A Study of 2-Additive Splitting for Solving Advection-Diffusion-Reaction 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

Adam Preuss

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

An initial-value problem consists of an ordinary differential equation subject to an initial condition. The right-hand side of the differential equation can be interpreted as additively split when it is comprised of the sum of two or more contributing factors. For instance, the right-hand sides of initial-value problems derived from advection-diffusion-reaction equations are comprised of the sum of terms emanating from three distinct physical processes: advection, diffusion, and reaction. In some cases, solutions to initial-value problems can be calculated analytically, but when an analytic solution is unknown or nonexistent, methods of numerical integration are used to calculate solutions. The runtime performance of numerical methods is problem dependent; therefore, one must choose an appropriate numerical method to achieve favourable performance, according to characteristics of the problem. Additive methods of numerical integration apply distinct methods to the distinct contributing factors of an additively split problem. Treating the contributing factors with methods that are known to perform well on them individually has the potential to yield an additive method that outperforms single methods applied to the entire (unsplit) problem. Splittings of the right-hand side can be physics-based, i.e., based on physical characteristics of the problem, such as advection, diffusion, or reaction terms. Splittings can also be based on linearization, called Jacobian splitting in this thesis, where the linearized part of the problem is treated with one method and the rest of the problem is treated with another. A comparison of these splitting techniques is performed by applying a set of additive methods to a test suite of problems. Many common non-additive methods are also included to serve as a performance baseline. To perform this numerical study, a problem-solving environment was developed to evaluate permutations of problems, methods, and their associated parameters. The test suite is comprised of several distinct advection-diffusion-reaction equations that have been chosen to represent a wide range of common problem characteristics. When solving split problems in the test suite, it is found that additive Runge–Kutta methods of orders three, four, and five using Jacobian splitting generally outperform those same methods using physics-based splitting. These results provide evidence that Jacobian splitting is an effective approach when solving such initial-value problems in practice.

Acknowledgements

Thanks to Professor Raymond Spiteri for his guidance, funding, all of our helpful conversations and, in general, for his vision of this project. His research area offers an enjoyable combination of math, computer science, and many other scientific disciplines. Thanks to all the members of the Numerical Simulation Laboratory for their support, advice, and friendly chit-chat. Thank you to the department of computer science for its funding and resources. Thank you to my fiancée, Kaylee Bohaychuk, for her support and for putting up with me being a workaholic for the past few months. Thank you to my parents, Lowell Preuss and Shelagh Watson-Preuss, for encouraging my academic pursuits since before I can remember. Special thanks to my uncle, Iain Watson, for first introducing me to computer programming years ago. Finally, I offer thanks to my friends who are always in favour of a much-needed beer after long hours of work and research.

To Lowell, Shelagh, and Kaylee.

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents v

List of Tables viii

List of Figures ix

List of Abbreviations xii

1 Introduction 1
- 1.1 Some Quick Words on Notation . . . 7
- 1.2 Outline of the Thesis . . . 8

2 Theoretical Background 9
- 2.1 Autonomous Form . . . 10
- 2.2 Accuracy and Tolerances . . . 10
- 2.3 Runge–Kutta Methods . . . 12
  - 2.3.1 Classes of Runge–Kutta Methods . . . 14
  - 2.3.2 Error Estimation . . . 17
  - 2.3.3 Step Control . . . 19
  - 2.3.4 Stability and Stiffness . . . 21
  - 2.3.5 Runge–Kutta–Chebyshev Methods . . . 25
  - 2.3.6 Rosenbrock Methods . . . 29
  - 2.3.7 Exponential Runge–Kutta Methods . . . 31
- 2.4 Methods for 2-Additively Split Problems . . . 32
  - 2.4.1 Additive Runge–Kutta Methods . . . 33
  - 2.4.2 Stability for 2-Additive Runge–Kutta Methods . . . 35
  - 2.4.3 The Partitioned Runge–Kutta–Chebyshev Method . . . 36
  - 2.4.4 The Implicit-Explicit Runge–Kutta–Chebyshev Method . . . 38
  - 2.4.5 Additive Exponential Runge–Kutta Methods . . . 40

3 Methods and Software 43
- 3.1 Details of `pythODE++` . . . 44
  - 3.1.1 Classes of Methods . . . 46
  - 3.1.2 Software Components . . . 47
  - 3.1.3 Supporting Classes . . . 51

3.1.4 Sparsity . . . 53
3.1.5 Automatic Differentiation . . . 54
3.1.6 Solving Initial-Value Problems Simultaneously . . . 55
3.1.7 Analysis . . . 57
3.2 Discretization of Partial Differential Equations . . . 59
3.2.1 Finite Difference Methods . . . 59
3.2.2 Finite Volume Methods . . . 60
3.3 Advection-Diffusion-Reaction Equations . . . 61
3.4 Test Suite of Problems . . . 63
3.4.1 Advection-Diffusion Problems . . . 65
3.4.2 Diffusion-Reaction Problems . . . 67
3.4.3 Advection-Diffusion-Reaction Problems . . . 70

4 Results of Numerical Experiments **76**
4.1 Cluster Specifications . . . 76
4.2 Findings regarding Interprocess Communication . . . 77
4.3 Sparse Jacobian Computation . . . 79
4.4 Verification, Validation, and Solution Plots . . . 79
4.4.1 Methods . . . 80
4.4.2 Problem Suite . . . 81
4.5 Comparisons of Numerical Methods . . . 88
4.5.1 Advection-Diffusion Problems . . . 89
4.5.2 Reaction-Diffusion Problems . . . 92
4.5.3 Advection-Diffusion-Reaction Problems . . . 95

5 Conclusions and Future Work **104**
5.1 Summary of Results . . . 104
5.2 Contributions of this Thesis . . . 105
5.3 Future Work . . . 106
5.3.1 Extension to Three-Dimensional Models . . . 106
5.3.2 Merging `pythODE++` and `pythODE` . . . 107
5.3.3 Extension to 3-Additive Methods . . . 107
5.3.4 Additional Methods . . . 108
5.3.5 Parallelized Methods . . . 108

References **109**

A Derivations of Order Conditions for Runge–Kutta Methods **113**
A.1 Taylor Expansion of the True Solution . . . 113
A.2 Taylor Expansion of the Numerical Solution . . . 114
A.3 Order Conditions . . . 115

B Derivations of Finite Difference Methods **116**

C Derivation of Additive Exponential Runge–Kutta Methods **117**

D Order Conditions for Additive Exponential Runge–Kutta Methods **119**
D.1 Derivatives of the Exact Solution 119
D.2 Derivatives of the Numerical Solution 120
D.2.1 First Derivatives 120
D.2.2 Second Derivatives 121
D.3 Order Conditions 122

E Steps versus Accuracy Plots **123**

F Eigenvalue Plots **131**

List of Tables

2.1 The Butcher tableau of an RK method. 14
2.2 Butcher tableau for RK4. 16
2.3 Butcher tableau for the implicit fourth-order Gauss method. 16
2.4 The Butcher tableaux for a 2-additive RK method. 34
2.5 Butcher tableaux for a third-order ARK method. 34

3.1 The methods used in this thesis. 47
3.2 Components to define an IVP in the `pythODE++` PSE. 48
3.3 Components to define a 2-additively split IVP in the `pythODE++` PSE. 49
3.4 Components to define a method in the `pythODE++` PSE. 50
3.5 List of all numerical studies. These are all ADR equations that are spatially discretized and solved in `pythODE++`. The last column gives the method that was used to calculate reference solutions. 64

List of Figures

2.1 Regions of absolute stability, calculated from the stability function, for Runge–Kutta methods ranging from order two to five. . . . 23
2.2 Regions of absolute stability, calculated from the stability function, for the forward and backward Euler methods. . . . 24
2.3 Region of absolute stability, calculated from the stability function, for the RADAU5 method. . . . 25
2.4 Regions of absolute stability, calculated from the stability function, for the RKC1 method of stages two through seven. . . . 28
2.5 Regions of absolute stability, calculated from the stability function, for the RKC2 method of stages two through seven. . . . 29
2.6 Region of absolute stability, calculated from the stability function, for the RODAS method. . . . 31
2.7 Regions of absolute stability, calculated from the stability function, for IMEX methods ranging from orders three to five. . . . 37
2.8 Regions of absolute stability, calculated from the stability function, for the PRKC method of stages two through seven. . . . 39
2.9 Regions of absolute stability, calculated from the stability function, for the IRKC method of stages two through seven. . . . 40
2.10 Butcher tableaux for the DIRK-CF1 and DIRK-CF2 methods. The classical RK tableaux are given on the left and the ExpRK tableaux are given on the right. . . . 42

3.1 Software organization of the `pythODE++` PSE 51

4.1 Solution plots for the linear advection-diffusion equation defined by (3.7), spatially discretized using 500 points. . . . 81
4.2 Solution plots for the heat transfer equation defined by (3.9), spatially discretized using 280x40 points. . . . 82
4.3 Solution plot for the `CUSP` problem defined by (3.10), using 32 discretized grid points. . . . 83
4.4 Solution plots for the one-dimensional Brusselator equation defined by (3.11), using 1000 discretized points. . . . 83
4.5 Solution plot for the two-dimensional Brusselator equation defined by (3.11), using 60x60 discretized points. The function $u(x, y, t)$ is given by the red meshes and the function $v(x, y, t)$ is given by blue meshes. . . . 84
4.6 Solution plots for the combustion equations defined by (3.12), using 1600 discretized points. . . . 85
4.7 Solution plots for the tumour angiogenesis model defined by (3.13). The domain has been discretized using 1000 points. Plots are shown for times 0, 0.1, 0.3, 0.5, 0.7, moving from light to dark. . . . 86

4.8 Solution plots for the concrete-rewetting problem. The hydration front is shown for 10 equally spaced times in [0,28], moving from light to dark. . . . 87
4.9 CPU time versus accuracy of IMEX and RKC methods applied to a series of one-dimensional advection-diffusion problem. Tolerances range from 10^{-4} to 10^{-8}. . . . 91
4.10 CPU time versus accuracy of IMEX and RKC methods applied to the heat transfer problem. Tolerances range from 10^{-4} to 10^{-8}. . . . 93
4.11 CPU time versus accuracy of IMEX and RKC methods applied to the `CUSP` problem. Tolerances range from 10^{-4} to 10^{-8}. . . . 94
4.12 CPU time versus accuracy of IMEX and RKC methods applied to a one-dimensional Brusselator problems. Tolerances range from 10^{-4} to 10^{-8}. . . . 96
4.13 CPU time versus accuracy of IMEX and RKC methods applied to a two-dimensional Brusselator problems. Tolerances range from 10^{-4} to 10^{-8}. . . . 97
4.14 CPU time versus accuracy of IMEX and RKC methods applied to the set of one-dimensional combustion problems. Tolerances range from 10^{-4} to 10^{-8}. . 99
4.15 CPU time versus accuracy of IMEX and RKC methods applied to the tumour angiogenesis model. Tolerances range from 10^{-5} to 10^{-11}. . . . 100
4.16 CPU time versus accuracy of IMEX and RKC methods applied to the concrete rewetting problem for both insulated and sink boundaries. Tolerances range from 10^{-4} to 10^{-9}. . . . 103

E.1 The number of steps versus accuracy of IMEX and RKC methods applied to a series of one-dimensional advection-diffusion problem. Tolerances range from 10^{-4} to 10^{-8}. . . . 123
E.2 The number of steps versus accuracy of IMEX and RKC methods applied to the heat transfer problem. Tolerances range from 10^{-4} to 10^{-8}. . . . 124
E.3 The number of steps versus accuracy of IMEX and RKC methods applied to the `CUSP` problem. Tolerances range from 10^{-4} to 10^{-8}. . . . 125
E.4 The number of steps versus accuracy of IMEX and RKC methods applied to a one-dimensional Brusselator problems. Tolerances range from 10^{-4} to 10^{-8}. 126
E.5 The number of steps versus accuracy of IMEX and RKC methods applied to a two-dimensional Brusselator problems. Tolerances range from 10^{-4} to 10^{-8}. 127
E.6 The number of steps versus accuracy of IMEX and RKC methods applied to the set of one-dimensional combustion problems. Tolerances range from 10^{-4} to 10^{-8}. . . . 128
E.7 The number of steps versus accuracy of IMEX and RKC methods applied to the tumour angiogenesis model. Tolerances range from 10^{-5} to 10^{-11}. . . . 129
E.8 The number of steps versus accuracy of IMEX and RKC methods applied to the concrete rewetting problem for both insulated and sink boundaries. Tolerances range from 10^{-4} to 10^{-9}. . . . 130

F.1 Distribution of eigenvalues for the linear advection-diffusion ($d = 1$, $a = 1/10$ and 64 unknowns) and the heat transfer problem (70×10 unknowns). . . . 131

F.2 Distribution of eigenvalues for the one-dimensional Brusselator ($\alpha = 1/50$ and 500 unknowns), the two-dimensional Brusselator ($\alpha = 1/50$ and 2500 unknowns), and the `CUSP` model ($\sigma = 1/144$ and 500 unknowns). 132
F.3 Distribution of eigenvalues for the combustion model, for $U_0 = 0.99$ and 40 unknowns. 133
F.4 Distribution of eigenvalues for the tumour angiogenesis problem for $d = 1$ and with 200 unknowns. 134
F.5 Distribution of eigenvalues for the concrete-rewetting problem using a either sink or insulated boundary condition, both with 100 unknowns. 135

List of Abbreviations

AD	Automatic differentiation
ARK	Additive Runge–Kutta
ADR	Advection-diffusion-reaction
BE	Backward Euler
CFERK	Commutator-free exponential Runge–Kutta
CRS	Compressed row storage
C-S-H	Calcium-silicate hydrate
DIRK	Diagonally implicit Runge–Kutta
ERK	Explicit Runge–Kutta
ExpRK	Exponential Runge–Kutta
FE	Forward Euler
FKPP	Fisher–Kolmogorov–Petrovskii–Piskunov
IMEX	Implicit-Explicit
IRK	Implicit Runge–Kutta
IRKC	Implicit Runge–Kutta–Chebyshev
IVP	Initial-value problem
MPI	Message-passing interface
ODE	Ordinary differential equation
PRKC	Partitioned Runge–Kutta–Chebyshev
PDE	Partial differential equation
PSE	Problem-solving environment
RHS	Right-hand side
RK	Runge–Kutta
RKC	Runge–Kutta–Chebyshev
SDIRK	Singly diagonally implicit Runge–Kutta

Chapter 1

Introduction

Differential equations are used in many situations to mathematically model the evolution of systems. They apply to a wide range of disciplines such as physics, chemistry, biology, and economics. Specific examples of differential equations include modelling of physical processes in the form of advection-diffusion-reaction equations [25]; fluid simulation using the incompressible Navier–Stokes equations [11]; special effects in cinematography such as the simulation of sand, smoke, fire, or water [5]; the electrophysiology of ion movement in epithelial cells [24]; chemotaxis models of tumour growth in cellular biology [29]; rewetting of hardened concrete [10]; and bone regrowth [19]. In each of these examples, a differential equation is solved to give an explicit representation of the dynamic process that is being modelled. The process of solving a differential equation is often informally called *integration*.

Initial-value problems (IVPs) consist of an ordinary differential equation that describes how the system changes over time and an associated initial state, known as the *initial condition*. The IVPs studied in this thesis are written as

$$\frac{d\mathbf{y}}{dt}(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \tag{1.1}$$

where $t \in \mathbb{R}$ is the simulation time, $\mathbf{y}(t) \in \mathbb{R}^m$ is the solution to the IVP at t, $\mathbf{f} : \mathbb{R} \times \mathbb{R}^m \mapsto \mathbb{R}^m$ is a function defining the differential equation that is often referred to as the *right-hand side* (RHS), t_0 is the initial time, and $\mathbf{y}_0$ is the initial condition (at time t_0). The solution $\mathbf{y}(t)$ is sought over a time interval $[t_0, t_\mathrm{f}]$, where t_f is the final simulation time.

Existence and *uniqueness* of solutions are both important concepts in the study of IVPs [39]. It is possible for IVPs not to have solutions; in such cases, a solution is said not to exist. If a solution exists, it might not be the only solution to the IVP, and therefore, the solution

is not unique. For the purpose of this thesis, all IVPs are assumed to have a RHS that is continuous in t and Lipschitz continuous in $\mathbf{y}$; i.e., $\forall t \in \mathbb{R}, \mathbf{y}_1 \in \mathbb{R}^m, \mathbf{y}_2 \in \mathbb{R}^m, \exists L \in \mathbb{R}$ such that $||\mathbf{f}(t, \mathbf{y}_1) - \mathbf{f}(t, \mathbf{y}_2)|| \leq L||\mathbf{y}_1 - \mathbf{y}_2||$ [1]. Consequently, the IVPs have unique solutions. Further, it is assumed that solutions to IVPs are not sensitive to small changes in the model parameters, particularly the initial condition. Due to perturbations caused by floating-point arithmetic, it would be impossible to numerically solve IVPs without making these assumptions. Together, these assumptions define a *well-posed* problem [39].

Many IVPs have known solutions that are generally calculated by solving the differential equations analytically. However, assuming that a solution exists, it is often infeasible to calculate an analytic solution, either due to computational constraints or because the differential equation does not lend itself to the application of known analytical solution techniques. In practice, most differential equations do not have analytical solutions. Therefore, methods of numerical integration are applied to find *approximate* solutions to IVPs. The *accuracy* of a numerical solution is, thus, an important metric to the study of numerical methods for differential equations. In the context of this thesis, accuracy is defined to be how well an approximate numerical solution matches the *exact* solution at a specified point. In the real world, approximate solutions to IVPs with appropriate accuracy are sufficient and are often the only possible means of finding a solution to a problem. For the analysis of numerical methods, exact solutions used as a reference for approximate solutions can be calculated analytically, if they are known; otherwise, they can be calculated via different numerical methods that have been shown to be reliable.

Methods that numerically solve IVPs are often called integrators because they integrate the RHS of an IVP equation from the initial condition to find $\mathbf{y}(t)$ at some later time. Numerically, this integration process is accomplished using a sequence of discrete temporal steps, advancing until the specified final simulation time has been reached. For each step, the solution is advanced from a known state $\mathbf{y}_{n-1} \approx \mathbf{y}(t_{n-1})$ to a new state $\mathbf{y}_n \approx \mathbf{y}(t_n)$. The stepsize is defined as $\Delta t_n = t_n - t_{n-1}$. A numerical method is *consistent* if it produces a solution that converges to the true solution as the stepsize approaches zero. For a consistent numerical method, smaller stepsizes lead to more accurate solutions.

Many IVPs can be interpreted as *additively split*; i.e., the RHS is defined by the sum of two

or more distinct terms. *Physics-based splitting* often arises naturally from an IVP, where each additively split term represents a physically significant contributing factor to the RHS of the IVP. Each additively split term of an IVP can have different numerical properties; therefore, it may be advantageous to apply an *additive* method, which is a method that applies distinct numerical methods to the contributing factors of the additively split problem. Treating the contributing factors with methods that are known to perform well on them individually has the potential to outperform single methods applied to the problem as a whole.

This thesis limits study to *2-additive splitting* [2], i.e., IVPs with an RHS that is defined as the sum of two contributing factors, written as

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}_I(t, \mathbf{y}(t)) + \mathbf{f}_E(t, \mathbf{y}(t)), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{1.2}$$

where $t \in \mathbb{R}$, $\mathbf{y} \in \mathbb{R}^m$, $\mathbf{f}_I : \mathbb{R} \times \mathbb{R}^m \mapsto \mathbb{R}^m$, and $\mathbf{f}_E : \mathbb{R} \times \mathbb{R}^m \mapsto \mathbb{R}^m$. IVPs that are 2-additively split can thus be solved with a 2-additive method. Subscripts I and E in (1.2) refer to terms that are respectively treated with implicit and explicit methods; see Section 2.3. In this thesis, IVPs with ostensibly more than two contributing factors must be consolidated into $\mathbf{f}_I$ or $\mathbf{f}_E$.

It is also possible to create a splitting of the form (1.2) that does not explicitly reflect the physical characteristics of the IVP. Such a splitting might be useful to capture mathematically significant characteristics of the problem that are not represented physically. This thesis investigates *Jacobian splitting* as an alternative to physics-based splitting for IVPs where one wishes to use an additive method. Jacobian splitting is accomplished by linearizing an IVP so that linear and non-linear components can be treated with separate numerical methods [44]. Jacobian splitting for the problem (1.1) is defined as

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{J}_\mathbf{f}\mathbf{y}(t) + \left[\mathbf{f}(t, \mathbf{y}(t)) - \mathbf{J}_\mathbf{f}\mathbf{y}(t)\right], \tag{1.3}$$

where $\mathbf{f} = \mathbf{f}_I + \mathbf{f}_E$ and $\mathbf{J}_\mathbf{f}$ is the Jacobian of $\mathbf{f}$ that has been evaluated at some t and $\mathbf{y}(t)$. Using the notation of (1.2), $\mathbf{f}_I(t, \mathbf{y}(t))$ corresponds to $\mathbf{J}_\mathbf{f}\mathbf{y}(t)$ and $\mathbf{f}_E(t, \mathbf{y}(t))$ corresponds to $\mathbf{f}(t, \mathbf{y}(t)) - \mathbf{J}_\mathbf{f}\mathbf{y}(t)$. Numerical methods typically update $\mathbf{J}_\mathbf{f}$ at the beginning of every step, a process known as *freezing* the Jacobian. Therefore, $\mathbf{J}_\mathbf{f}$ in (1.3) is a constant matrix for

a given step. Jacobian splitting uses a frozen Jacobian as opposed to the true Jacobian, which is a function of t and $\mathbf{y}$. Otherwise, $\mathbf{J_f y}(t)$ would generally not be linear. Linearity is desirable because methods often involve $\mathbf{J_f}$ in systems of equations; it is generally less expensive to solve linear systems of equations. Jacobian splitting has the added advantage of being applicable to any IVP of the form (1.1), whereas a numerical method using physics-based splitting requires that the RHS of the IVP be explicitly additive.

It is often difficult to predict which numerical method best suits a particular differential equation. During the last century, many advancements have been made regarding numerical integration methods [6]. Due to the complexity of modern methods and the size of modern IVPs, one can only make general predictions based on problem characteristics regarding which type of method is best suited for an IVP. Experimentation is thus a natural approach to empirically discover those methods that are most efficient and practical for different types of IVPs.

In this thesis, a suite of numerical methods is applied to several advection-diffusion-reaction problems to investigate the benefits of Jacobian splitting versus physics-based splitting for additive methods. Jacobian splitting has not been extensively used as an alternative to physics-based splitting. The suite of methods includes three 2-additive Runge–Kutta methods [2, 30], as well as additive Runge–Kutta methods based on the Chebyshev polynomials [46, 49, 40]. The suite includes six standard Runge–Kutta methods that serve as a performance baseline for the more advanced additive methods [20, 21]. Additive exponential Runge–Kutta methods are also applied to investigate whether the benefits of Jacobian splitting extend to these so-called structure-preserving methods [8].

The suite of methods is applied to several distinct advection-diffusion-reaction equations over a number of problem-specific parameters. These equations have been chosen because they represent a wide range of common characteristics that arise in the study of advection-diffusion-reaction equations. This study aims to be rigorous, and therefore, does not simply choose a few select problems that give favourable results. Numerical methods are applied to the following list of problems:

- a one-dimensional advection-diffusion problem with both linear advection and linear diffusion [25],

- a two-dimensional advection-diffusion problem that models heat transfer inside a fluid-filled pipe with a known velocity profile [28],
- one- and two-dimensional Brusselator problems, which are theoretical models of auto-catalytic reactions with linear diffusion and non-linear reaction [21],
- the `CUSP` problem, which is a one-dimensional diffusion-reaction equation that models a combination of Zeeman's "cusp catastrophe" and the Van der Pol oscillator with linear diffusion and non-linear reaction [25],
- a one-dimensional theoretical model of a combustion front that consists of linear advection, linear diffusion, and non-linear reaction [4],
- a one-dimensional chemotaxis model of tumour angiogenesis that consists of non-linear advection, linear diffusion, and non-linear reaction [25], and
- a one-dimensional mathematical model of concrete-rewetting that consists of non-linear advection, non-linear diffusion, and non-linear reaction [10].

The suite of methods includes the RODAS and RADAU5 methods, which are used to generate reference solutions for all of the IVPs considered in this work [21]. It is important that reference solution methods be efficient because IVPs must be solved numerous times with different tolerances to achieve a reliable reference solution. The choice between RODAS and RADAU5 for a given problem is dependent on which method has superior performance on that particular problem. Both of these methods are highly optimized and are known to provide accurate solutions for a wide variety of problems. It is thus reasonable to use them to generate the reference solutions. These reference solutions are used to evaluate the accuracy of all other numerical methods in this study.

In this thesis, the overall runtime is used as the primary metric to compare numerical methods. Results show that additive Runge–Kutta methods applied with Jacobian splitting outperform those applied with physics-based splitting. On some of the problems, the (non-additive) Runge–Kutta–Chebyshev methods outperform all other methods. Neither additive Runge–Kutta–Chebyshev nor additive exponential methods perform as well as any of the

additive Runge–Kutta methods. The problems in this thesis are evaluated over a set of tolerances for timestepping and a range of spatial discretizations. These form a large parameter set that depicts the benefits of Jacobian splitting versus physics-based splitting.

Evaluation of multiple numerical methods requires extensive computational power because each set of methods, IVPs, and other parameters forms an exponentially large parameter space. The analysis approach is to run tests across a coarse discretization of the parameter space, after which further analysis can focus on specific areas of interest. Such requirements motivate the use of a problem-solving environment (PSE), which is a platform designed to allow the detailed study of specific types of scientific problems [36]. PSEs provide infrastructure that is specialized to their intended problems to improve the efficiency with which one can implement and compare problems and methods. A performance-focused PSE for ordinary differential equations was developed in C++ to evaluate the suite of methods on IVPs. The PSE is named `pythODE++`; it is heavily based on and is intended to be used in conjunction with `pythODE`, a PSE written in Python, designed to perform extensive analysis on numerical methods for IVPs [31].

The most important aspect of comparing numerical methods is to ensure that all methods are implemented fairly in regard to code optimization. Otherwise, runtime measurements are not meaningful to the comparison of methods. Hidden parallelism in external libraries or compiled code embedded into a higher-level language can also be disastrous when trying to maintain fairness among implementations of methods. The solvers in `pythODE++` are written entirely in C++ to avoid such problems and to take advantage of the speed of a compiled language. The surrounding infrastructure was developed in Python. There is a performance trade-off between robust infrastructure and optimized code. The developed PSE is written in favour of a robust infrastructure, where all methods share the same supporting code, to allow comparisons to be made as equally as possible.

When numerical experiments are performed over a large parameter set, analysis of the experiments can be time consuming if not implemented efficiently. The `pythODE++` PSE is designed to efficiently perform extensive analysis of the numerical methods that have been applied to the IVPs. Experimental runs are organized in a tree-like structure, so that searching for the result of a particular run is efficient; an associative search is too computationally

expensive due to the large number of runs in a numerical experiment. Common usage of the analysis component of the PSE is to generate figures comparing methods for a specific IVP. The `pythODE++` PSE extracts runs that match user-specified criteria. Therefore, a straightforward approach is to develop scripts that generate figures of the experimental runs; parameters for analysis are specified using a simple set of matching directives. Analysis can also discard methods that take too long or become unstable, thus making the analysis phase more efficient.

1.1 Some Quick Words on Notation

In this thesis, all scalars are written in lower-case and are italicized, e.g., time t. All vectors are written in bold and lower-case, e.g., a position vector $\mathbf{x}$, and matrices are written in upper-case and bold, e.g., a linear system is written as $\mathbf{Ax} = \mathbf{b}$. Whenever a variable is a function, it is written in terms of its independent variables, such as $\mathbf{y}(t)$, when it first appears. Indices for vectors and matrices are written using subscripts starting from one, e.g., a vector is written $\mathbf{y} = (y_1, y_2, \ldots, y_m)^T$. In cases where more than one index is required, some sections use tensor notation for clarity, using superscripts to denote indices. These sections are specifically highlighted.

Throughout this thesis, many descriptions of numerical methods and problems use the same variables. In general, m refers to the dimension of a system, n refers to the current timestep, and s refers to the number of stages.

Derivatives are generally written in Leibniz notation; for instance, the time-derivative of q is written $\frac{\mathrm{d}q}{\mathrm{d}t}$. The gradient operator ∇ denotes $\left(\frac{\partial}{\partial y_1}, \frac{\partial}{\partial y_2}, \ldots, \frac{\partial}{\partial y_m}\right)^{\mathrm{T}}$ for a system of size m. Variables representing spatial or temporal discretization size are prefixed with a Δ symbol; e.g., a numerical method takes time steps of size Δt. Jacobian matrices (i.e., the matrix of

all first partial derivatives) of a function $\mathbf{f}(t, \mathbf{y})$ are written

$$\frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t, \mathbf{y}) = \mathbf{J}_{\mathbf{f}}(t, \mathbf{y}) = \begin{pmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial y_m} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial y_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial y_1} & \frac{\partial f_m}{\partial y_2} & \cdots & \frac{\partial f_m}{\partial y_m} \end{pmatrix}.$$

1.2 Outline of the Thesis

Chapter 2 introduces definitions for differential equations, the development of various classes of Runge–Kutta methods to solve IVPs, and the application of 2-additive Runge–Kutta methods to IVPs. Chapter 3 introduces the `pythODE++` PSE, provides mathematical definitions for advection-diffusion-reaction equations, and explains the discretization of differential equations such that they can be solved by the numerical methods described in this thesis. Chapter 4 discusses results, comparing how each tested numerical method behaves on various differential equations. Chapter 5 provides a list of scientific contributions, a summary of results, and some possible directions for future work.

Chapter 2

Theoretical Background

A differential equation defines a relationship between an unknown function and its derivatives. Any function that satisfies a differential equation is called a *solution* to the differential equation. For example, any function of the form $y(t) = -2/(c+t^2)$, where $c \in \mathbb{R}$, is a solution to the differential equation $\frac{\mathrm{d}y}{\mathrm{d}t} = ty^2$. In this thesis, all differential equations that are solved numerically are restricted to using real numbers. Their solutions are also real.

An *ordinary differential equation* (ODE) is comprised of a function of a single independent variable and derivatives of that function with respect to its independent variable. When the function (and hence dependent variable) is vector-valued, the system of ODEs can be written as

$$\boldsymbol{f}\left(t,\ \mathbf{y}(t),\ \frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t}(t)\right) = \mathbf{0},$$

where the solution $\mathbf{y}(t) : \mathbb{R} \mapsto \mathbb{R}^m$ is the dependent variable, t is the independent variable, and $\boldsymbol{f} \in \mathbb{R}^m$ is a function defining the relationship between variables in the ODE.

A *partial differential equation* (PDE) is comprised of a function of multiple independent variables and derivatives of various orders with respect to any of the independent variables. For example, the vector-valued PDE system in terms of one-dimensional position x and time t can be written as

$$\boldsymbol{f}\left(x,\ t,\ \mathbf{y}(x,t),\ \frac{\partial \mathbf{y}}{\partial x}(x,t),\ \frac{\partial \mathbf{y}}{\partial t}(x,t),\ \frac{\partial^2 \mathbf{y}}{\partial x^2}(x,t),\ \frac{\partial^2 \mathbf{y}}{\partial x \partial t}(x,t),\ \frac{\partial^2 \mathbf{y}}{\partial t^2}(x,t), \dots\right) = \mathbf{0},$$

where the solution $\mathbf{y}(x,t) : \mathbb{R}\times\mathbb{R} \mapsto \mathbb{R}^m$ is the dependent variable, x and t are the independent variables, and $\boldsymbol{f} \in \mathbb{R}^m$ is a function defining the relationship between variables in the PDE.

2.1 Autonomous Form

Recall that this thesis is focused on methods for solving problems of the form (1.1). The RHS of the differential equation in (1.1) is a function of both the independent variable t and dependent variable $\mathbf{y}(t)$. To simplify analysis, IVPs can be written in an *autonomous* form, where the RHS is only a function of the dependent variable. The transformation to autonomous form is straightforward. Letting $\tau = t$ be an additional dependent variable, the solution vector to the IVP becomes $(y_1, y_2, \ldots, y_m, \tau)^T$. Thus, (1.1) can be written as

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \\ \tau \end{pmatrix} = \begin{pmatrix} f_1(\tau, y_1, y_2, \ldots, y_m) \\ f_2(\tau, y_1, y_2, \ldots, y_m) \\ \vdots \\ f_m(\tau, y_1, y_2, \ldots, y_m) \\ 1 \end{pmatrix}, \quad \left.\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \\ \tau \end{pmatrix}\right|_{t=t_0} = \begin{pmatrix} y_{0,1} \\ y_{0,2} \\ \vdots \\ y_{0,m} \\ t_0 \end{pmatrix}, \tag{2.1}$$

because the derivative of time with respect to itself is one. It is therefore sufficient to state an IVP as

$$\frac{d\mathbf{y}}{dt}(t) = \mathbf{f}(\mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \tag{2.2}$$

where $\mathbf{f} : \mathbb{R}^m \mapsto \mathbb{R}^m$ without loss of generality. The development and analysis of numerical methods for IVPs are often based on the form (2.2).

2.2 Accuracy and Tolerances

The accuracy of a numerical method can be assessed using the *global error*, which is a measure of how close the calculated numerical solution is to the true solution. The global error at

each timestep n is defined as

$$\mathbf{e}_{n,\text{global}} = \mathbf{y}(t_n) - \mathbf{y}_n.$$

The global error represents the accumulated error from all steps of a numerical method. In practice, it is generally impossible to calculate because the exact solution to an IVP is unknown. However, another useful quantity is the *local error*, which is a measure of how much error is introduced from a single step of a numerical method. For each timestep n, the local error is defined as

$$\mathbf{e}_{n,\text{local}} = \tilde{\mathbf{y}}(t_n) - \mathbf{y}_n,$$

where $\tilde{\mathbf{y}}(t_n)$ is the local solution to the IVP, i.e., $\tilde{\mathbf{y}}(t_{n-1}) = \mathbf{y}_{n-1}$; see Section 2.3.2 for approaches to estimating the local error. In summary, the global error refers to the overall error of the numerical solution, whereas the local error refers to the error introduced by a single step.

Numerical methods are often classified by their order of accuracy. The numerical solution at each step is compared to the Taylor expansion of the local solution of (2.2), written as

$$\mathbf{y}(t_n) = \mathbf{y}(t_{n-1}) + \sum_{i=1}^{p} \frac{(\Delta t_n)^i}{i!} \frac{\mathrm{d}^i \mathbf{y}}{\mathrm{d}t^i}(t_{n-1}) + \mathcal{O}\big((\Delta t_n)^{p+1}\big), \tag{2.3}$$

where $p \in \mathbb{Z}^+$. A numerical method is said to be of order p if it matches the Taylor expansion of the local solution up to the term of order $p+1$.

Error in a numerical solution can arise because a numerical method represents a truncated Taylor expansion. Error can be also caused by internal components of a method that might not be exact, such as the numerical solution of non-linear systems of equations; see Section 2.3.1. Numerical methods are assigned *tolerances* that are used to govern the amount of acceptable error when calculating a solution. As tolerances become more strict, they generally increase the accuracy of a numerical solution at the expense of overall computational cost. However, it is important to note that a method subject to given tolerances is not guaranteed to yield a solution accurate to those tolerances. In practice, the error at each

step is only an estimate; it is not exact. Tolerances are typically presented as a combination of (i) *absolute* tolerances, which give a maximal value of allowable numerical error between quantities, and (ii) *relative* tolerances, which give a maximal value of allowable error between two quantities relative to their magnitude.

2.3 Runge–Kutta Methods

This section introduces the construction of numerical methods for IVPs. Using the fundamental theorem of calculus, each step of the solution to an IVP of the form (2.2) can be written as

$$\mathbf{y}_n - \mathbf{y}_{n-1} = \int_{t_{n-1}}^{t_n} \mathbf{f}(\mathbf{y}(t))\,\mathrm{d}t. \tag{2.4}$$

This integral cannot be solved using methods of quadrature due to the presence of $\mathbf{y}(t)$ in the integral. However, it can, for example, be approximated by $\Delta t_n \mathbf{f}(\mathbf{y}_{n-1})$, which uses a single evaluation of the RHS of the IVP. Each step of an IVP solution can thus be written as

$$\mathbf{y}_n = \mathbf{y}_{n-1} + \Delta t_n\, \mathbf{f}(\mathbf{y}_{n-1}), \tag{2.5}$$

which is the forward Euler (FE) method. FE is the simplest method to solve an IVP. It is first order with respect to a Taylor expansion of the local solution; thus, FE often requires tiny stepsizes to be sufficiently accurate. An alternative approach is to approximate (2.4) with $\Delta t_n \mathbf{f}(\mathbf{y}_n)$, yielding the backward Euler (BE) method (sometimes called implicit Euler). The BE method is written as

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

where the RHS evaluation is at t_n, rather than t_{n-1}. Note that the RHS of (2.6) contains $\mathbf{y}_n$. Thus, when implementing BE, a system of equations must be solved at every step of the method because $\mathbf{y}_n$ is not defined explicitly. Typically, such systems are solved using Newton's method; see Section 2.3.1. The advantage of BE is that it allows for larger stepsizes

in comparison to FE for certain IVPs; see Section 2.3.4 for details. However, to be sufficiently accurate, BE might still require small stepsizes to be accurate because it is a first-order method.

A small stepsize can be computationally expensive because it requires a large number of overall steps to solve an IVP over a desired time interval. Therefore, the concept of the forward and backward Euler methods is extended to a larger class of methods known as Runge–Kutta (RK) methods, which were first introduced circa 1900 [6]. These methods use one or more evaluations of the RHS of the differential equation in the calculation of a single step. Higher-order and potentially more accurate solutions are generated, increasing the allowable stepsize while maintaining desired tolerances. RK methods are said to be *single-step* methods because they use only solution information from the current timestep when advancing the numerical solution.

