improvements in the test suite. Expansion of the number of IVPs in the test suite and automating checking will help guard against code regressions as development proceeds. Finally, development of a GUI interface for `pythODE` will eventually allow many of the potential numerical experiments to be conducted with little to no programming on the part of the user.

REFERENCES

- [1] A. Abdulle. On roots and error constants of optimal stability polynomials. *BIT*, 40(1):177–182, 2000.
- [2] A. Abdulle. Fourth-order Chebyshev methods with recurrence relation. *SIAM J Sci Comput*, 23(6):2041–2054, 2002.
- [3] A. Abdulle. Large stiff systems solved by Chebyshev methods. *PAMM*, 1(1):508–509, 2002.
- [4] A. Abdulle and A. A. Medovikov. Second-order Chebyshev methods based on orthogonal polynomials. *Numer Math*, 90(1):1–18, 2001.
- [5] R. A. Adams and C. Essex. *Calculus: A complete course*. Pearson Education Canada, 7 edition, 2009.
- [6] M. Al-lawatia, R. C. Sharpley, and H. Wang. Second-order characteristic methods for advection-diffusion equations and comparison to other schemes. *Adv Wat Resour*, 22(7), 1999.
- [7] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri. Implicit-explicit Runge–Kutta methods for time-dependent partial differential equations. *Appl Numer Math*, 25(2-3):151–167, 1997. Special issue on time integration (Amsterdam, 1996).
- [8] R. Bermejo and J. Carpio. An adaptive finite element semi-Lagrangian implicit-explicit Runge–Kutta–Chebyshev method for convection dominated reaction-diffusion problems. *Appl Numer Math*, 58(1):16–39, 2008.
- [9] R. Bermejo and M. El Amrani. A finite element semi-Lagrangian explicit Runge–Kutta–Chebyshev method for convection dominated reaction-diffusion problems. *J Comput Appl Math*, 154(1):27–61, 2003.
- [10] P. Bogacki and L. F. Shampine. An efficient Runge–Kutta (4, 5) pair. *Comput Math Appl*, 32(6):15–28, 1996.
- [11] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language. *World Wide Web J*, 2:29–66, November 1997.
- [12] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh. VODE: a variable-coefficient ODE solver. *SIAM J Sci Stat Comput*, 10(5):1038–1051, 1989.
- [13] J. C. Butcher. On Runge–Kutta processes of high order. *J Austral Math Soc*, 4:179–194, 1964.
- [14] J. C. Butcher. The nonexistence of ten-stage eighth-order explicit Runge–Kutta methods. *BIT*, 25(3):521–540, 1985.
- [15] J. C. Butcher. *The numerical analysis of ordinary differential equations: Runge–Kutta and general linear methods*. A Wiley-Interscience Publication. John Wiley & Sons Ltd., 1987.
- [16] J. C. Butcher and W. M. Wright. The construction of practical general linear methods. *BIT*, 43(4):695–721, 2003.

- [17] J. H. E. Cartwright and O. Piro. The dynamics of Runge–Kutta methods. *Internat J Bifur Chaos Appl Sci Engrg*, 2(3):427–449, 1992.
- [18] E. Celledoni. Eulerian and semi-Lagrangian schemes based on commutator-free exponential integrators. In *Group theory and numerical analysis*, volume 39 of *CRM Proc & Lect Note*, pages 77–90. Amer Math Soc, Providence, RI, 2005.
- [19] E. Celledoni. MA8404-notes on IMEX methods and exponential integrators. <http://www.math.ntnu.no/~elenac/imexexpint.pdf>, 2009.
- [20] E. Celledoni, A. Iserles, S. P. Nørsett, and B. Orel. Complexity theory for Lie group solvers. *J Complex*, 18(1):242–286, 2002.
- [21] E. Celledoni and B. K. Kometa. Order conditions for the semi-Lagrangian exponential integrators. Technical report, Norwegian University of Science and Technology, 2009.
- [22] E. Celledoni and B. K. Kometa. Semi-Lagrangian Runge–Kutta exponential integrators for convection dominated problems. *J Sci Comput*, 41(1):139–164, 2009.
- [23] E. Celledoni and B. K. Kometa. Semi-Lagrangian multistep exponential integrators for index 2 differential algebraic systems. *J Comput Phys*, 230(9):3413–3429, 2011.
- [24] E. Celledoni, A. Marthinsen, and B. Owren. Commutator-free Lie group methods. *Future Gener Comput Syst*, 19(3):341–352, 2003.
- [25] M. Chhay and A. Hamdouni. Lie symmetry preservation by finite difference schemes for the Burgers equation. *Symmetry*, 2(2):868–883, 2010.
- [26] G. J. Cooper and A. Sayfy. Additive Runge–Kutta methods for stiff ordinary differential equations. *Math Comp*, 40(161):207–218, 1983.
- [27] G. J. Cooper and J. H. Verwer. Some explicit Runge–Kutta methods of high order. *SIAM J Numer Anal*, 9:389–405, 1972.
- [28] G. Corliss and Y. Chang. Solving ordinary differential equations using Taylor series. *ACM Trans Math Software*, 8(2):114–144, 1982.
- [29] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen Differenzgleichungen der mathematischen Physik. *Math Ann*, 100(1):32–74, 1928.
- [30] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM J Res Develop*, 11:215–234, 1967.
- [31] P. E. Crouch and R. Grossman. Numerical integration of ordinary differential equations on manifolds. *J Nonlinear Sci*, 3(1):1–33, 1993.
- [32] K. S. Das and A. J. Weaver. Semi-Lagrangian advection algorithms for ocean circulation models. *J Atmos Ocean Tech*, 12(4):935–950, 1995.
- [33] R. Dean. Numerical methods for simulation of electrical activity in the myocardial tissue. Master’s thesis, Department of Computer Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, 2009.
- [34] M. El-Amrani and M. Seaïd. Eulerian–Lagrangian time-stepping methods for convection-dominated problems. *Int J Comput Math*, 85(3-4):421–439, 2008.
- [35] W. H. Enright, T. E. Hull, and B. Lindenberg. Comparing numerical methods for stiff systems of ODEs. *BIT*, 15(2):10–48, 1975.
- [36] R. E. Ewing and H. Wang. An optimal-order estimate for Eulerian–Lagrangian localized adjoint methods for variable-coefficient advection-reaction problems. *SIAM J Numer Anal*, 33(1):318–348, 1996.