RK methods use a number of intermediate states called *stages*, where each stage corresponds to a function evaluation of the RHS. Each stage is associated with a specific simulation time and a solution to the differential equation at that time. Each stage of the timestep is generally calculated using other stages in that timestep.

The general form of an s-stage RK method applied to an IVP is written as

$$\mathbf{k}_i = \Delta t_n \mathbf{f}(t_{n-1} + c_i \Delta t_n, \mathbf{y}_{n-1} + \sum_{j=1}^{s} a_{ij}\mathbf{k}_j), \quad i = 1, 2, \ldots, s,$$

$$\mathbf{y}_n = \mathbf{y}_{n-1} + \sum_{j=1}^{s} b_j \mathbf{k}_j,$$

where the $\mathbf{k}_i$ are the stages, the coefficients a_{ij} and b_i determine weightings for the linear combination of stages, and the coefficients c_i determine the times at which the stages are evaluated. The coefficients are stored in a convenient format known as a *Butcher tableau*, shown in Table 2.1. For example, the respective Butcher tableaux for forward and backward Euler methods, from (2.5) and (2.6), are

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

c_1	$a_{1,1}$	$a_{1,2}$	$\dots$	$a_{1,s}$
c_2	$a_{2,1}$	$a_{2,2}$	$\dots$	$a_{2,s}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_s	$a_{s,1}$	$a_{s,2}$	$\dots$	$a_{s,s}$
	b_1	b_2	$\dots$	b_s

Table 2.1: The Butcher tableau of an RK method.

Note that from (2.1) and the definition of an RK method, it follows that $c_i = \sum_{j=1}^{s} a_{ij}$ because the last element of the RHS of (2.1) is one. The coefficients a_{ij} and b_i are determined from *order conditions* that are derived by matching the expanded RK method to terms of the Taylor series, up to a specified order. When the order conditions are satisfied, the numerical method is designated to be of that order. An overview of RK order conditions that form the Butcher tableau is shown in Appendix A. Higher-order methods generally yield more accurate solutions to IVPs for a given stepsize.

Linear multistep methods are an alternative class of methods that use information from previous steps to gain a higher order of accuracy, rather than by the introduction of stages. The focus of this work is to solve IVPs efficiently and accurately; therefore, higher-order numerical methods are desired. Multistep methods are not considered in this thesis because RK methods have better theoretical stability properties at higher order [2]; see Section 2.3.4, which discusses stability. RK and multistep methods are both part of an even larger class of numerical methods known as general linear methods (GLMs), which make use of past steps and stages when calculating a new step [26]. The theory of GLMs is much less developed compared to RK methods. GLMs are beyond the scope of this thesis.

2.3.1 Classes of Runge–Kutta Methods

If a given stage depends only on prior stages (i.e., a matrix of the Butcher tableau is a strictly lower-triangular matrix), the method is an *explicit Runge–Kutta* (ERK) method. It is relatively inexpensive to compute a step using an explicit method. Stages can be

calculated directly from the definition of an RK method because all stages depend only on previous stages. An example of the Butcher tableau for an explicit RK method is shown in Table 2.2. This RK method is a popular fourth-order RK method, RK4, commonly known as *the Runge–Kutta method* to those outside the field of numerical analysis. The method is algorithmically written

$$\begin{aligned}
\mathbf{k}_1 &= \Delta t_n \mathbf{f}(t_{n-1}, \mathbf{y}_{n-1}), \\
\mathbf{k}_2 &= \Delta t_n \mathbf{f}(t_{n-1} + \frac{1}{2}\Delta t_n, \mathbf{y}_{n-1} + \frac{1}{2}\mathbf{k}_1), \\
\mathbf{k}_3 &= \Delta t_n \mathbf{f}(t_{n-1} + \frac{1}{2}\Delta t_n, \mathbf{y}_{n-1} + \frac{1}{2}\mathbf{k}_2), \\
\mathbf{k}_4 &= \Delta t_n \mathbf{f}(t_{n-1} + \Delta t_n, \mathbf{y}_{n-1} + \mathbf{k}_3), \\
\mathbf{y}_n &= \mathbf{y}_{n-1} + \frac{1}{6}\mathbf{k}_1 + \frac{1}{3}\mathbf{k}_2 + \frac{1}{3}\mathbf{k}_3 + \frac{1}{6}\mathbf{k}_4.
\end{aligned}$$

If stages of an RK method depend on current or future stages, the numerical method is an *implicit Runge–Kutta* (IRK) method. In an implicit method, some of the stages must be solved simultaneously as part of a larger system of algebraic equations, possibly of size $s \times m$. The stages are dependent on each other; therefore, they do not have explicit representations. For example, the Gauss method of order 4 is an implicit method; its Butcher tableau is shown in Table 2.3. The equations for each step of the Gauss method are written as

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}\left(t_{n-1} + \left[\frac{1}{2} - \frac{\sqrt{3}}{6}\right]\Delta t_n, \mathbf{y}_{n-1} + \Delta t_n \left[\frac{1}{4}\mathbf{k}_1 + \left(\frac{1}{4} - \frac{\sqrt{3}}{6}\right)\mathbf{k}_2\right]\right), \\
\mathbf{k}_2 &= \mathbf{f}\left(t_{n-1} + \left[\frac{1}{2} + \frac{\sqrt{3}}{6}\right]\Delta t_n, \mathbf{y}_{n-1} + \Delta t_n \left[\left(\frac{1}{4} + \frac{\sqrt{3}}{6}\right)\mathbf{k}_1 + \frac{1}{4}\mathbf{k}_2\right]\right), \\
\mathbf{y}_n &= \mathbf{y}_{n-1} + \Delta t_n \left[\frac{1}{2}\mathbf{k}_1 + \frac{1}{2}\mathbf{k}_2\right].
\end{aligned}$$

The steps of an implicit method are generally more expensive than the steps of an explicit method because the implicit method involves solving systems of equations. However, implicit methods are often are able to meet desired tolerances using greater stepsizes because implicit methods generally have better stability properties [20, 21]. For example, BE often allows for greater stepsizes than FE, as previously discussed in Section 2.3. In certain cases, implicit

methods mitigate the increased computation per step. For example, if the IVP is linear, the implicit method is *linearly implicit*. The system of equations to be solved at each step or stage is linear; therefore, it can be solved with a single Newton iteration.

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
	1/6	1/3	1/3	1/6

Table 2.2: Butcher tableau for RK4.

$\frac{1}{2} - \frac{\sqrt{3}}{6}$	$\frac{1}{4}$	$\frac{1}{4} - \frac{\sqrt{3}}{6}$
$\frac{1}{2} + \frac{\sqrt{3}}{6}$	$\frac{1}{4} + \frac{\sqrt{3}}{6}$	$\frac{1}{4}$
	$\frac{1}{2}$	$\frac{1}{2}$

Table 2.3: Butcher tableau for the implicit fourth-order Gauss method.

Diagonally implicit Runge–Kutta (DIRK) methods are a special class of IRK methods that can have non-zero values on the diagonal of their Butcher tableaux but have an upper triangle that is comprised only of zeros. DIRK methods compute each stage separately, solving a smaller system (in comparison to a fully implicit method) of m equations for each stage. Solving s systems of size m is less computationally expensive than solving larger systems required when calculating multiple stages simultaneously. DIRK methods mitigate some of the performance issues of general IRK methods, while maintaining some of the desirable properties. Therefore, DIRK methods can be more efficient than general IRK methods.

Singly diagonally implicit Runge–Kutta (SDIRK) methods are a further optimization to DIRK methods, whereby all diagonal elements in the Butcher tableau are identical. SDIRK methods can then use the same Jacobian for all stages because the implicit function evaluations are all multiplied by the same factor. Therefore, operations such as matrix factorization need only be done once per step, resulting in an overall method that is more efficient.

When calculating stages, implicit methods must solve systems of equations that are non-linear if the differential equation is non-linear. In this thesis, these systems are computed using the modified Newton's method, shown in Algorithm 1. This algorithm is iterative, solving the stages to a desired tolerance. This algorithm differs from the standard Newton's method such that Jacobian is evaluated only once, rather than at each iteration. This modified approach using a frozen Jacobian might require more iterations; however, the computational cost of additional iterations is low in comparison to the cost of re-evaluating the Jacobian multiple times. Note that, if the system is linear, the solution converges after a single iteration.

Algorithm 1 Modified Newton's method used for implicit RK methods, designed to solve a non-linear system of the form $\mathbf{g}(\mathbf{y}) = \mathbf{0}$. In the algorithm, τ is the absolute tolerance of the method (note that when this algorithm applied to a general non-linear system, a relative tolerance should also be used), $\tau_{\max}$ is upper limit for the norm if the iteration diverges, and $i_{\max}$ is the maximum number of iterations.

$\mathbf{x} \leftarrow$ initial guess
$\mathbf{J} \leftarrow \frac{\partial \mathbf{g}}{\partial \mathbf{y}}(\mathbf{x})$
for $i = 1 \rightarrow i_{\max}$ **do**
  $\mathbf{r} \leftarrow \mathbf{g}(\mathbf{x})$
  if $||\mathbf{r}||_\infty < \tau$ **then**
    return $\mathbf{x}$
  end if
  if $||\mathbf{r}||_\infty > \tau_{\max}$ **then**
    break
  end if
  Solve $\mathbf{J}\mathbf{z} = -\mathbf{r}$ for $\mathbf{z}$
  $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{z}$
end for
return convergence failure

2.3.2 Error Estimation

The exact solution to an IVP is generally unknown; subsequently, the exact error of a numerical solution is also unknown. A common way of predicting local error at each step n is to solve the differential equation with two different methods at each step, producing solution $\tilde{\mathbf{y}}_n$ in addition to $\mathbf{y}_n$. The local error prediction is thus written $\mathbf{e}_{n,\text{local}} = \tilde{\mathbf{y}}_n - \mathbf{y}_n$. A second method that produces $\tilde{\mathbf{y}}_n$ is commonly known as an *auxiliary* method. The goal of a step

controller is to ensure that each component $y_{n,j}$ (n is the step and j is the vector index) of the solution satisfies $|\tilde{y}_{n,j} - y_{n,j}| \leq \tau_{\text{abs}} + \tau_{\text{rel}} \max(y_{n,j}, y_{n-1,j})$, where τ_{abs} is the absolute tolerance and τ_{rel} is the relative tolerance. A prediction of error for a step can thus be calculated using the root mean square [20]

$$\varepsilon_n = \sqrt{\frac{1}{m}\sum_{j=1}^{m}\left(\frac{\tilde{y}_{n,j} - y_{n,j}}{\tau_{\text{abs}} + \tau_{\text{rel}} \max(y_{n,j}, y_{n-1,j})}\right)^2}, \tag{2.7}$$

where ε_n is a scalar that represents local error at every step n with respect to some desired tolerances. Recall that m is the dimension of the system.

Solving the IVP with two methods is computationally expensive. Therefore, it is common to use an *embedded* method for error prediction, where two RK methods of different order use the same stages because stages are generally the most computationally expensive part of an RK method. The two methods for embedded error control share the same a_{ij} coefficients from the Butcher tableau but use different b_i coefficients to obtain methods of different order; i.e., the main method is embedded into the tableau of the auxiliary method. The representation of an embedded method includes an additional row of coefficients $\tilde{b}_i$ in the Butcher tableau for the auxiliary method. The modified tableau for an s-stage embedded method can be written as

$$\begin{array}{c|cccc} c_1 & a_{1,1} & a_{1,2} & \dots & a_{1,s} \\ c_2 & a_{2,1} & a_{2,2} & \dots & a_{2,s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s,1} & a_{s,2} & \dots & a_{s,s} \\ \hline & b_1 & b_2 & \dots & b_s \\ & \tilde{b}_1 & \tilde{b}_2 & \dots & \tilde{b}_s \end{array}$$

where $\tilde{b}_i$ are the quadrature weights for the embedded method.

Step-doubling is another common approach to the estimation of local error when a numerical method does not have an associated auxiliary method [21]. The solution from taking

two steps of size Δt_n is compared to advancing the solution using a single step of size $2\Delta t_n$. The difference between the two solutions can be used as a local error estimate.

2.3.3 Step Control

The goal of using numerical methods for solving IVPs is to obtain a solution to some desired level of accuracy in a reasonable amount of time. When numerically solving differential equations, there is a trade-off between the accuracy of a numerical solution and the time required to calculate the solution. For a given numerical method, the stepsize plays an important role in this trade-off because the time required to generate a numerical solution relies heavily on the number of steps taken. If the stepsize is too large, the numerical method introduces too much error. If the stepsize is too small, the numerical method performs unnecessary computation (resulting in a solution that might be unnecessarily accurate) and is inefficient; i.e., the IVP could have been solved to an acceptable accuracy with a larger stepsize and, consequently, in less time.

It is often advantageous for numerical methods to use an *adaptive* stepsize rather than a constant stepsize while timestepping through an IVP. If the stepsize is allowed to change, the numerical method can use smaller stepsizes on "exciting" regions of the solution that require smaller stepsizes to remain accurate, and the method can use larger stepsizes on the less exciting regions of the solution that remain sufficiently accurate with larger stepsizes. Adaptive step control can thus allow numerical methods to be much more efficient because time is not wasted using small steps on sections of the problem that could easily allow for larger steps; fewer overall steps are taken.

Step controllers automatically govern the change in stepsize, adjusting the step size based on the calculated value for ε_n, given by (2.7). After each step, the step controller decides whether or not to *accept* the step and proceed to computing subsequent steps, or to *reject* the step, in which case the step controller decreases the stepsize and makes another attempt at the step. A step is accepted when ε_n is less than one; otherwise, it is rejected. A step can also be rejected for other reasons. For example, Newton's method might fail to converge or undefined values might arise while computing the numerical solution. It is desirable to minimize the number of rejected steps because rejected steps are a computational cost that

does not advance the numerical solution.

Recall that ε_n is only an estimate of the local error, not the true value. Therefore, limits are imposed so that the stepsize does not shrink or grow too quickly. If stepsize grows too quickly, the IVP solver might reject the following step due an estimate of local error that is not sufficiently accurate. A stepsize can also shrink too quickly due to an inaccurate estimation of local error, resulting in poor performance because a larger stepsize would have been sufficient. The change in stepsize is also multiplied by a safety factor to provide better error prediction at the next step. *Standard step control* is thus managed by the following formula [42]

$$\Delta t_{n+1} = \Delta t_n \cdot \min\left(\alpha_{\max}, \max\left(\alpha_{\min}, \alpha\left(\frac{1}{\varepsilon_{n+1}}\right)^{1/(p+1)}\right)\right), \tag{2.8}$$

where α is the safety factor, $\alpha_{\min}$ is a minimum allowable factor, $\alpha_{\max}$ is a maximum allowable factor, and p is the order of the numerical method. In this thesis, the numerical experiments set the value of α to 0.9, $\alpha_{\min}$ to 0.2, and $\alpha_{\max}$ to 5. These values are standard choices. Additionally, the step controller places a hard minimum on the stepsize such that the stepsize does not approach machine epsilon, resulting in inaccuracies or a solution that can no longer be advanced because the stepsize is too small.

A common issue arises from the following situation: the IVP solver takes a step, accepts the step with a small error estimate, increases the stepsize by the maximum amount, rejects the subsequent step due to a large error estimate, and decreases the stepsize by the maximum amount to approximately what it was at the beginning of the previously accepted step. The standard step controller is prone to repeating this process, resulting in a numerical method that is inefficient because it rejects approximately half of the overall steps. Often, this issue can be mitigated by changing values for $a_{\min}$ and $a_{\max}$, but for some IVPs, the problem persists.

To address this, numerical methods (typically those that are implicit) often perform better

using a *predictive step controller* scheme [42], written as

$$\Delta t_{n+1} = \Delta t_n \cdot \min\left(\alpha_{\max}, \max\left(\alpha_{\min}, \alpha\left(\frac{1}{\varepsilon_n}\right)^{1/(p+1)} \frac{\Delta t_n}{\Delta t_{n-1}}\left(\frac{\varepsilon_{n-1}}{\varepsilon_n}\right)^{1/(p+1)}\right)\right). \tag{2.9}$$

The predictive controller often provides a smoother change in stepsize compared to the standard controller, in part, because it relies on Δt_{n-1} in addition to Δt_n.

2.3.4 Stability and Stiffness

Stability analysis is important to the study of numerical methods because it facilitates understanding and prediction of the types of methods best suited to different types of IVPs. Different methods impart different errors to a numerical solution, affecting how the numerical solution behaves over time. Each step of a numerical method introduces error into the solution. Throughout the many steps required to calculate the solution to an IVP over a desired time interval, error accumulates because error is introduced at every step. In the context of numerical integration methods, the numerical solution is stable if its error remains bounded. Instability is often apparent when the numerical solution rapidly diverges.

To analyze the stability an RK method, the test equation $\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y$ is considered, where $\lambda \in \mathbb{C}$ is known as an *eigenvalue* of the problem. The corresponding function $\exp(\lambda t)$ is a solution to the differential equation corresponding to eigenvalue λ. The stability behaviour of integration methods applied to the test equation is known as *linear stability analysis*. Analysis of the test equation can be meaningful for IVPs of the form (1.1) despite the fact that IVPs are generally non-linear [20].

Application of an RK method to the test equation is written as

$$\begin{aligned} k_i &= \lambda y_{n-1} + \Delta t_n \lambda \sum_{j=1}^{s} a_{ij} k_i, \quad i = 1, 2, \ldots, s, \\ y_n &= y_{n-1} + \Delta t_n \sum_{j=1}^{s} b_j k_j. \end{aligned} \tag{2.10}$$

Using matrix notation to coalesce all stages, (2.10) can be rewritten as

$$\begin{aligned} \mathbf{k} &= \lambda y_{n-1}\mathbf{1} + \Delta t_n \lambda \mathbf{A}\mathbf{k}, \\ y_n &= y_{n-1} + \Delta t_n \mathbf{b}^{\mathrm{T}}\mathbf{k}, \end{aligned} \tag{2.11}$$

where $\mathbf{k} = (k_1, k_2, \ldots, k_s)^{\mathrm{T}}$, $\mathbf{1} = (1, 1, \ldots, 1)^{\mathrm{T}} \in \mathbb{R}^s$, and $\mathbf{b} = (b_1, b_2, \ldots, b_s)^{\mathrm{T}}$. By solving for $\mathbf{k}$, (2.11) can be rearranged to

$$y_n = [1 + \Delta t_n \lambda \mathbf{b}^{\mathrm{T}}(\mathbf{I} - \Delta t_n \lambda \mathbf{A})^{-1}\mathbf{1}]y_{n-1},$$

where $\mathbf{I} \in \mathbb{R}^{s\times s}$ is the identity matrix. The IVP step can then be written $y_n = R\big(\Delta t_n \lambda\big) y_{n-1}$ where $R(z) = 1 + z\mathbf{b}^{\mathrm{T}}(\mathbf{I} - z\mathbf{A})^{-1}\mathbf{1}$ is the stability function.

For $\mathrm{Re}(\lambda) \leq 0$, a method is *absolutely stable* when $|R(\Delta t_n \lambda)| \leq 1$. Otherwise, exponential accumulation of error can cause the numerical solution to rapidly become inaccurate.[1] This criterion for stability forms regions on the complex plane for which the method is absolutely stable, based on both the timestep Δt_n and the (complex) eigenvalue λ. The shaded areas in Figure 2.1 show the regions of stability for some explicit RK methods of orders two through five, applied to the test equation, i.e., the regions for which $\Delta t_n \lambda$ satisfies the stability constraint $|R(\Delta t_n \lambda)| \leq 1$. Note that these plots hold for all RK methods of those orders, respectively. When λ is large, the estimate of local error typically causes the step controller to reduce Δt_n so that $\Delta t_n \lambda$ is within the region of stability, and therefore, a stable numerical solution is maintained.

With regard to IVPs of the form (1.1), each eigenvalue of the Jacobian of the RHS should generally be contained within the stability region for a numerical method to perform well. Therefore, these eigenvalues are indicative of how well a given method performs on the associated IVP, based how the stability region adapts to contain the eigenvalues. Linear stability analysis cannot generally be used a sole justification for the selection of numerical methods; it is, nonetheless, a useful tool to provide insight into which methods might perform well on a given IVP.

[1] Note that when $\mathrm{Re}(\lambda) > 0$, it is actually desirable to have $|R(\Delta t_n \lambda)| \geq 1$; otherwise, the numerical solution is damped when it should be increasing in magnitude.

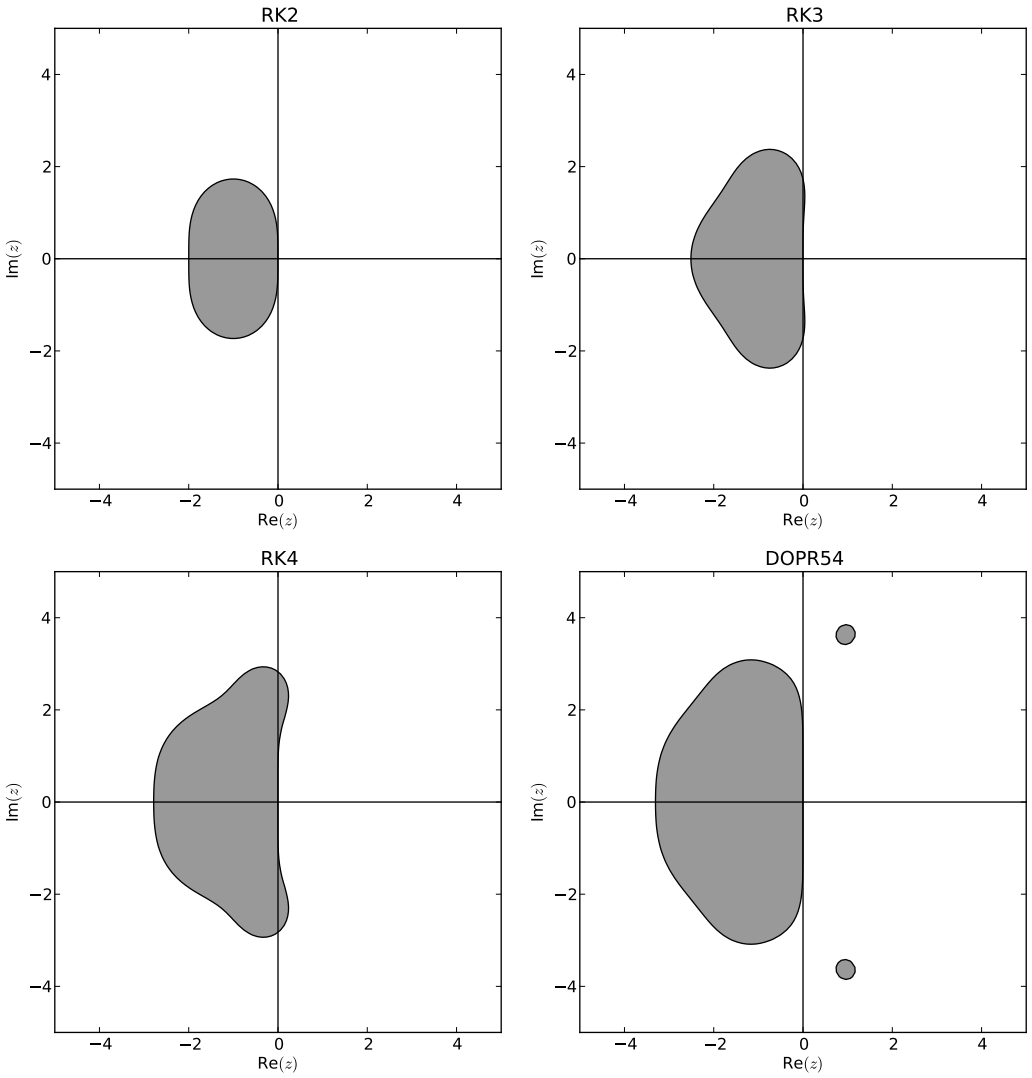

Figure 2.1: Regions of absolute stability, calculated from the stability function, for Runge–Kutta methods ranging from order two to five.

Directly related to stability is a property of IVPs known as *stiffness*, which can be thought of as referring to how difficult it is for an explicit numerical method to solve a problem using reasonably sized timesteps, i.e., timesteps that are not too small [21]. In order to maintain a stable numerical solution on a stiff problem, the stepsize of an explicit method is limited by the stiffness of the problem rather than by accuracy constraints. As an IVP becomes more stiff, it becomes increasingly inefficient for the IVP to be solved using explicit methods. Implicit methods generally perform much better on stiff problems, despite the trade-off of having to solve a generally non-linear system at every step. Larger, more costly steps can be taken with an implicit method, and the overall performance gain of fewer overall steps usually outweighs the additional cost per step. For example, Figure 2.2 shows stability regions for the FE and BE methods, comparing explicit and implicit methods. Notice that forward Euler is explicit and thus has a bounded region of absolute stability. Backward Euler has an unbounded region of absolute stability, demonstrating why implicit methods can perform better on stiff problems. The RADAU5 method is also fully implicit and has an unbounded region of stability, shown in Figure 2.3.

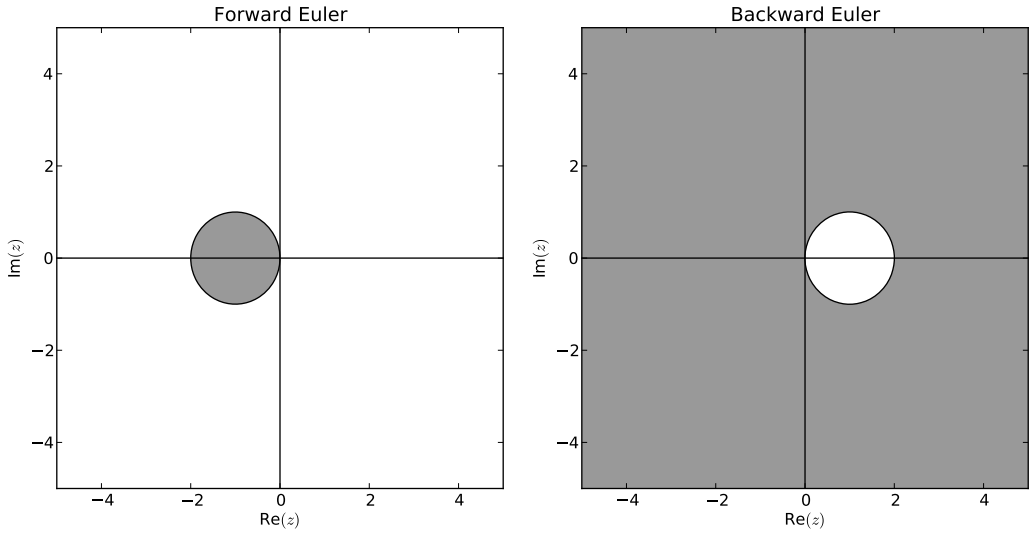

Figure 2.2: Regions of absolute stability, calculated from the stability function, for the forward and backward Euler methods.

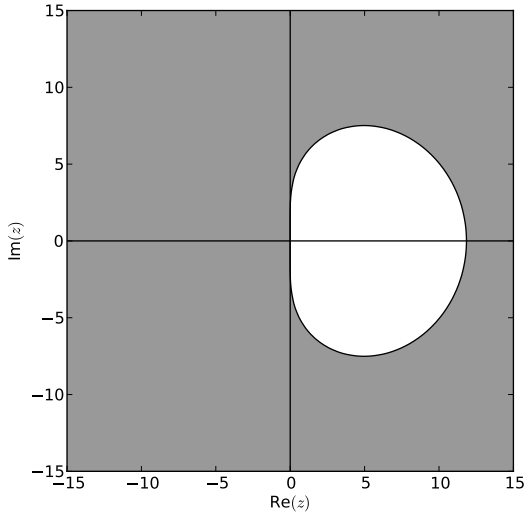

Figure 2.3: Region of absolute stability, calculated from the stability function, for the RADAU5 method.

2.3.5 Runge–Kutta–Chebyshev Methods

Thus far, RK methods have been defined by a Butcher tableau, containing a set of precomputed coefficients that specifies the number of stages, the dependencies among stages, and how to weight them when taking a step. Note that the number of stages in these methods is fixed for all steps. Runge–Kutta–Chebyshev (RKC) methods are a special class of ERK methods that aim to overcome the stability constraints of general ERK methods by using an extended region of absolute stability. RKC methods have a variable number of stages, which are used to control the stability region. RKC methods first allow a step controller to choose an appropriate stepsize with respect to a prediction of local error (much like general RK methods), and then they choose an appropriate number of stages to maintain a stable solution (unlike general RK methods).

RKC methods are well suited for some problems that are moderately stiff. These methods maintain the speed of general ERK methods because they do not have any implicit stages. RKC methods are particularly applicable to parabolic PDEs that, when discretized, have eigenvalues along the negative real axis. The region of absolute stability extends along the negative real axis, growing quadratically with the number of stages.

All stages are defined recursively; thus, the storage requirements of RKC are minimal.

At each stage, the order conditions are satisfied explicitly using the Chebyshev polynomials. An arbitrary number of stages can be computed on-the-fly without solving order condition coefficients for each new number of desired stages. Each step of an RKC method can be represented by a Butcher tableau because RKC methods form a subset of general RK methods; however, for the purposes of implementation, it is more efficient and convenient to use a recursive definition for an RKC method.

Once an appropriate timestep Δt_n has been chosen by the step controller, the number of stages s of the RKC method is optimally chosen based on the spectral radius σ of $\frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_{n-1}, \mathbf{y}_{n-1})$, i.e., the largest eigenvalue of the Jacobian of the RHS evaluated at the beginning of the timestep. The calculation for the number of stages for each step is written [46]

$$s = 1 + \left\lfloor \sqrt{1 + \frac{\Delta t_n \sigma}{0.653}} \right\rfloor.$$

The s-stage formula for an RKC method is written as [46]

$$\begin{aligned}
\mathbf{k}_0 &= \mathbf{y}_{n-1}, \\
\mathbf{k}_1 &= \mathbf{k}_0 + \kappa_1 \Delta t_n \mathbf{f}_0, \\
\mathbf{k}_j &= (1 - \mu_j - \nu_j)\mathbf{k}_0 + \mu_j \mathbf{k}_{j-1} + \nu_j \mathbf{k}_{j-2} + \kappa_j \Delta t_n \mathbf{f}_{j-1} - a_{j-1}\kappa_j \Delta t_n \mathbf{f}_0, \\
j &= 2, 3, \ldots, s, \\
\mathbf{y}_n &= \mathbf{k}_s,
\end{aligned}$$

$$\begin{aligned}
&\mu_j = \frac{2 b_j \omega_0}{b_{j-1}}, \qquad \nu_j = \frac{-b_j}{b_{j-2}}, \qquad \mathbf{f}_j = \mathbf{f}(t_{n-1} + c_j \Delta t_n, \mathbf{k}_j), \\
&T_j(z) = 2 z T_{j-1}(z) - T_{j-2}(z), \qquad T_1(z) = z, \qquad T_0(z) = 1,
\end{aligned}$$

where T_j is the jth Chebyshev polynomial. The following coefficients for each stage j pertain

to a first-order RKC method; they are written as

$$\kappa_j = 0, \quad a_j = 0, \quad b_j = T_j^{-1}(\omega_0), \quad c_j = \omega_1 \frac{T_j'(\omega_0)}{T_j(\omega_0)},$$
$$\omega_0 = 1 + \frac{\eta}{s^2}, \quad \omega_1 = \frac{T_s(\omega_0)}{T_s'(\omega_0)},$$

and coefficients for a second-order RKC method for each stage j are written as

$$\kappa_1 = b_1\omega_1, \quad \kappa_j = \frac{2b_j\omega_1}{b_{j-1}},$$
$$a_j = 1 - b_j T_j(\omega_0), \quad b_j = \frac{T_j''(\omega_0)}{\left(T_j'(\omega_0)\right)^2}, \quad b_0 = b_1 = b_2,$$
$$c_j = \omega_1 \frac{T_j''(\omega_0)}{T_j'(\omega_0)}, \quad c_1 = \frac{c_2}{T_2'(\omega_0)}, \quad c_0 = 0,$$
$$\omega_0 = 1 + \frac{\eta}{s^2}, \quad \omega_1 = \frac{T_s'(\omega_0)}{T_s''(\omega_0)}.$$

Derivatives of the Chebyshev polynomials required by each stage j are calculated as

$$T_0'(z) = 0 \quad T_1'(z) = 1,$$
$$T_j'(z) = 2T_{j-1}(z) + 2zT_{j-1}'(z) - T_{j-2}'(z),$$
$$T_0''(z) = 0 \quad T_1''(z) = 0,$$
$$T_j''(z) = 2T_{j-1}'(z) + 2T_{j-1}'(z) + 2zT_{j-1}''(z) - T_{j-2}''(z).$$

RKC is intended for problems with Jacobian matrices that have all of their eigenvalues near the negative real axis [43]. The stability polynomial for an s-stage RKC method is written as [46]

$$R(z) = a_j + b_j T_j(\omega_0 + \omega_1 z),$$

where $z = \Delta t_n \lambda$. Recall that $\lambda \in \mathbb{C}$. Stability regions of the RKC1 and RKC2 methods for stages two through five are are shown in Figures 2.4 and 2.5, respectively.

All Chebyshev polynomials in the method can be calculated recursively; each stage of the RKC method calculates the next required Chebyshev polynomial in the sequence. The only

exception to this recursive calculation is ω_1, which depends on the last Chebyshev polynomial, $T_s(\omega_0)$. RKC methods use ω_1 in all stages, but the last Chebyshev polynomial is not calculated until the final stage. Taking note that $\omega_0 \geq 1$, the values of $T_s'(\omega_0)$ and $T_s''(\omega_0)$, required to compute ω_1, are calculated using the trigonometric definition of Chebyshev polynomials:

$$T_s(x) = \cosh(s \operatorname{arccosh}(x))$$

RKC calculates error for the step controller differently from the previously discussed embedded method. An estimate for error is calculated using an additional function evaluation. An expression for the local error corresponding to $\tilde{\mathbf{y}} - \mathbf{y}$ and used to calculate ε_n in (2.7) can be written for the RKC method as

$$\frac{1}{15}\left[12(\mathbf{y}_{n-1} - \mathbf{y}_n) + 6\Delta t_n(\mathbf{f}(t_{n-1}, \mathbf{y}_{n-1}) + \mathbf{f}(t_n, \mathbf{y}_n))\right].$$

After an error estimate has been calculated, the step controller functions according to (2.8).

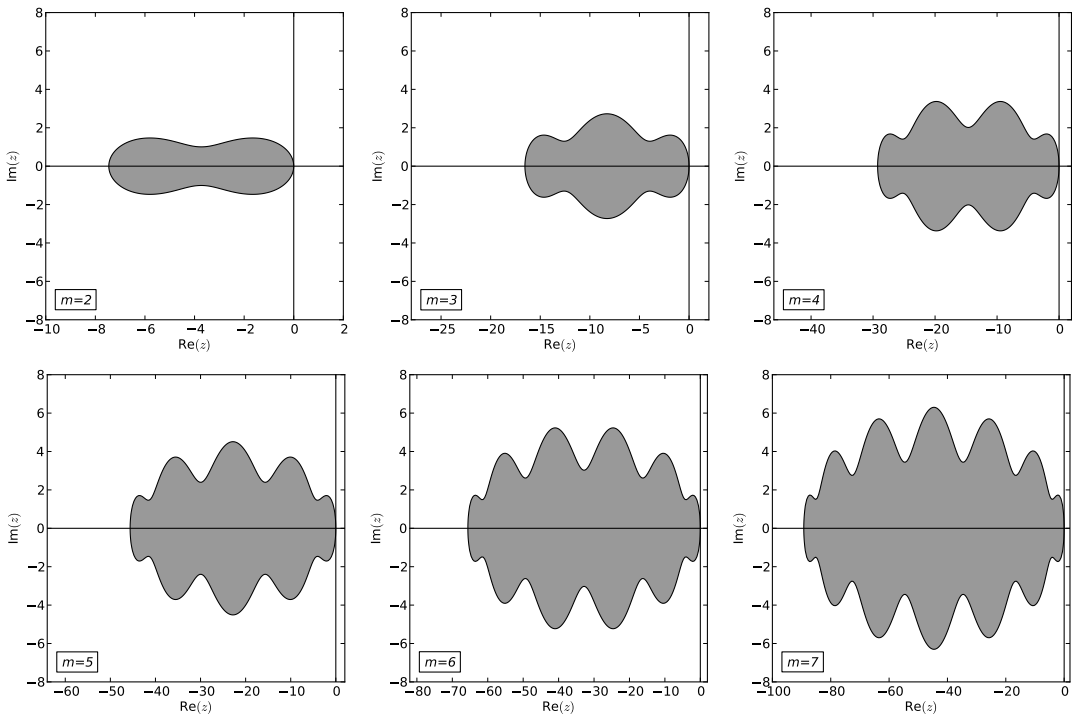

Figure 2.4: Regions of absolute stability, calculated from the stability function, for the RKC1 method of stages two through seven.

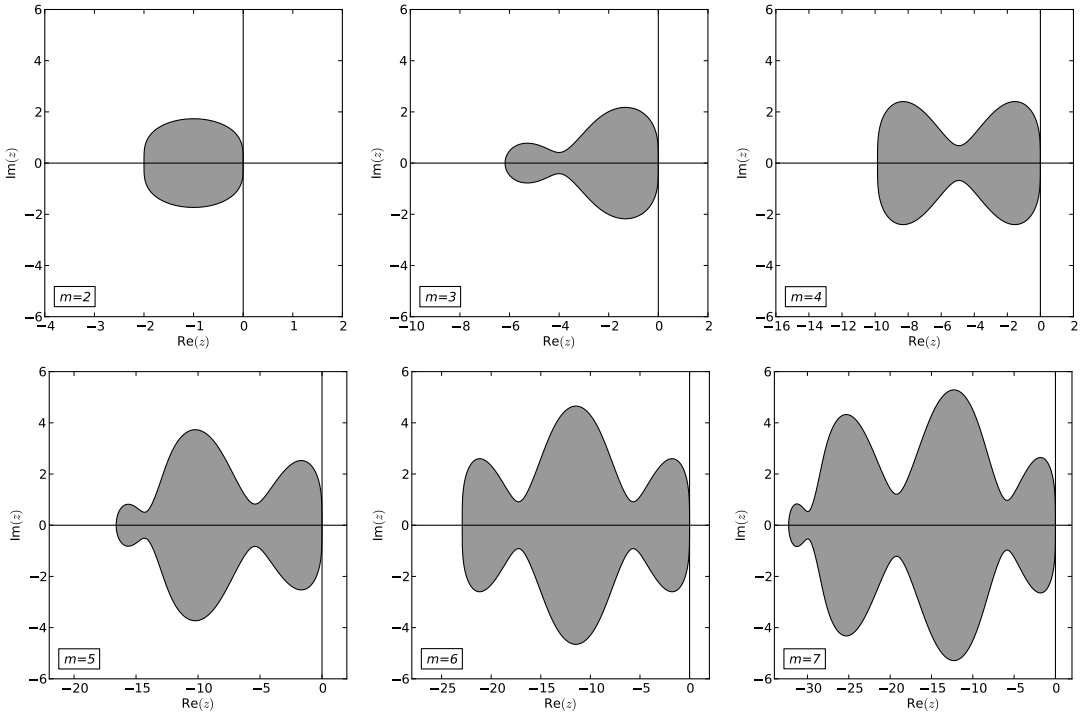

Figure 2.5: Regions of absolute stability, calculated from the stability function, for the RKC2 method of stages two through seven.

2.3.6 Rosenbrock Methods

Steps for implicit methods can be expensive because they often require the solution of large non-linear systems. DIRK methods can be more efficient but still require the solution of a non-linear system at each stage. The class of Rosenbrock methods attempts to mitigate the issue by linearly approximating RK stages of a DIRK method. Therefore, the DIRK method can be transformed such that it only needs to solve linear systems. This approach departs from the traditional class of RK methods because, due to the linear approximation, it cannot be written as a single Butcher tableau. As an optimization, the Jacobian in the linear approximation is evaluated at $\mathbf{y}_{n-1}$; thus, stages for Rosenbrock methods are derived as

$$\mathbf{k}_i = \Delta t_n \mathbf{f}\left(t_{n-1} + \alpha_i \Delta t_n, \mathbf{y}_{n-1} + \sum_{j=1}^{i-1} \alpha_{ij}\mathbf{k}_j\right) \tag{2.12}$$
$$+ (\Delta t_n)^2 \frac{\partial \mathbf{f}}{\partial t}(t_{n-1}, \mathbf{y}_{n-1}) + \Delta t_n \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_{n-1}, \mathbf{y}_{n-1}) \sum_{j=1}^{i} \gamma_{ij}\mathbf{k}_i,$$

where the coefficients α_i, α_{ij}, and γ_{ij} satisfy similar order conditions to those of an RK method. In this formulation of a Rosenbrock method, $\mathbf{y}_n$ is calculated identically to RK method, using coefficients b_i to weight stages.

Implementation of a Rosenbrock method can be much more efficient than directly using the above definition. With the substitution $\mathbf{u}_i = \sum_{j=1}^{i} \gamma_{ij}\mathbf{k}_j$, (2.12) can be rewritten as [21]

$$\begin{aligned}\left(\frac{1}{\gamma_{ii}\Delta t_n}\mathbf{I} - \mathbf{J}\right)\mathbf{u}_i = \mathbf{f}&\left(t_{n-1} + \alpha_i\Delta t_n, \mathbf{y}_{n-1} + \sum_{j=1}^{i-1} a_{ij}\mathbf{u}_j\right) \\ &+ \sum_{j=1}^{i-1}\left(\frac{c_{ij}}{\Delta t_n}\right)\mathbf{u}_j + \gamma_i\Delta t_n\frac{\partial \mathbf{f}}{\partial t}(t_{n-1}, \mathbf{y}_{n-1}), \\ \mathbf{y}_n = \mathbf{y}_{n-1} &+ \sum_{j=1}^{s} m_j\mathbf{u}_j,\end{aligned}$$

where $\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_{n-1}, \mathbf{y}_{n-1})$, $\alpha_i = \sum_{i=1}^{i-1}\alpha_{ij}$, $\gamma_i = \sum_{i=1}^{i}\gamma_{ij}$, the matrix of a_{ij} coefficients is defined as

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1s} \\ a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ a_{s1} & a_{s2} & \dots & a_{ss} \end{pmatrix} = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1s} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{s1} & \alpha_{s2} & \dots & \alpha_{ss} \end{pmatrix} \begin{pmatrix} \gamma_{11} & \gamma_{12} & \dots & \gamma_{1s} \\ \gamma_{21} & \gamma_{22} & \dots & \gamma_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{s1} & \gamma_{s2} & \dots & \gamma_{ss} \end{pmatrix}^{-1},$$

and the vector of m_j coefficients is defined as

$$\begin{pmatrix} m_1 & m_2 & \dots & m_s \end{pmatrix} = \begin{pmatrix} b_1 & b_2 & \dots & b_s \end{pmatrix} \begin{pmatrix} \gamma_{11} & \gamma_{12} & \dots & \gamma_{1s} \\ \gamma_{21} & \gamma_{22} & \dots & \gamma_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{s1} & \gamma_{s2} & \dots & \gamma_{ss} \end{pmatrix}^{-1}.$$

Rosenbrock methods with strict tolerances are used as a way to generate reference solutions for problems in this thesis. The RODAS method is a fourth-order Rosenbrock method with coefficients chosen for computational and stability optimization. The stability function for RODAS is shown in Figure 2.6.

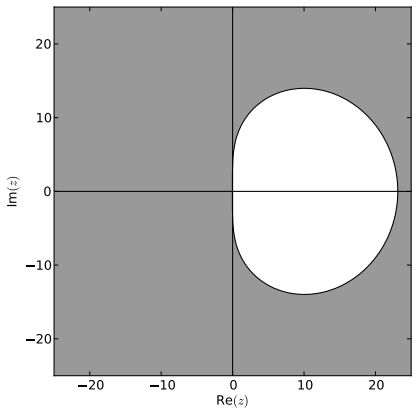

Figure 2.6: Region of absolute stability, calculated from the stability function, for the RODAS method.

2.3.7 Exponential Runge–Kutta Methods

A number of generalized methods exists for solving IVPs; the solution to the IVP is evolved in $\mathbb{R}^m$, where m is the size of the system. However, if the solution of the IVP is known to evolve on a specific sub-manifold of $\mathbb{R}^m$, perhaps better methods can be derived that use information from this sub-manifold [22]. Methods that preserve structure are particularly applicable to mechanical systems involving conservation of mass, energy, or momentum. For example, consider a problem where the solution is constrained to a fixed, circular orbit. That orbit is the manifold on which the solution evolves; thus, it is desirable for each step of an IVP integrator to yield a result on that manifold. Another example is rigid-body dynamics, where bodies move in SE(3) [22].

Exponential RK (ExpRK) methods are an example of these so-called structure-preserving methods [35]. This thesis restricts analysis of ExpRK methods to those based on matrix multiplications and exponentials. The IVPs studied are those of the form

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t}(t) = \mathbf{A}(t, \mathbf{y})\mathbf{y}(t), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \tag{2.13}$$

where $\mathbf{y} \in \mathbb{R}^m$ and $\mathbf{A} : \mathbb{R} \times \mathbb{R}^m \mapsto \mathbb{R}^{m \times m}$, because performing operations on matrices is well understood and arises in many applications. Many IVPs exist that can be written in this form. If $\mathbf{A}$ is constant, this problem has an exact solution of $\mathbf{y}(t) = \exp(t\mathbf{A})\,\mathbf{y}_0$. When $\mathbf{A}$ is not constant, freezing $\mathbf{A}$ at the beginning of the timestep and taking a step using the exact linearized solution yields an ExpRK method that is analogous to FE for classical RK methods. The method, known as exponential Euler, is written as [8]:

$$\mathbf{y}_n = \mathbf{y}_{n-1} + \exp\left[\Delta t_n \mathbf{A}(t_{n-1}, \mathbf{y}_{n-1})\right] \mathbf{y}_{n-1}.$$

As with classical RK, the ExpRK methods can be extended to higher order, as first introduced by Crouch and Grossman [12]. In 2005, Celledoni introduced the commutator-free exponential Runge-Kutta (CFERK) method [7]. For an IVP of the form (2.13), the s-stage CFERK method is written

$$\mathbf{k}_i = \left[\prod_{k=J}^{1} \exp\left(\Delta t_n \sum_{j=1}^{s} a_{ij}^{[k]} \mathbf{A}(\mathbf{k}_j)\right)\right] \mathbf{y}_{n-1},$$
$$\mathbf{y}_n = \left[\prod_{k=J}^{1} \exp\left(\Delta t_n \sum_{j=1}^{s} b_{j}^{[k]} \mathbf{A}(\mathbf{k}_j)\right)\right] \mathbf{y}_{n-1},$$

where $i = 1, 2, \ldots, s$ and there are J Butcher tableaux; $a_{ij}^{[k]}$ refers to a_{ij} in tableau k and $b_j^{[k]}$ refers to b_j in tableau k.

2.4 Methods for 2-Additively Split Problems

There exist many numerical methods for IVPs that are 2-additively split. In general, these methods are equally applicable to all 2-additive IVPs, regardless of whether the IVP has been split using physics-based splitting or Jacobian splitting. However, the performance of methods might vary, depending on the manner in which the IVP was split.

Numerical methods for 2-additive IVPs are particularly useful for problems that are split such that one split part is relatively stiff and the other is relatively non-stiff. Typically, when considering an additive splitting of the form of (1.2), $\mathbf{f}_I$ is assumed to be stiff and $\mathbf{f}_E$

is assumed to be non-stiff. Applying an implicit method to $\mathbf{f}_I$ and an explicit method to $\mathbf{f}_E$ could result in a method that performs better than a non-additive method applied to the entire, non-split IVP.

Physics-based splitting only uses an implicit method on the term that is more stiff. Jacobian splitting is thought to be superior to physics-based splitting because it treats the stiffness from both terms implicitly and the non-stiff remainder explicitly. It is based on the premise that the Jacobian of the RHS captures the stiffness of an IVP and should be treated implicitly. This may be reasonable because linear stability analysis is generally effective in predicting the suitability of a numerical method for a given IVP. The stability regions of methods influence how well a method performs on a problem. Those methods with unbounded regions of absolute stability typically perform better on relatively stiff problems. It is therefore reasonable that a 2-additive method should treat the (stiff) linearized part of the RHS with an implicit method and the (nonstiff) remainder of the RHS explicitly.

2.4.1 Additive Runge–Kutta Methods

An additively split IVP can be solved by applying a distinct RK method to each of its contributing factors. Such an approach forms a class of numerical methods known as additive Runge–Kutta (ARK) methods. In practice, the most common form of ARK methods are 2-additive. Assuming the IVP has already been split (using physical characteristics or using Jacobian splitting) into two functions $\mathbf{f}_I(t, \mathbf{y})$ and $\mathbf{f}_E(t, \mathbf{y})$, as in (1.2), the general form of a 2-additive ARK method is written as

$$\begin{aligned}
\mathbf{k}_i &= \mathbf{f}_I\left(t_{n-1} + c_i\Delta t_n, y_{n-1} + \Delta t_n \sum_{j=1}^{s}\left[a_{ij}\mathbf{k}_j + \tilde{a}_{ij}\tilde{\mathbf{k}}_j\right]\right), \\
\tilde{\mathbf{k}}_i &= \mathbf{f}_E\left(t_{n-1} + \tilde{c}_i\Delta t_n, y_{n-1} + \Delta t_n \sum_{j=1}^{s}\left[a_{ij}\mathbf{k}_j + \tilde{a}_{ij}\tilde{\mathbf{k}}_j\right]\right), \\
i &= 1, 2, \ldots, s \\
\mathbf{y}_n &= \mathbf{y}_{n-1} + \Delta t_n \sum_{i=1}^{s}\left[b_i\mathbf{k}_i + \tilde{b}_i\tilde{\mathbf{k}}_\mathbf{i}\right].
\end{aligned}$$

The associated Butcher tableaux are given in Table 2.4.

c_1	$a_{1,1}$	$\dots$	$a_{1,s}$	$\tilde{c}_1$	$\tilde{a}_{1,1}$	$\dots$	$\tilde{a}_{1,s}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_s	$a_{s,1}$	$\dots$	$a_{s,s}$	$\tilde{c}_s$	$\tilde{a}_{s,1}$	$\dots$	$\tilde{a}_{s,s}$
	b_1	$\dots$	b_s		$\tilde{b}_1$	$\dots$	$\tilde{b}_s$

Table 2.4: The Butcher tableaux for a 2-additive RK method.

Each RK method has its own Butcher tableau subject to standard order conditions; see Appendix A. The coefficients are also subject to *coupling conditions* between the two RK methods. The ARK method is able to achieve order p provided each method is independently of at least order p and that the appropriate coupling conditions are satisfied. The Butcher tableaux for a third-order ARK method is given in Table 2.5 [30].

DIRK Method

0	0	0	0	0
$\frac{1767732205903}{2027836641118}$	$\frac{1767732205903}{4055673282236}$	$\frac{1767732205903}{4055673282236}$	0	0
$\frac{3}{5}$	$\frac{2746238789719}{10658868560708}$	$-\frac{640167445237}{6845629431997}$	$\frac{1767732205903}{4055673282236}$	0
1	$\frac{1471266399579}{7840856788654}$	$-\frac{4482444167858}{7529755066697}$	$\frac{11266239266428}{11593286722821}$	$\frac{1767732205903}{4055673282236}$
Main	$\frac{1471266399579}{7840856788654}$	$-\frac{4482444167858}{7529755066697}$	$\frac{11266239266428}{11593286722821}$	$\frac{1767732205903}{4055673282236}$
Aux.	$\frac{2756255671327}{12835298489170}$	$-\frac{10771552573575}{22201958757719}$	$\frac{9247589265047}{10645013368117}$	$\frac{2193209047091}{5459859503100}$

ERK Method

0	0	0	0	0
$\frac{1767732205903}{2027836641118}$	$\frac{1767732205903}{2027836641118}$	0	0	0
$\frac{3}{5}$	$\frac{5535828885825}{10492691773637}$	$\frac{788022342437}{10882634858940}$	0	0
1	$\frac{6485989280629}{16251701735622}$	$-\frac{4246266847089}{9704473918619}$	$\frac{10755448449292}{10357097424841}$	0
Main	$\frac{1471266399579}{7840856788654}$	$-\frac{4482444167858}{7529755066697}$	$\frac{11266239266428}{11593286722821}$	$\frac{1767732205903}{4055673282236}$
Aux.	$\frac{2756255671327}{12835298489170}$	$-\frac{10771552573575}{22201958757719}$	$\frac{9247589265047}{10645013368117}$	$\frac{2193209047091}{5459859503100}$

Table 2.5: Butcher tableaux for a third-order ARK method.

IVPs often have stiff and non-stiff additively split contributing factors. The stiff and non-stiff factors are usually treated with implicit and explicit methods, respectively. Methods

of this type form a class known as implicit-explicit (IMEX) Runge–Kutta methods. IMEX methods are a subset of ARK methods; they can be defined using the same notation as a general ARK method. Recall that the matrix representation of the Butcher tableau for an ERK method is lower triangular. Many of the methods in this thesis are of the IMEX class; for the stiff term of the IVP, they require the solution of a generally non-linear system.

When applying Jacobian splitting to ARK methods, the Jacobian is computed once at the beginning of each step and then frozen for the duration of the step. The stiff term $\mathbf{f}_I$ of the split IVP is linear; therefore, it can be solved by implicit methods using a single Newton iteration. Jacobian splitting might have additional performance gains over physics-based splitting if $\mathbf{f}_I$ from the physics-based splitting is non-linear.

Note that the computation of a Jacobian matrix is generally required at every step by methods that are implicit or use Jacobian splitting. For physics-based splitting, the computation is of $\frac{\partial \mathbf{f}_I}{\partial \mathbf{y}}$; for Jacobian splitting, the computation is of $\frac{\partial \mathbf{f}}{\partial \mathbf{y}}$. Even if no systems of equations are solved, i.e., the numerical method is fully explicit, the Jacobian is still required at each step to perform Jacobian splitting; however, using a fully explicit approach in conjunction with Jacobian splitting is likely not practical. Computing Jacobian matrices can be computationally expensive, but it is necessary for all IMEX methods, regardless of whether Jacobian splitting is used. Jacobian splitting is no more complex to implement than physics-based splitting because both splittings involve computation of a Jacobian matrix and are otherwise generally comparable in implementational difficulty. For the numerical experiments in this thesis, the computation of Jacobian matrices is further discussed in Section 3.1.4.

2.4.2 Stability for 2-Additive Runge–Kutta Methods

Stability analysis for ARK methods is similar to that of RK methods. The test equation $\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y + i\mu y$ is used, where $\lambda, \mu \in \mathbb{R}$ and $i = \sqrt{-1}$. It is assumed that the first split term has real eigenvalues and the second has imaginary eigenvalues; therefore, stability plots can still be represented on a two-dimensional plot. Application of an ARK method to the test

equation is written as

$$
\begin{aligned}
k_i &= \lambda y_{n-1} + \Delta t_n \lambda \sum_{j=1}^{s} (a_{ij} k_i + \tilde{a}_{ij} \tilde{k}_i), \\
\tilde{k}_i &= i\mu y_{n-1} + \Delta t_n i\mu \sum_{j=1}^{s} (a_{ij} k_i + \tilde{a}_{ij} \tilde{k}_i), \\
y_n &= y_{n-1} + \Delta t_n \sum_{i=1}^{s} (b_i k_i + \tilde{b}_i \tilde{k}_i).
\end{aligned} \tag{2.14}
$$

Using matrix notation, (2.14) can be rewritten as

$$
\begin{aligned}
\mathbf{k} &= \lambda y_{n-1} \mathbf{1} + \Delta t_n \lambda (\mathbf{A}\mathbf{k} + \tilde{\mathbf{A}}\tilde{\mathbf{k}}), \\
\tilde{\mathbf{k}} &= i\mu y_{n-1} \mathbf{1} + \Delta t_n i\mu (\mathbf{A}\mathbf{k} + \tilde{\mathbf{A}}\tilde{\mathbf{k}}), \\
y_n &= y_{n-1} + \Delta t_n (\mathbf{b}^{\mathrm{T}}\mathbf{k} + \tilde{\mathbf{b}}^{\mathrm{T}}\tilde{\mathbf{k}}),
\end{aligned} \tag{2.15}
$$

where $\mathbf{k} = (k_1, k_2, \ldots, k_s)^{\mathrm{T}}$, $\mathbf{1} = (1, 1, \ldots, 1)^{\mathrm{T}} \in \mathbb{R}^s$, and $\mathbf{b} = (b_1, b_2, \ldots, b_s)^{\mathrm{T}}$. Solving for $\mathbf{k}$ and $\tilde{\mathbf{k}}$, (2.15) can be rearranged as

$$
\begin{aligned}
\mathbf{k} &= \lambda y_{n-1} (\mathbf{I} - \Delta t_n \lambda \mathbf{A} - \Delta t_n i\mu \tilde{\mathbf{A}})^{-1} \mathbf{1}, \\
\tilde{\mathbf{k}} &= i\mu y_{n-1} (\mathbf{I} - \Delta t_n \lambda \mathbf{A} - \Delta t_n i\mu \tilde{\mathbf{A}})^{-1} \mathbf{1}.
\end{aligned}
$$

Substituting the stages into (2.15) yields

$$
y_n = \left[1 + \Delta t_n (\lambda \mathbf{b}^{\mathrm{T}} + i\mu \tilde{\mathbf{b}}^{\mathrm{T}}) (\mathbf{I} - \Delta t_n \lambda \mathbf{A} - \Delta t_n i\mu \tilde{\mathbf{A}})^{-1} \mathbf{1} \right] y_{n-1},
$$

where $\mathbf{I} \in \mathbb{R}^{s \times s}$ is the identity matrix. The IVP step can now be written $y_n = R\big(\Delta t_n \lambda, \Delta t_n \mu\big) y_{n-1}$. Stability functions for IMEX methods ARK3, ARK4, and ARK5 are shown in Figure 2.7.

2.4.3 The Partitioned Runge–Kutta–Chebyshev Method

For additively split problems, there might be a dominating stiff component and a non-stiff component. It might be possible to solve a split IVP accurately with fewer function evaluations of the non-stiff component in comparison to those of the stiff component. One approach

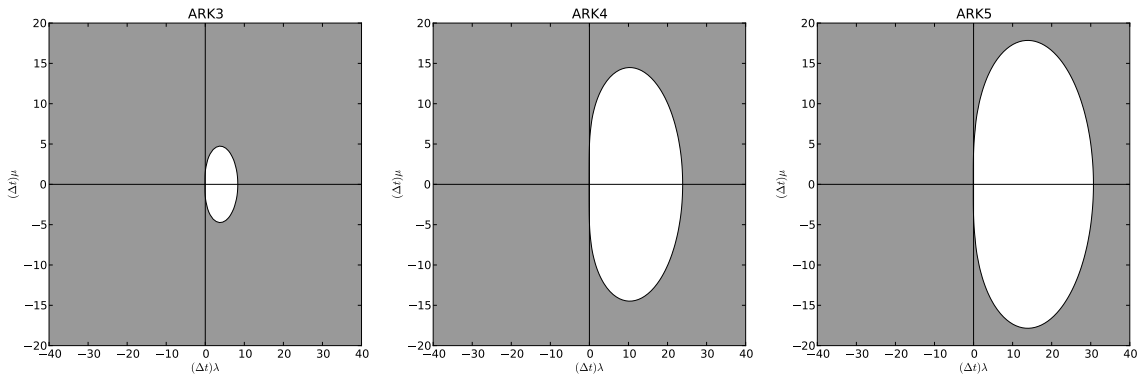

Figure 2.7: Regions of absolute stability, calculated from the stability function, for IMEX methods ranging from orders three to five.

is to apply RKC to the stiff component of the IVP (possibly taking many function evaluations) and have only four function evaluations of the non-stiff component [49]. This method is known as the partitioned Runge–Kutta–Chebyshev (PRKC) method. Like RKC, it is aimed at problems with eigenvalues that are near the negative real axis. The s-stage formula for the second-order PRKC method is written as

$$
\begin{aligned}
\mathbf{k}_{-1} &= \mathbf{y}_{n-1}, \\
\mathbf{k}_0 &= \mathbf{k}_{-1} + \alpha_0 \Delta t_n \mathbf{g}_{-1}, \\
\mathbf{k}_1 &= \mathbf{k}_0 + \kappa_1 \Delta t_n \mathbf{f}_0, \\
\mathbf{k}_j &= (1 - \mu_j - \nu_j)\mathbf{k}_0 + \mu_j \mathbf{k}_{j-1} + \nu_j \mathbf{k}_{j-2} + \kappa_j \Delta t_n \mathbf{f}_{j-1} - a_{j-1}\kappa_j \Delta t_n \mathbf{f}_0, \\
j &= 1, 2, \ldots, s
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{k}_s &= (1 - \mu_s - \nu_s)\mathbf{k}_0 + \mu_s \mathbf{k}_{s-1} + \nu_s \mathbf{k}_{s-2} + \kappa_s \Delta t_n \mathbf{f}_{s-1} - a_{s-1}\kappa_j \Delta t_n \mathbf{f}_0 \\
&\quad + \alpha_1 \Delta t_n \mathbf{g}_{-1} + \alpha_2 \Delta t_n \mathbf{g}_0 + \alpha_3 \Delta t_n \mathbf{g}_{s-1}, \\
\mathbf{k}_{s+1} &= (1 - \mu_s - \nu_s)\mathbf{k}_0 + \mu_s \mathbf{k}_{s-1} + \nu_s \mathbf{k}_{s-2} + \kappa_s \Delta t_n \mathbf{f}_{s-1} - a_{s-1}\kappa_s \Delta t_n \mathbf{f}_0, \\
&\quad + \alpha_4 \Delta t_n \mathbf{g}_{-1} + \alpha_5 \Delta t_n \mathbf{g}_0 + \alpha_6 \Delta t_n \mathbf{g}_{s-1} + \alpha_7 \Delta t_n \mathbf{g}_s, \\
\mathbf{y}_n &= \mathbf{k}_{s+1}.
\end{aligned}
$$

For $j = 1, 2, \ldots, s$, the coefficients μ_j, ν_j, κ_j, a_j, b_j, c_j, ω_0, and ω_1 are identical to the

second-order RKC method. The remaining coefficients are defined as follows:

$$\mathbf{f}_i = \mathbf{f}_I(t_{n-1} + c_i\Delta t_n, \mathbf{k}_i), \qquad \mathbf{g}_{-1} = \mathbf{f}_I(t_{n-1}, \mathbf{k}_{-1}),$$
$$\mathbf{g}_0 = \mathbf{f}_E(t_{n-1} + \alpha_0\Delta t_n, \mathbf{k}_0), \quad \mathbf{g}_{s-1} = \mathbf{f}_E(t_{n-1} + \alpha_0\Delta t_n, \mathbf{k}_{s-1}), \quad \mathbf{g}_s = \mathbf{f}_E(t_{n-1} + \Delta t_n, \mathbf{k}_s),$$

$$\alpha_0 = \frac{1}{2}, \qquad \alpha_1 = -\frac{1}{2} + v(3-4v), \qquad \alpha_2 = 2v(2v-1) - \alpha_3, \qquad \alpha_4 = \frac{1-3v}{6v},$$
$$\alpha_5 = \frac{1-3v}{6v}, \quad \alpha_6 = \frac{1+3v(1-2v)+4c_{m-1}v(3v-2)}{6c_{m-1}v(2v-1)}, \qquad \alpha_7 = \frac{1}{6v(2v-1)},$$

where α_3 and v are free parameters. Stability plots for PRKC are shown in Figure 2.8. The stability polynomial for PRKC is written as [49]

$$\begin{aligned} R(\Delta t_n\lambda, \Delta t_n\mu) =& R_s(\Delta t_n\lambda)\left[1 + (\alpha_0+\alpha_7)\Delta t_n i\mu + \alpha_0\alpha_7(\Delta t_n i\mu)^2\right] \\ &+ R_{s-1}(\Delta t_n\lambda)\left[\alpha_6\Delta t_n i\mu + (\alpha_0\alpha_6 + \alpha_3\alpha_7)(\Delta t_n i\mu)^2 + \alpha_0\alpha_3\alpha_7(\Delta t_n i\mu)^3\right] \\ &+ (\alpha_4+\alpha_5)\Delta t_n i\mu + (\alpha_0\alpha_5 + (\alpha_1+\alpha_2)\alpha_7)(\Delta t_n i\mu)^2 + \alpha_0\alpha_2\alpha_7(\Delta t_n i\mu)^3. \end{aligned}$$

2.4.4 The Implicit-Explicit Runge–Kutta–Chebyshev Method

The implicit-explicit Runge-Kutta-Chebyshev (IRKC) method is based on RKC methods. IRKC is designed for 2-additively split IVPs whereby one contributing factor is treated with RKC and the other contributing factor is treated with the BE method [40]. Conceptually, a fundamental difference between PRKC and IRKC is that the PRKC method integrates the second contributing factor explicitly, whereas the IRKC method integrates the second contributing factor implicitly. IRKC is particularly applicable to IVPs in where the elements of the term treated with BE are generally independent, i.e., $\frac{\mathrm{d}y_i}{\mathrm{d}t}$ only depends on y_i (and t) for all $i = 1, 2, \ldots, m$. Therefore, at every step, m one-dimensional linear systems are solved, making the method much more efficient in comparison to solving a single, larger $m \times m$ system.

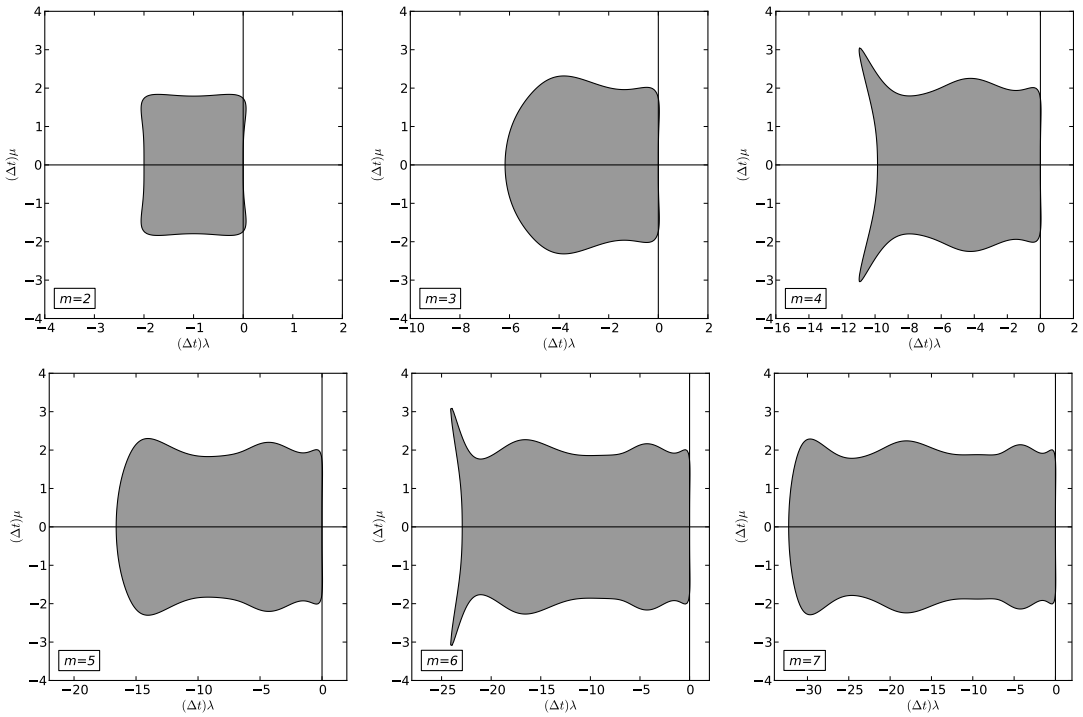

Figure 2.8: Regions of absolute stability, calculated from the stability function, for the PRKC method of stages two through seven.

The algorithm for a step for the IRKC method is written as [40]

$$
\begin{aligned}
\mathbf{k}_0 &= \mathbf{y}_{n-1},\\
\mathbf{k}_1 &= \mathbf{k}_0 + \kappa_1 \Delta t_n (\mathbf{f}_0 + \mathbf{g}_1),\\
\mathbf{k}_j &= (1-\mu_j-\nu_j)\mathbf{k}_0 + \mu_j \mathbf{k}_{j-1} + \nu_j \mathbf{k}_{j-2} + \kappa_j \Delta t_n \mathbf{f}_{j-1} - a_{j-1}\kappa_j \Delta t_n \mathbf{f}_0\\
&\quad - [a_{j-1}\kappa_j + (1-\mu_j-\nu_j)\kappa_1]\Delta t_n \mathbf{g}_0 - \nu_j \kappa_1 \Delta t_n \mathbf{g}_{j-2} + \kappa_1 \Delta t_n \mathbf{g}_j,\\
\mathbf{f}_j &= \mathbf{f}\big(t_{n-1} + c_j \Delta t_n, \mathbf{k}_j\big), \qquad \mathbf{g}_j = \mathbf{g}\big(t_{n-1} + c_j \Delta t_n, \mathbf{k}_j\big),\\
j &= 1, 2, \ldots, s
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{y}_n &= \mathbf{k}_s,\\
c_0 &= 0, \qquad c_1 = \frac{\omega_1}{\omega_0}, \qquad c_j = \mu_j c_{j-1} + \nu_j c_{j-2} + a_j(1+\kappa_j).
\end{aligned}
$$

All of the coefficients of IRKC are identical to the RKC method described in Section 2.3.5 with the exception of $b_0 = \frac{1}{4\omega_0^2}$ and $b_1 = \frac{1}{\omega_0}$. The stability polynomial for IRKC is written

as [40]

$$R(z) = 1 - b_s T_s(w_0) + b_s T_s \left(w_0 + w_1 \frac{z}{1 - \frac{w_1}{w_0}\mathrm{Im}(z)} \right), \tag{2.16}$$

and is shown for stages two through seven in Figure 2.9.

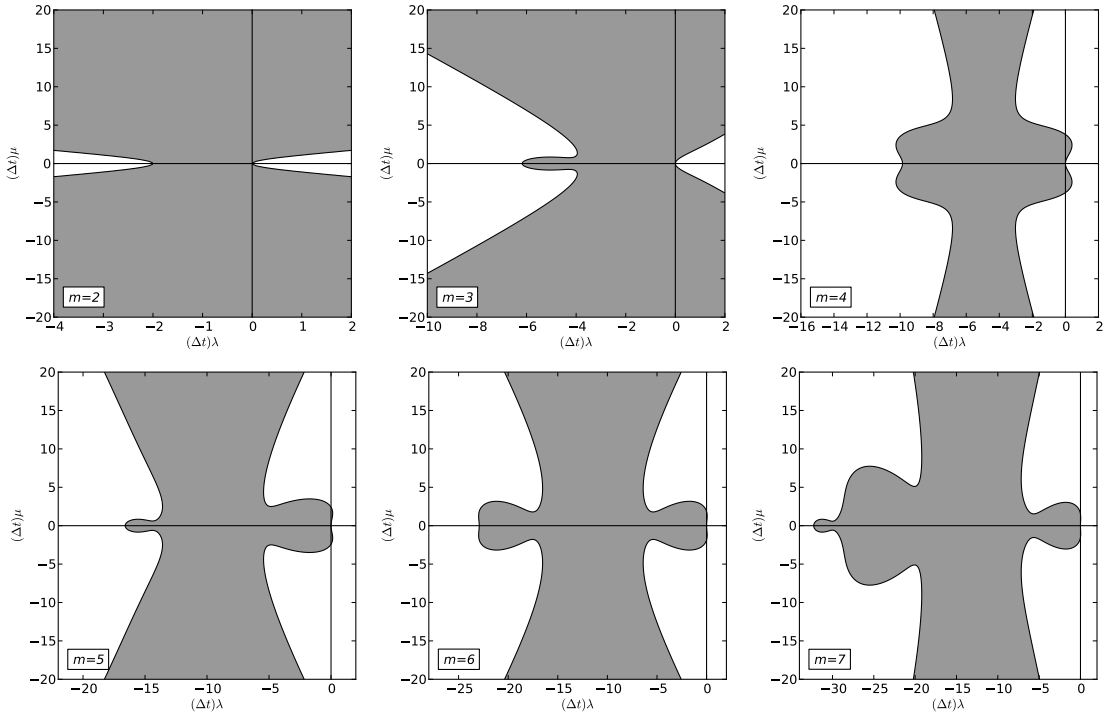

Figure 2.9: Regions of absolute stability, calculated from the stability function, for the IRKC method of stages two through seven.

2.4.5 Additive Exponential Runge–Kutta Methods

An ExpRK method can be used in conjunction with classical RK method to derive a 2-additive method [8]. Many IVPs have a physics-based splitting of the form

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}_I(t, \mathbf{y}) + \mathbf{A}(t, \mathbf{y})\mathbf{y}, \qquad \mathbf{y}(0) = \mathbf{y}_0, \tag{2.17}$$

where $\mathbf{y} \in \mathbb{R}^m$, $\mathbf{f}_I : \mathbb{R} \times \mathbb{R}^m \mapsto \mathbb{R}^m$, and $\mathbf{A} : \mathbb{R} \times \mathbb{R}^m \mapsto \mathbb{R}^{m \times m}$. This form lends itself well to the application of 2-additive ExpRK methods. One approach is to treat the first term $\mathbf{f}_I$ with a (generally implicit) classical RK method and the second term $\mathbf{A}(t, \mathbf{y})\mathbf{y}$ with a (generally

explicit) CFERK method. It is of interest to determine whether an additive ExpRK method has superior performance compared to classical ARK methods (using either physics-based or Jacobian splitting).

The additive s-stage ExpRK method applied to (2.17) is written [8]

$$\begin{aligned}
\mathbf{y}_i &= \mathbf{\Phi}_i \mathbf{y}_{n-1} + \Delta t_n \sum_{j=1}^{s} a_{ij} \mathbf{\Phi}_i \mathbf{\Phi}_j^{-1} \mathbf{f}(\mathbf{y}_j), \qquad \mathbf{\Phi}_i = \prod_{k=J}^{1} \exp\left(\Delta t_n \sum_{j=1}^{s} \alpha_{ij}^{[k]} \mathbf{C}(\mathbf{y}_j) \right), \\
\mathbf{y}_n &= \mathbf{\Phi}_n \mathbf{y}_{n-1} + \Delta t_n \sum_{j=1}^{s} b_j \mathbf{\Phi}_n \mathbf{\Phi}_j^{-1} \mathbf{f}(\mathbf{y}_j), \qquad \mathbf{\Phi}_n = \prod_{k=J}^{1} \exp\left(\Delta t_n \sum_{j=1}^{s} \beta_j^{[k]} \mathbf{C}(\mathbf{y}_j) \right),
\end{aligned} \tag{2.18}$$

where $i = 1, 2, ..., s$, the a_{ij} and b_j are the coefficients of the classical RK method, and the $\alpha_{ij}^{[k]}$ and $\beta_j^{[k]}$ are the coefficients of the ExpRK method.

A common approach is to use a DIRK method as the classical RK method and an explicit ExpRK method, forming a class of methods known as the DIRK-CF methods; otherwise, the system of equations to solve the implicit stages involves a matrix exponential, which can be computationally expensive. Specifics on the derivation of DIRK-CF methods is given in Appendix C. The numerical stability of these methods is largely determined by the classical RK method; the stability function is given as [8]

$$R(\Delta t_n \lambda, \Delta t_n \mu) = \exp(i \Delta t_n \mu) R(\Delta t_n \lambda),$$

where $R(\Delta t \lambda)$ is the stability function of the classical RK method.