- [37] T. Feagin. High-order explicit Runge–Kutta methods. <http://sce.uhcl.edu/rungekutta/>.
- [38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1994.
- [39] K. Gustafsson. Control-theoretic techniques for stepsize selection in implicit Runge–Kutta methods. *ACM T Math Software*, 20(4):496–517, 1994.
- [40] E. Hairer. A Runge–Kutta method of order 10. *J Inst Math Appl*, 21(1):47–59, 1978.
- [41] E. Hairer. Origin of the term elementary differentials. Personal communication, August 2009.
- [42] E. Hairer, C. Lubich, and G. Wanner. *Geometric numerical integration*, volume 31 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 2002. Structure-preserving algorithms for ordinary differential equations.
- [43] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I: Nons-tiff problems.*, volume 8 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 1993.
- [44] 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, 1996.
- [45] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
- [46] D. J. Higham and L. N. Trefethen. Stiffness of ODEs. *BIT*, 33(2):285–303, 1993.
- [47] M. Hochbruck, C. Lubich, and H. Selhofer. Exponential integrators for large systems of differential equations. *SIAM J Sci Comput*, 19(5):1552–1574 (electronic), 1998.
- [48] M. Hochbruck and J. Niehoff. Approximation of matrix operators applied to multiple vectors. *Math Comput Simulat*, 79(4):1270–1283, 2008.
- [49] Y. C. Hon and X. Z. Mao. An efficient numerical scheme for Burgers’ equation. *Appl Math Comput*, 95(1):37–50, 1998.
- [50] T. E. Hull, W. H. Enright, B. Fellen, and A. E. Sedgwick. Comparing numerical methods for ordinary differential equations. *SIAM J Num Anal*, 9(4):603–607, 1972.
- [51] W. Hundsdorfer and J. G. Verwer. *Numerical solution of time-dependent advection-diffusion-reaction equations*. Springer-Verlag, Berlin, 2003.
- [52] P. E. Hydon. *Symmetry methods for differential equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2000. A beginner’s guide.
- [53] Z. Jackiewicz and S. Tracogna. A general class of two-step Runge–Kutta methods for ordinary differential equations. *SIAM J Numer Anal*, 32(5):1390–1427, 1995.
- [54] C. T. Kelley. *Iterative methods for linear and nonlinear equations*, volume 16 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995.
- [55] C. A. Kennedy and M. H. Carpenter. Additive Runge–Kutta schemes for convection–diffusion–reaction equations. *Appl Numer Math*, 44(1-2):139–181, 2003.
- [56] I. P. E. Kinnmark and W. G. Gray. One step integration methods of third-fourth order accuracy with large hyperbolic stability limits. *Math Comput Simulat*, 26(3):181–188, 1984.
- [57] I. P. E. Kinnmark and W. G. Gray. One step integration methods with maximum stability regions. *Math Comput Simulat*, 26(2):87–92, 1984.

- [58] I. P. E. Kinnmark and W. G. Gray. Fourth-order accurate one-step integration methods with large imaginary stability limits. *Numer Meth Part D E*, 2(1):63–70, 1986.
- [59] J. T. Kirby, G. Wei, Q. Chen, A. B. Kennedy, and R. A. Dalrymple. FUNWAVE 1.0 fully non-linear Boussinesq wave model documentation and user’s manual. Technical report, University of Delaware, 1998.
- [60] S. Krogstad. Generalized integrating factor methods for stiff PDEs. *J Comput Phys*, 203(1):72–88, 2005.
- [61] J. D. Lambert. *Numerical methods for ordinary differential systems*. John Wiley & Sons Ltd., Chichester, 1991. The initial value problem.
- [62] J. D. Lawson. An order five Runge–Kutta process with extended region of stability. *SIAM J Numer Anal*, 3:593–597, 1966.
- [63] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans Program Lang Syst*, 16(6):1811–1841, 1994.
- [64] C. Luo and Y. Rudy. A model of ventricular cardiac action potential. *Circ Res*, 68(6):1501–1526, 1991.
- [65] Y. Maday, A. T. Patera, and E. M. Rønquist. An operator-integration-factor splitting method for time-dependent problems: application to incompressible fluid flow. *J Sci Comput*, 5(4):263–292, 1990.
- [66] A. Malevsky and S. Thomas. Parallel algorithms for semi-Lagrangian advection. *Int J Numer Meth Fl*, 25(4):455–473, 1997.
- [67] R. C. Martin. Design principles and design patterns. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.
- [68] F. Mazzia and C. Magherini. Test set for initial value problem solvers, release 2.4. Technical Report 4, Department of Mathematics, University of Bari, Italy, 2008.
- [69] R. I. McLachlan and R. Quispel. Splitting methods. *Acta Numer*, 11:341–434, 2002.
- [70] Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467, 2010.
- [71] B. V. Minchev and W. Wright. A review of exponential integrators for first order semi-linear problems. *Preprint Numerics*, 2, 2005.
- [72] K. W. Morton and D. F. Mayers. *Numerical solution of partial differential equations*. Cambridge University Press, New York, NY, USA, 1995.
- [73] H. Munthe-Kaas. Runge–Kutta methods on Lie groups. *BIT*, 38(1):92–111, 1998.
- [74] H. Munthe-Kaas. High order Runge–Kutta methods on manifolds. In *Proceedings of the NSF/CBMS regional conference on numerical analysis of Hamiltonian differential equations (Golden, CO, 1997)*, volume 29, pages 115–127, 1999.
- [75] B. Owren. Order conditions for commutator-free Lie group methods. *J Phys A*, 39(19):5585–5599, 2006.
- [76] B. Owren and A. Marthinsen. Runge–Kutta methods adapted to manifolds and based on rigid frames. *BIT*, 39(1):116–142, 1999.
- [77] O. Pironneau. On the transport-diffusion algorithm and its applications to the Navier–Stokes equations. *Numer Math*, 38(3):309–332, 1982.