This thesis considers two DIRK-CF methods of orders 1 and 2, which are named DIRK-CF1 and DIRK-CF2. The Butcher tableaux for these methods are given in Figure 2.10. Many IVPs are defined such that $\mathbf{f}_I$ in (2.17) can be further split, either using physics-based or Jacobian splitting. Such splittings transform (2.17) into a 3-additive problem, which is discussed as part of potential future work, but is beyond the scope of this thesis.

DIRK-CF1

0	0
	1

1	1
	1

DIRK-CF2

0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0
	1	0

0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$
	0	1

Figure 2.10: Butcher tableaux for the DIRK-CF1 and DIRK-CF2 methods. The classical RK tableaux are given on the left and the ExpRK tableaux are given on the right.

Chapter 3

Methods and Software

Numerical methods are applied to additively split IVPs to determine which methods have the best performance. The most important metric is a method's runtime with respect to the accuracy of the solution it produces. One can then choose an appropriate method when solving other, similar IVPs. Other metrics include statistics gathered from method runs, such as the number of steps, stages per step, or number of Newton iterations per step. These additional statistics can be used to identify bottlenecks in PSE code and can guide future work for improving the overall performance of a method. For instance, if a method takes fewer but more expensive steps, optimizations to decrease its time per step can be expected to improve its overall performance.

As part of this research, the `pythODE++` PSE was developed to perform the experiments in this thesis, rather than using existing PSEs, such as `MAPLE` or `MATLAB`, to solve IVPs. Although those PSEs are excellent tools for solving IVPs, they focus mainly on providing solutions to problems using a set of well-tested solvers designed for different types of problems. They are not designed to conveniently run extensive evaluations over a parameter set to evaluate how well methods perform on different types of problems.

The infrastructure of the developed software for this thesis makes it possible to generate data over large parameter sets and to make fair comparisons between numerical methods. In this study, `pythODE++` is used to generate reference solutions, run experiments, plot eigenvalues of the Jacobian of the RHS of IVPs, generate stability plots, and generate plots of CPU time and the number of steps versus accuracy.

3.1 Details of pythODE++

Experiments are conducted using the pythODE++ PSE, which is designed to evaluate permutations of numerical methods, tolerances, and IVPs. The pythODE++ PSE iterates over all specified parameter combinations, and, after all runs are completed, it performs a thorough analysis of experimental data. An analysis tool is customizable to generate statistics for all aspects of the experiment.

The pythODE++ PSE is a sophisticated software suite that has been developed to support the vast number of numerical experiments required to perform a rigorous study of physics-based splitting versus Jacobian splitting. It is heavily based on pythODE, a PSE for ODEs written entirely in Python [31] (rather than a combination of Python and C++). Many parts of the code in pythODE++ are direct ports from pythODE. The pythODE PSE is designed to be a general tool, giving a user fine-grained control over the entire solution process for manipulating methods and gathering statistics. However, as a piece of software becomes more robust and generalized, its infrastructure and supporting codebase dramatically increase in size and complexity. A specialized piece of software requires less infrastructure and is more suitable to specific tasks. General-purpose software often involves some sacrifice of performance. The pythODE++ PSE used in this thesis is more specialized than pythODE and has been designed to support a subset of the functionality that pythODE offers. It solves IVPs much faster than pythODE due to the decreased overhead. It is meant to be used in conjunction with pythODE for experiments that require greater runtime efficiency than what pythODE can provide.

Choice of language is important to software development. In this thesis, high-level languages are assumed to be interpreted languages, whereas low-level languages run natively and require code to be compiled. High-level languages offer more features and flexibility, allowing code to be produced quickly. Additionally, they are generally much better suited to portability across different operating systems. However, high-level languages achieve these feats at the expense of runtime speed and efficiency. Typically, high-level languages have a runtime overhead that is an order of magnitude larger than languages that are compiled. Although many high-level languages contain vectorized operations that provide significant

performance increases, high-level languages still cannot compete with the speed of low-level, compiled languages.

Different implementations of numerical methods can produce the same numerical result but have drastically different runtimes due to the manner in which they are implemented. For example, one implementation of an RK method might dynamically allocate memory at every stage, whereas another might allocate an initial block of memory to be used by all stages. Allocating an initial block of memory is clearly more efficient. It is important to consider the fairness with which numerical methods are implemented, such that runtime bias caused by implementation is mitigated. However, the most efficient implementations of numerical methods are not required to perform meaningful comparisons. Such efficiency is not achievable while maintaining the versatility of the PSE in this work, nor is it possible to know whether an implementation is truly optimal.

The implementation of `pythODE++` uses concepts from inheritance and polymorphism. There are three important base classes of the PSE software from which all other classes are derived. First, an *IVP class* defines the problem being solved. Second, a *method class* defines the algorithms used for steps and for error prediction. Finally, a *solver class* serves as the master controller that instructs the method class to step through the IVP with a specified timestep. Polymorphism allows for a common interface in order to access functions and attributes of all types of IVPs, methods, and solvers. For instance, a method can evaluate the RHS of an IVP without knowledge of problem-specific details of the IVP. Solvers can also instruct methods to take timesteps without knowledge of method-specific details.

In this thesis, the goal of `pythODE++` is to strike a balance between versatility and efficiency. All solvers, methods, and IVPs are developed entirely in C++ because it is a fast and powerful language [38]. C++ interfaces directly with memory and low-level libraries; at the same time, it allows the use of many complex data structures such as templates and polymorphic classes. The infrastructure of the PSE is heavily influenced by object-oriented software engineering principles; all components are modularized to allow for maximum flexibility, yet they all draw from a central set of parameters, thus permitting easy inter-module communication and coherence. The supporting infrastructure of the PSE, involving the specification of parameters and post-run analysis, is written in Python. The runtime measurement for

pre- and post-processing of experiments is not important to the behaviour of the numerical methods; those components do not need to be optimized using a low-level language.

3.1.1 Classes of Methods

When one is required to solve an IVP, a straightforward approach such as the FE method or the RK4 method is commonly used; see Section 2.3. For many small or simple problems, these approaches are sufficient because modern computers can solve problems so quickly and accurately. In such cases, there is no reason to investigate better methods for solving the problem. However, as simulations become larger and more complex, simple methods become inefficient and cannot always provide solutions with acceptable accuracy in reasonable amounts of time.

A complete list of the methods evaluated in this thesis is shown in Table 3.1. The table also includes the orders of the methods as well as the orders of their associated auxiliary methods used for error prediction; see Section 2.3.2. FE, BE, and RK4 are included to provide a basis for comparison. If those three methods were to outperform newer, more complicated, higher-order methods, there would no reason to investigate further. The Dormand–Prince, Zonneveld, Merson, and Verner methods are all explicit RK methods with steps that are controlled using standard step controllers. They are not expected to perform well once numerical experiments become stiff but are evaluated for as a basis for comparison because they represent simple higher-order methods. RKC1 and RKC2 are methods of the RKC family. PRKC and IRKC are respectively the fully explicit (so-called partitioned) and IMEX methods based on RKC methods. The ARK methods are IMEX methods used to study which type of splitting is superior; these IMEX methods treat $\mathbf{f}_I$ with an SDIRK method and $\mathbf{f}_E$ with an explicit method. The DIRK-CF1 and DIRK-CF2 methods are IMEX additive ExpRK methods of orders one and two. RODAS and RADAU5 are chosen to be the methods for generating reference solutions. Recall that RODAS is a six-stage, fourth-order Rosenbrock method and RADAU5 is a three-stage, fifth-order IRK method. RODAS is a first choice for all IVPs; it is less expensive per step in comparison to RADAU5 because RODAS only requires the solution of a single linear system per step; see Section 2.3.6. However, for sufficiently stiff IVPs, RADAU5 is more efficient. In Section 3.4, Table 3.5 shows the problems for which

RODAS is used and the problems for which RADAU5 is used to find reference solutions. In principle, any consistent numerical method can be used to calculate reference solutions. The RODAS and RADAU5 methods were chosen because they are known to perform well on many different types of IVPs.

Method Name	**Order**	**Auxiliary Order**
Forward Euler	1	N/A
Backward Euler	1	N/A
Dormand–Prince [20]	5	4
Zonneveld [20]	4	3
Merson [20]	4	3
Verner [20]	6	5
RKC1 [46]	1	N/A
RKC2 [46]	2	N/A
PRKC [49]	2	N/A
IRKC [40]	2	N/A
ARK3 [2]	3	2
ARK4 [30]	4	3
ARK5 [30]	5	4
DIRK-CF1 [8]	1	N/A
DIRK-CF2 [8]	2	N/A
RODAS [21]	4	3
RADAU5 [21]	4	3

Table 3.1: The methods used in this thesis.

3.1.2 Software Components

Each IVP must define components identified by a base IVP class, such that the IVP can be used by other entities in the PSE, e.g., numerical methods or solvers. The required compo-

nents are a class constructor, a definition of the RHS, and a unique name for the IVP; the optional components are definitions for N-additive splittings, functions to evaluate analytic Jacobian matrices, and functions to evaluated the analytic time derivatives. The components are described in Table 3.2. All problems in this thesis that are used to compare Jacobian and physics-based splittings have RHSs that are 2-additively split; for ease of implementation, a class derived from the base IVP class has been developed to define an interface of components specific to 2-additive splittings. These components include the RHSs of the two contributing factors and, optionally, functions to evaluate their respective analytic Jacobian matrices and analytic time derivatives. Each 2-additively split IVP should define the components listed in Table 3.3, rather than those of Table 3.2.

Required Components	
Constructor	Responsible for allocating and setting an initial condition, the degree of additive splitting, initial and final times, and any other problem variables.
Right-hand side	A function that defines the RHS of an IVP, i.e., it is given arguments t and $\mathbf{y}$ and it returns the derivative $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t}$.
Unique name for the IVP	A function that returns a human-readable identifier for the IVP that should be unique to avoid confusion during analysis.
Optional Components	
Split components	Definitions for physics-based split components of the RHS, similar to the definition of the entire RHS.
Analytic Jacobian matrices	Definitions that specify the analytic Jacobian of the RHS, as well as the Jacobian for additively split components of the RHS.
Analytic time derivatives	Definitions that specific the analytic time derivatives of the RHS, as well as the analytic time derivatives for additively split components of the RHS.

Table 3.2: Components to define an IVP in the `pythODE++` PSE.

Similar to IVPs, numerical methods their own sets of components, identified by a base

Required Components	
Constructor	Responsible for allocating and setting an initial condition, initial and final times, and any other problem variables.
Split component 1	A function that evaluates the first split component of the RHS of a 2-additively split IVP. The function is of the same form as the RHS for an IVP that is not split.
Split component 2	A function that evaluates the second split component of the RHS of a 2-additively split IVP. The function is of the same form as the RHS for an IVP that is not split.
Unique name for the IVP	Human-readable identifier for the 2-additively split IVP that should be unique to avoid confusion during analysis.
Optional Components	
Analytic Jacobian 1	Function returning the Jacobian of the first term in a physics-based splitting.
Analytic Jacobian 2	Function returning the Jacobian of the second term in a physics-based splitting.
Analytic Time Derivative 1	Function returning the time-derivative of the RHS of the first term in a physics-based splitting.
Analytic Time Derivative 2	Function returning the time-derivative of the RHS of the second term in a physics-based splitting.

Table 3.3: Components to define a 2-additively split IVP in the `pythODE++` PSE.

method class, that must be implemented. The mandatory components include a class constructor, a function returning the order of the method, a unique name for the method, and the algorithm by which the method computes a step. The optional components include definitions for functions that are called post and prior to each step (useful for methods like RKC where the number of stages is calculated before taking a step) and a function defining the auxiliary order of the method. These components are described in Table 3.4.

Required Components	
Constructor	Responsible for allocating method-specific memory buffers.
Order for the method	A function that specifies the order of the method for use in analysis and for step controllers.
Unique name for the method	Human-readable identifier for the numerical method that should be unique to avoid confusing during analysis.
Method step	Defines how a method steps the solution $\mathbf{y}_{n-1}$ to $\mathbf{y}_n$, given a timestep.
Optional Components	
Pre-step	Function to calculate memory, number of stages, adjust timestep, etc., before the method is called.
Post-step	Function to adjust solutions after the step has been called.
Auxiliary Order	A function that specifies the auxiliary order of the method for use in analysis and for step controllers. This function is required for methods that use embedded error control.

Table 3.4: Components to define a method in the `pythODE++` PSE.

Three solvers have been implemented. The first solves IVPs using a constant stepsize, used primarily for verification of numerical methods. The second solver is an embedded solver that adjusts step-size based on an error estimate given by the numerical method. The third solver uses step-doubling, appropriate for methods such as FE or BE that do not have embedded methods to compute an error estimate. These solvers are responsible for

conducting timings, which are measured using wall-clock time from when the solver begins taking steps to after all steps have been completed. Having the solver responsible for timings, rather than measuring the execution time of an entire process, removes the time required for initializing solver processes and other startup costs that are not relevant to the numerical study.

The object hierarchy of `pythODE++` is shown in Figure 3.1. Square boxes represent derived classes that are fully implemented. Boxes with rounded corners denote problems and methods that are templates for use in implementing the methods and problems. Users can easily add new methods or problems into the framework, merging them into the existing code base.

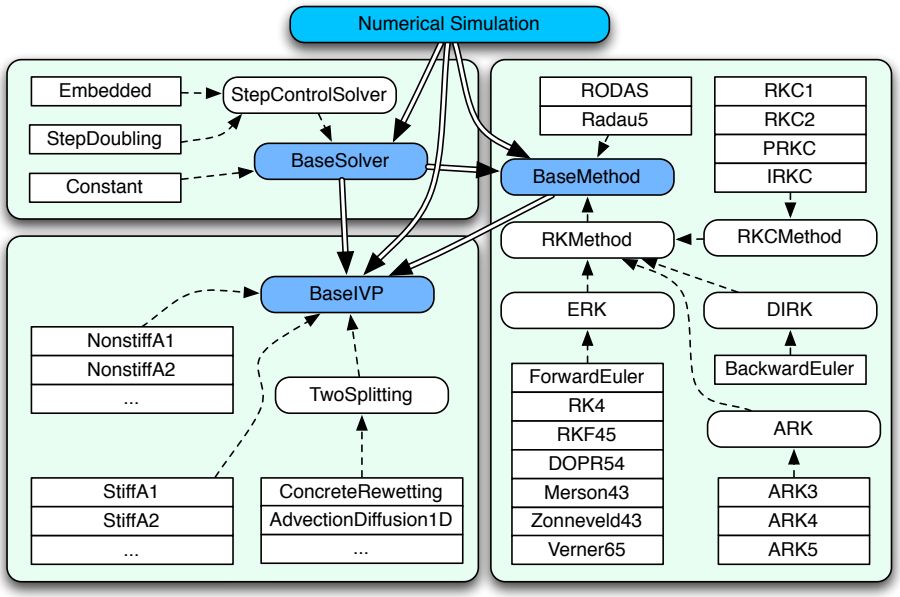

Figure 3.1: Software organization of the `pythODE++` PSE

3.1.3 Supporting Classes

The software developed for this thesis is comprised of a significant amount of supporting infrastructure, including classes and functions to represent vectors, matrices, hashes, and heaps. All classes inherit from a common input/output class, which is used to output solutions and is also useful for debugging. Output can be directed to disk or to the terminal. A powerful infrastructure is one of the primary advantages to using a PSE, as opposed to

implementing a set of methods and problems independently for each numerical study. Significant development time is saved when constructs, such as vectors, are reusable across many problem and method implementations.

The `pythODE++` PSE codebase relies on relatively few external libraries, such that it can be finely tuned to the specific needs of numerical methods, without unnecessary overhead that could be incurred by irrelevant computation in external libraries. If implementation is abstracted, it is difficult to see whether there are any application-specific optimizations that can be performed. Writing implementations for the supporting infrastructure from scratch facilitates gathering specific statistics by instrumenting them directly into the code. Subsequently, it is much easier to identify potential bottlenecks in the code, e.g., unnecessary matrix copies that consume a large amount of CPU time. The C++ standard template library is used for sections of the code that are not performance sensitive, such as output and post-run analysis. Note that when conducting timings, all disk input/output is disabled.

The vector class contains implementation for many common vector-vector and vector-scalar arithmetic operations. Examples include addition, subtraction, scalar multiplication, inner product, and norms. The vector class is templated; therefore, vectors can be created for any recognized data type (both built-in and custom). Although the internal format is stored as a flattened array, this class also supports two-dimensional indexing, which is useful for many IVPs.

The matrix class contains implementation for the matrix-matrix, matrix-vector, and matrix-scalar operations, such as multiplication operations. It also contains a direct solver for dense linear systems that is based on LU factorization [45]. The reason behind this decision over the use of an external library is that direct linear solvers are notorious for having hidden parallelism built into their code. Therefore, the use of external libraries for common operations in such performance sensitive code can affect the fairness in which methods are implemented.

An implicit method requires the solution of a generally non-linear system of equations that is solved using modified Newton's method; see Section 2.3.1. Newton's method requires the Jacobian of the non-linear system. The `pythODE++` PSE supports three methods of Jacobian computation: finite differences (centred and forward), automatic differentiation

(AD), and manual analytic implementation. The finite difference approach computes each value of the Jacobian using a finite difference approximation; see Section 3.2.1. Note that computing the Jacobian using central differences is relatively inefficient[1] because it requires on the order of twice as many function evaluations in comparison to a forward finite difference. AD calculates the Jacobian matrix (more generally, any derivative) by applying the rules of differential calculus to the source code of a function, either via code transformation or via operator overloading. The AD and manual analytic approaches both provide true Jacobian matrices.

3.1.4 Sparsity

At each iteration of Newton's method (used by implicit methods), a system of linear equations must be solved. The matrix of this linear system is the Jacobian of the non-linear equation being solved; recall Algorithm 1 that shows the modified Newton's method. The discussion pertains to the Jacobian of the non-linear system, rather than the Jacobian of the RHS of an IVP (although, the structure of the former is generally based on the latter). Jacobian matrices of stages in an RK method are comprised mostly of zeros; i.e., they are sparse. Therefore, it is desirable to exploit sparsity during the Newton iterations. In practice, large systems should be solved using routines that exploit sparsity to increase the computational efficiency. Additionally, Jacobian matrices of large systems of non-linear equations can quickly become too large to be stored in a dense format.

A sparse matrix class was developed for `pythODE++` to store elements of the matrix in the compressed row storage (CRS) format [37]. This format represents the matrix using three arrays: the first array contains all of the elements in row-major format, the second array specifies the column of each corresponding element, and the third array contains a set of offsets describing at what point in the value and column arrays each row starts. Such a format significantly decreases the amount of memory required to store the matrix in comparison to storing each of the zero entries.

[1] Computing Jacobian matrices using central differences is more costly but more accurate than using forward differences. However, a cheap estimate to the Jacobian is usually sufficient. Although Newton's method may require more iterations when using a Jacobian that is less accurate, in practice, the additional cost of these iterations is low in comparison to a Jacobian that is relatively inexpensive to compute.

This sparse matrix class supports the same set of operations as the dense matrix class; therefore, it is easy to substitute sparsity at any location in the methods or solvers. Infrastructure has also been implemented in `pythODE++` to seamlessly convert between the dense and sparse formats. Note that when using the CRS format, operations such as direct element access are slow; however, such direct-access operations are uncommon for the Jacobian of RK stages. Direct-access operations are much more common for accessing the **A** matrix in a Butcher tableau. Butcher tableaux are small and dense; therefore, it is not advantageous to use sparsity for these matrices.

The `pythODE++` PSE contains implementations for both dense and sparse linear solvers. The dense solvers are included for purposes of verification. When performing timings for method comparisons, all systems of equations are solved by taking advantage of sparsity. The software uses `UMFPACK` [13], which is a library that contains routines for direct solutions to sparse matrices. It has been verified that all internal operations of `UMFPACK` run in serial; therefore, no interference occurs when solving multiple IVPs simultaneously.

3.1.5 Automatic Differentiation

All implicit methods and those that use Jacobian splitting require the computation of the Jacobian at the beginning of each timestep. This computation is a significant part of the overall time-cost per step. It is, therefore, crucial that computation of Jacobian matrices be done efficiently; otherwise, the majority of the computation time per step is spent calculating the Jacobian, and benefit due to Jacobian splitting might go unnoticed.

The `pythODE++` PSE supports computation of (sparse and dense) Jacobian matrices using `ADOL-C` [47], an operator-overloading approach to automatic differentiation (rather than an approach that instruments the code during compilation). `ADOL-C` supports sparse matrices via `ColPack` [18], a library designed to solve graph colouring problems, which lend themselves well to the computation of sparsity patterns for sparse Jacobian matrices.

`ADOL-C` is easy to implement into pre-existing C++ code. All that is required is to substitute floating-point variables in the code with the `ADOL-C` data type, `adouble`. Such substitutions are made easy with C++ templates. The following example of an RHS shows how, with templates, a single function can represent the RHS if it is called with a built-in

floating-point type (typically `double`) and the Jacobian if it is called with the `adouble` type.

Consider the following RHS:

$$
\begin{aligned}
y_1' &= y_3, \\
y_2' &= y_4, \\
y_3' &= -y_1(y_1^2 + y_2^2)^{-3/2}, \\
y_4' &= -y_2(y_1^2 + y_2^2)^{-3/2}.
\end{aligned}
$$

This function describes the RHS of a two-dimensional orbital problem, which is part of a set of IVPs used for method verification in the `pythODE++` PSE. The corresponding code using C++ templates might look something like:

```
template<typename T>
void RHS(const T t, const Vec<T>& y, Vec<T>& yp) {
    T denom = pow(sqr(y[0]) + sqr(y[1]), -1.5);
    yp[0] = y[2];
    yp[1] = y[3];
    yp[2] = -y[0]*denom;
    yp[3] = -y[1]*denom;
}
```

This code seamlessly allows for both built-in floating-point types and `ADOL-C` data types because the compiler generates separate code for the cases when `T` is of type `adouble` and `T` is of type `double`. The functions `pow` and `sqr` are part of the C++ standard math library; they are overloaded by `ADOL-C` to support the `adouble` type in addition to the supported built-in types. The alternative to `ADOL-C` is to manually create sparse Jacobian matrices for each IVP and each of the split components. Although this approach is error-prone, it can often be more efficient compared to using `ADOL-C`.

3.1.6 Solving Initial-Value Problems Simultaneously

The numerical experiments in this thesis involve the solution of many independent IVPs. Therefore, these experiments are well suited to parallelization; there is no need for process synchronization because each IVP is independent. Processes distributed across several nodes

are dispatched instructions, which take the form of parameter lists. These parameters specify which IVPs, methods, and solvers to use for a given run.

Suppose one wanted to run a numerical experiment for the concrete-rewetting IVP using the RODAS method with an embedded solver, a spatial domain discretized into 100 unknowns, a sink boundary condition, an isopropanol solution, a Jacobian calculated using forward finite differences, absolute and relative tolerances of 10^{-5}, and an initial time-step of 0.001. Section 3.2.1 describes the parameters specific to the concrete-rewetting problem, i.e., the number of unknowns, the boundary condition, and the choice of isopropanol solution. The code given to `pythODE++` specifying the experiment might look like:

```
{
  'ivp': 'ConcreteRewetting',
  'method': 'RODAS',
  'solver': 'EmbeddedSolver',
  'N': 100,
  'sink␣bc': True,
  'isopropanol': True,
  'jacobian': Forward,
  'atol': 1e-5,
  'rtol': 1e-5,
  'dt': 1e-3
}
```

This list of parameters instructs a process to solve the specified IVP using the `ivp` key. It chooses the RODAS method based on the `method` key and an embedded solver using the `solver` key. The keys `N`, `sink bc`, and `isopropanol` are IVP-specific parameters handled explicitly by the IVP code. The method of calculating the Jacobian matrix is specified using the `jacobian` key. Absolute tolerances, relative tolerances, and the timestep are specified by the `atol`, `rtol`, and `dt` keys, respectively.

One master process manages the distribution of work to maintain a balanced load across all processes. After the experimental runs are complete, there is an added expense of transferring data to the master process to be used for analysis. If all processes write to the same file system, this last step is not required because it is implicitly part of the run; otherwise, each process must transmit its solution data to the master process.

The solvers for `pythODE++` mimic the design of `pythODE`. They are designed to evaluate

problems in a specified parameter space across multiple machines. A major advantage to having solvers that are completely independent of each other is that they can be run across heterogeneous, non-clustered machines. Support for heterogeneity is important during development because scientists often do not have unrestricted access to large computing clusters for extended periods of time. Algorithm 2 defines the work-flow of the master process. Each worker process is comprised of two threads. The first awaits instruction from the master process and reports the current status to the master process. The second computes the solution to the IVP.

Once experimental runs are complete, `pythODE++` performs statistical analysis, examining the CPU time, the number of steps, and the average number of stages per step. To avoid biasing results when running timings, it is important that all machines have the same hardware specifications and use scratch memory (memory on a local disc) to avoid network latency incurred by transmitting data to the master node.

Algorithm 2 Logic for the manager process that dispatches runs to worker processes.

```
R ← set of runs
H ← set of hosts
F ← ∅                                   ▷ Initialize the set of free hosts as empty.
for r in R do                           ▷ Loop over each set of run parameters.
    while F = ∅ do                      ▷ Loop until there are free hosts.
        F ← GetFreeHosts(H)
        if F = ∅ then
            Sleep(500ms)                ▷ Sleep to avoid aggressive polling.
        end if
    end while
    h ←pop(H)
    Dispatch r to h                     ▷ Dispatch problem run.
end for
while F ≠ H do                          ▷ Loop until all hosts have completed.
    Sleep(500ms)                        ▷ Sleep to avoid aggressive polling.
    F ← GetFreeHosts(H)
end while
```

3.1.7 Analysis

After all runs are complete, a separate program that is part of the PSE is responsible for gathering run information and presenting plots according to specified parameters. Plots are

generated using `gnuplot` [48]. The data for plots are specified using sets of search criteria, specifying the relevant experimental runs.

The distinct advantage to performing analysis with the PSE is that the storage and analysis of data collected from runs are abstracted from the user. The user simply provides selection criteria to determine which runs are relevant. The analyzer is implemented to select all runs that match a set of selection parameters. For example, one can specify all tolerances for a subset of all methods for one IVP. There is also a list of parameter combinations, known as a discard list, which allows certain runs to be discarded after they have been matched based on the selection parameters. For example, one might wish to discard tolerances that are more lax than a specific threshold for a specific method because those tolerances were found to produce oscillations in the solution. Sample analysis parameters are presented as follows:

```
{
  'mode': 'Accuracy',
  'comparison': 'time',
  'reference␣run': {'ivp': 'ConcreteRewetting',
                    'N':    100,
                    'sink␣bc': True,
                    'isopropanol': True,
                    'method': 'RODAS',
                    'atol': 1e-12 },
  'match': {'ivp': 'ConcreteRewetting',
            'N':    100,
            'sink␣bc': True,
            'isopropanol': True },
  'group': ['method','solver','jacobian␣splitting'],
  'discard': []
}
```

These parameters produce an accuracy plot with respect to time for the concrete-rewetting problem. The RODAS method, with an absolute tolerance of 10^{-12}, is used as a reference solution. When finding a single match for the reference solution, it is not necessary to specify the relative tolerance because, in this example, a single relative tolerance is used in association with each absolute tolerance; therefore, only one run matches the specified absolute tolerance. Likewise, it is not necessary in this example to specify the solver class that was used because only the embedded solver was used. The resulting accuracy plot then consists of all concrete-rewetting problems with parameters specified by `match`, grouping them such that all groups

contain the same value for method, solver, and Jacobian splitting, as specified by `group`. In this example, no specific combinations of parameters are discarded.

The analysis software currently supports several types of plots. It supports plots of the entire solution by a specified method, phase portraits of any two solution components, and the plot of any attribute versus accuracy with respect to some other reference values or reference solution method.

3.2 Discretization of Partial Differential Equations

Numerical methods for IVPs of the form (1.1) are not directly applicable to PDEs because PDEs contain spatial derivatives. PDEs must be spatially discretized such that the spatial domain is represented by a set of discrete values rather than continuously. This transforms the PDEs to a (large) set of coupled ODEs for each point in the spatial discretization. Once a PDE has been transformed into a system of ODEs, numerical methods for IVPs can be applied.

The discretization technique is known as the *method of lines*, where each "line" refers to the evolution of the state of the system in time at a given point in space. The PDE must therefore hold for all discretized points. Each discrete point is a function of time and it is advanced by the numerical method. Approximations to the spatial derivatives at the discretized points can be computed using the value at that point and the values of nearby points. This thesis considers two such approaches to the computation of spatial derivatives on a discretized domain: finite differences and finite volume.

3.2.1 Finite Difference Methods

Finite difference methods are methods to numerically approximate derivatives of various orders using linear combinations of the points on a discretized grid. Although extensible to non-uniform grids, finite differences are often applied to a grid with uniform point spacing. The idea is to use a *stencil* for each point indicating which other points are to be used in calculation of the derivative. The points in the stencil are weighted such that the stencil yields approximations to the derivative of a certain order of accuracy in comparison to a

Taylor expansion (recall (2.3), but substitute temporal variables with spatial variables). The first few terms of the expansion are written as

$$u(x+\Delta x) = u(x) + (\Delta x)\frac{\partial u}{\partial x}(x) + \frac{(\Delta x)^2}{2}\frac{\partial^2 u}{\partial x^2}(x) + \mathcal{O}\big((\Delta x)^3\big). \tag{3.1}$$

The simplest form is a first-order forward finite difference method, which can be easily derived by rearranging (3.1) to solve for $\frac{\partial u}{\partial x}(x)$. The result is

$$\frac{\partial u}{\partial x}(x) = \frac{u(x+\Delta x) - u(x)}{\Delta x} + \mathcal{O}\big((\Delta x)^2\big). \tag{3.2}$$

An alternative first-order scheme is the backward difference method, derived with the substitution $\Delta x \to -\Delta x$ into equation (3.2). The backward difference method is written

$$\frac{\partial u}{\partial x}(x) = \frac{u(x) - u(x-\Delta x)}{\Delta x} + \mathcal{O}\big((\Delta x)^2\big).$$

A centred finite difference is derived by combining forward and backward differences. It requires grid points on either side of x. It is written as

$$\frac{\partial u}{\partial x}(x) = \frac{u(x+\Delta x) - u(x-\Delta x)}{2\Delta x} + \mathcal{O}\big((\Delta x)^3\big).$$

Notice that such a combination cancels the second-order terms in both Taylor series; therefore, the centred difference is second order (the terms of order three and higher still remain).

Higher-order finite difference methods can be derived by similarly combining lower-order finite differences with different substitutions for Δx, e.g., $\Delta x \to 2\Delta x$ and $\Delta x \to -2\Delta x$ to include an additional two data points on either side of x. A general method for calculating asymmetric finite differences in one dimension given in Appendix B.

3.2.2 Finite Volume Methods

Finite volume methods constitute another approach to numerically approximating derivatives via discretization. Finite differences use a differential form, i.e., the Taylor series, whereas finite volume methods compute derivatives using an integral form. For example, consider the

one-dimensional PDE

$$\frac{\partial q}{\partial t}(x,t) = \frac{\partial q}{\partial x}(x,t), \tag{3.3}$$

where x is defined on some continuous domain of length L. The domain can be uniformly discretized into N cells, each of which has width $\Delta x = L/N$. The centre of cell i is denoted by x_i and borders of cell i are denoted by $x_{i\pm\frac{1}{2}}$. The average "volume" of q in each cell i can be written

$$q_i(t) = \frac{1}{\Delta x}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} q(x,t)\,\mathrm{d}x.$$

Differentiating both sides of the equation with respect to time and using (3.3) yields

$$\frac{\mathrm{d}q_i}{\mathrm{d}t}(t) = \frac{1}{\Delta x}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \frac{\partial q}{\partial x}(x,t)\,\mathrm{d}x.$$

Applying the fundamental theorem of calculus yields a discretization of (3.3):

$$\frac{\partial q_i}{\partial t}(t) = \frac{q_{i+\frac{1}{2}}(t) - q_{i-\frac{1}{2}}(t)}{\Delta x}. \tag{3.4}$$

Note that the form (3.4) does not yet contain any approximation. The values $q_{i\pm\frac{1}{2}}$ lie on the borders between the cells; there are no solutions stored at those points because the solutions are stored at cell centres. One approach is to compute those values at half-indices using averages of the adjacent cells. This approach yields the same result as a second-order centred finite difference. An advantage of the finite volume method over the finite difference method is that the finite volume method can use a non-uniform discretization more easily than the finite differences can; however, such unstructured meshes are beyond the scope of this thesis.

3.3 Advection-Diffusion-Reaction Equations

This thesis focuses on the study of methods for advection-diffusion-reaction (ADR) equations, which are a class of additively split problems where *advection*, *diffusion*, and *reaction*

terms form the contributing factors. ADR equations are PDEs and, thus, must be spatially discretized as described in Section 3.2. In the following definitions, n refers to the spatial dimension of the problem; i.e., generally one, two, or three dimensions, although, in principle, ADR equations are defined for arbitrary $n \in \mathbb{Z}^+$. The contributing factors of an ADR equation are defined as follows.

Advection is the time-dependent transport of a quantity in a domain caused by the bulk motion of the medium in which the quantity resides [25]. The velocity of the medium is represented as a vector field of dimension n and is defined over the entire domain, determining the direction in and magnitude by which the quantity is transported. Mathematically, for a quantity q defined for all points in the domain, an advection equation is written

$$\frac{\partial q}{\partial t} + \nabla \cdot [q\mathbf{u}(t, \mathbf{x}, q)] = 0, \tag{3.5a}$$

where $t \in \mathbb{R}$ is time, $\mathbf{x} \in \mathbb{R}^n$ represents a position in the domain, and $\mathbf{u} \in \mathbb{R}^n$ is the velocity field of the medium. The advection equation is linear when $\mathbf{u}$ is not a function of q and can thus be written as $\mathbf{u}(t, \mathbf{x})$.

Diffusion is the time-dependent movement of a quantity in a domain along its concentration gradient [25]. The quantity in regions of higher concentration generally moves toward regions with lower concentration, with the aim of equalizing the distribution of the quantity over the entire domain. The diffusion quantity can be a continuous substance, such as heat, or it can refer to discrete particles that move from regions of high particle density to low particle density. In physical models, particle density is often treated continuously as a concentration defined over a domain. A quantification of how well particles or quantities diffuse is known as *diffusivity*. A diffusion equation is written

$$\frac{\partial q}{\partial t} = \nabla \cdot [\mathbf{D}(t, \mathbf{x}, q)\nabla q], \tag{3.5b}$$

where $\mathbf{D}(t, \mathbf{x}, q) \in \mathbb{R}^{n \times n}$ is the diffusivity tensor. Similar to the advection equation, the diffusion equation is linear when $\mathbf{D}$ is not a function of q and can thus be written as $\mathbf{D}(t, \mathbf{x})$.

Reaction is the physical transformation of one or more quantities in a domain [25]. Reaction typically refers to some form of chemical reaction, but it also can be used to represent

source and sink terms of the quantity. A reaction equation is written

$$\frac{\partial q}{\partial t} = r(t, \mathbf{x}, q), \tag{3.5c}$$

where $r(t, \mathbf{x}, q)$ is a function representing the reaction. A reaction is linear when r is linear with respect to q.

Combining the contributions from (3.5) forms the complete ADR equation. It is written

$$\frac{\partial q}{\partial t} + \nabla \cdot [q\mathbf{u}(t, \mathbf{x}, q)] = \nabla \cdot [\mathbf{D}(t, \mathbf{x}, q)\nabla q] + r(t, \mathbf{x}, q). \tag{3.6}$$

In ADR equations, the quantity q is scalar because it corresponds only to the behaviour of a single quantity. In order to model multiple quantities subject to ADR effects, a new ADR equation is required for each additional quantity. For example, an ADR model for $\mathbf{q} = (q_1, q_2)$ is written as

$$\begin{aligned}
\frac{\partial q_1}{\partial t} + \nabla \cdot [q_1\mathbf{u}_1(t, \mathbf{x}, \mathbf{q})] &= \nabla \cdot [\mathbf{D}_1(t, \mathbf{x}, \mathbf{q})\nabla q_1] + r_1(t, \mathbf{x}, \mathbf{q}), \\
\frac{\partial q_2}{\partial t} + \nabla \cdot [q_2\mathbf{u}_2(t, \mathbf{x}, \mathbf{q})] &= \nabla \cdot [\mathbf{D}_2(t, \mathbf{x}, \mathbf{q})\nabla q_2] + r_2(t, \mathbf{x}, \mathbf{q}).
\end{aligned}$$

Notice how the functions $\mathbf{u}$, r, and $\mathbf{D}$ from (3.6) have been extended to become functions of both q_1 and q_2, allowing for interaction between those two quantities.

3.4 Test Suite of Problems

The methods described in Section 3.1.1 are evaluated on a number of test problems over a range of tolerances, spatial discretization orders, and numbers of unknowns. These test problems comprise a wide range of numerical models with contributing factors arising from advection, diffusion, and reaction. This wide range of problems aims to show that Jacobian splitting, in comparison to physics-based splitting, is a desirable technique that should be seriously considered for practical computation. It is not simply an approach that is desirable for a few select problems.

The full list of numerical studies is defined in Table 3.5, where L signifies that the com-

ponent is linear with respect to $\mathbf{y}$, and NL signifies that the component is non-linear. The reference solution method used for each problem is also listed. The problems are fully defined in the following sections.

Numerical Study	**Advection**	**Diffusion**	**Reaction**	**Reference Solution Method**
Linear advection-diffusion	L	L		RODAS
2D Heat transfer	L	L		RODAS
`CUSP` model		L	NL	RODAS
1D Brusselator		L	NL	RODAS
2D Brusselator		L	NL	RADAU5
Combustion model	L	L	NL	RADAU5
Angiogenesis model	NL	L	NL	RADAU5
Concrete-Rewetting	NL	NL	NL	RADAU5

Table 3.5: List of all numerical studies. These are all ADR equations that are spatially discretized and solved in `pythODE++`. The last column gives the method that was used to calculate reference solutions.

All methods are verified on thirty problems from a test suite of non-stiff problems, thirty problems from a test suite of stiff problems, as well as the HIRES, Beam, and van der Pol problems from the Bari test set [15, 34]. During verification, the 2-additive methods apply Jacobian splitting to the IVPs from the test sets because those IVPs do not have physics-based splittings. All remaining problems can be defined using 2-additive splittings and are thus solved using physics-based splitting and Jacobian splitting.

The following problems are all PDEs with initial and boundary conditions. Sample discretizations for each of the problems are shown. Note that the parameters and unknowns for all IVPs are non-dimensionalized; in practice, it is common to solve IVPs using non-dimensionalized quantities and, after the IVPs have been solved, transform their solutions into quantities with physical units. Non-dimensionalized quantities are useful because they can be used to represent any physical scale.

3.4.1 Advection-Diffusion Problems

One-Dimensional Advection-Diffusion

Many physical processes are based on effects that arise from advection and diffusion. The first numerical study is a one-dimensional, constant-coefficient advection-diffusion equation, which is one of the most basic models. Subsequent numerical studies increase in complexity. Following the form (3.6), the advection-diffusion equation for a quantity q is written

$$\begin{aligned} \frac{\partial q}{\partial t} + a\frac{\partial q}{\partial x} &= d\frac{\partial^2 q}{\partial x^2}, \\ q(0,x) &= \sin(2\pi x), \\ q(t,0) &= q(t,1), \end{aligned} \tag{3.7}$$

where $x \in [0,1]$, and $a \in \mathbb{R}$ and $d \in \mathbb{R}$ are constant scalars representing the strength of advection and diffusion, respectively.

The spatial domain is uniformly discretized into N points with spacing $\Delta x = 1/N$; a second-order centred finite difference method yields a system of ODEs written as

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = d\left(\frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2}\right) - a\left(\frac{y_{i+1} - y_{i-1}}{\Delta x}\right), \tag{3.8}$$

where $i = 1, 2, \ldots, N$. Boundary conditions are periodic, as given in the problem definition; therefore, $y_0(t) = y_N(t)$ and $y_{N+1}(t) = y_1(t)$.

The one-dimensional, linear advection-diffusion IVP has been evaluated for varying levels of diffusivity and strength of the advection term. The evaluated parameters are $d = \{10, 100\}$ and $a = \{10, 100\}$, using 500 unknowns. The final simulation time is 0.1.

Two-Dimensional Heat Transfer Model

A model of heat transfer in a long, fluid-filled, cylindrical pipe with a known velocity profile is presented in [28]. This model is a two-dimensional linear advection-diffusion PDE. Such heat transfer models are often have application to mechanical engineering. In this model, heat is transferred down the pipe due to advection and diffusion processes. The interior of

the pipe is initially at a constant, relatively cool temperature. The input temperature to the pipe is hot and the conductance of the walls is prescribed; the output temperature is an unknown that is sought after some time interval.

The pipe is defined to be of length l and of height (diameter) h. It is aligned lengthwise in the x-direction; thus, the domain is given as $x \in [0, l]$ and $y \in [0, h]$. The PDE, boundary conditions, and initial condition of the model for temperature T are written

$$\begin{aligned} \frac{\partial T}{\partial t} &= d\,\nabla^2 T - \mathbf{u} \cdot \nabla T, \\ \frac{\partial T}{\partial y}(t, x, 0) &= -\frac{\partial T}{\partial y}(t, x, h) = \sigma, \\ T(t, 0, y) &= T_{\text{in}}, \qquad T(0, x, y) = T_0, \end{aligned} \tag{3.9}$$

where d represents the diffusivity of heat, $\mathbf{u}$ is the velocity field in the pipe, T_{in} is the input temperature, T_0 is the initial temperature of the pipe, and σ is the heat transfer across the pipe wall. Note that there is no prescribed boundary condition at $x = l$; this boundary is unknown and it is sought as part of the solution.

The fluid travels lengthwise down the pipe; therefore, velocity in the y-direction is zero and velocity in the x-direction is dependent on the distance to the pipe wall. The fluid velocity $\mathbf{u}$ is given based on average velocity u_{avg} as

$$\mathbf{u} = \begin{pmatrix} \frac{3}{2} u_{\text{avg}} \left[1 - \left(\frac{y}{h/2} \right)^2 \right] \\ 0 \end{pmatrix}.$$

This PDE is uniformly discretized on a two-dimensional grid, where the unknown grid points consist of N_x points in the x-direction and N_y points in the y-direction. These points are located in the middle-right side of each cell. Applying second-order centred differences to the second derivatives and first-order backward differences to the first derivatives yields a

discretized system that is written

$$\frac{\mathrm{d}T_{ij}}{\mathrm{d}t} = d\left(\frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{(\Delta x)^2} + \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{(\Delta y)^2}\right) - \frac{3}{2}u_{\text{avg}}\left[1 - \left(\frac{(i+\frac{1}{2})\Delta y}{h/2}\right)^2\right]\frac{T_{i,j} - T_{i-1,j}}{\Delta x}$$

where $i = 1, 2, \ldots, N_x$, $j = 1, 2, \ldots, N_y$, $\Delta x = \frac{l}{N_x}$, and $\Delta y = \frac{h}{N_y}$. The boundary values are, therefore, written

$$T_{0,j}(t) = T_{\text{in}}(t), \qquad T_{i,0}(t) = T_{i,1}(t) - (\Delta y)\sigma, \qquad T_{i,N+1}(t) = T_{i,N}(t) - (\Delta y)\sigma.$$

The heat transfer problem is evaluated with four sets of unknowns: 70×10, 140×20, 210×30, and 280×40. The problem parameters are chosen such that diffusivity d is 1.38×10^{-7}, average velocity u_{avg} is 4.5×10^{-6}, the pipe height h is 0.1, the pipe length l is 0.7, the initial temperature is 298.15, the inlet temperature is 303.15, the heat transfer through the pipe walls σ is 0, and the final simulation time t_f is 50.

3.4.2 Diffusion-Reaction Problems

The `CUSP` Problem

The `CUSP` problem is the combination of Zeeman's cusp catastrophe model and the van der Pol oscillator [25]. It is a diffusion-reaction equation that consists of three coupled PDEs. The diffusion of each PDE is linear (and non-coupled) and the reaction terms are non-linear. This particular system is of interest due to stiffness in the reaction term that is appropriately captured when using Jacobian splitting but not by physics-based splitting. The `CUSP` problem is defined over the domain $x \in [0, 1]$ and is written

$$\begin{aligned}
\frac{\partial y}{\partial t} &= \sigma\frac{\partial^2 y}{\partial x^2} - \frac{1}{\epsilon}(y^3 + ay + b), \\
\frac{\partial a}{\partial t} &= \sigma\frac{\partial^2 a}{\partial x^2} + b + \frac{0.07(y-0.7)(y-1.3)}{(y-0.7)(y-1.3)+0.1}, \\
\frac{\partial b}{\partial t} &= \sigma\frac{\partial^2 b}{\partial x^2} + (1-a^2)b - a - 0.4y + \frac{0.035(y-0.7)(y-1.3)}{(y-0.7)(y-1.3)+0.1},
\end{aligned} \tag{3.10}$$

where σ represents the diffusivity and ϵ is a parameter controlling the stiffness of the reaction. As ϵ approaches zero, the reaction in y becomes more stiff. These PDEs are uniformly discretized into N grid points using second-order centred differences; they are written as

$$\begin{aligned}
\frac{\mathrm{d}y_i}{\mathrm{d}t} &= \sigma\left(\frac{y_{i-1} - 2y_i + y_{i+1}}{(\Delta x)^2}\right) - \frac{1}{\epsilon}(y_i^3 + a_i y_i + b_i), \\
\frac{\mathrm{d}a_i}{\mathrm{d}t} &= \sigma\left(\frac{a_{i-1} - 2a_i + a_{i+1}}{(\Delta x)^2}\right) + b_i + \frac{0.07(y_i - 0.7)(y_i - 1.3)}{(y_i - 0.7)(y_i - 1.3) + 0.1}, \\
\frac{\mathrm{d}b_i}{\mathrm{d}t} &= \sigma\left(\frac{b_{i-1} - 2b_i + b_{i+1}}{(\Delta x)^2}\right) + (1 - a_i^2)b_i - a_i - 0.4y_i + \frac{0.035(y_i - 0.7)(y_i - 1.3)}{(y_i - 0.7)(y_i - 1.3) + 0.1},
\end{aligned}$$

where $i = 1, 2, \ldots, N$ and $\Delta x = 1/N^2$. The boundary conditions are given as periodic, and are, therefore, written as

$$\begin{aligned}
y_0(t) &= y_N(t), & a_0(t) &= a_N(t), & b_{N+1}(t) &= b_1(t), \\
y_{N+1}(t) &= y_1(t), & a_{N+1}(t) &= a_1(t), & b_{N+1}(t) &= b_1(t).
\end{aligned}$$

Initial conditions are given as

$$y_i(0) = 0, \qquad a_i(0) = -2\cos\left(\frac{2i\pi}{N}\right), \qquad b_i(0) = 2\sin\left(\frac{2i\pi}{N}\right).$$

The `CUSP` problem has been run for 500 and 1000 unknown grid points, where σ is $1/144$, ϵ is 10^{-4}, and the final simulation time t_f is 1.1. Therefore, the systems of ODEs have 1500 and 3000 unknowns, respectively, because there are three ODEs (for y, a, and b) at each point.

The Brusselator

The Brusselator is a theoretical model of an autocatalytic reaction consisting of two reacting quantities [25]. The model consists of two coupled diffusion-reaction equations, with linear

diffusion and non-linear reaction. They are written as

$$\begin{aligned} \frac{\partial u}{\partial t} &= \alpha \nabla^2 u + A + u^2 v - (B+1)u, \\ \frac{\partial v}{\partial t} &= \alpha \nabla^2 v + Bu - u^2 v, \end{aligned} \tag{3.11}$$

where A and B are parameters governing the reaction terms and α is the diffusivity. In this thesis, the Brusselator model is defined in both one and two dimensions. If $B > 1 + A^2$, the solution is oscillatory; otherwise, it approaches a steady state over time.

In the one-dimensional case, this thesis considers parameter selections where $A = 1$, $B = 3$, $\alpha = \{1/50, 1/500\}$, the spatial domain is given by $x \in [0, 1]$ and ∇ simplifies to $\frac{\partial}{\partial x}$. These parameters introduce oscillatory behaviour in the solution. Prescribed initial and Dirichlet boundary conditions are written as

$$\begin{aligned} u(0,t) &= u(1,t) = 1, & v(0,t) &= v(1,t) = 3, \\ u(x,0) &= 1 + \sin(2\pi x), & v(x,0) &= 3. \end{aligned}$$

Using second-order centred differences, the uniform discretization of the one-dimensional Brusselator equation using N unknown grid points is written

$$\begin{aligned} \frac{\mathrm{d}u_i}{\mathrm{d}t} &= \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1}) + A + u_i^2 v_i - (B+1)u_i, \\ \frac{\mathrm{d}v_i}{\mathrm{d}t} &= \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1}) + Bu_i - u_i^2 v_i, \end{aligned}$$

where $i = 1, 2, \ldots, N$, with initial conditions and boundary points written as

$$\begin{aligned} u_0(t) &= u_{N+1}(t) = 1, & v_0(t) &= v_{N+1}(t) = 3, \\ u_i(0) &= 1 + \sin\left(\frac{2\pi i}{N+1}\right), & v_i(0) &= 3. \end{aligned}$$

In this discretized form, the Brusselator model is now an IVP that can be solved using the numerical methods described in this thesis. The IVP has $2N$ unknowns because the original problem definition is comprised of two PDEs.

The two-dimensional form of the Brusselator is similar to the one-dimensional form, with

an additional forcing (reaction) term f that acts on u; otherwise, the diffusion quickly dominates any of the interesting solution characteristics. This thesis considers parameter selections for the two-dimensional Brusselator where $A = 1$, $B = 3.4$, $\alpha = \{1/50, 1/500\}$, $x \in [0, 1]$, and $y \in [0, 1]$. Boundary conditions are taken to be periodic; the initial conditions are given by:

$$u(x, y, 0) = 22y(1 - y)^{3/2}, \qquad v(x, y, 0) = 27x(1 - x)^{3/2}.$$

The uniformly discretized two-dimensional Brusselator model using second-order centred differences and $N \times N$ unknown grid points is written

$$\begin{aligned}
\frac{\mathrm{d}u_{i,j}}{\mathrm{d}t} &= \alpha N^2(u_{i,j-1} + u_{i-1,i} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}) + f_{i,j}(t) + A + u_{i,j}^2 v_{i,j} - (B + 1)u_{i,j}, \\
\frac{\mathrm{d}v_{i,j}}{\mathrm{d}t} &= \alpha N^2(v_{i,j-1} + v_{i-1,i} - 4v_{i,j} + v_{i+1,j} + v_{i,j+1}) + Bu_{i,j} - u_{i,j}^2 v_{i,j},
\end{aligned}$$

where $i = 1, 2, \ldots, N$, $j = 1, 2, \ldots, N$, and the forcing term $f_{i,j}(t)$ is written

$$f_{i,j}(t) = \begin{cases} 5 & \text{if } (i\Delta x - 0.3)^2 + (j\Delta y - 0.5)^2 \leq 0.1^2 \text{ and } t \geq 1.1, \\ 0 & \text{otherwise.} \end{cases}$$

Therefore, this ODE system is of size $2N^2$. The initial conditions on the discretized grid are given by

$$u_{i,j}(0) = 22y_j(1 - y_j)^{3/2}, \qquad v_{i,j}(0) = 27x_i(1 - x_i)^{3/2},$$

where $x_i = i\Delta x$ and $y_j = j\Delta y$.

3.4.3 Advection-Diffusion-Reaction Problems

Combustion Models

ADR equations can be used to simulate combustion. Combustion simulations are generally complex, involving many coupled PDEs that represent interactions among multiple react-

ing species. These PDEs typically describe conservation of energy, conservation of mass (commonly known as continuity equations), and conservation of momentum [14].

A simplified combustion model is presented in [4], where a single PDE is used to represent a combustion reaction. The flow in the domain is prescribed, rather than being influenced by reactions that occur in the simulation. This simplified model is written

$$\frac{\partial c}{\partial t} = \left[1 + U_0 \sin\left(\frac{\pi x}{L}\right)\right]\left(\frac{\gamma}{\beta}\frac{\partial^2 c}{\partial x^2} - \frac{\partial c}{\partial x}\right) + f(c), \tag{3.12}$$

where $x \in [10, 50]$, c is the mass fraction of the reacted products, $U_0 \in [0, 1]$ is a parameter that controls density and velocity changes of the fluid domain, L is the length scale of the domain, and the parameters γ and β are representative of the average diffusivity. Following the numerical experiments of [4], the model has been evaluated for $U_0 = \{0, 0.75, 0.99\}$. Additionally, it has been evaluated for the Fisher–Kolmogorov–Petrovskii–Piskunov (FKPP) reaction type, the ignition reaction type, and the general mth order Fisher reaction type. These three reaction models are written as

$$\begin{aligned} f_{\text{FKPP}}(c) &= \alpha_1 c(1-c), \\ f_{\text{Ignition}}(c) &= \alpha_2 (1-c)(c-c_s)^+, \\ f_{\text{Fisher}}(c) &= \alpha_3 c^m (1-c). \end{aligned}$$

where α_1, α_2, and α_3 represent rates of reaction, c_s is a reaction threshold, $(c - c_s)^+ = \max(c - c_s, 0)$, and m is the general Fisher non-linearity [4]. In this model, α_1 is 0.1, c_s is 0.6, and m is 10. Values α_2 and α_3 are calculated from the relation $\max(f(c)) = \frac{\alpha_1}{4}$ given by [4], such that all three reaction types are of approximately the same magnitude. The calculations are as follows:

$$\begin{aligned} f'_{\text{Ignition}}(c) = 0 \quad &\implies c = \frac{c_s + 1}{2} \quad &&\implies \alpha_2 = \frac{\alpha_1}{(1-c_s)^2} \\ f'_{\text{Fisher}}(c) = 0 \quad &\implies c = \frac{m}{m+1} \quad &&\implies \alpha_3 = \frac{\alpha_1}{4(\frac{m}{m+1})^m(1-\frac{m}{m+1})} \end{aligned}$$

Boundary conditions are periodic. The initial condition of the system is given as

$$c(x, t_0) = \exp\left[-\frac{(x - x_0)^2}{\sigma_0}\right].$$

This problem can be uniformly discretized using N unknown grid points, applying centred differences to the spatial derivatives. Therefore, the combustion model is written

$$\frac{\mathrm{d}c_i}{\mathrm{d}t} = \left[1 + U_0 \sin\left(\frac{i\pi\Delta x}{L}\right)\right]\left[\frac{\gamma}{\beta(\Delta x)^2}(c_{i-1} + c_i + c_{i+1}) + \frac{1}{2\Delta x}(c_{i+1} - c_{i-1})\right] + f(c_i),$$

where $i = 1, 2, \ldots, N$ and the boundary conditions are given as $c_0(t) = c_N(t)$ and $c_{N+1}(t) = c_1(t)$.

Tumour Angiogenesis

Differential equation models are prevalent in mathematical biology because they are useful in modelling the dynamics of biological processes. Such models can be used to verify that a biological process is understood and subsequently can make useful predictions.

Mathematical biology contains many models that involve ADR equations. One example is a model of tumour angiogenesis. This model tracks the growth of a tumour, simplified to one spatial dimension. The underlying ADR equations are comprised of non-linear advection, linear diffusion, and non-linear reaction terms. Two quantities are modelled in this domain: (i) ρ is a measurement of blood vessel density and (ii) c is the concentration of the tumour angiogenesis factor, which stimulates the growth of blood vessels. The model is scaled to the domain $x \in [0, 1]$ and is given as a system of two PDEs. They are written as

$$\frac{\partial \rho}{\partial t} = \epsilon\frac{\partial^2 \rho}{\partial x^2} - \frac{\partial}{\partial x}\left(\kappa\frac{\partial c}{\partial x}\rho\right) + \mu\rho(1 - \rho)(c - c^*)^+ - \beta\rho, \tag{3.13a}$$

$$\frac{\partial c}{\partial t} = \delta\frac{\partial^2 c}{\partial x^2} - \lambda c - \frac{\alpha\rho c}{\gamma + c}, \tag{3.13b}$$

where the parameter values of the PDE are

$$\epsilon = 10^{-3},\ \delta = 1,\ \alpha = 10,\ \beta = 4,\ \gamma = 1,\ \kappa = 0.75,\ \lambda = 1,\ \mu = 100,\ c^* = 0.2.$$

Initial and boundary conditions for ρ and c are given as

$$\rho(x,0) = \begin{cases} 0 & \text{if } 0 \leq x \leq 1, \\ 1 & \text{if } x = 1, \end{cases} \qquad c(x,0) = \cos\left(\frac{1}{2}\pi x\right),$$

$$\rho(0,t) = 0, \quad \rho(1,t) = 1, \qquad c(0,t) = 1, \quad c(1,t) = 0.$$

This PDE model can be transformed into an IVP by first applying the product rule to the second term of the RHS of (3.13a) and second, discretizing the spatial derivatives of (3.13a) and (3.13b) using centred finite differences. The spatial domain is comprised of N unknown grid points; therefore, $\Delta x = \frac{1}{N+1}$. This spatial discretization method is simple and is not expected to provide an accurate solution to the PDE; however, it is sufficient to compare ARK methods on the model.

The discretized form of (3.13) can now be written as

$$\begin{aligned} \frac{\mathrm{d}\rho_i}{\mathrm{d}t} &= \epsilon(N+1)^2(\rho_{i-1} - 2\rho_i + \rho_{i+1}) - \kappa(N+1)^2(c_{i-1} - 2c_i + c_{i+1}) \\ &\quad - \frac{\kappa(N+1)^2}{4}(c_{i+1} - c_{i-1})(\rho_{i+1} - \rho_{i-1}), \\ \frac{\mathrm{d}c_i}{\mathrm{d}t} &= \delta(N+1)^2(c_{i-1} - 2c_i + c_{i+1}) - \lambda c_i - \frac{\alpha\rho_i c_i}{\gamma + c_i}, \end{aligned}$$

where $i = 1, 2, \ldots, N$. The discretized initial and boundary conditions are written

$$\rho_{0,i} = 0 \qquad \rho_0(t) = 0, \qquad \rho_{N+1}(t) = 1,$$

$$c_{0,i} = \cos\left(\frac{\pi}{2(N+1)}\right), \qquad c_0(t) = 1, \qquad c_{N+1}(t) = 0.$$

Concrete-Rewetting

Concrete is a popular construction material because it is strong, inexpensive, and easy to use. It is created through a physicochemical transformation of cement, aggregate, and water. These ingredients form a wet mix that dries into a solid, porous mass. Concrete is cured several hours after the initial reaction has occurred; however, at this stage, the process is not

complete because a significant amount of cement remains unreacted. Thus, it is important to study concrete after the curing stage [3].

A hydration front passing through dry concrete, perhaps as a result of rainfall or water runoff, causes unreacted particles of cement to react [23]. This process is known as concrete-rewetting. As reactions occur, concrete becomes stronger; the reacting components clog the porous concrete causing the hydration front to move more slowly. Chapwanya et al. proposed a mathematical model of concrete-rewetting, i.e., a model describing the physical and chemical structure of dry concrete that has been exposed to a boundary of new moisture [10]. The model is a system of coupled PDEs, comprised of contributing factors from advection, diffusion, and reaction. The PDEs have one spatial dimension: a simplification of a column of concrete with an insulated or sink boundary on the top and a hydrated boundary on the bottom. The model applies to the movement of (i) isopropanol, a non-reactive solution that diffuses without altering the porosity of the concrete, and (ii) water, a solution that reacts with cement. This thesis is a study of additive splitting; therefore, case (i) is not considered because its reaction term is zero.

The equations for the rewetting model are written as:

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial x}\left[D(\theta,\epsilon)\frac{\partial \theta}{\partial x}\right] - \nu(\theta-\theta_r)^+\frac{m_w r_{\text{csh}}}{\rho_w m_{\text{csh}}}, \tag{3.14}$$

$$\frac{\partial(\theta C_\alpha)}{\partial t} = \frac{\partial}{\partial x}\left(\theta D_\alpha \frac{\partial C_\alpha}{\partial x}\right) - \frac{\partial(uC_\alpha)}{\partial x} - (\theta-\theta_r)^+ r_\alpha, \tag{3.15}$$

$$\frac{\partial(\theta C_\beta)}{\partial t} = \frac{\partial}{\partial x}\left(\theta D_\beta \frac{\partial C_\beta}{\partial x}\right) - \frac{\partial(uC_\beta)}{\partial x} - (\theta-\theta_r)^+ r_\beta, \tag{3.16}$$

$$\frac{\partial(\theta C_q)}{\partial t} = \frac{\partial}{\partial x}\left(\theta D_q \frac{\partial C_q}{\partial x}\right) - \frac{\partial(uC_q)}{\partial x} + (\theta-\theta_r)^+(r_{\text{csh}} - k_{\text{prec}}C_q + k_{\text{diss}}C_g), \tag{3.17}$$

$$\frac{\partial(\theta C_g)}{\partial t} = (\theta-\theta_r)^+(k_{\text{prec}}C_q - k_{\text{diss}}C_g), \tag{3.18}$$

where $(\theta-\theta_r)^+ = \max(0,\theta-\theta_r)$; θ is the volumetric water content; C_α, C_β, C_q, and C_g are the respective constituent concentrations of C_3S in concrete, C_2S in concrete, calcium-silicate hydrate (C-S-H) in liquid, and solid C-S-H gel; r_α, r_β, and r_{csh} are functions that govern the rates of reaction; D is the effective diffusivity; D_α, D_β, and D_q are the respective diffusivities of C_3S in concrete, C_2S in concrete, and C-S-H in liquid. The remaining variables

are substance-specific constants, which are defined in [10].

The domain is one-dimensional, of length L. It is discretized into N cells of equal width $\Delta x = \frac{L}{N}$ using the centred finite volume approach, described in Section 3.2.2. The N cells are indexed by $i = 1, 2, \ldots, N$ and are denoted using subscripts. The discretized form is written as follows:

$$
\begin{aligned}
\frac{d\theta_i}{dt} &= \frac{1}{(\Delta x)^2}\left[D_{i+\frac{1}{2}}\left(\theta_{i+1}-\theta_i\right) - D_{i-\frac{1}{2}}\left(\theta_i-\theta_{i-1}\right)\right] - \nu(\theta_i-\theta_r)^+\frac{m_w r_{\text{csh}}}{\rho_w m_{\text{csh}}},\\
\frac{d(\theta_i C_{\alpha,i})}{dt} &= \frac{D_\alpha}{(\Delta x)^2}\left[\theta_{i+\frac{1}{2}}\left(C_{\alpha,i+1}-C_{\alpha,i}\right) - \theta_{i-\frac{1}{2}}\left(C_{\alpha,i}-C_{\alpha,i-1}\right)\right]\\
&\quad - \frac{u_{i+\frac{1}{2}}C_{\alpha,i+\frac{1}{2}} - u_{i-\frac{1}{2}}C_{\alpha,i-\frac{1}{2}}}{\Delta x} - (\theta_i-\theta_r)^+ r_\alpha,\\
\frac{d(\theta_i C_{\beta,i})}{dt} &= \frac{D_\beta}{(\Delta x)^2}\left[\theta_{i+\frac{1}{2}}\left(C_{\beta,i+1}-C_{\beta,i}\right) - \theta_{i-\frac{1}{2}}\left(C_{\beta,i}-C_{\beta,i-1}\right)\right]\\
&\quad - \frac{u_{i+\frac{1}{2}}C_{\beta,i+\frac{1}{2}} - u_{i-\frac{1}{2}}C_{\beta,i-\frac{1}{2}}}{\Delta x} - (\theta_i-\theta_r)^+ r_\beta,\\
\frac{d(\theta_i C_{q,i})}{dt} &= \frac{D_q}{(\Delta x)^2}\left[\theta_{i+\frac{1}{2}}\left(C_{q,i+1}-C_{q,i}\right) - \theta_{i-\frac{1}{2}}\left(C_{q,i}-C_{q,i-1}\right)\right]\\
&\quad - \frac{u_{i+\frac{1}{2}}C_{q,i+\frac{1}{2}} - u_{i-\frac{1}{2}}C_{q,i-\frac{1}{2}}}{\Delta x} + (\theta_i-\theta_r)^+(r_{\text{csh}} - k_{\text{prec}}C_{q,i} + k_{\text{diss}}C_{g,i}),\\
\frac{d(\theta_i C_{g,i})}{dt} &= (\theta_i-\theta_r)^+(k_{\text{prec}}C_{q,i} - k_{\text{diss}}C_{g,i}),
\end{aligned}
$$

where $u_{i+\frac{1}{2}} = -D_{i+\frac{1}{2}}(\theta_{i+1}-\theta_i)/\Delta x$ and $D_i = D(\theta_i, \epsilon_i)$.

Points at half-indices are calculated as the average between cells, e.g., $\theta_{i+\frac{1}{2}} = (\theta_i+\theta_{i+1})/2$. The concrete-rewetting problem is thus converted to the form (1.1) and solved over the time interval $[0, 28]$.

Chapter 4

Results of Numerical Experiments

Numerical experiments are performed on the ADR models described in Section 3.4 using the test set of methods described in Section 3.1.1. Each experiment is run ten times; the minimum time measurement of the ten runs is used for statistical analysis because a minimum time is indicative of how well a given method performs under ideal conditions. Running experiments multiple times mitigates experimental error introduced by processing artifacts such as caching effects or task-switching to background processes. Although it is assumed that the IVP solver is the only process that significantly contributes to CPU load, there are always background processes that occasionally require minimal CPU and memory resources. In general, there are no significant differences observed in the CPU times, i.e., less than 1% difference; however, the occasional outlying measurement can be up to 20% longer than the minimum. The presence of these outlying measurements is precisely the reason for using minimum rather than average runtimes.

4.1 Cluster Specifications

All experiments are conducted on a 4-node cluster. Each node has the following specifications:

- 2x Xeon E5-2620 (6C 2.0GHz 15MB 1333MHz 95W),
- 4x 8GB RAM (16GB per processor),
- 500GB local disc space,
- Gigabit Ethernet, and
- Ubuntu 12.04.2 LTS.

The cluster does not use a batch scheduler because a scheduler might negatively impact performance. The processors on the machine are hyper-threaded and therefore allow the execution of two simultaneous threads per processor. Each machine is capable of executing 24 simultaneous threads (2 processors per machine $\times$ 6 cores per processor $\times$ 2 threads per core). It is theoretically possible to solve 96 IVPs simultaneously on the entire cluster (4 machines $\times$ 24 threads per machine). However, one must be mindful of hardware when running numerical experiments that involve timing. Hyper-threading shares cache at all levels but presents the threads to the operating system as independent, virtual processors [33]. If all threads on a machine are at full capacity, runtimes of numerical experiments might be significantly affected because IVP solvers would be competing for cache space. Therefore, the number of simultaneous IVPs is restricted to 12 per machine (48 total).

4.2 Findings regarding Interprocess Communication

The message passing interface (MPI) consists of a protocol and external libraries designed to ease the development of parallel applications for scientific computing [41]. In this thesis, `pythODE++` launches processes on multiple machines using the MPI implementation Open-MPI [17]. MPI launches multiple processes on multiple nodes. In principle, `pythODE++` is scalable to arbitrarily large clusters such that many IVPs can be solved simultaneously. However, load distribution of IVP runs is not trivial. Once processes have been established on all machines in the cluster, it is inefficient to distribute experimental runs evenly across all processes. Unlike many MPI applications where each processor is solving its own section of a common problem, each process in the `pythODE++` PSE could be solving a completely different problem that requires a significantly different amount of time to complete. Therefore, one process is designated to be the master process, which is responsible for load balancing. It dispatches tasks to all other worker processes.

As introduced in Section 3.1.6, each worker process is comprised of two components: a communication thread and a worker thread. The communication thread is responsible for responding to requests from the master node, for example, queries for runtime statistics or queries regarding whether the process is free to accept another IVP to solve. This thread uses

few computational resources and is generally in a sleep state while it is awaiting instruction from the master node. The worker thread is responsible for running the IVP solver; therefore, it uses essentially 100% of its CPU while it is solving a problem.

Although processes for `pythODE++` are initialized using MPI, once solvers have begun to solve problems, interprocess communication is done solely using sockets. The reason for this seemingly peculiar choice is as follows. When using a cluster, MPI is an extremely easy way with which to initialize processes across multiple nodes. It is also incredibly easy to use MPI functions for interprocess communication because MPI allows for the reliable transmission of messages of an arbitrary size. In this work, an initial implementation of the `pythODE++` PSE used such an approach. However, MPI interprocess communication inflicts severe performance penalties on the PSE. MPI polls extremely aggressively while it is waiting for incoming messages; in extreme cases, it uses effectively 100% of CPU resources for a single thread. Generally, aggressive polling is desired because applications aim to minimize the time delay due to interprocess communication. Valuable time is wasted if a message arrives and the receiving process does not immediately begin processing the message. However, such usage is neither ideal for the efficient use of resources when running independent problems, nor for maintaining accurate timing operations during simultaneous timings performed by the PSE.

In this work, interprocess communication is implemented using TCP sockets [9]. TCP sockets ensure reliable communication; i.e., they ensure that all data sent to the receiving process are actually received and that the data are received in the order in which they are sent. At the application level, TCP sockets transmit data using a continuous stream of data (to allow reliable communication or load balancing, for instance), rather than abstracting transmission of arbitrarily sized messages corresponding to the model of MPI. Stream communication is more difficult in comparison to the MPI implementation because message alignment might not coincide with packet alignment. Therefore, implementation of an "end of message" marker is necessary in all messages transmitted between master and worker nodes.

4.3 Sparse Jacobian Computation

Initially, Jacobians were computed using forward finite differences. Such an approach requires $\mathcal{O}(m)$ function evaluations each time a Jacobian matrix is calculated. Forward finite differences are sufficient so long as the problem size is small. However, once the number of unknowns grows sufficiently large, it is no longer feasible to compute finite difference Jacobian matrices. The time required for the necessary function evaluations quickly becomes the majority of the time per step of the numerical method and, more importantly, machines do not have enough memory to store a dense Jacobian. For example, even a two-dimensional problem of 500×500 unknowns requires 200^4 values to form the finite-difference Jacobian. When using double-precision values, this equals approximately 500GB, which is impractical with current resources.

When calculating Jacobians with sparsity, such problems disappear, and the majority of time per step is no longer spent calculating Jacobian matrices. Therefore, benchmarks of Jacobian splitting versus physics-based splitting are now meaningful. However, the sparse library `ADOL-C` still has significant overhead that can be avoided by coding sparse analytic Jacobian matrices for each IVP. The downside of manually coding Jacobian matrices is that it is tedious and error-prone. Therefore, `ADOL-C` was used to verify the implementation of analytic Jacobians, so the `pythODE++` PSE could take advantage of the speed of manually implemented sparse Jacobian with the assurance that the implementation was correct.

4.4 Verification, Validation, and Solution Plots

This section discusses the verification and validation of numerical methods and differential equations. As part of the verification, solution plots for each of the numerical experiments are presented, showing that the models behave as expected and produce physically plausible results.

4.4.1 Methods

The numerical methods must be verified to ensure that they are solving IVPs correctly; otherwise, the runtime required for a method to step through an IVP is meaningless. All methods in this work are verified against their implementations in `pythODE` and are validated on a series of test problems consisting of thirty stiff and thirty non-stiff IVPs having known exact solutions; see [15] for a complete list of the test problems. These test problems range from linear, autonomous problems, to those that are non-linear, time-dependent, and contain discontinuities; therefore, they are well suited for ensuring the numerical methods perform correctly.

Numerical methods designed for 2-additively split problems have applied Jacobian splitting to the stiff and non-stiff IVPs because those IVPs do not have natural physics-based splittings. Two further verifications have been implemented: the *verification of order* and the *convergence of reference solutions*. These verifications are now described.

Numerical methods for IVPs have a certain order of accuracy with respect to a Taylor series; see Section 2.2. It must be verified that each method converges to the true solution at the correct rate. This verification is extremely important because small mistakes in a method's coefficients might still provide a convergent solution to an IVP. A numerical method of order p exhibits an approximate decrease in error by a factor of 2^p when its stepsize is halved. To perform the verification of order, one can solve an IVP with a given stepsize and continue halving it to show that error decreases at the appropriate rate. Such verification demonstrates that error incurred by the method is due to order, i.e., the matching of terms in the Taylor series, rather than an implementation error or round-off error incurred by floating-point operations in the method. The expected order of convergence has been verified for all methods in this thesis.

Numerical reference solutions must be generated for IVPs that do not have analytic solutions with which to compare the accuracy of methods. However, a single solution of a problem using a strict tolerance does not give rigorous evidence that the solution is accurate. Verification of convergence for reference solutions demonstrates that reference solution methods converge to a common solution as tolerances are decreased. Therefore, IVPs must

be solved many times to generate a reference solution. In this thesis, the RODAS method and the RADAU5 method have been used to generate reference solutions for all problems that do not have analytical solutions. The process for verification involves solving an IVP with a given tolerance, then gradually decreasing the tolerance until the numerical solution has sufficiently converged. If numerical solutions do not converge as tolerances are decreased, there might be an error in the numerical method or the IVP might not be well-posed.

4.4.2 Problem Suite

Implementations for each of the ADR problems have been verified by comparing graphs of reference solutions to an expected result or to external implementations. Each ADR problem is shown to produce reasonable results.

The two advection-diffusion equations are both problems that have been designed specifically for this thesis. Figure 4.1 shows a solution plot for the one-dimensional advection problem defined by (3.7). Note from the problem definition that the initial condition is chosen to be a harmonic wave. Notice how the advection process moves the wave and the diffusion process makes it more smooth. Figure 4.2 shows solution plots for the two-dimensional heat transfer problem defined by (3.9). Heat travels down the pipe in the shape of the velocity profile as expected.

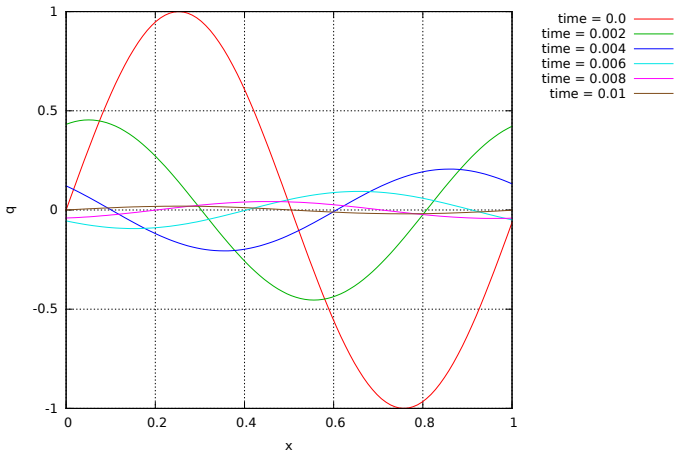

Figure 4.1: Solution plots for the linear advection-diffusion equation defined by (3.7), spatially discretized using 500 points.