- [78] D. A. Pope. An exponential method of numerical integration of ordinary differential equations. *Comm ACM*, 6:491–493, 1963.
- [79] J. A. Pudykiewicz. Numerical solution of the reaction-advection-diffusion equation on the sphere. *J Comput Phys*, 213(1):358–390, 2006.
- [80] K. Radhakrishnan and A. C. Hindmarsh. Description and use of LSODE, the Livermore solver for ordinary differential equations. Technical report, UCRL-ID-113855, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 1993.
- [81] J. R. Rice and R. F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Comput Sci Eng*, 3(3):44–53, 1996.
- [82] A. Rigo. Representation-based just-in-time specialization and the Psycho prototype for Python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM.
- [83] Y. Saad. Analysis of some Krylov subspace approximations to the matrix exponential operator. *SIAM J Numer Anal*, 29(1):209–228, 1992.
- [84] R. R. Schaller. Moore’s law: past, present, and future. *IEEE Spectr*, 34(6):52–59, 1997.
- [85] H. Sellers, R. J. Spiteri, and M. Perrone. CO₂ + CH₄ chemistry over Pd: Results of kinetic simulations relevant to environmental issues. *J Phys Chem-US*, 113(6):2340–2346, 2009.
- [86] L. F. Shampine. *Numerical solution of ordinary differential equations*. Chapman & Hall, New York, 1994.
- [87] L. F. Shampine, I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge University Press, Cambridge, 2003.
- [88] L. F. Shampine and M. W. Reichelt. The MATLAB ODE suite. *SIAM J Sci Comput*, 18(1):1–22, 1997. Dedicated to C. William Gear on the occasion of his 60th birthday.
- [89] L. F. Shampine, B. P. Sommeijer, and J. G. Verwer. IRKC: an IMEX solver for stiff diffusion-reaction PDEs. *J Comput Appl Math*, 196(2):485–497, 2006.
- [90] Siraj-ul-Islam, A. Ali, and S. Haq. A computational modelling of the behavior of the two-dimensional reaction diffusion Brusselator system. *Appl Math Model*, pages 3896–3909, 2010.
- [91] J. D. Skufca. Analysis still matters: a surprising instance of failure of Runge–Kutta–Fehlberg ODE solvers. *SIAM Rev*, 46(4):729–737 (electronic), 2004.
- [92] G. Söderlind. Digital filters in adaptive time-stepping. *ACM Trans Math Software*, 29(1):1–26, 2003.
- [93] G. Söderlind and L. Wang. Evaluating numerical ODE/DAE methods, algorithms and software. *J Comput Appl Math*, 185(2):244–260, 2006.
- [94] B. P. Sommeijer, L. F. Shampine, and J. G. Verwer. RKC: an explicit solver for parabolic PDEs. *J Comput Appl Math*, 88(2):315–326, 1998.
- [95] R. J. Spiteri and T.-P. Ter. pythNon: a PSE for the numerical solution of nonlinear algebraic equations. *JNAIAM J Numer Anal Ind Appl Math*, 3(1-2):123–137, 2008.
- [96] A. Staniforth and J. Côté. Semi-Lagrangian integration schemes for atmospheric models: A review. *Mon Weather Rev*, 119(9):2206–2223, 1991.
- [97] L. N. Trefethen and D. Bau. *Numerical linear algebra*. Society for Industrial Mathematics, 1997.

- [98] E. H. Twizell, A. B. Gumel, and Q. Cao. A second-order scheme for the “Brusselator” reaction-diffusion system. *J Math Chem*, 26(4):297–316 (2000), 1999.
- [99] J. Tyson. Some further studies of nonlinear oscillations in chemical systems. *J Chem Phys*, 58:3919, 1973.
- [100] P. J. van der Houwen. Explicit Runge–Kutta formulas with increased stability boundaries. *Numer Math*, 20:149–164, 1972/73.
- [101] J. G. Verwer and B. P. Sommeijer. An implicit-explicit Runge–Kutta–Chebyshev scheme for diffusion-reaction equations. *SIAM J Sci Comput*, 25(5):1824–1835 (electronic), 2004.
- [102] J. G. Verwer, B. P. Sommeijer, and W. Hundsdorfer. RKC time-stepping for advection-diffusion-reaction problems. *J Comput Phys*, 201(1):61–79, 2004.
- [103] H. Wang, M. Al-Lawatia, and A. S. Telyakovskiy. Runge–Kutta characteristic methods for first-order linear hyperbolic equations. *Numer Meth Part D E*, 13(6):617–661, 1997.
- [104] W. Weckesser. VFGEN: A code generation toolkit. *JNAIAM*, 3(1-2):151–165, 2008.
- [105] D. Xiu and G. E. Karniadakis. A semi-Lagrangian high-order method for Navier–Stokes equations. *J Comput Phys*, 172(2):658–684, 2001.