Figure 4.3 shows a solution plot for the `CUSP` problem defined by (3.10). This plot matches

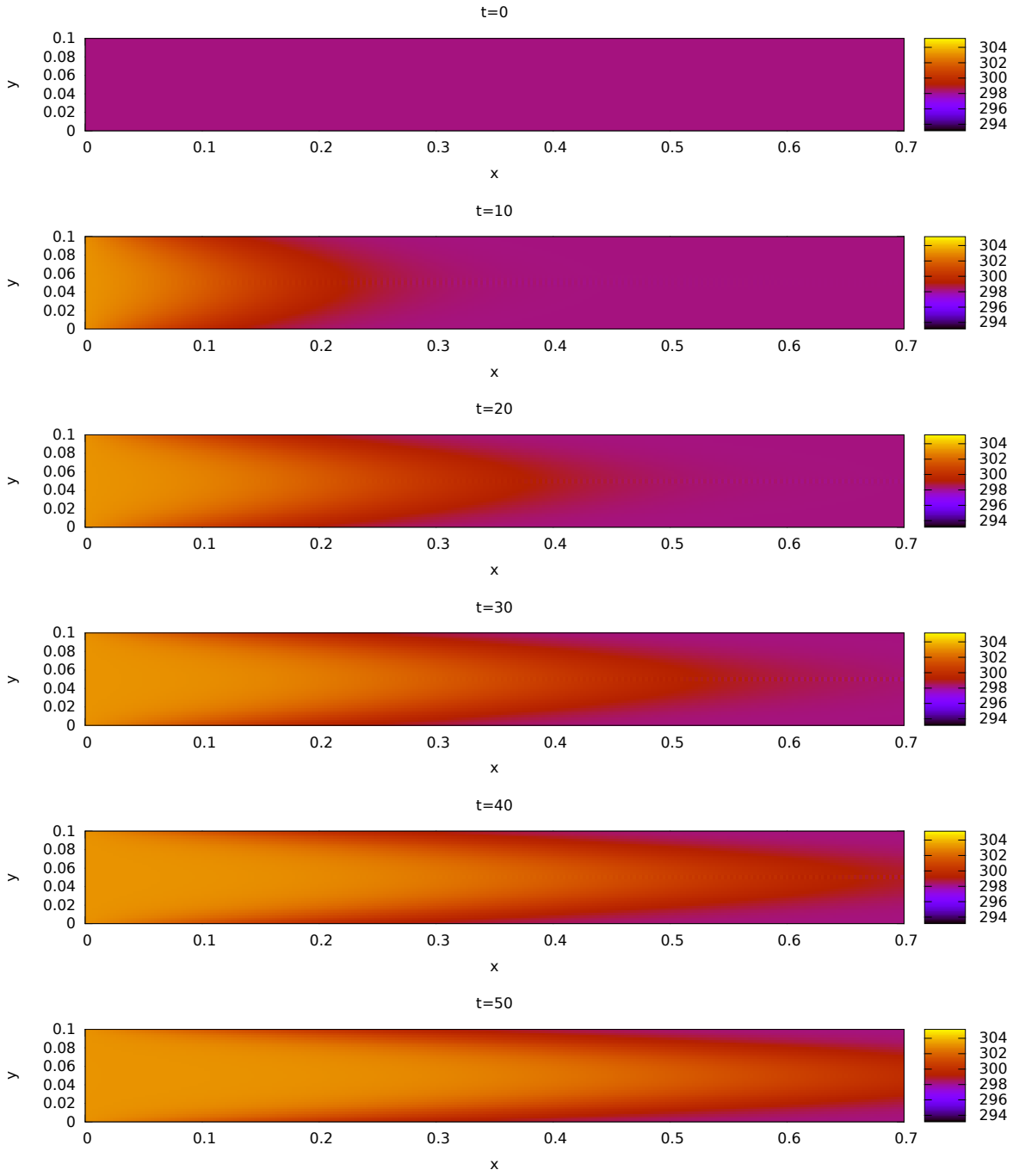

Figure 4.2: Solution plots for the heat transfer equation defined by (3.9), spatially discretized using 280x40 points.

results that are shown in [21]. Figures 4.4 and 4.5 show solution plots of the two reactants present in the one and two-dimensional Brusselator problems, both of which are defined by (3.11). The one-dimensional version matches results that are shown in [21]; the two-dimensional version is verified against the `BRUSS-2D` Fortran code from [21].

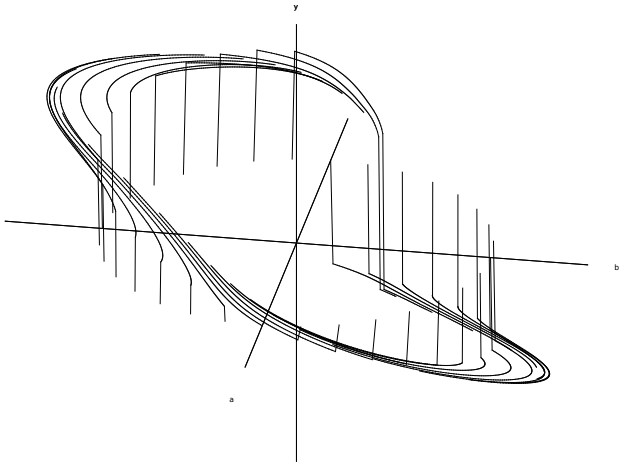

Figure 4.3: Solution plot for the `CUSP` problem defined by (3.10), using 32 discretized grid points.

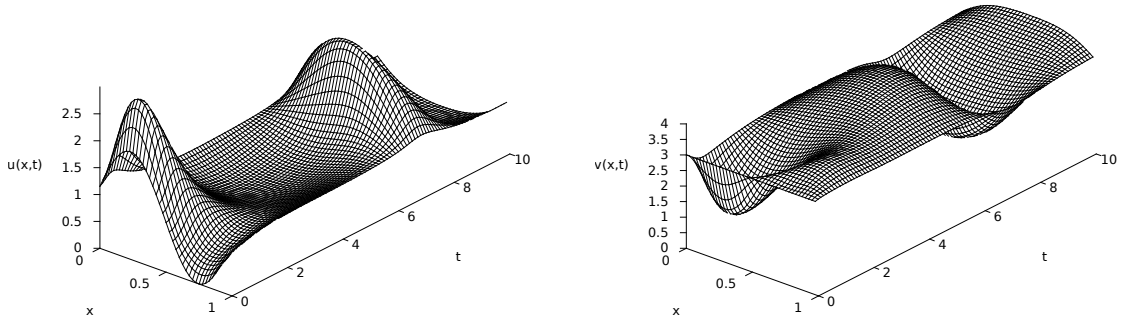

Figure 4.4: Solution plots for the one-dimensional Brusselator equation defined by (3.11), using 1000 discretized points.

Figure 4.6 shows solution plots for each permutation of parameters for the combustion problem defined by (3.12); these graphs match the results of the numerical experiment in [4]. Figure 4.7 shows solution plots for the tumour angiogenesis model defined by (3.13) for two different diffusivity values; these plots match those given in [25]. Figure 4.8 shows solution plots of four sets of parameters for the concrete-rewetting problem defined by (3.14)–

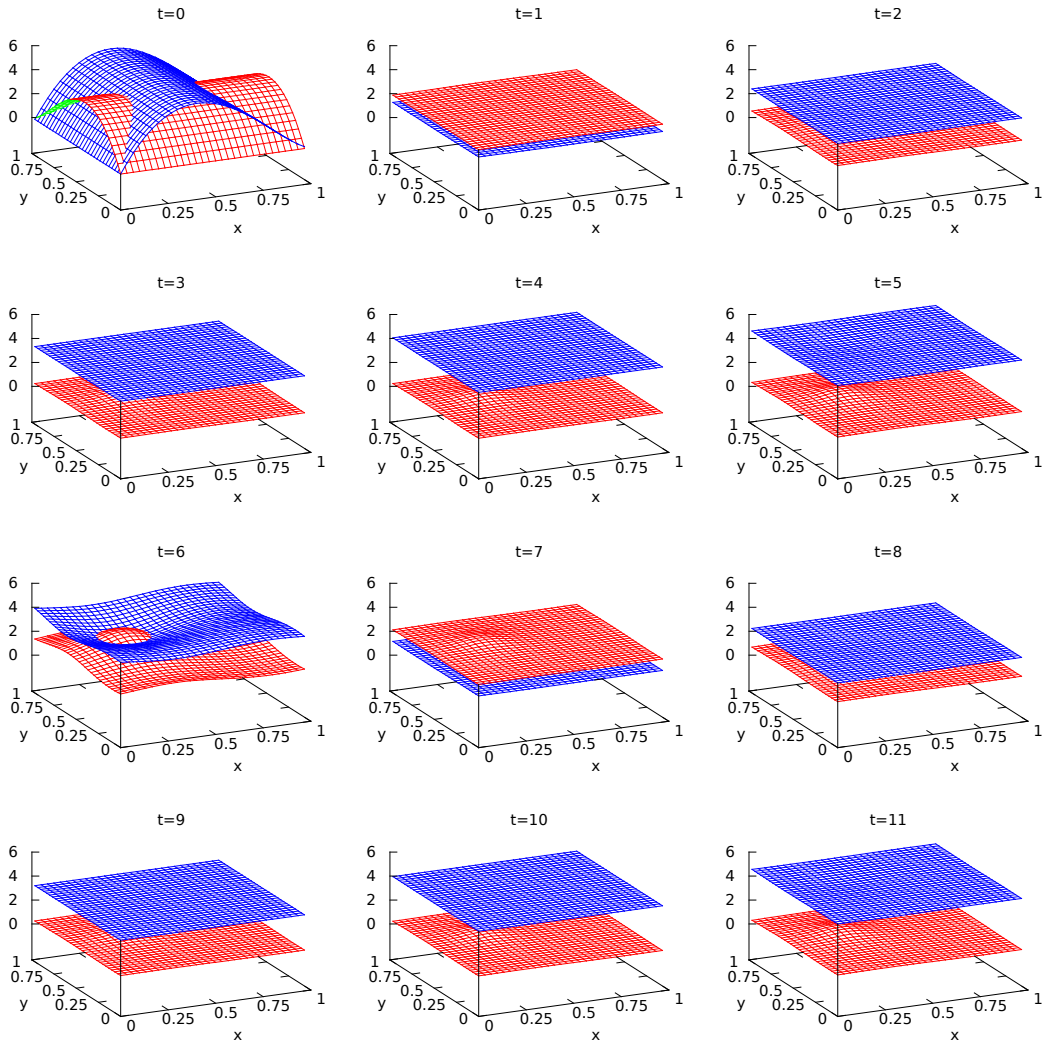

Figure 4.5: Solution plot for the two-dimensional Brusselator equation defined by (3.11), using 60x60 discretized points. The function $u(x, y, t)$ is given by the red meshes and the function $v(x, y, t)$ is given by blue meshes.

(3.18). These sets involve using either water or isopropanol, and either a sink boundary or a free boundary. Although this study is only concerned with using water (not isopropanol) when preforming experiments related to splitting, all four sets of parameters are used during verification to ensure that the model behaves appropriately. The solution plots of the concrete-rewetting problem generated for this work agree with plots from [10]. Notice how isopropanol travels much more quickly through the concrete pores in comparison to water because isopropanol does not react with cement.

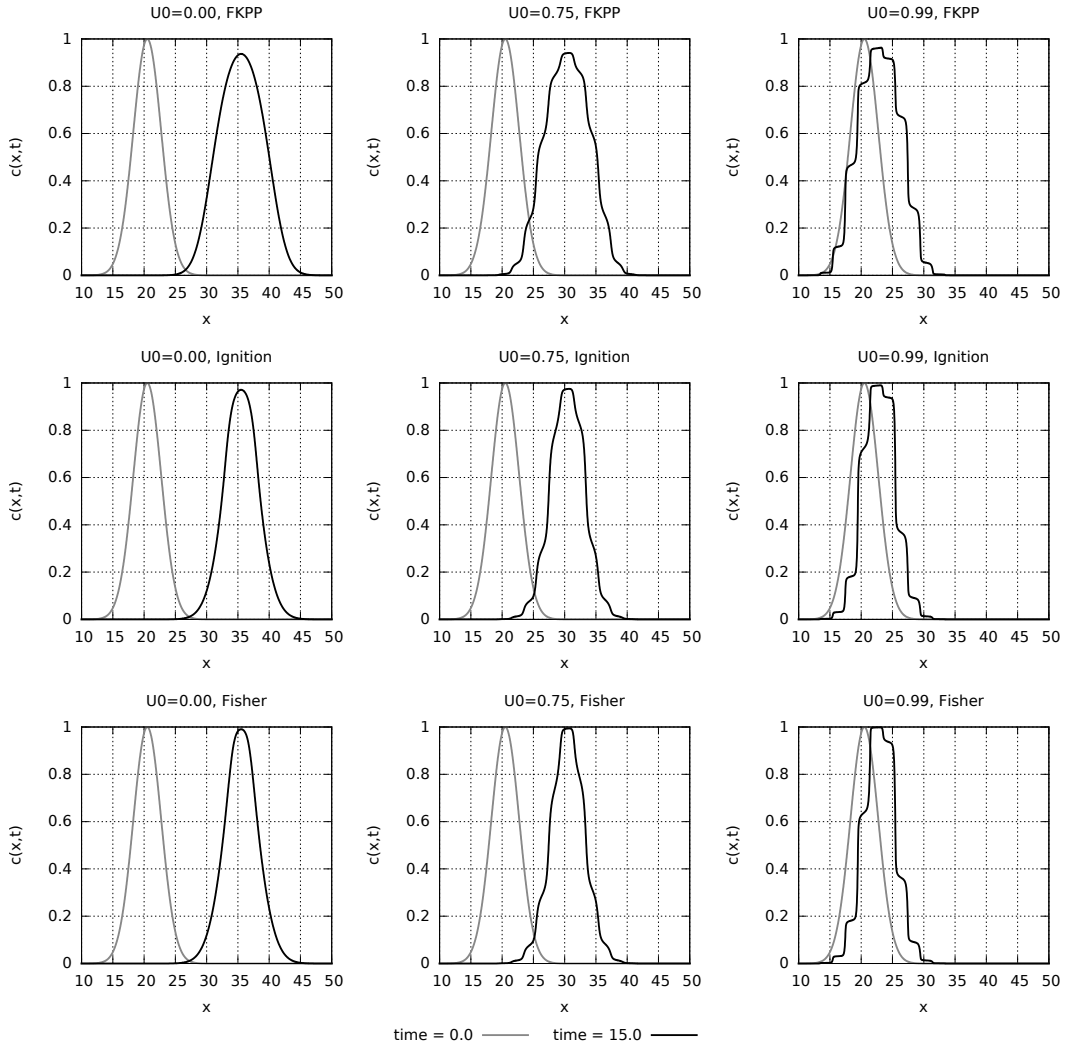

Figure 4.6: Solution plots for the combustion equations defined by (3.12), using 1600 discretized points.

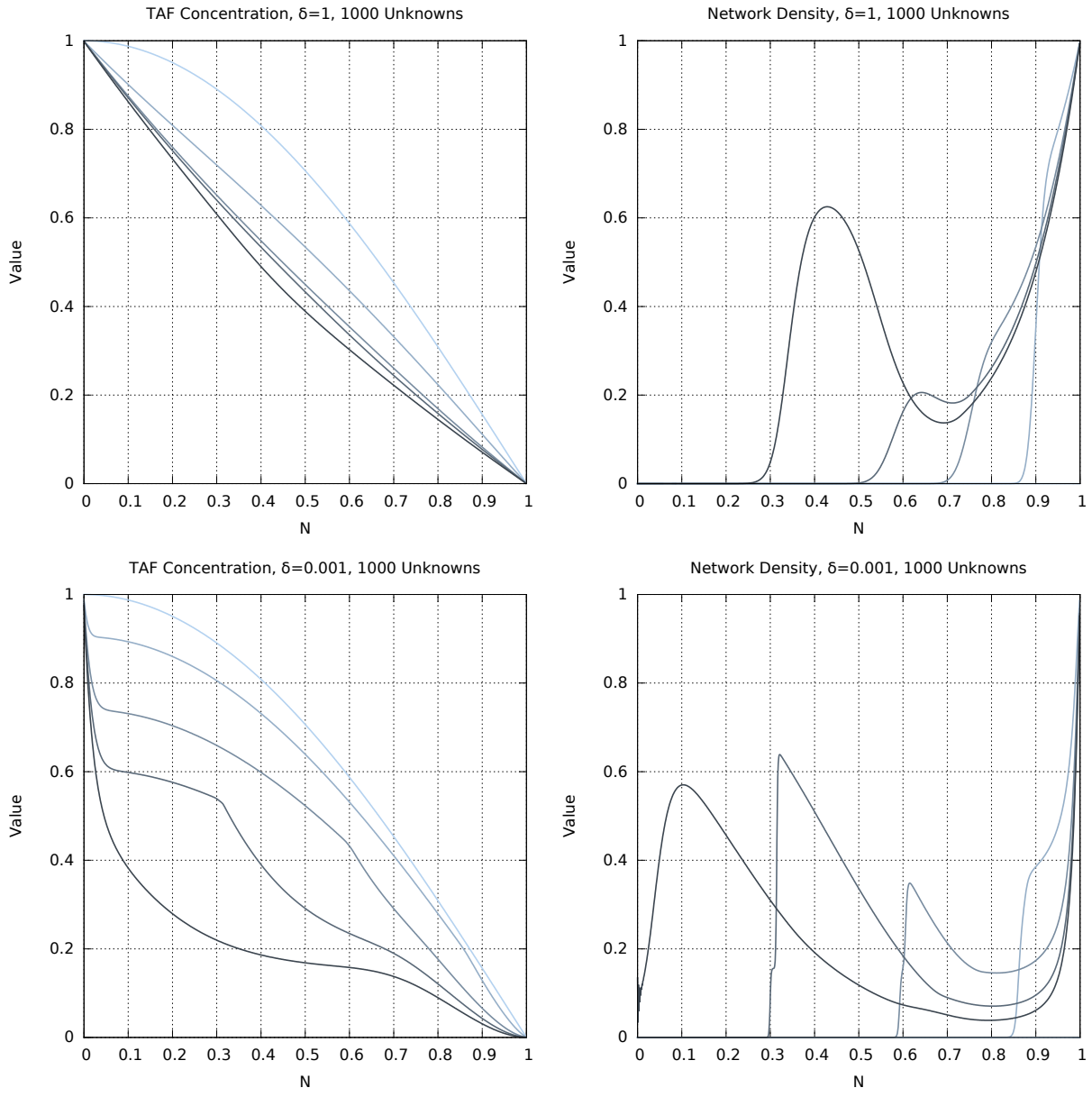

Figure 4.7: Solution plots for the tumour angiogenesis model defined by (3.13). The domain has been discretized using 1000 points. Plots are shown for times 0, 0.1, 0.3, 0.5, 0.7, moving from light to dark.

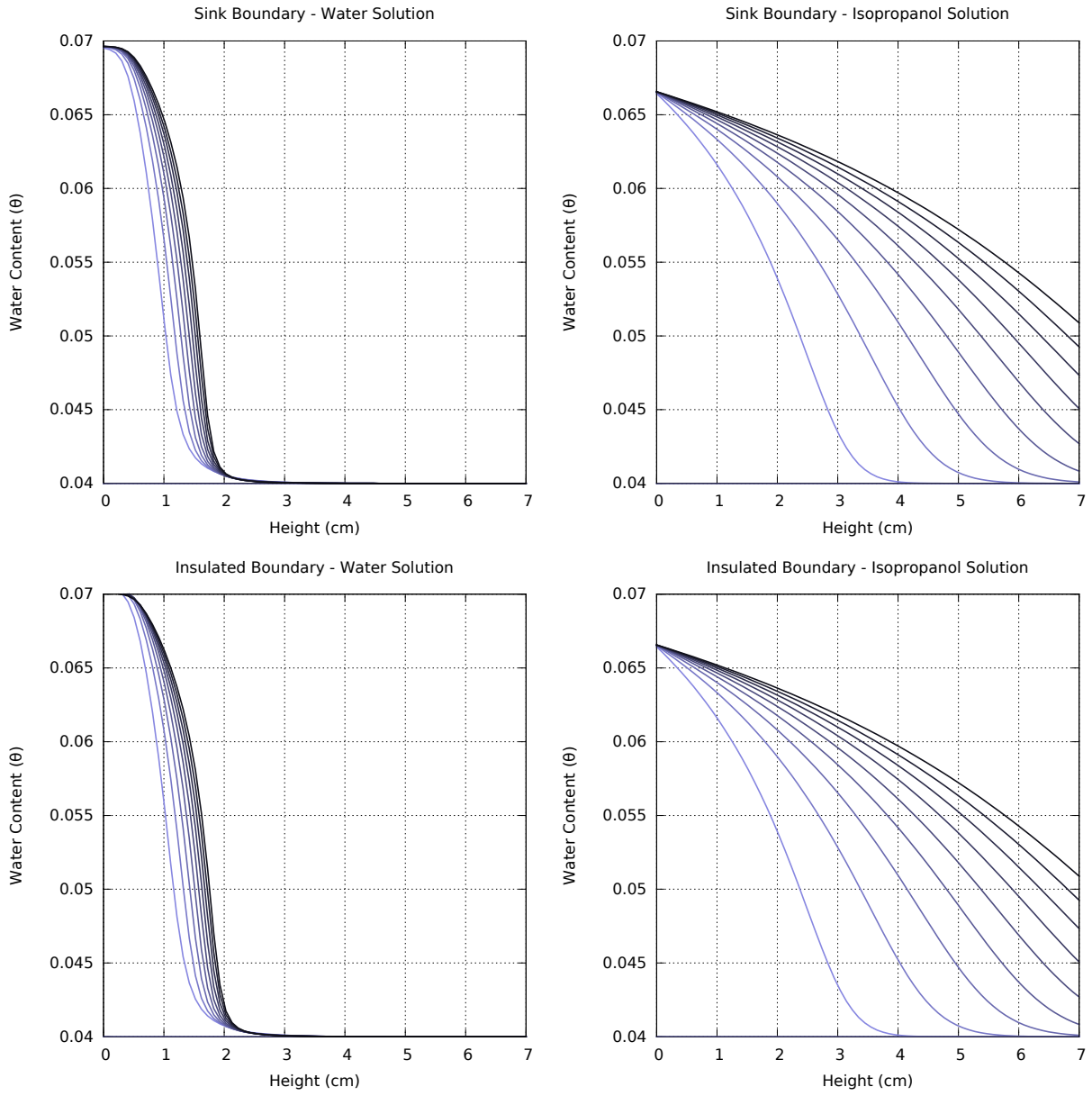

Figure 4.8: Solution plots for the concrete-rewetting problem. The hydration front is shown for 10 equally spaced times in [0,28], moving from light to dark.

4.5 Comparisons of Numerical Methods

This section describes results of the comparison between physics-based splitting and Jacobian splitting used by IMEX methods. *Work-precision* plots are used to compare the performance of numerical methods, where the accuracy of the numerical methods are given by the x-axis and the total CPU time is given by the y-axis. These plots consist of data points, that show the accuracy and CPU time of each numerical experiment. The legends in the graph provide the method name, the splitting type (where applicable), and the associated range of tolerances that were evaluated. In all experiments, absolute and relative tolerances are taken to be equal. Jacobian splitting is shown to outperform physics-based splitting on a test suite of IVPs. Each of the IVPs is various evaluated on sets of parameters.

In the following sections, work-precision plots are shown comparing the ARK3, ARK4, and ARK5 using both physics-based splitting and Jacobian splitting. The PRKC, IRKC, and DIRK-CF methods are not competitive with any of the IMEX methods and are not shown in the following graphs so as not to complicate the results. However, on some of the IVPs, the (non-additive) RKC1 and RKC2 methods are shown to be competitive with the IMEX methods and, for the heat transfer, `CUSP`, two-dimensional Brusselator, and concrete-rewetting problems, RKC2 in fact outperforms the IMEX methods. The RKC methods are included in graphs for which results of the RKC experiments are of interest. In these cases, the IMEX methods generally take fewer steps than the RKC methods; however, the RKC methods have faster runtimes because their steps are relatively inexpensive. Were it possible to compute steps of the IMEX methods more quickly, the IMEX methods might experience shorter run times than those of RKC. Appendix F shows the eigenvalue plots for each of the IVPs. As expected, the RKC methods perform well on those IVPs with eigenvalues near the negative real axis. Associated plots for each of the numerical experiments of the number of steps versus accuracy are given in Appendix E.

IVPs solved using Jacobian splitting have theoretically more expensive function evaluations than problems that have been split based on physical characteristics. Although the numerical method remains the same when comparing physics-based splitting to Jacobian splitting, calls to the RHS are more expensive for Jacobian splitting because of the addi-

tional matrix multiplications (of the Jacobian); i.e., each evaluation of the RHS requires two additional computations of $\mathbf{J_f y}$. However, results show the increase in time for evaluation of the RHS is more than offset because methods using Jacobian splitting can take significantly larger step sizes. For problems where $\mathbf{f}_I$ is non-linear, Jacobian splitting theoretically has a further advantage over a step using physics-based splitting because Jacobian splitting requires only a single Newton iteration, whereas physics-based splitting might require several.

The range of specified tolerances is given for each of the work-precision plots. The plot lines begin with the most lax tolerances and progress to more strict tolerances, up to the specified tolerance limit. In general, it is desirable for data points to fall in the bottom-left of the plots, signifying that the numerical method has solved the IVP with relatively high accuracy, while requiring a relatively low amount of time or relatively low number of steps.

4.5.1 Advection-Diffusion Problems

One-Dimensional Advection-Diffusion Model

The IMEX methods are evaluated on the one-dimensional, linear advection-diffusion equation, defined by (3.7). To apply physics-based splitting, the diffusion term is assumed to be relatively stiff and is treated implicitly; the advection term is assumed to be relatively non-stiff and is treated explicitly. Therefore, using the notation of (1.2), physics-based splitting treats the advection-diffusion IVP from (3.8) as

$$\mathbf{f}_I = \mathbf{D}\mathbf{y}, \qquad \mathbf{f}_E = -\mathbf{A}\mathbf{y},$$

where $\mathbf{D}$ is a constant matrix representing the discretization of the diffusion term and $\mathbf{A}$ is a matrix representing the discretization of the advection term. Note that $\mathbf{D}$ and $\mathbf{A}$ are tridiagonal when the problem is discretized using second-order central differences because the problem is one-dimensional. This advection-diffusion IVP is simple because it is linear. It is used only as a proof of concept for Jacobian splitting. Jacobian splitting applied to the

IVP yields

$$\mathbf{f}_I = (\mathbf{D} - \mathbf{A})\mathbf{y}, \qquad \mathbf{f}_E = \mathbf{0}.$$

On this IVP, the application of Jacobian splitting causes IMEX methods to reduce to DIRK methods because the term that would be treated explicitly is zero. This simple case, nonetheless, demonstrates how Jacobian splitting captures the stiffness of the IVP. In this example, physics-based splitting does not treat any part of the advection term implicitly and thus performs poorly.

The models are evaluated using second-, fourth-, and sixth-order finite difference methods. Only minute differences are observed in the results; the results for a second-order discretization are presented. The work-precision plots are shown in Figure 4.9, demonstrating that Jacobian splitting also has a shorter runtime. Notice that as the advection term becomes less significant in comparison to diffusion, physics-based splitting begins to approach the runtime performance of Jacobian splitting. In all instances, the methods using Jacobian splitting are superior in comparison to those same methods using physics-based splitting. The RKC methods are significantly outperformed.

Two-Dimensional Heat Transfer Model

The heat transfer model defined by (3.9) uses physics-based splitting in the same way as for the one-dimensional advection-diffusion problem. The diffusion term is treated implicitly, and the advection term is treated explicitly. The primary difference between this model and the one-dimensional advection-diffusion problem is that the Jacobian of the RHS of the heat transfer problem contains two additional bands because the PDE is two-dimensional. Numerical experiments were conducted using second- and fourth-order central differences to approximate the second derivatives in the diffusion term, and first-, second-, and third-order backward differences for the advection term. Near the boundaries, asymmetric differences were used when the higher-order finite difference stencils would otherwise extend past the boundary. There was no observed difference between the relative performance of the methods when using higher-order spatial discretization; therefore, results are shown using first-order

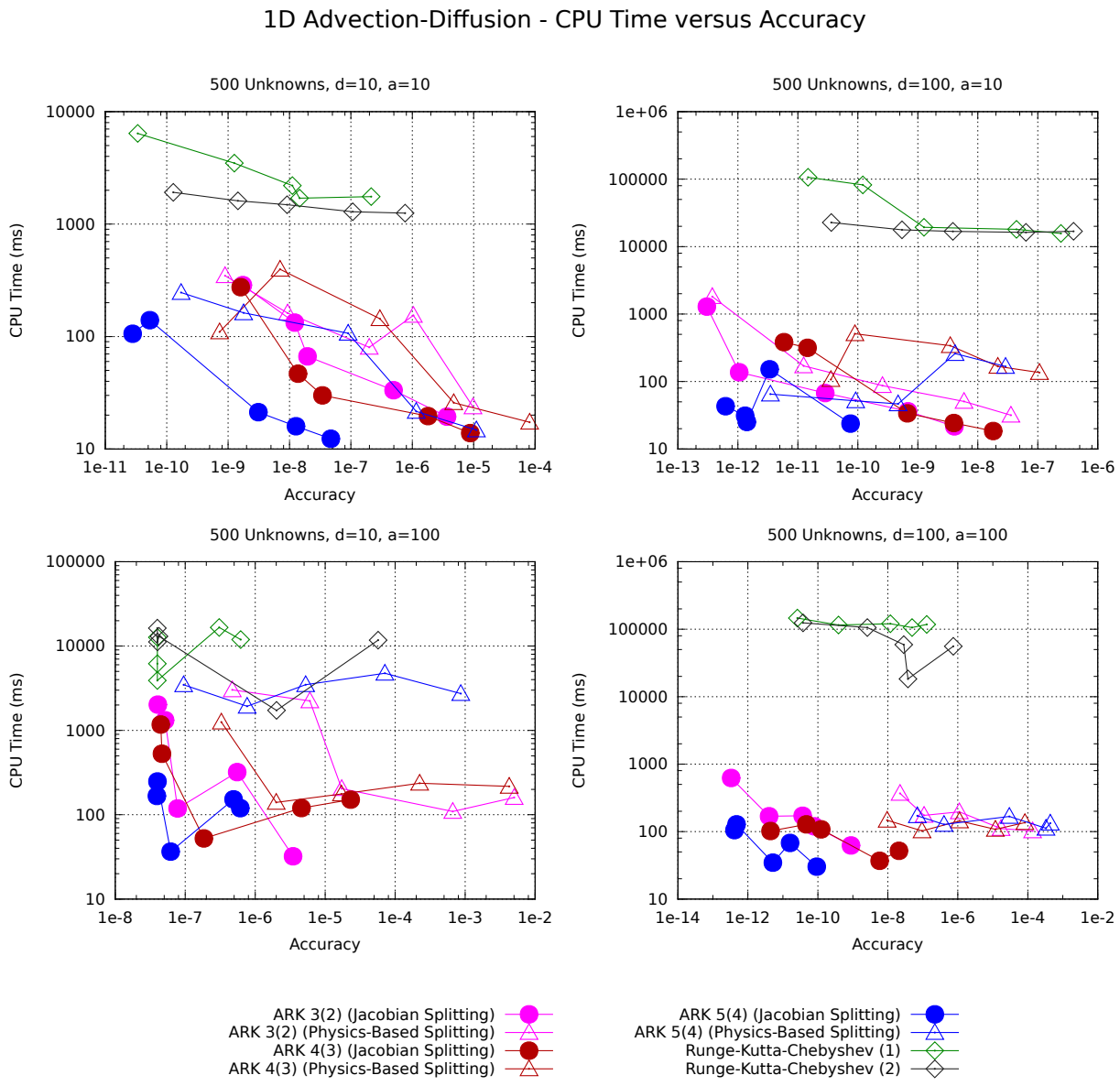

Figure 4.9: CPU time versus accuracy of IMEX and RKC methods applied to a series of one-dimensional advection-diffusion problem. Tolerances range from 10^{-4} to 10^{-8}.

backward differences and second-order central differences for the first and second derivatives, respectively.

The work-precision plots for each of the four tested cases shows Jacobian splitting to have superior performance compared to physics-based splitting. CPU times are shown in Figure 4.10. Unlike the one-dimensional case, the RKC1 and RKC2 methods are competitive with the IMEX methods. RKC2 solves the problems more quickly but it is not able to attain the same level of accuracy as the IMEX methods; this is not a surprising result because it is a lower-order method compared to the tested IMEX methods.

4.5.2 Reaction-Diffusion Problems

The `CUSP` Problem

The `CUSP` model defined by (3.10) is a more complicated example than the previous advection-diffusion problems. Much like the advection-diffusion IVP, the diffusion term is linear; however, the reaction term is now non-linear. In reference to the variables of (1.2), a physics-based splitting for the `CUSP` problem is defined by treating diffusion as $\mathbf{f}_I$ because it is relatively stiff. The remaining reaction terms are placed into $\mathbf{f}_E$ because they are relatively non-stiff.

Work-precision plots are shown in Figure 4.11. The IMEX methods using Jacobian splitting perform better than virtually all IMEX methods using physics-based splitting by up to an order of magnitude. RKC2 significantly outperforms the IMEX method on three of the four tests by at least an order of magnitude; however, the solution produced by RKC2 is not able to achieve the same levels of accuracy.

The Brusselator

The Brusselator problems in one and two dimensions are solved using 2-additive methods by applying physics-based splitting similarly to the `CUSP` problem. Diffusion is treated implicitly and reaction is treated explicitly. Both one- and two-dimensional PDEs were discretized using second-, fourth-, and sixth-order finite difference methods. Results are shown for the second-order discretization scheme because there was no noticeable difference between these three schemes.

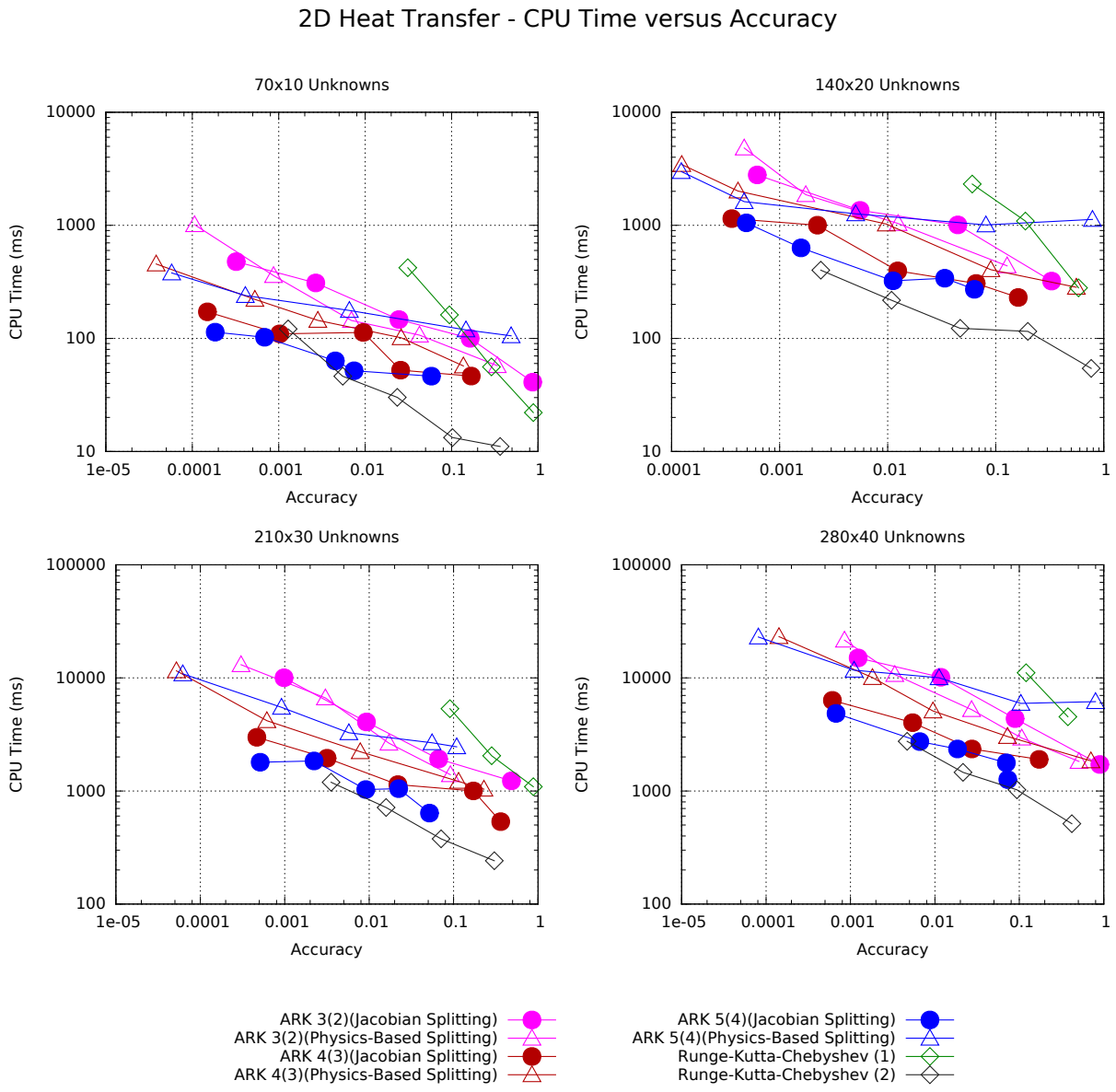

Figure 4.10: CPU time versus accuracy of IMEX and RKC methods applied to the heat transfer problem. Tolerances range from 10^{-4} to 10^{-8}.