APPENDIX A

EXAMPLES RELATED TO THE DERIVATION OF ORDER CONDITIONS

The rooted trees and corresponding elementary differentials corresponding to the fifth derivative of the RHS of an ODE (2.2a) are given by

$$\frac{d^5}{dt^5} \mathbf{f}(\mathbf{y}) = \sum_{\mathbf{g}^5 \in \mathbf{G}^5(\mathbf{f}(\mathbf{y}(t)))} \alpha(\mathbf{g}^5) \mathbf{g}^5 \sim \begin{array}{c} \bullet \\ \diagup \quad \diagdown \\ \bullet \quad \bullet \end{array} + 6 \begin{array}{c} \bullet \\ \diagup \quad \diagdown \\ \bullet \quad \bullet \\ | \\ \bullet \end{array} + 6 \begin{array}{c} \bullet \quad \bullet \\ \diagup \quad \diagdown \\ \bullet \quad \bullet \end{array} + 4 \begin{array}{c} \bullet \quad \bullet \\ \diagup \quad \diagdown \\ \bullet \quad \bullet \\ | \\ \bullet \end{array} + 3 \begin{array}{c} \bullet \\ \diagup \quad \diagdown \\ \bullet \quad \bullet \\ | \\ \bullet \end{array} + 4 \begin{array}{c} \bullet \\ | \\ \bullet \end{array} + 3 \begin{array}{c} \bullet \quad \bullet \\ \diagup \quad \diagdown \\ \bullet \quad \bullet \\ | \\ \bullet \end{array} + 3 \begin{array}{c} \bullet \quad \bullet \\ \diagup \quad \diagdown \\ \bullet \quad \bullet \\ | \\ \bullet \end{array} + \begin{array}{c} \bullet \\ | \\ \bullet \\ | \\ \bullet \\ | \\ \bullet \\ | \\ \bullet \end{array}$$

See Section 2.4 for description of the notion and theory. The expansion of the elementary differentials for certain trees with $m = 2$ are given below.

APPENDIX B

REPRESENTATIVE BEHAVIOUR OF THE BURGERS, BRUSSELATOR, AND BURGERS-BRUSSELATOR EQUATIONS

The following figures are representative of the time evolution of the Burgers (4.5), Brusselator (4.24), and Burgers-Brusselator (4.25) equations with some parameters of interest for this thesis.

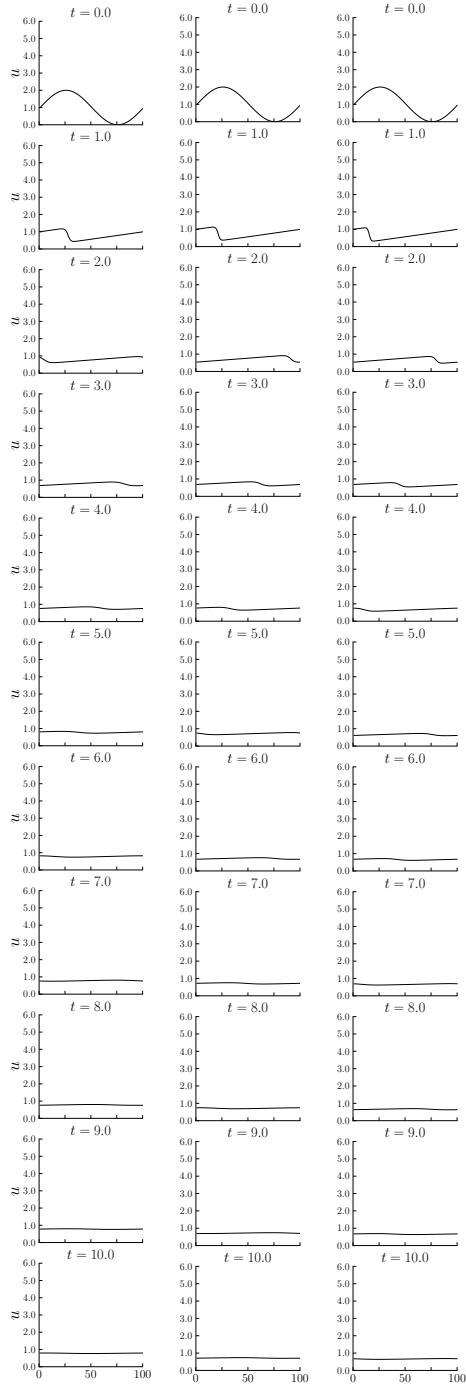


Figure B.1: Solutions to the Burgers equation (4.5) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.005, 0.002, 0.001\}$ (left to right), and $n_{\text{grid}} = 100$.

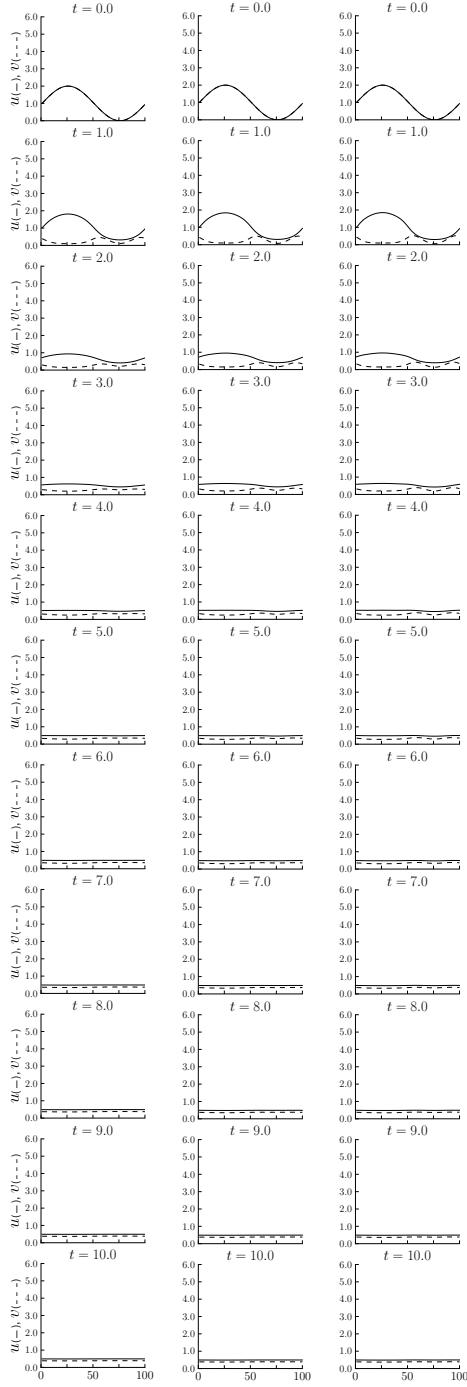


Figure B.2: Solutions to the Brusselator equation (4.24) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 0.5$, $\beta = 0.2$ (4.29a) that have asymptotically decaying behaviour at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$.

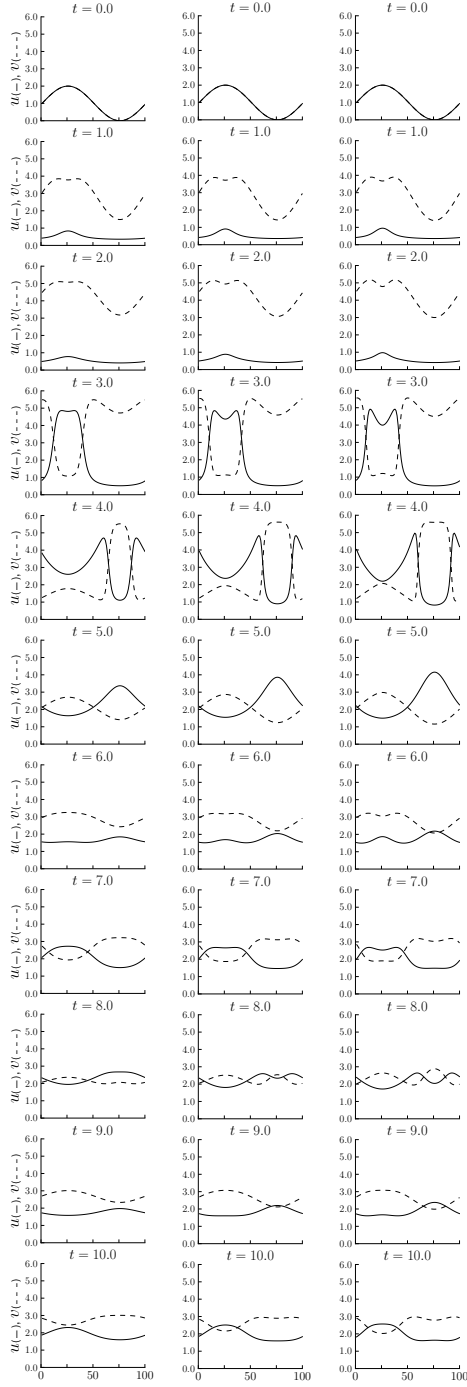


Figure B.3: Solutions to the Brusselator equation (4.24) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 2.0$, $\beta = 5.0$ (4.29b) that are stable and oscillatory at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$.

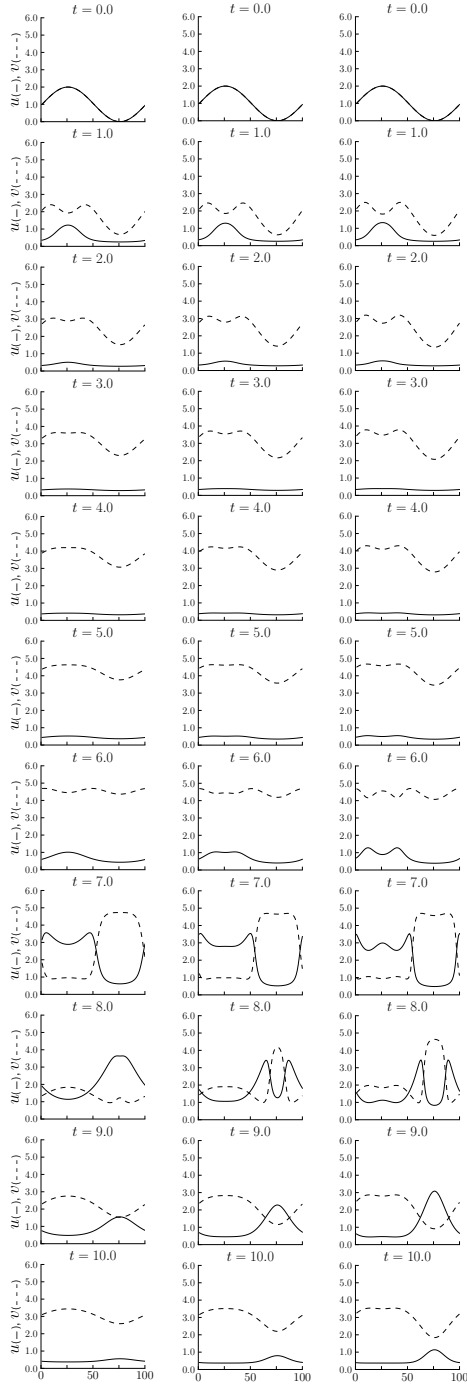


Figure B.4: Solutions to the Brusselator equation (4.24) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 1.0$, $\beta = 3.0$ (4.29c) that are unstable at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$.