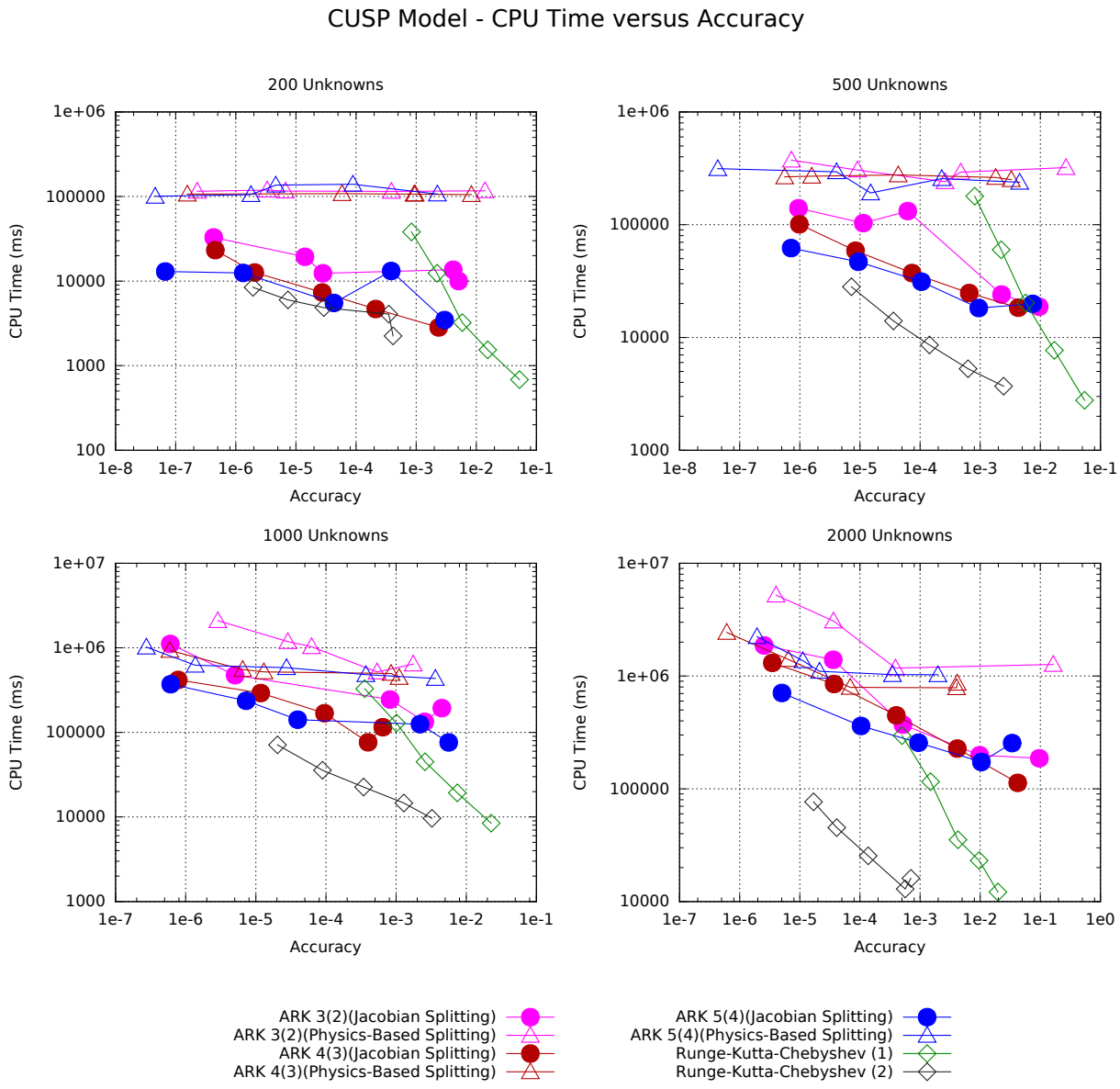

Figure 4.11: CPU time versus accuracy of IMEX and RKC methods applied to the `CUSP` problem. Tolerances range from 10^{-4} to 10^{-8}.

Results for the one-dimensional Brusselator problem show that the better type of splitting is highly dependent on the order of the IMEX method. For the ARK5 method, Jacobian splitting generally takes fewer steps and less CPU time than physics-based splitting; i.e., use of Jacobian splitting results in a more efficient method. Conversely, when $\alpha = 0.002$, the ARK3 method using Jacobian splitting is outperformed by the ARK3 method using physics-based splitting. In the other cases, there is no discernible difference between the performance of Jacobian splitting and physics-based splitting. In all cases, the RKC methods are underperformed. The work-precision plots for the one-dimensional Brusselator problem are shown in Figure 4.12.

When the Brusselator problem is extended into two dimensions, the results are similar. In all cases, the ARK4 and ARK5 methods using Jacobian splitting outperform those same methods using physics-based splitting. The ARK3 method using Jacobian splitting is outperformed by the ARK3 method using physics-based splitting; however, the difference in two dimensions is not nearly as severe. The work-precision plots for the two-dimensional Brusselator problem are shown in Figure 4.13. These results demonstrate how, even though the reaction term of the Brusselator problem is relatively non-stiff, it is still beneficial to use Jacobian splitting. Unlike the one-dimensional case, the RKC2 method now outperforms all of the IMEX methods; however, it cannot achieve the highest levels of accuracy. RKC1 underperforms relative to the IMEX methods.

4.5.3 Advection-Diffusion-Reaction Problems

Combustion Models

The combustion model defined by (3.12) involves contributing factors from advection, diffusion, and reaction. The diffusion term is treated implicitly and its discretization corresponds to $\mathbf{f}_I$; the advection and reaction terms are treated explicitly, and their discretizations are grouped together as part of $\mathbf{f}_E$. In every case, IMEX methods of orders four and five using Jacobian splitting outperform the same methods using physics-based splitting. As U_0 increases, the relative performance of Jacobian splitting also appears to increase, to the point that all methods using Jacobian splitting outperform those that do not. When the combustion model

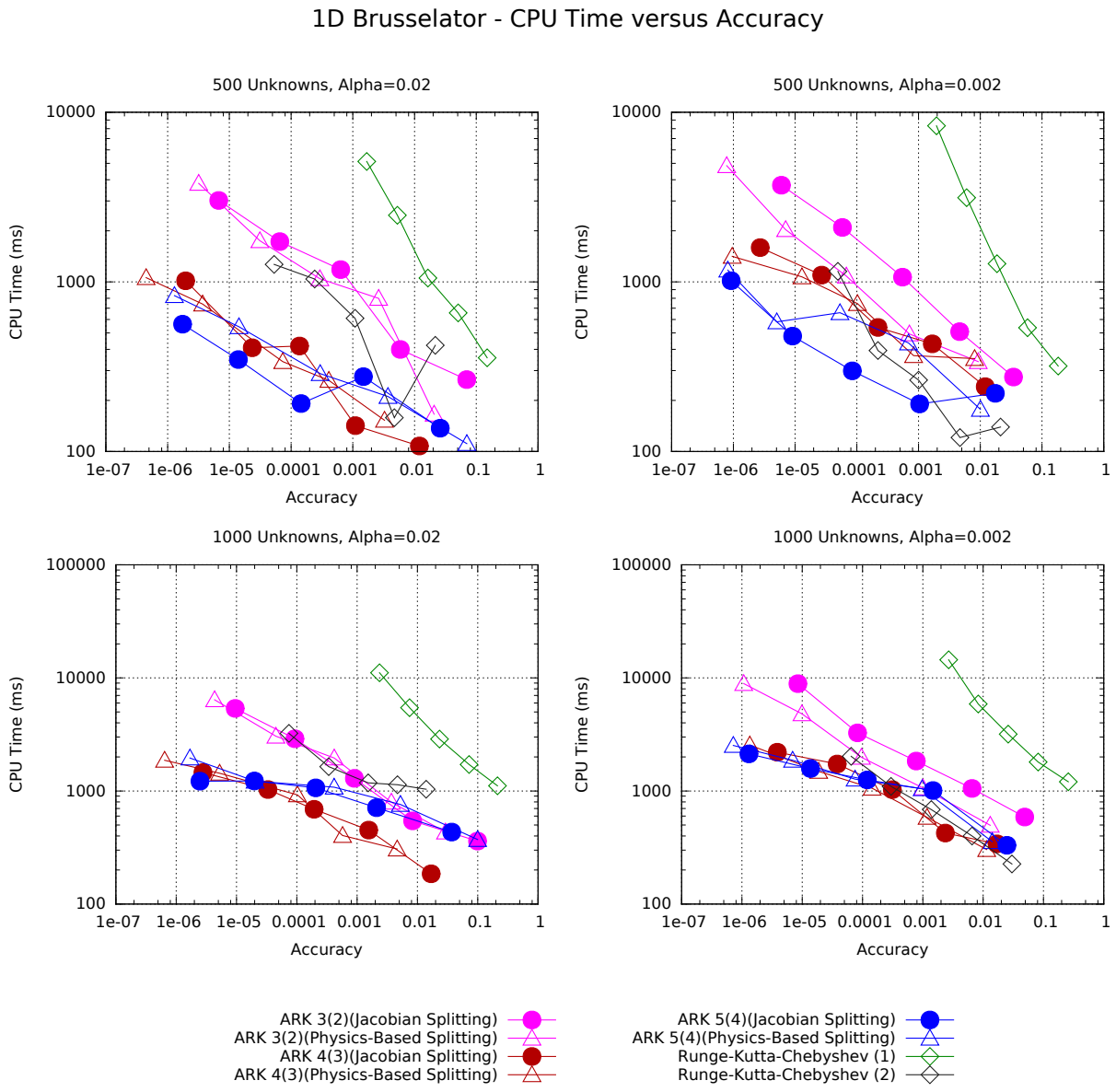

Figure 4.12: CPU time versus accuracy of IMEX and RKC methods applied to a one-dimensional Brusselator problems. Tolerances range from 10^{-4} to 10^{-8}.

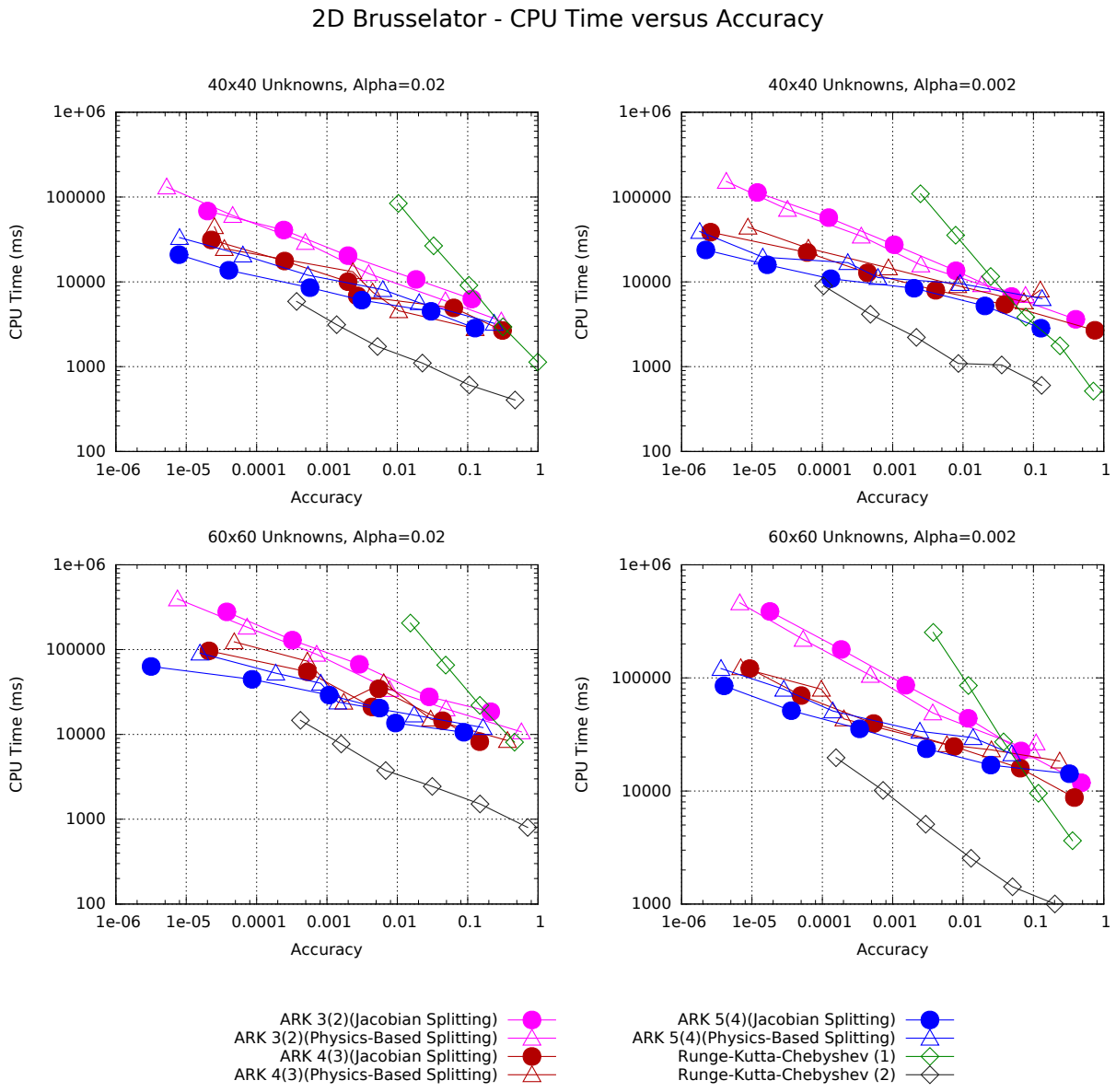

Figure 4.13: CPU time versus accuracy of IMEX and RKC methods applied to a two-dimensional Brusselator problems. Tolerances range from 10^{-4} to 10^{-8}.

involves FKPP and Fisher reactions, the relative performance of Jacobian splitting is also more significant than for the ignition reaction. The ignition problem nonetheless demonstrates the superior performance of Jacobian splitting. In all cases, the RKC methods of orders one and two are outperformed by the IMEX methods.

Figure 4.14 shows the work-precision plots for each of the parameter combinations. These results are for the FKPP, ignition, and Fisher combustion models, where each problem discretized the spatial domain with 1600 unknowns. Higher-order spatial discretizations were evaluated on the combustion models but did not show any significant difference in the relative performance of the results.

Tumour Angiogenesis

A physics-based splitting of the tumour angiogenesis model defined by (3.13) treats the diffusion term implicitly and the remaining terms explicitly. Figure 4.15 shows the work-precision plots of the numerical experiments associated with tumour angiogenesis model. In general, methods using Jacobian splitting perform better than physics-based splitting on these models. However, the tumour angiogenesis model yields a much more significant result.

At strict tolerances, IMEX methods using physics-based splitting are incapable of solving the tumour angiogenesis problems. The recorded errors for methods using physics-based splitting either show that a minimum stepsize of 10^{-12} was reached or that a maximum number of steps (1,000,000) was reached. The methods using Jacobian splitting successfully solve the problem at all requested tolerances. These results suggest that physics-based splitting does not appropriately capture the stiffness of this problem, whereas Jacobian splitting does.

The Concrete-Rewetting Problem

The concrete-rewetting problem in (3.14)–(3.18) is comprised of advection, diffusion, and reaction terms that are all non-linear. To perform physics-based splitting, the discretized

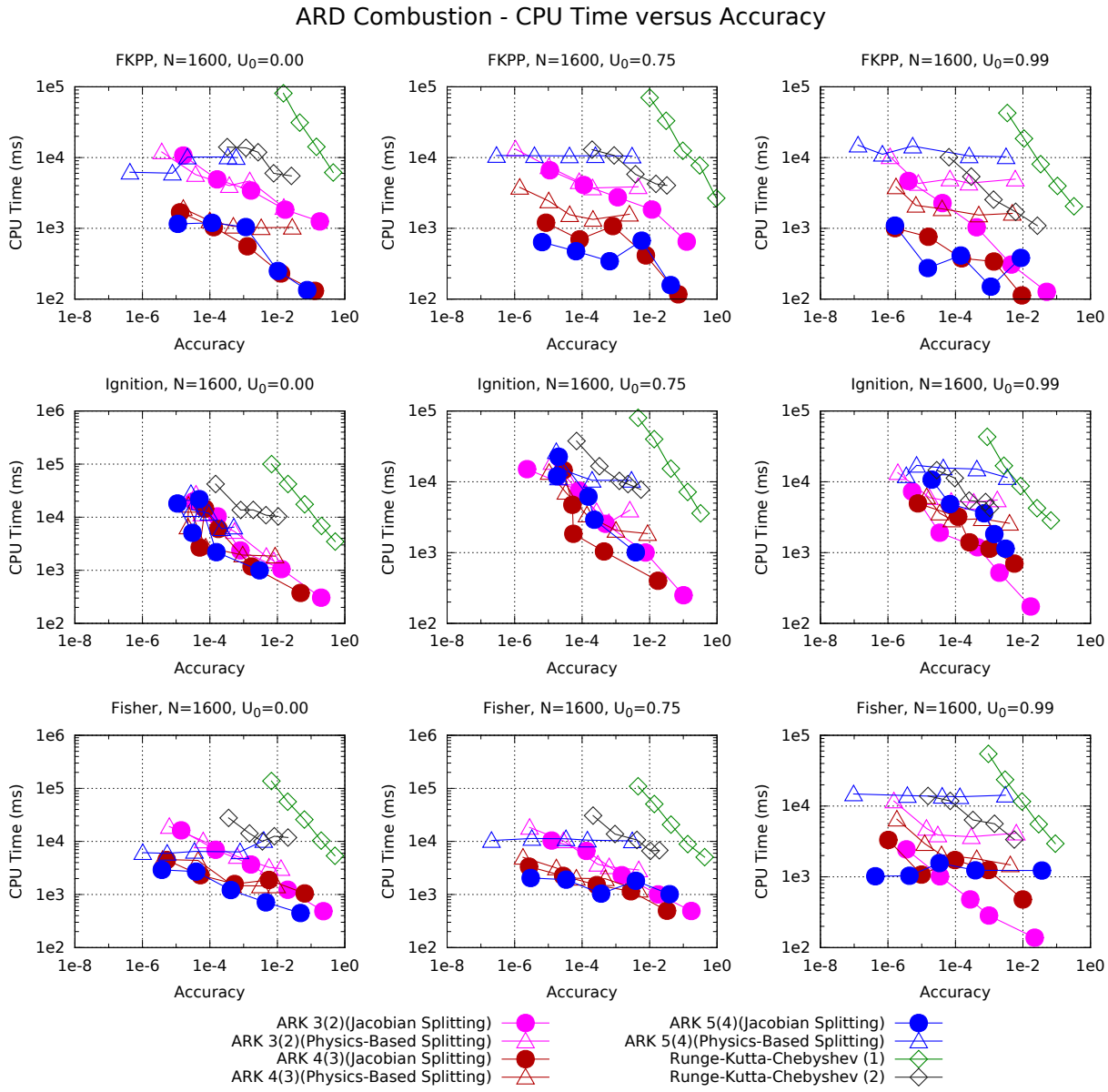

Figure 4.14: CPU time versus accuracy of IMEX and RKC methods applied to the set of one-dimensional combustion problems. Tolerances range from 10^{-4} to 10^{-8}.

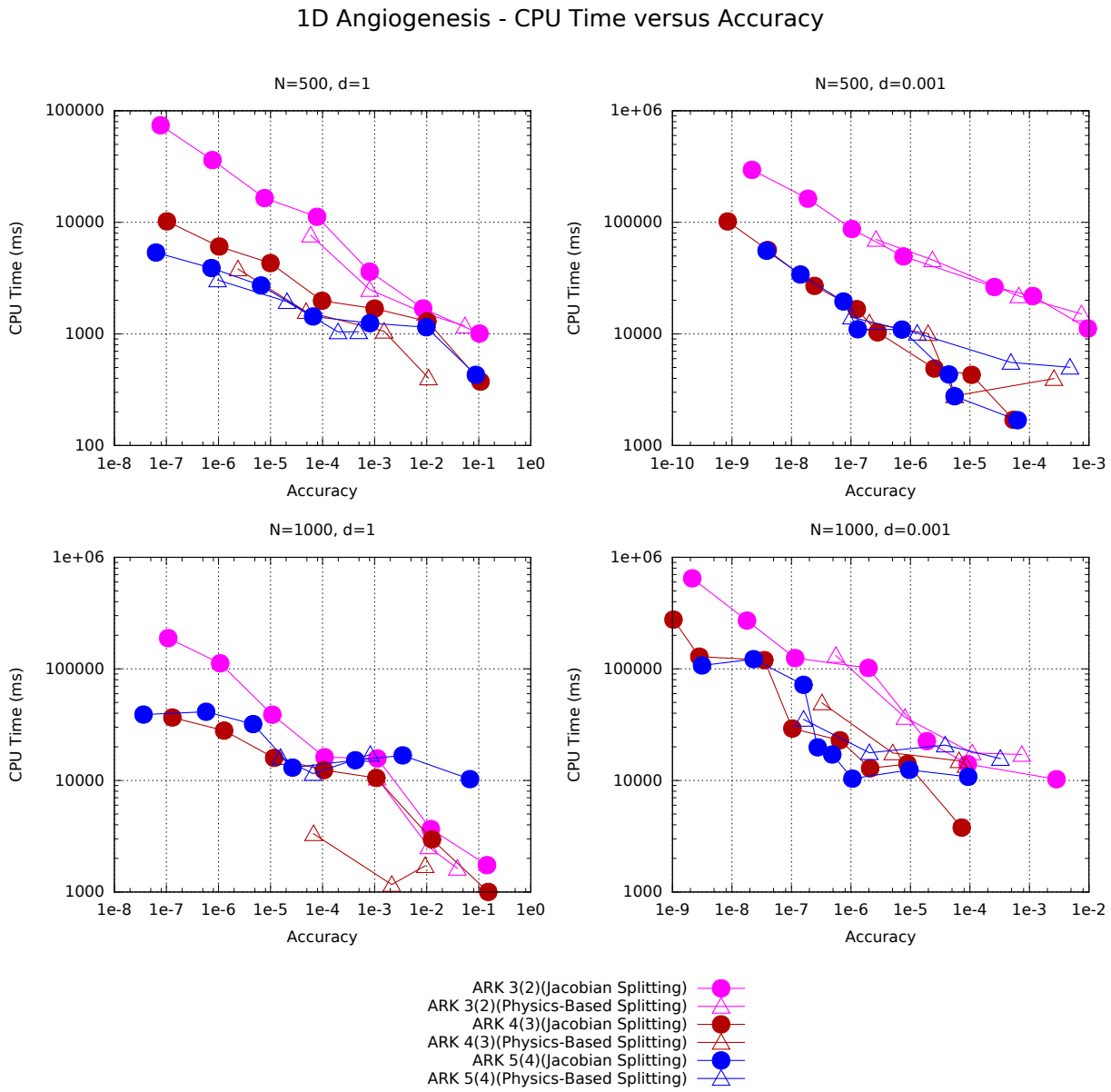

Figure 4.15: CPU time versus accuracy of IMEX and RKC methods applied to the tumour angiogenesis model. Tolerances range from 10^{-5} to 10^{-11}.

diffusion terms that are grouped as part of $\mathbf{f}_I$ are written as

$$\begin{aligned}
\frac{d\theta_j}{dt} &= \frac{1}{(\Delta x)^2}\left[D_{j+\frac{1}{2}}\left(\theta_{j+1}-\theta_j\right) - D_{j-\frac{1}{2}}\left(\theta_j-\theta_{j-1}\right)\right], \\
\frac{d(\theta_j C_{\alpha,j})}{dt} &= \frac{D_\alpha}{(\Delta x)^2}\left[\theta_{j+\frac{1}{2}}\left(C_{\alpha,j+1}-C_{\alpha,j}\right) - \theta_{j-\frac{1}{2}}\left(C_{\alpha,j}-C_{\alpha,j-1}\right)\right], \\
\frac{d(\theta_j C_{\beta,j})}{dt} &= \frac{D_\beta}{(\Delta x)^2}\left[\theta_{j+\frac{1}{2}}\left(C_{\beta,j+1}-C_{\beta,j}\right) - \theta_{j-\frac{1}{2}}\left(C_{\beta,j}-C_{\beta,j-1}\right)\right], \\
\frac{d(\theta_j C_{q,j})}{dt} &= \frac{D_q}{(\Delta x)^2}\left[\theta_{j+\frac{1}{2}}\left(C_{q,j+1}-C_{q,j}\right) - \theta_{j-\frac{1}{2}}\left(C_{q,j}-C_{q,j-1}\right)\right], \\
\frac{d(\theta_j C_{g,j})}{dt} &= 0,
\end{aligned}$$

and the discretized reaction terms that are grouped as part of $\mathbf{f}_E$ are written as

$$\begin{aligned}
\frac{d\theta_j}{dt} &= -\nu(\theta_j-\theta_r)^+\frac{m_w r_{\text{csh}}}{\rho_w m_{\text{csh}}}, \\
\frac{d(\theta_j C_{\alpha,j})}{dt} &= -\frac{u_{j+\frac{1}{2}}C_{\alpha,j+\frac{1}{2}} - u_{j-\frac{1}{2}}C_{\alpha,j-\frac{1}{2}}}{\Delta x} - (\theta_j-\theta_r)^+ r_\alpha, \\
\frac{d(\theta_j C_{\beta,j})}{dt} &= -\frac{u_{j+\frac{1}{2}}C_{\beta,j+\frac{1}{2}} - u_{j-\frac{1}{2}}C_{\beta,j-\frac{1}{2}}}{\Delta x} - (\theta_j-\theta_r)^+ r_\beta, \\
\frac{d(\theta_j C_{q,j})}{dt} &= -\frac{u_{j+\frac{1}{2}}C_{q,j+\frac{1}{2}} - u_{j-\frac{1}{2}}C_{q,j-\frac{1}{2}}}{\Delta x} + (\theta_j-\theta_r)^+(r_{\text{csh}} - k_{\text{prec}}C_{q,j} + k_{\text{diss}}C_{g,j}), \\
\frac{d(\theta_j C_{g,j})}{dt} &= (\theta_j-\theta_r)^+(k_{\text{prec}}C_{q,j} - k_{\text{diss}}C_{g,j}),
\end{aligned}$$

where the subscript $j = 1, 2, \ldots, N$ refers to each discretized point.

The RKC1 and RKC2 methods perform orders of magnitude than any of the IMEX methods, regardless of whether Jacobian splitting or physics-based splitting is used. The eigenvalues for the sink and insulated boundaries are shown in Appendix F. Notice that the eigenvalues are almost exclusively located near the negative real axis. It is easy to see that the stability region of RKC methods, discussed in Section 2.3.5, encompasses the eigenvalues when it is appropriately scaled by the timestep, perhaps explaining why the RKC method performs so well.

Numerical experiments are conducted for both sets of boundary conditions: a sink boundary and an insulated boundary. Results are shown for discretized problems with 100 unknowns and with 200 unknowns, including the runtimes for RKC methods of orders 1 and 2.

Work-precision plots for both sets of problem size are shown in Figure 4.16.

Jacobian splitting outperforms physics-based splitting on the concrete-rewetting problem for most tolerances with both types of boundary conditions. The sink boundary shows significant performance gains when using Jacobian splitting. The IMEX methods of orders three and four generally require fewer steps when using Jacobian splitting in comparison to physics-based splitting. They subsequently have shorter runtimes as well. The ARK5 methods perform poorly with respect to the other IMEX methods.

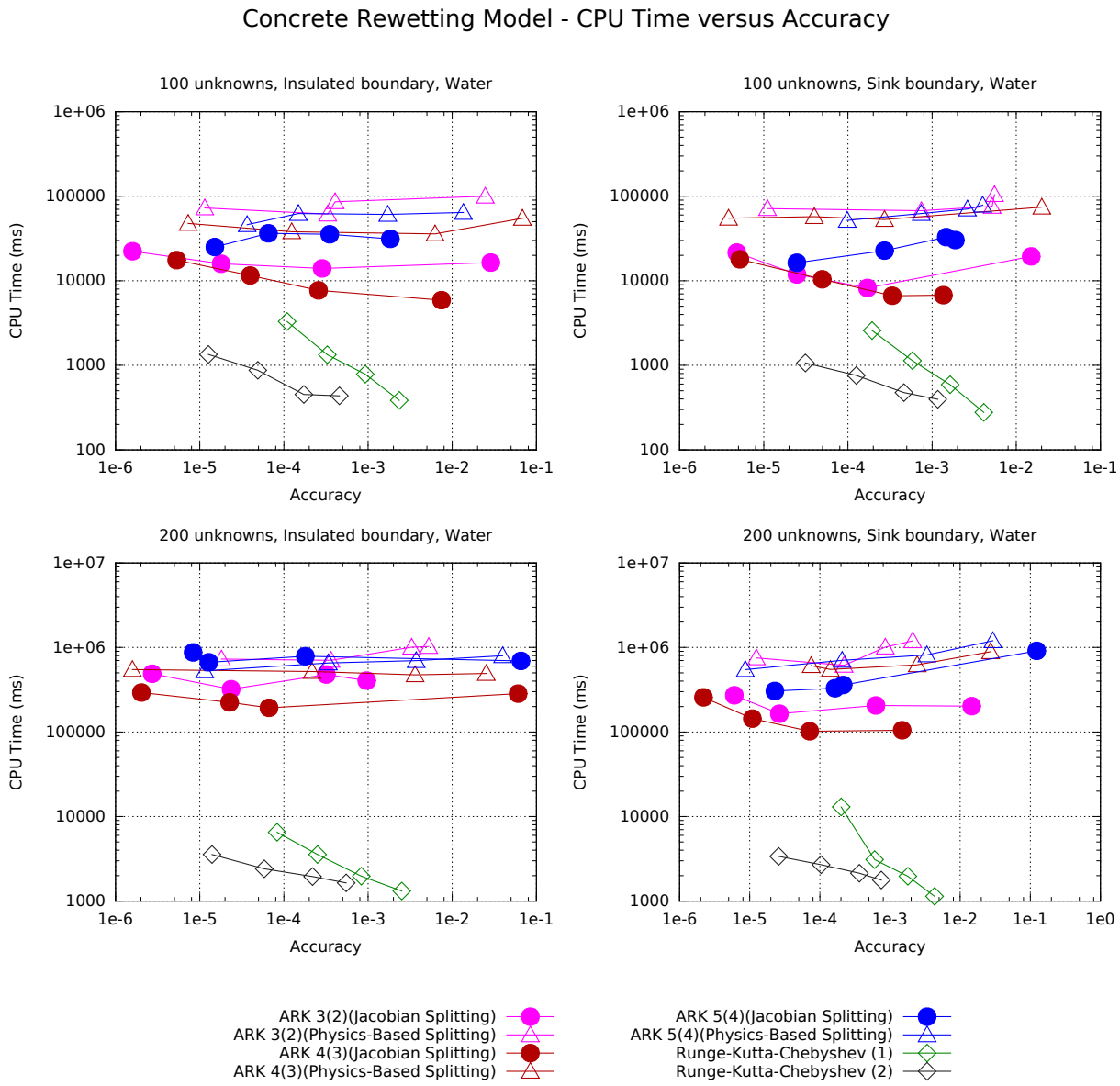

Figure 4.16: CPU time versus accuracy of IMEX and RKC methods applied to the concrete rewetting problem for both insulated and sink boundaries. Tolerances range from 10^{-4} to 10^{-9}.

Chapter 5

Conclusions and Future Work

This thesis conducts a study of 2-additive numerical methods on IVPs that represent the discretization of ADR equations. These methods have been applied to a test set of ADR equations to compare the performance of IMEX methods using Jacobian splitting to a variety of methods (including those IMEX methods) using physics-based splitting. The ARK3, ARK4, and ARK5 methods were evaluated using Jacobian splitting. Those three methods, in addition to PRKC, IRKC, DIRK-CF1, and DIRK-CF2, were evaluated using physics-based splitting. Several other standard non-additive methods were evaluated as well, the most notable of which are the RKC1 and RKC2 methods. Several standard RK methods were used as a baseline for comparisons. In general, the ARK methods outperformed the standard RK methods.

5.1 Summary of Results

The primary metric to gauge the performance of a numerical is the CPU time as a function of accuracy of the solution. Other metrics include the number of steps required, versus the accuracy of the solution. Eigenvalue plots are used to help understand why some methods perform particularly well on problems whereas others do not. Accuracy calculations are performed with respect to reference solutions that have been generated using the RODAS and RADAU5 methods, depending on which method is more efficient on a given problem.

The test set of ADR problems consists of two advection-diffusion models, three diffusion-reaction models, and three models with contributing factors from advection, diffusion, and reaction. These problems all arise from PDEs that are spatially discretized such that the above numerical methods for IVPs can be applied to them. Discretization schemes include

centred, forward, and backward finite difference methods and a second-order centred finite volume method.

The IMEX methods that use Jacobian splitting are shown to outperform those that use physics-based splitting on the suite of test problems. Further, the IMEX methods outperform PRKC, IRKC, and the DIRK-CF methods, regardless of which type of splitting is used. It is also found that different spatial discretization orders of the PDEs do not impact the relative performance of Jacobian splitting versus physics-based splitting. On many of the IVPs, RKC1 and RKC2 are competitive with the IMEX methods. In some cases, RKC2 is faster the IMEX methods; however, it is not able to attain the same level of accuracy.

The `pythODE++` PSE has been developed to evaluate numerical methods on IVPs. It is based on `pythODE`, which is a PSE written entirely in Python. The `pythODE++` PSE is designed to run experiments over many sets of parameters, such as a range of tolerances or problem sizes. It measures the runtimes of a numerical method as well as other statistics, such as the number of steps or the average number of Newton iterations per step. The PSE is written in favour of performance rather than in favour of a robust codebase; however, its numerical methods all share the same support code (e.g., vectors, matrices, etc.) to mitigate implementation-dependent runtime bias. The infrastructure of `pythODE++` is designed to use parallel computing to solve many IVPs simultaneously on a cluster of machines, vastly increasing the size of the parameter space that can be evaluated in a reasonable amount of time. Methods and IVPs in the PSE are written exclusively in C++ because C++ is a relatively low-level language. Supporting code, used for analysis of problem runs and the distribution of problem runs across a cluster, is written in Python for ease of development. The analysis and distribution code does not affect runtime measurements because these measurements are conducted internally by the IVP solvers.

5.2 Contributions of this Thesis

This thesis makes two distinct contributions to the fields of numerical analysis and scientific computing.

First, this thesis provides a rigorous study that compares physics-based splitting to Ja-

cobian splitting on a wide variety of ADR problems. Both of these splitting techniques are known; however, there does not presently exist a systematic and comprehensive study that compares the two splitting approaches. Knowledge of which splitting type to use can be useful when one is solving a problem and wishes to use some form of splitting. Jacobian splitting is shown to be a superior approach in terms of overall runtime with respect to the accuracy of the solution. Further, the implementation difficulty of a method using Jacobian splitting is comparable to physics-based splitting, an equally important fact. Although an approach may be shown to be superior, in practice, it might not be used if the implementation is too complicated. Jacobian splitting is also useful because it is an approach for splitting the RHS of an IVP that is not amenable to a physics-based splitting; e.g., see [44].

Second, this thesis develops a PSE, `pythODE++`, that is used to perform the comparisons between splitting approaches. It would be tedious and incredibly time consuming to perform a study of this magnitude without the supporting infrastructure of a PSE. This software is important because it is capable of performing analysis on IVPs that are relatively large over a large parameter space. It is also specifically designed for the analysis of numerical methods for IVPs; therefore, it is able to make optimizations that more generalized PSEs are unable to provide. Although there are numerous existing PSEs, it is difficult to find a suitable one that provides all of these capabilities, giving motivation for the development of `pythODE++`.

5.3 Future Work

This section outlines future work that might extend the current study.

5.3.1 Extension to Three-Dimensional Models

The models in this thesis consist of problems in one and two spatial dimensions. Future work involves extending the set of IVPs to include one or more problems involving advection, diffusion, and reaction in three spatial dimensions. Many physical models are three dimensional; therefore, it is desirable to evaluate numerical methods on the underlying IVPs of three-dimensional PDE models. It is likely that the benefits of Jacobian splitting extend to three dimensions, but this hypothesis should be verified directly. An example of a

three-dimensional ADR problem is a combustion simulation with compressible flow and more than one reacting species. Such a system is incredibly complex, consisting of many coupled PDEs including momentum equations for each spatial dimension, an energy equation, and continuity equations for each quantity present in the simulation [14].

For these problems, it might be advantageous to support alternative methods of spatial discretization than finite differences, such as finite element methods [16]. Three-dimensional models become complex due to the large number of unknowns required by a fixed grid. Therefore, a non-uniform, adaptive discretization might be desirable.

5.3.2 Merging `pythODE++` and `pythODE`

Although much of the code and surrounding infrastructure of `pythODE++` is based on `pythODE`, `pythODE++` functions independently. It would be advantageous for both PDEs to be merged. They both have the same goal of running numerical experiments to compare sets of numerical methods and IVPs. The `pythODE` PSE allows for more complex and customizable numerical studies, whereas `pythODE++` is more specialized and more optimized. Future work involves developing `pythODE` and `pythODE++` to share a common Python front-end that is responsible for distributing numerical experiments to many processes on many different hosts. The back-ends of the two PSEs would remain completely separate, but they would implement a common protocol. Further, a code transformation tool would be written such that one could implement an IVP or method in either C++ or Python (using some subset of the languages), thereby allowing one to seamlessly solve IVPs using either the Python back-end or the C++ back-end.

5.3.3 Extension to 3-Additive Methods

This thesis only considers 2-additive methods. For an ADR equation, methods that treat advection, diffusion, and reaction each with a separate methods, i.e., 3-additive methods, might outperform the 2-additive methods introduced in this work. Specifically, one approach is to treat the discretized advection terms with an (explicit) ExpRK method, the discretized reaction terms with an ERK method, and the discretized diffusion terms with an IRK method.

Such an approach might also be a candidate for Jacobian splitting, where advection would be treated with the same ExpRK method, but the remaining terms would be 2-additively split using Jacobian splitting, thereby treating the linear term implicitly and the non-linear term explicitly. Future work involves deriving order conditions and developing implementations for these 3-additive methods. These methods can be incorporated into the current suite of methods and evaluated.

5.3.4 Additional Methods

The coefficients of the ARK3, ARK4, and ARK5 methods represent only single solutions to their respective order conditions. There are many degrees of freedom in the order conditions; investigating these degrees of freedom might produce methods that are better than the presently chosen coefficients for ARK3, ARK4, ARK5. Unfortunately, calculating these coefficients is not trivial; as order increases, the number of order conditions increases dramatically, resulting in a large non-linear algebraic system with possibly hundreds of unknowns [30]. Additionally, the DIRK-CF1 and DIRK-CF2 methods do not show many signs of improvement over their classical RK equivalents. Coefficients should be derived for higher-order ExpARK methods, so that these methods can be compared fairly to ARK3, ARK4, and ARK5.