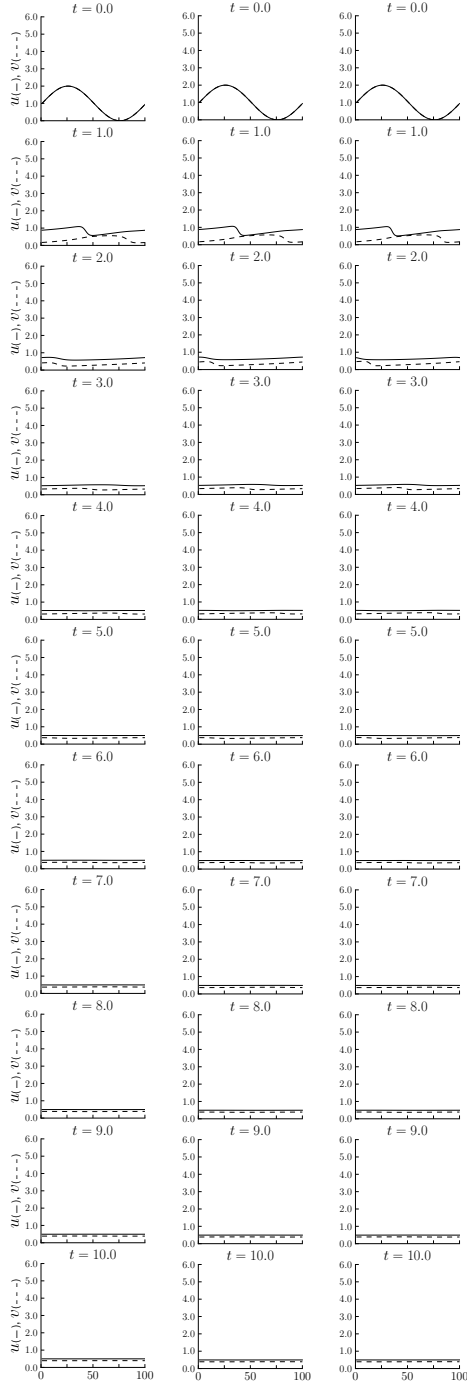


Figure B.5: Solutions to the Burgers–Brusselator equation (4.25) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 0.5$, $\beta = 0.2$ (4.29a) that have asymptotically decaying behaviour at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$.

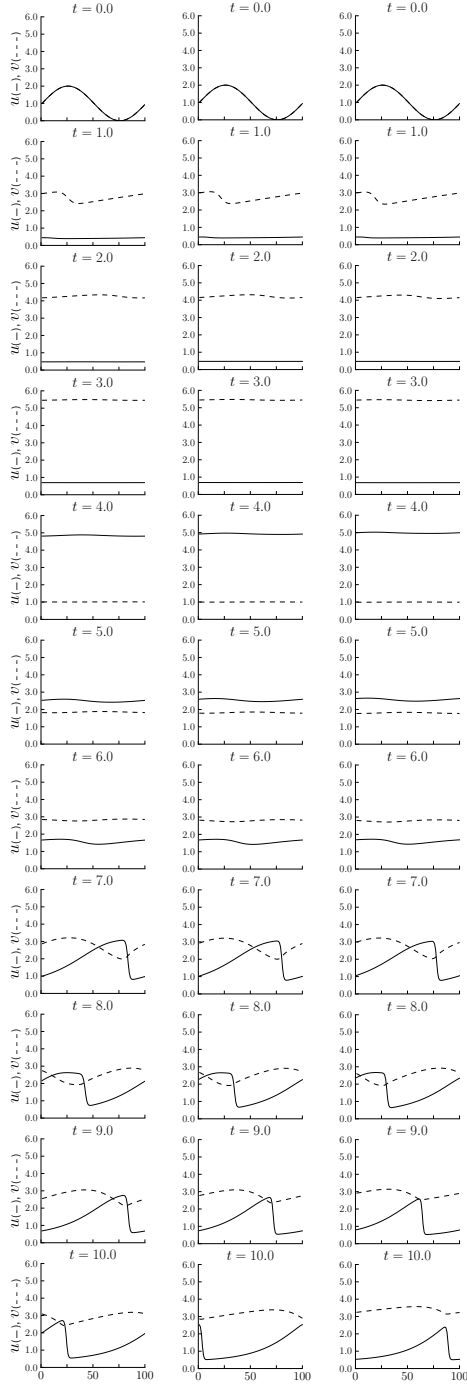


Figure B.6: Solutions to the Burgers–Brusselator equation (4.25) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 2.0$, $\beta = 5.0$ (4.29b) that are stable and oscillatory at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$.

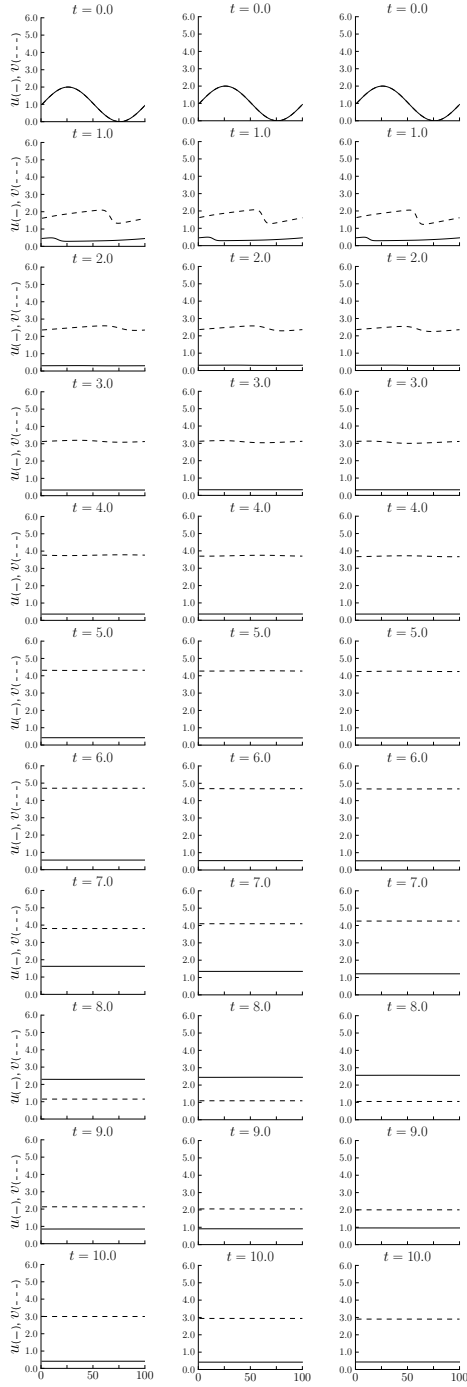


Figure B.7: Solutions to the Burgers–Brusselator equation (4.25) with the initial condition (4.28c), boundary conditions (4.28d), $\epsilon = \{0.002, 0.001, 0.0005\}$ (left to right), $\alpha = 1.0$, $\beta = 3.0$ (4.29c) that are unstable at the diffusion-free Brusselator steady state, and $n_{\text{grid}} = 100$.

APPENDIX C

DETAILS ON THE IF-2-ARK METHODS

C.1 Derivation of IF-2-ARK methods

The full derivation of the method (4.22) based on the OIF splitting of Section 4.3.1 is covered in this section. Substitute the auxiliary variables into (4.2) based on OIF splitting:

$$\begin{aligned} \mathbf{y}(t) &= \mathcal{W}(t) \cdot \mathbf{z}(t), \\ \frac{d}{dt} \mathbf{y}(t) &= \frac{d}{dt} \mathcal{W}(t) \cdot \mathbf{z}(t) + \mathcal{W}(t) \cdot \frac{d}{dt} \mathbf{z}(t), \\ \frac{d}{dt} \mathcal{W}(t) &= \mathcal{A}(\mathcal{W}(t) \cdot \mathbf{z}(t)) \cdot \mathcal{W}(t), \quad \mathcal{W}(t_0) = \mathbb{I}, \\ \frac{d}{dt} \mathbf{z}(t) &= \mathcal{W}^{-1}(t) \cdot \mathcal{D}(\mathcal{W}(t) \cdot \mathbf{z}(t)) \cdot \mathcal{W}(t) \cdot \mathbf{z}(t) + \mathcal{W}^{-1}(t) \cdot \mathbf{r}(\mathcal{W}(t) \cdot \mathbf{z}(t)), \end{aligned}$$

with further details given in Section 4.3.1 and [22]. Applying a CFERK method to $\mathcal{W}(t)$ and a 2-ARK method to $\mathbf{z}(t)$ yields

$$\begin{aligned} \mathcal{Q}_i &= \exp \left(\Delta t \sum_{j=1}^{i-1} \alpha_{ij}^{[2]} \mathcal{A}(\mathcal{Q}_j \cdot \mathbf{Z}_j) \right) \cdot \exp \left(\Delta t \sum_{j=1}^{i-1} \alpha_{ij}^{[1]} \mathcal{A}(\mathcal{Q}_j \cdot \mathbf{Z}_j) \right) \cdot \mathcal{W}_n, \quad i = 1, 2, \dots, s, \\ \mathbf{Z}_i &= \mathbf{z}_n + \Delta t \sum_{j=1}^s (\mathbf{A}_{\mathcal{D}})_{ij} \mathcal{Q}_j^{-1} \cdot \mathcal{D}(\mathcal{Q}_j \cdot \mathbf{Z}_j) \cdot \mathcal{Q}_j \cdot \mathbf{z}_j + \Delta t \sum_{j=1}^s (\mathbf{A}_{\mathbf{r}})_{ij} \mathcal{Q}_j^{-1} \cdot \mathbf{r}(\mathcal{Q}_j \cdot \mathbf{Z}_j), \quad i = 1, 2, \dots, s. \\ \mathcal{W}_{n+1} &= \exp \left(\Delta t \sum_{i=1}^s \beta_i^{[2]} \mathcal{A}(\mathcal{Q}_i \cdot \mathbf{Z}_i) \right) \cdot \exp \left(\Delta t \sum_{i=1}^s \beta_i^{[1]} \mathcal{A}(\mathcal{Q}_i \cdot \mathbf{Z}_i) \right) \cdot \mathcal{W}_n, \quad i = 1, 2, \dots, s, \\ \mathbf{z}_{n+1} &= \mathbf{z}_n + \Delta t \sum_{i=1}^s (\mathbf{b}_{\mathcal{D}})_i \mathcal{Q}_i^{-1} \cdot \mathcal{D}(\mathcal{Q}_i \cdot \mathbf{Z}_i) \cdot \mathcal{Q}_i \cdot \mathbf{z}_i + \Delta t \sum_{i=1}^s (\mathbf{b}_{\mathbf{r}})_i \mathcal{Q}_i^{-1} \cdot \mathbf{r}(\mathcal{Q}_i \cdot \mathbf{Z}_i), \quad i = 1, 2, \dots, s, \end{aligned}$$

where $(\mathbf{A}_{\mathcal{A}})_{i,j} = \alpha_{i,j}^{[1]} + \alpha_{i,j}^{[2]}$, $(\mathbf{b}_{\mathcal{A}})_i = \beta_i^{[1]} + \beta_i^{[2]}$ and the rest of the variables are given in the definition of (4.22).

Some further variable substitutions yield

$$\mathbf{Y}_i := \mathcal{Q}_i \cdot \mathbf{Z}_i, \quad \rho_i := \mathcal{Q}_i \cdot \mathcal{W}_n^{-1}, \quad \mathbf{y}_{n+1} := \mathcal{W}_{n+1} \cdot \mathbf{z}_{n+1}, \quad \rho_{n+1} := \mathcal{W}_{n+1} \cdot \mathcal{W}_n^{-1}, \quad i = 1, 2, \dots, s,$$

which results in a method free of the auxiliary variables and given by

$$\begin{aligned}\rho_i \cdot \mathcal{W}_n &= \exp \left(\Delta t \sum_{j=1}^{i-1} \alpha_{ij}^{[2]} \mathcal{A}(\mathbf{Y}_j) \right) \cdot \exp \left(\Delta t \sum_{j=1}^{i-1} \alpha_{ij}^{[1]} \mathcal{A}(\mathbf{Y}_j) \right) \cdot \mathcal{W}_n, \quad i = 1, 2, \dots, s, \\ \rho_i &= \exp \left(\Delta t \sum_{j=1}^{i-1} \alpha_{ij}^{[2]} \mathcal{A}(\mathbf{Y}_j) \right) \cdot \exp \left(\Delta t \sum_{j=1}^{i-1} \alpha_{ij}^{[1]} \mathcal{A}(\mathbf{Y}_j) \right), \quad i = 1, 2, \dots, s,\end{aligned}$$

$$\rho_{i,j} := \rho_i \cdot \rho_j^{-1},$$

$$\begin{aligned}\mathcal{Q}_i^{-1} \cdot \mathbf{Y}_i &= \mathcal{W}_n^{-1} \cdot \mathbf{y}_n + \Delta t \sum_{j=1}^s (\mathbf{A}_{\mathcal{D}})_{i,j} \mathcal{Q}_j^{-1} \cdot \mathcal{D}(\mathbf{Y}_j) \cdot \mathbf{Y}_j + \Delta t \sum_{j=1}^s (\mathbf{A}_{\mathbf{r}})_{i,j} \mathcal{Q}_j^{-1} \cdot \mathbf{r}(\mathbf{Y}_j), \quad i = 1, 2, \dots, s, \\ \mathbf{Y}_i &= \mathcal{Q}_i \cdot \mathcal{W}_n^{-1} \cdot \mathbf{y}_n + \Delta t \sum_{j=1}^s (\mathbf{A}_{\mathcal{D}})_{i,j} \mathcal{Q}_i \cdot \mathcal{Q}_j^{-1} \cdot \mathcal{D}(\mathbf{Y}_j) \cdot \mathbf{Y}_j + \Delta t \sum_{j=1}^s (\mathbf{A}_{\mathbf{r}})_{i,j} \mathcal{Q}_i \cdot \mathcal{Q}_j^{-1} \cdot \mathbf{r}(\mathbf{Y}_j), \\ i &= 1, 2, \dots, s, \\ \mathbf{Y}_i &= \rho_i \cdot \mathbf{y}_n + \Delta t \sum_{j=1}^s (\mathbf{A}_{\mathcal{D}})_{i,j} \rho_{i,j} \cdot \mathcal{D}(\mathbf{Y}_j) \cdot \mathbf{Y}_j + \Delta t \sum_{j=1}^s (\mathbf{A}_{\mathbf{r}})_{i,j} \rho_{i,j} \cdot \mathbf{r}(\mathbf{Y}_j), \quad i = 1, 2, \dots, s,\end{aligned}$$

$$\begin{aligned}\rho_{n+1} \cdot \mathcal{W}_n &= \exp \left(\Delta t \sum_{i=1}^s \beta_i^{[2]} \mathcal{A}(\mathbf{Y}_i) \right) \cdot \exp \left(\Delta t \sum_{i=1}^s \beta_i^{[1]} \mathcal{A}(\mathbf{Y}_i) \right) \cdot \mathcal{W}_n. \\ \rho_{n+1} &= \exp \left(\Delta t \sum_{i=1}^s \beta_i^{[2]} \mathcal{A}(\mathbf{Y}_i) \right) \cdot \exp \left(\Delta t \sum_{i=1}^s \beta_i^{[1]} \mathcal{A}(\mathbf{Y}_i) \right),\end{aligned}$$

$$\rho_{n+1,i} := \rho_{n+1} \cdot \rho_i^{-1},$$

$$\begin{aligned}\mathcal{W}_{n+1}^{-1} \cdot \mathbf{y}_{n+1} &= \mathcal{W}_n^{-1} \cdot \mathbf{y}_n + \Delta t \sum_{i=1}^s (\mathbf{b}_{\mathcal{D}})_i \mathcal{Q}_i^{-1} \cdot \mathcal{D}(\mathbf{Y}_i) \cdot \mathbf{Y}_i + \Delta t \sum_{i=1}^s (\mathbf{b}_{\mathbf{r}})_i \mathcal{Q}_i^{-1} \cdot \mathbf{r}(\mathbf{Y}_i). \\ \mathbf{y}_{n+1} &= \mathcal{W}_{n+1} \cdot \mathcal{W}_n^{-1} \cdot \mathbf{y}_n + \Delta t \sum_{i=1}^s (\mathbf{b}_{\mathcal{D}})_i \mathcal{W}_i \cdot \mathcal{Q}_i^{-1} \cdot \mathcal{D}(\mathbf{Y}_i) \cdot \mathbf{Y}_i + \Delta t \sum_{i=1}^s (\mathbf{b}_{\mathbf{r}})_i \mathcal{W}_i \cdot \mathcal{Q}_i^{-1} \cdot \mathbf{r}(\mathbf{Y}_i). \\ \mathbf{y}_{n+1} &= \rho_{n+1} \cdot \mathbf{y}_n + \Delta t \sum_{i=1}^s (\mathbf{b}_{\mathcal{D}})_i \rho_{n+1,i} \cdot \mathcal{D}(\mathbf{Y}_i) \cdot \mathbf{Y}_i + \Delta t \sum_{i=1}^s (\mathbf{b}_{\mathbf{r}})_i \rho_{n+1,i} \cdot \mathbf{r}(\mathbf{Y}_i).\end{aligned}$$

C.2 Tableaux of constituent methods used to propose IF-2-ARK methods

The linearly implicit IMEX (2, 3, 2) from [7], used in the experiments in Section 4.7, is given by

$$\begin{array}{c|ccc|ccc}
 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \gamma & 0 & \gamma & 0 & \gamma & \gamma & 0 & 0 \\
 1 & 0 & 1-\gamma & \gamma & 1 & \delta & 1-\delta & 0 \\
 \hline
 & 0 & 1-\gamma & \gamma & & 0 & 1-\gamma & \gamma
 \end{array} \tag{C.4}$$

where $\gamma = (2 - \sqrt{2})/2$, $\delta = -2\sqrt{2}/3$. These parameters give the explicit method the stability region of a three-stage, third-order method, and an L -stable implicit method.