5.3.5 Parallelized Methods

The `pythODE++` PSE is presently capable of solving many permutations of parameters such as the IVP, method, solver, and tolerances in parallel. Conceptually, it is relatively simple to parallelize this component of the PSE because each problem run is inherently parallel. However, all numerical methods in this study are serial. It might be possible to take advantage of a finer-grained parallelism by using parallelized numerical methods. An overview of basic strategies for parallelizing RK methods is presented in [27].

References

[1] U. M. Ascher and L. R. Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. SIAM: Society for Industrial and Applied Mathematics, 1998.

[2] U. M. Ascher, S. Ruuth, and R. J. Spiteri. Implicit-explicit Runge–Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25:151–167, 1997.

[3] A. Bentur. Cementitious materials - nine millennia and a new century: past, present, and future. *J Mater Civ Eng*, 14:2–22, 2002.

[4] F. Bianco, S. Chibbaro, and R. Prud'homme. Étude d'une équation de convection-réaction-diffusion en écoulement compressible. Presented at: Rencontre du non linéaire, March 2011.

[5] R. Bridson. *Fluid simulation for computer graphics*. A K Peters, Wellesley, 2007.

[6] J. C. Butcher. A history of Runge–Kutta methods. *Appl. Numer. Math.*, 20(3):247–260, March 1996.

[7] 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.

[8] E. Celledoni and B. K. Kometa. Semi-lagrangian Runge-Kutta Exponential integrators for convection dominated problems. *Journal of Scientific Computing*, 41(1):139–164, 2009.

[9] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *Communications, IEEE Transactions on*, 22(5):637–648, 1974.

[10] M. Chapwanya, W. Liu, and J. M. Stockie. A model for reactive porous transport during re-wetting of hardened concrete. *J Eng Math*, 65:53–73, January 2009.

[11] A. J. Chorin. Numerical solution of the Navier–Stokes equations. *Math Comp*, 22:745–762, 1968.

[12] P. E. Crouch and R. Grossman. Numerical integration of ordinary differential equations on manifolds. *J Nonlinear Sci*, 3:1–33, 1993.

[13] T. A. Davis. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, June 2004.

[14] T. Echekki and E. Mastorakos. *Turbulent Combustion Modeling*, volume 95 of *Fluid Mechanics and Its Applications*. Springer Netherlands, 2011.

[15] W. H. Enright and J. D. Pryce. Two FORTRAN packages for assessing initial value methods. *ACM Trans. Math. Softw.*, 13(1):1–27, March 1987.

[16] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Studentlitteratur, 1996.

[17] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[18] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen. Colpack: Software for graph coloring and related problems in scientific computing. *ACM Trans. Math. Softw.*, 40(1):1:1–1:31, Oct. 2013.

[19] I. Goda, M. Assidi, S. Belouettar, and J. Ganghoffer. A micropolar anisotropic constitutive model of cancellous bone from discrete homogenization. *Journal of the Mechanical Behavior of Biomedical Materials*, 16(0):87–108, 2012.

[20] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I: Nonstiff problems*, volume 8 of *Springer series in computational mathematics*. Springer-Verlag, Berlin, 1993.

[21] 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.

[22] E. Hairier, C. Lubich, and G. Wanner. *Geometric numerical integration*, volume 31 of *Springer series in computational mathematics*. Springer-Verlag, Berlin, 2002.

[23] C. Hall. Anomalous diffusion in unsaturated flow: fact or fiction? *Cement and Concrete Research*, 37(3):378–385, 2007.

[24] T. Hartmann and A. S. Verkman. Model of ion transport regulation in chloride-secreting airway epithelial cells. *Biophys J*, 58(2):391–401, August 1990.

[25] W. Hundsdorfer and J. G. Verwer. *Numerical Solution of time-dependent advection-diffusion-reaction equations*, volume 33 of *Springer series in computational mathematics*. Springer-Verlag, Berlin, 2003.

[26] Z. Jackiewicz. *General linear methods for ordinary differential equations*. John Wiley & Sons, Inc, Hoboken, New Jersey, 2009.

[27] K. R. Jackson. A survey of parallel numerical methods for initial value problems for ordinary differential equations. *IEEE Trans. on Magnetics*, 27:3792–3797, 1991.

[28] L. M. Jiji. *Heat Convection*. Springer Berlin Heidelberg, 2nd edition, 2009.

[29] D. S. Jones, M. Plank, and B. D. Sleeman. *Differential Equations and Mathematical Biology*. Chapman & Hall/CRC, Taylor & Francis Group, second edition, 2010.

[30] C. A. Kennedy and M. H. Carpenter. Additive Runge–Kutta schemes for convection-diffusion-reaction equations. Technical report, 2001.

[31] A. Kroshko. Integrating-factor-based 2-additive Runge–Kutta methods for advection-reaction-diffusion equations. Master's thesis, Department of Computer Science, University of Saskatchewan, May 2011.

[32] Y. Maday, A. T. Patera, and E. M. Rønquist. An operator-integration-factor splitting method for time-dependent problems: Application to incompressible fluid flow. *Journal of Scientific Computing*, 5(4):263–292, 1990.

[33] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, A. J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[34] 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.

[35] B. V. Minchev and W. Wright. A review of exponential integrators for first order semi-linear problems. *Preprint Numerics*, 2, 2005.

[36] J. Rice and R. F. Boisvert. From scientific software libraries to problem solving environments. *IEEE Computational Science and Engineering*, 3:44–53, 1996.

[37] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[38] G. Satir and D. Brown. *C++ - the core language: a foundation for C programmers*. O'Reilly, 1995.

[39] L. F. Shampine, I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge University Press, 2003.

[40] L. F. Shampine, B. P. Sommeijer, and J. G. Verwer. IRKC: An IMEX solver for stiff diffusion-reaction PDEs. *Journal of Computational and Applied Mathematics*, 196(2):485–497, 2006.

[41] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[42] G. Söderlind. Automatic control and adaptive time-stepping. *Numerical Algorithms*, 31(1-4):281–310, 2002.

[43] 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, January 1997.

[44] R. J. Spiteri and R. C. Dean. On the performance of implicit-explicit Runge–Kutta methods in models of cardiac electrical activity. *IEEE Transactions on Biomedical Engineering*, 55(5):1488–1495, May 2008.

[45] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[46] J. G. Verwer, W. Hundsdorfer, and B. P. Sommeijer. Convergence properties of the Runge–Kutta–Chebyshev method. *Numerische Mathematik*, 57(1):157–178, 1990.

[47] A. Walther and A. Griewank. Getting started with adol-c. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, chapter 7, pages 181–202. Chapman-Hall CRC Computational Science, 2012.

[48] T. Williams, C. Kelly, et al. `gnuplot`. http://www.gnuplot.info.

[49] C. J. Zbinden. Partitioned Runge–Kutta–Chebyshev methods for diffusion-advection-reaction problems. *SIAM J Sci Comput*, 33(4):1707–1725, July 2011.

Appendix A

Derivations of Order Conditions for Runge–Kutta Methods

This material is largely based on derivations for order conditions in [20]. Order conditions for a method of order p are calculated by matching the Taylor expansion of the numerical solution to the Taylor expansion of the local solution up to the term of order p. Both expansions are taken at $\Delta t = 0$; therefore, it is sufficient to derive order conditions by matching derivatives of the numerical and local solutions. Derivations are shown for RK methods up to third order. Higher-order methods can be derived by continuing the process of calculating higher derivatives. In this section, tensor notation is used to represent the components of vectors; i.e., f^J corresponds to the Jth component of $\mathbf{f}$. Additionally, note that $f^J = f^J(\mathbf{y}_{n-1})$ unless specific function arguments are given.

A.1 Taylor Expansion of the True Solution

The Taylor expansion an IVP of the form (2.2) is written as

$$y_n^J = y_{n-1}^J + \sum_{i=1}^{p} \frac{(\Delta t)^i}{i!} \frac{\mathrm{d}^i y_{n-1}^J}{\mathrm{d}t^i} + \mathcal{O}\big((\Delta t)^{p+1}\big), \tag{A.1}$$

where $J = 1, \ldots, m$. By definition of the IVP, the first derivative is written

$$\frac{\mathrm{d}y_{n-1}^J}{\mathrm{d}t} = f^J.$$

Recall the chain rule for vector-valued functions:

$$\frac{\mathrm{d}f^J}{\mathrm{d}t} = \frac{\partial f^J}{\partial y^1}\frac{\mathrm{d}y^1}{\mathrm{d}t} + \cdots + \frac{\partial f^J}{\partial y^m}\frac{\mathrm{d}y^m}{\mathrm{d}t}.$$

By applying the chain rule to the time derivatives, the subsequent derivatives required by A.1 can be calculated as

$$\begin{aligned}
\frac{\mathrm{d}^2 y_{n-1}^J}{\mathrm{d}t^2} &= \sum_{K=1}^{m} \frac{\partial f^J}{\partial y^K} f^K, \\
\frac{\mathrm{d}^3 y_{n-1}^J}{\mathrm{d}t^3} &= \sum_{K=1}^{m}\sum_{L=1}^{m} \left(\frac{\partial^2 f^J}{\partial y^K \partial y^L} f^K f^L + \frac{\partial f^J}{\partial y^K}\frac{\partial f^K}{\partial y^L} f^L \right).
\end{aligned}$$

A.2 Taylor Expansion of the Numerical Solution

One step of an s-stage RK method (see Section 2.3) is written as

$$k_i^J = f^J\left(\mathbf{y}_{n-1} + \Delta t \sum_{j=1}^{s} a_{ij}\mathbf{k}_j\right),$$
$$y_n^J = y_{n-1}^J + \Delta t \sum_{j=1}^{s} b_j k_j^J,$$

where $i = 1, \ldots, s$. It is convenient for analysis to use the transformation Y_i^J, such that $k_i^J = f^J(Y_i^1, \ldots, Y_i^m)$. Therefore, the RK method can be rewritten as

$$Y_i^J = y_{n-1}^J + \Delta t \sum_{j=1}^{s} a_{ij} f^J(Y_j^1, \ldots, Y_j^m),$$
$$y_n^J = y_{n-1}^J + \Delta t \sum_{j=1}^{s} b_j f^J(Y_j^1, \ldots, Y_j^m).$$

Due to the similarity of the formula for Y_i^J and y_n^J, it is sufficient to treat y_n^J as an additional stage. The RK method is now written as

$$Y_i^J = y_{n-1}^J + \Delta t \sum_{j=1}^{s} a_{ij} f^J(Y_j^1, \ldots, Y_j^m),$$

where $i = 1, 2, \ldots, s+1$, $a_{s+1,j} = b_j$, and $y_n^J = Y_{s+1}^J$. The general Leibniz rule is now useful to compute derivatives of Y_i^J because applying the chain rule would result in an infinite recursion. A simplified version of the rule that are useful to the following calculations is written

$$\frac{\mathrm{d}^\alpha [t\beta(t)]}{\mathrm{d}t^\alpha} = \alpha \frac{\mathrm{d}^{\alpha-1}\beta(t)}{\mathrm{d}t^{\alpha-1}}.$$

A new substitution $\xi_j^J = f^J(Y_j^1, \ldots, Y_j^m)$ is introduced to simplify notation. Note that $\xi_j^J|_{\Delta t=0} = f^J$. Derivatives of Y_i^J are thus calculated as

$$\left.\frac{\mathrm{d}Y_i^J}{\mathrm{d}t}\right|_{\Delta t=0} = \left.\sum_{j=1}^{s} a_{ij}\xi_j^J\right|_{\Delta t=0} = \sum_{j=1}^{s} a_{ij} f^J,$$

$$\left.\frac{\mathrm{d}^2 Y_i^J}{\mathrm{d}t^2}\right|_{\Delta t=0} = 2\sum_{j=1}^{s} a_{ij} \sum_{K=1}^{m} \frac{\partial \xi_j^J}{\partial y^K} \left.\frac{\mathrm{d}Y_j^K}{\mathrm{d}t}\right|_{\Delta t=0} = 2\sum_{j=1}^{s}\sum_{k=1}^{s} a_{ij}a_{jk} \sum_{K=1}^{m} \frac{\partial f^J}{\partial y^K} f^K,$$

$$
\begin{aligned}
\left.\frac{\mathrm{d}^3 Y_i^J}{\mathrm{d}t^3}\right|_{\Delta t=0} &= 3\sum_{j=1}^{s} a_{ij}\frac{\mathrm{d}}{\mathrm{d}t}\left(\sum_{K=1}^{m}\frac{\partial \xi_j^J}{\partial y^K}\frac{\mathrm{d}Y_j^K}{\mathrm{d}t}\right)\Bigg|_{\Delta t=0} \\
&= 3\sum_{j=1}^{s} a_{ij}\sum_{K=1}^{m}\left(\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \xi_j^J}{\partial y^K}\right)\frac{\mathrm{d}Y_j^K}{\mathrm{d}t} + \frac{\partial \xi_j^J}{\partial y^K}\frac{\mathrm{d}^2 Y_j^K}{\mathrm{d}t^2}\right)\Bigg|_{\Delta t=0} \\
&= 3\sum_{j=1}^{s} a_{ij}\sum_{K=1}^{m}\left(\sum_{L=1}^{s}\frac{\partial^2 \xi_j^J}{\partial y^K \partial y^L}\frac{\mathrm{d}Y_j^K}{\mathrm{d}t}\frac{\mathrm{d}Y_j^L}{\mathrm{d}t} + \frac{\partial \xi_j^J}{\partial y^K}\frac{\mathrm{d}^2 Y_j^K}{\mathrm{d}t^2}\right)\Bigg|_{\Delta t=0} \\
&= 3\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} a_{ij}a_{jk}a_{jl}\sum_{K=1}^{m}\sum_{L=1}^{m}\frac{\partial^2 f^J}{\partial y^K \partial y^L} f^K f^L \\
&\quad + 3\cdot 2\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} a_{ij}a_{jk}a_{kl}\sum_{K=1}^{m}\sum_{L=1}^{m}\frac{\partial f^J}{\partial y^K}\frac{\partial f^K}{\partial y^L} f^L.
\end{aligned}
$$

A.3 Order Conditions

Matching the above derivatives between the numerical and local expansions yields the order conditions for a third-order RK method. One condition comes from matching the first derivative, one comes from matching the second derivative, and two more come from matching the third derivative. The goal is to match expansions of $\mathbf{y}_n^J$, which corresponds to Y_{s+1}^J in the numerical solution. The coefficients $a_{ij} = a_{s+1,j}$ can be replaced with b_j to follow notation of the coefficients in the Butcher tableau. The four order conditions are therefore written

$$
\sum_{j=1}^{s} b_j = 1, \qquad \sum_{j=1}^{s}\sum_{k=1}^{s} b_j a_{jk} = \frac{1}{2},
$$

$$
\sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} b_j a_{jk}a_{jl} = \frac{1}{3}, \qquad \sum_{j=1}^{s}\sum_{k=1}^{s}\sum_{l=1}^{s} b_j a_{jk}a_{kl} = \frac{1}{6}.
$$

Appendix B

Derivations of Finite Difference Methods

Numerical approximation to derivatives of varying order can be calculated using an approach known as the method of finite differences. This method is applied to a function $u(x)$ that has been discretized on a generally uniform grid. The derivative of order q of $u(x)$ can be approximated by combining weighted values of points near to x on the grid. The finite difference method uses a stencil, which identifies which nearby points to use and the amount by which to weight them. The goal of finite differences is to choose points such that the approximation to the derivative matches the Taylor expansion of the exact solution up to a desired order of accuracy.

First, recall the Taylor expansion up to order p:

$$u(x+\Delta x) = u(x) + \sum_{i=1}^{p} \frac{(\Delta x)^i}{i!} \frac{\partial^i u}{\partial x^i}\bigg|_x + \mathcal{O}\big((\Delta x)^{p+1}\big).$$

When applied to a discretized grid and letting $u_n = u(x + n\Delta x)$ and writing derivatives in Lagrange notation, the expansion is written as:

$$u_n = u_0 + \sum_{i=1}^{p} \frac{n^i (\Delta x)^i u_0^{(i)}}{i!} + \mathcal{O}\big((\Delta x)^{p+1}\big). \tag{B.1}$$

Combining many equations of the form (B.1) with varying $n \in \mathbb{Z}$ can be used to eliminate all derivatives up to a desired order and solve for whichever derivative is sought. Varying n has the effect of including many points due to the u_n and u_0 terms from (B.1). The general problem for an arbitrary number of points and an arbitrary derivative of order q can be described as a linear system as follows:

$$\begin{bmatrix} \frac{n_1}{1!} & \frac{n_2}{1!} & \cdots & \frac{n_m}{1!} \\ \frac{n_1^2}{2!} & \frac{n_2^2}{2!} & \cdots & \frac{n_m^2}{2!} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{n_1^m}{m!} & \frac{n_2^m}{m!} & \cdots & \frac{n_m^m}{m!} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} \delta_{1,q} \\ \delta_{2,q} \\ \vdots \\ \delta_{m,q} \end{bmatrix}, \qquad w_0 = \sum_{i=1}^{m} w_i, \qquad \delta_{i,q} = \begin{cases} 0 & \text{when } i \neq q \\ 1 & \text{when } i = q \end{cases}$$

where m is the number of points in the stencil (not including the point at u_0) and w_i corresponds to the weighted value for each stencil point. The derivative of order qth can then be calculated as

$$\frac{\mathrm{d}^q u}{\mathrm{d}x^q} = \frac{\sum_{i=0}^{m} w_i u_i}{(\Delta x)^q}$$

Appendix C

Derivation of Additive Exponential Runge–Kutta Methods

The following calculations describe how to derive a 2-additive ExpRK methods by using operator integrating factor splitting [8, 32]. Many IVPs can be written as

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t}(t,\mathbf{y}(t)) = \mathbf{f}\big(t,\mathbf{y}(t)\big) + \mathbf{C}\big(t,\mathbf{y}(t)\big)\mathbf{y}(t), \tag{C.1}$$

where $\mathbf{f}(t,\mathbf{y}(t))$ contains what is treated with an classical RK method and $\mathbf{C}(t,\mathbf{y}(t)) : \mathbb{R} \times R^m \mapsto \mathbb{R}^m$ is a matrix representing what is treated with an ExpRK method.

Physics-based splitting allows a CFERK to be applied to the $\mathbf{C}\big(t,\mathbf{y}(t)\big)\mathbf{y}(t)$ term and a classical RK method to be applied to the $\mathbf{f}\big(t,\mathbf{y}(t)\big)$ term. The first step is to consider the substitution

$$\mathbf{y}(t) = \mathbf{\Phi}(t)\mathbf{z}(t), \tag{C.2}$$

$$\frac{\mathrm{d}\mathbf{\Phi}}{\mathrm{d}t}(t) = \mathbf{C}\big(t,\mathbf{\Phi}(t)\mathbf{z}(t)\big)\mathbf{\Phi}(t), \tag{C.3}$$

$$\mathbf{\Phi}(t_{n-1}) = \mathbf{I},$$

where $\mathbf{\Phi}(t) \in \mathbb{R}^{m\times m}$ is an unknown matrix, $\mathbf{z}(t) \in \mathbb{R}^m$ is an unknown vector, and $\mathbf{I} \in \mathbb{R}^{m\times m}$ is the identity matrix. Differentiating (C.2) and substituting it into (C.1) yields

$$\frac{\mathrm{d}}{\mathrm{d}t}\left[\mathbf{\Phi}(t)\right]\mathbf{z}(t) + \mathbf{\Phi}(t)\frac{\mathrm{d}}{\mathrm{d}t}\left[\mathbf{z}(t)\right] = \mathbf{C}\big(t,\mathbf{\Phi}(t)\mathbf{z}(t)\big)\mathbf{\Phi}(t)\mathbf{z}(t) + \mathbf{f}\big(t,\mathbf{\Phi}(t)\mathbf{z}(t)\big).$$

On substitution of (C.3), the above now becomes

$$\mathbf{\Phi}(t)\frac{\mathrm{d}}{\mathrm{d}t}\left[\mathbf{z}(t)\right] = \mathbf{f}\big(t,\mathbf{\Phi}(t)\mathbf{z}(t)\big).$$

Therefore, (C.1) can be split into two coupled equations:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{\Phi}(t) = \mathbf{C}(t,\mathbf{y}(t))\mathbf{\Phi}(t), \qquad \mathbf{\Phi}(t_{n-1}) = \mathbf{I},$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{z}(t) = \mathbf{\Phi}^{-1}(t)\mathbf{f}\big(t,\mathbf{\Phi}(t)\mathbf{z}(t)\big), \qquad \mathbf{z}(t_{n-1}) = \mathbf{y}_{n-1},$$

The initial condition for $\mathbf{z}(t)$ comes from the fact that $\mathbf{z}(t) = \mathbf{\Phi}^{-1}(t)\mathbf{y}(t)$. Applying an s-stage

CFERK to the first equation for $\mathbf{\Phi}(t)$ for a timestep from t_{n-1} to t_n is written

$$\mathbf{\Phi}_i = \prod_{k=J}^{1} \exp\left(\Delta t_n \sum_{j=1}^{s} \alpha_{ij}^{[k]} \mathbf{C}(\mathbf{y}_j)\right), \qquad \mathbf{\Phi}_n = \prod_{k=J}^{1} \exp\left(\Delta t_n \sum_{j=1}^{s} \beta_{j}^{[k]} \mathbf{C}(\mathbf{y}_j)\right),$$

where $i = 1, 2, ..., s$ and the coefficients $\alpha_{ij}^{[k]}$ and $\beta_j^{[k]}$ correspond to $a_{ij}^{[k]}$ and $b_j^{[k]}$ in the Butcher tableau of the CFERK method. Applying an s-stage classical RK method to the second equation for $\mathbf{z}(t)$ is written

$$\mathbf{z}_i = z_{n-1} + \Delta t_n \sum_{j=1}^{s} a_{ij} \mathbf{\Phi}_j^{-1} \mathbf{f}_j,$$
$$\mathbf{z}_n = z_{n-1} + \Delta t_n \sum_{j=1}^{s} b_j \mathbf{\Phi}_j^{-1} \mathbf{f}_j.$$

Therefore, by substituting into (C.2), the additive method becomes

$$\mathbf{y}_i = \mathbf{\Phi}_i z_{n-1} + \Delta t_n \sum_{j=1}^{s} a_{ij} \mathbf{\Phi}_i \mathbf{\Phi}_j^{-1} \mathbf{f}_j,$$
$$\mathbf{y}_n = \mathbf{\Phi}_i y_{n-1} + \Delta t_n \sum_{j=1}^{s} b_j \mathbf{\Phi}_i \mathbf{\Phi}_j^{-1} \mathbf{f}_j,$$

which is equivalent to the form (2.17).

Appendix D

Order Conditions for Additive Exponential Runge–Kutta Methods

The derivation of order conditions for additive exponential RK methods is significantly more complicated than for classical RK methods. However, the principle of matching the Taylor expansion of the numerical solution to that of the true solution is the same. Derivations are shown up to order two. Higher-order derivations are significantly more complicated and are not covered in this thesis. Consider an IVP of the form

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{C}(\mathbf{y})\mathbf{y} + \mathbf{f}(\mathbf{y}),$$

where $\mathbf{y} \in \mathbb{R}^m$, $\mathbf{C} \in \mathbb{R}^{m\times m}$, $\mathbf{f} \in \mathbb{R}^m$. In tensor notation, the IVP can be equivalently written

$$\frac{\mathrm{d}y^J}{\mathrm{d}t} = \sum_{K=1}^{m} c^{JK} y^K + f^J, \tag{D.1}$$

where $c^{JK} = c^{JK}(y^1, \ldots, y^m)$, $f^J = f^J(y^1, \ldots, y^m)$, and $J = 1, 2, \ldots, m$.

D.1 Derivatives of the Exact Solution

The first derivative is given by the problem definition in (D.1). Derivatives of c^{JK} and f^J are useful in further computations. They are written

$$\frac{\mathrm{d}c^{JK}}{\mathrm{d}t} = \sum_{L=1}^{m} \frac{\partial c^{JK}}{\partial y^L} \left(\sum_{M=1}^{m} c^{LM} y^M + f^L \right), \tag{D.2}$$

$$\frac{\mathrm{d}f^J}{\mathrm{d}t} = \sum_{K=1}^{m} \frac{\partial f^J}{\partial y^K} \left(\sum_{L=1}^{m} c^{KL} y^L + f^K \right). \tag{D.3}$$

The second derivative now follows from the first derivative:

$$\frac{\mathrm{d}^2 y^J}{\mathrm{d}t^2} = \sum_{K=1}^{m} \left(\frac{\mathrm{d}c^{JK}}{\mathrm{d}t} y^K + c^{JK} \frac{\mathrm{d}y^K}{\mathrm{d}t} \right) + \frac{\mathrm{d}f^J}{\mathrm{d}t}.$$

The terms from (D.1), (D.2), and (D.3) can be substituted as follows:

$$\begin{aligned}\frac{\mathrm{d}^2 y^J}{\mathrm{d}t^2} &= \sum_{K=1}^{m}\left[\sum_{L=1}^{m}\frac{\partial c^{JK}}{\partial y^L}\left(\sum_{M=1}^{m} c^{LM}y^M + f^L\right)y^K + c^{JK}\left(\sum_{L=1}^{m} c^{KL}y^L + f^K\right)\right] \\ &\quad + \sum_{K=1}^{m}\frac{\partial f^J}{\partial y^K}\left(\sum_{L=1}^{m} c^{KL}y^L + f^K\right) \\ &= \sum_{K,L,M}\frac{\partial c^{JK}}{\partial y^L}c^{LM}y^M y^K + \sum_{K,L}\frac{\partial c^{JK}}{\partial y^L}f^L y^K + \sum_{K,L} c^{JK}c^{KL}y^L + \sum_{K} c^{JK}f^K \\ &\quad + \sum_{K,L}\frac{\partial f^J}{\partial y^K}c^{KL}y^L + \sum_{K}\frac{\partial f^J}{\partial y^K}f^K.\end{aligned} \tag{D.4}$$

D.2 Derivatives of the Numerical Solution

The additive exponential RK method is written

$$\begin{aligned}\mathbf{y}_i &= \mathbf{\Phi}_i\mathbf{y}_n + \Delta t\sum_{j=1}^{s} a_{ij}\mathbf{\Phi}_i\mathbf{\Psi}_j\mathbf{f}(\mathbf{y}_j), \\ \mathbf{\Phi}_i &= \exp\left(\Delta t\sum_{k=1}^{s}\alpha_{ik}\mathbf{C}(\mathbf{y}_k)\right), \\ \mathbf{\Psi}_i &= \exp\left(-\Delta t\sum_{k=1}^{s}\alpha_{ik}\mathbf{C}(\mathbf{y}_k)\right).\end{aligned}$$

Using tensor notation, the method is written

$$y_i^J = \sum_{K=1}^{m}\varphi_i^{JK}y_n^K + \Delta t\sum_{j=1}^{s} a_{ij}\sum_{K=1}^{m}\sum_{L=1}^{m}\varphi_i^{JK}\psi_j^{KL}\xi_j^L.$$

using the substitution $\xi_j^L = f^L(y_j^1, \ldots, y_j^m)$.

D.2.1 First Derivatives

The first step is to calculate first derivatives. Future computations require derivatives of $\mathbf{\Phi}_i$ and $\mathbf{\Psi}_i$ evaluated at $\Delta t = 0$. Their first derivatives are written as

$$\left.\frac{\mathrm{d}\mathbf{\Phi}_i}{\mathrm{d}\Delta t}\right|_{\Delta t=0} = \mathbf{\Phi}_i\sum_{k=1}^{s}\alpha_{ik}\mathbf{C}(\mathbf{y}_k)\Big|_{\Delta t=0}, \qquad \left.\frac{\mathrm{d}\mathbf{\Psi}_i}{\mathrm{d}\Delta t}\right|_{\Delta t=0} = -\mathbf{\Psi}_i\sum_{k=1}^{s}\alpha_{ik}\mathbf{C}(\mathbf{y}_k)\Big|_{\Delta t=0},$$

or equivalently in tensor notation as

$$\left.\frac{\mathrm{d}\varphi_i^{JK}}{\mathrm{d}\Delta t}\right|_{\Delta t=0} = \sum_{k=1}^{s} \alpha_{ik} \sum_{L=1}^{m} \varphi_i^{JL} \omega_k^{LK},$$
$$\left.\frac{\mathrm{d}\psi_i^{JK}}{\mathrm{d}\Delta t}\right|_{\Delta t=0} = -\sum_{k=1}^{s} \alpha_{ik} \sum_{L=1}^{m} \psi_i^{JL} \omega_k^{LK},$$

using the substitution $\omega_k^{LK} = C^{LK}(y_k^1, \ldots, y_k^m)$. Using this information, the first derivative of y_i^J is now calculated

$$\left.\frac{\mathrm{d}y_i^J}{\mathrm{d}\Delta t}\right|_{\Delta t=0} = \sum_{K=1}^{m} \left.\frac{\mathrm{d}\varphi_i^{JK}}{\mathrm{d}\Delta t} y_n^K\right|_{\Delta t=0} + \sum_{j=1}^{s} a_{ij} f^J = \sum_{j=1}^{s} \alpha_{ij} \sum_{K=1}^{m} c^{JK} y_n^K + \sum_{j=1}^{s} a_{ij} f^J. \tag{D.5}$$

D.2.2 Second Derivatives

Calculations require the second derivative of $\mathbf{\Phi}$, which is written

$$\left.\frac{\mathrm{d}^2\mathbf{\Phi}_i}{\mathrm{d}\Delta t^2}\right|_{\Delta t=0} = \mathbf{\Phi}_i \left.\left(\sum_j \alpha_{ij} \mathbf{C}(\mathbf{y}_j)\right)^2\right|_{\Delta t=0} + 2\mathbf{\Phi}_i \left.\frac{\mathrm{d}}{\mathrm{d}\Delta t}\left(\sum_j^s \alpha_{ij} \mathbf{C}(\mathbf{y}_j)\right)\right|_{\Delta t=0}.$$

It can now be expanded in tensor notation as

$$\begin{aligned}
\left.\frac{\mathrm{d}^2\varphi_i^{JK}}{\mathrm{d}\Delta t^2}\right|_{\Delta t=0} &= \left(\sum_j \alpha_{ij}\right)^2 \sum_L c^{JL} c^{LK} + 2\sum_j \alpha_{ij} \sum_L \frac{\partial c^{JK}}{\partial y^L} \left(\sum_k \alpha_{jk} \sum_M c^{LM} y_n^M + \sum_k a_{jk} f^L\right) \\
&= \left(\sum_k \alpha_{ik}\right)^2 \sum_L c^{JL} c^{LK} + 2\sum_{j,k} \alpha_{ij}\alpha_{jk} \sum_{L,M} \frac{\partial c^{JK}}{\partial y^L} c^{LM} y_n^M \\
&\quad + 2\sum_{j,k} \alpha_{ij} a_{jk} \sum_{K,L} \frac{\partial c^{JK}}{\partial y^L} f^L.
\end{aligned}$$

Therefore, the second derivative of y_i^J is now calculated

$$\begin{aligned}
\left.\frac{\mathrm{d}^2 y_i^J}{\mathrm{d}\Delta t^2}\right|_{\Delta t=0} &= \sum_K \left.\frac{\mathrm{d}^2\varphi_i^{JK}}{\mathrm{d}\Delta t^2} y_n^K\right|_{\Delta t=0} + 2\frac{\mathrm{d}}{\mathrm{d}\Delta t}\left(\sum_j a_{ij}\sum_{K,L}\varphi_i^{JK}\psi_j^{KL}\xi_j^L\right)\Bigg|_{\Delta t=0} \\
&= \cdots + 2\sum_j a_{ij}\sum_{K,L}\left(\frac{\mathrm{d}\varphi_i^{JK}}{\mathrm{d}\Delta t}\psi_j^{KL}\xi_j^L + \varphi_i^{JK}\frac{\mathrm{d}\psi_j^{KL}}{\mathrm{d}\Delta t}\xi_j^L + \varphi_i^{JK}\psi_j^{KL}\frac{\mathrm{d}\xi_j^L}{\mathrm{d}\Delta t}\right)\Bigg|_{\Delta t=0} \\
&= \cdots + 2\sum_j a_{ij}\left(\sum_K \left.\frac{\mathrm{d}\varphi_i^{JK}}{\mathrm{d}\Delta t}\right|_{\Delta t=0} f^K + \sum_K \left.\frac{\mathrm{d}\psi_j^{JK}}{\mathrm{d}\Delta t}\right|_{\Delta t=0} f^K + \left.\frac{\mathrm{d}\xi_j^J}{\mathrm{d}\Delta t}\right|_{\Delta t=0}\right) \\
&= \left(\sum_k \alpha_{ik}\right)^2 \sum_{K,L} c^{JL}c^{LK}y^K + 2\sum_{j,k}\alpha_{ij}\alpha_{jk}\sum_{K,L,M}\frac{\partial c^{JK}}{\partial y^L}c^{LM}y_n^M y^K \\
&\quad + 2\sum_{j,k}\alpha_{ij}a_{jk}\sum_{K,L}\frac{\partial c^{JK}}{\partial y^L}f^L y_n^K + 2\sum_{j,k}a_{ij}\alpha_{ik}\sum_K c^{JK}f^K \\
&\quad - 2\sum_{j,k}a_{ij}a_{jk}\sum_K c^{JK}f^K + 2\sum_{j,k}a_{ij}\alpha_{jk}\sum_{K,L}\frac{\partial f^J}{\partial y^K}c^{KL}y_n^L \\
&\quad + 2\sum_{j,k}a_{ij}a_{jk}\sum_K \frac{\partial f^J}{\partial y^K}f^K.
\end{aligned} \tag{D.6}$$

D.3 Order Conditions

For first derivatives, there are two terms to match in (D.1) and (D.5). Taking $b_j = a_{i+1,j}$ and $\beta_j = \alpha_{i+1,j}$, there are two order conditions for order one, which are written as

$$\sum_k \beta_j = 1, \qquad \sum_k b_j = 1.$$

For second derivatives, there are seven terms to match in (D.4) and (D.6). There are duplicates with the first order conditions, leaving four conditions for second order. which are written

$$\sum_{j,k}\beta_j\alpha_{jk} = \frac{1}{2}, \qquad \sum_{j,k} b_j\alpha_{jk} = \frac{1}{2},$$
$$\sum_{j,k}\beta_j a_{jk} = \frac{1}{2}, \qquad \sum_{j,k} b_j a_{jk} = \frac{1}{2}.$$

Appendix E

Steps versus Accuracy Plots

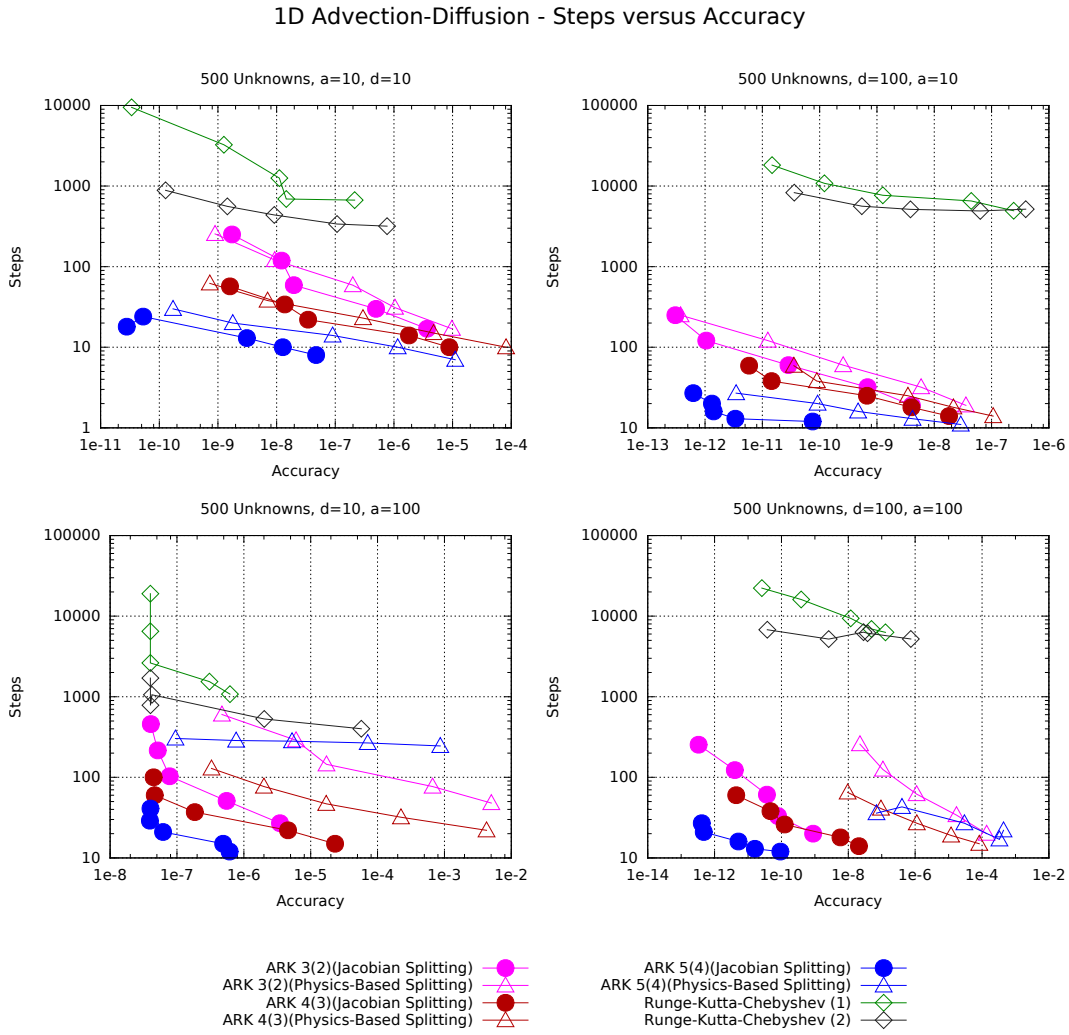

Figure E.1: The number of steps versus accuracy of IMEX and RKC methods applied to a series of one-dimensional advection-diffusion problem. Tolerances range from 10^{-4} to 10^{-8}.

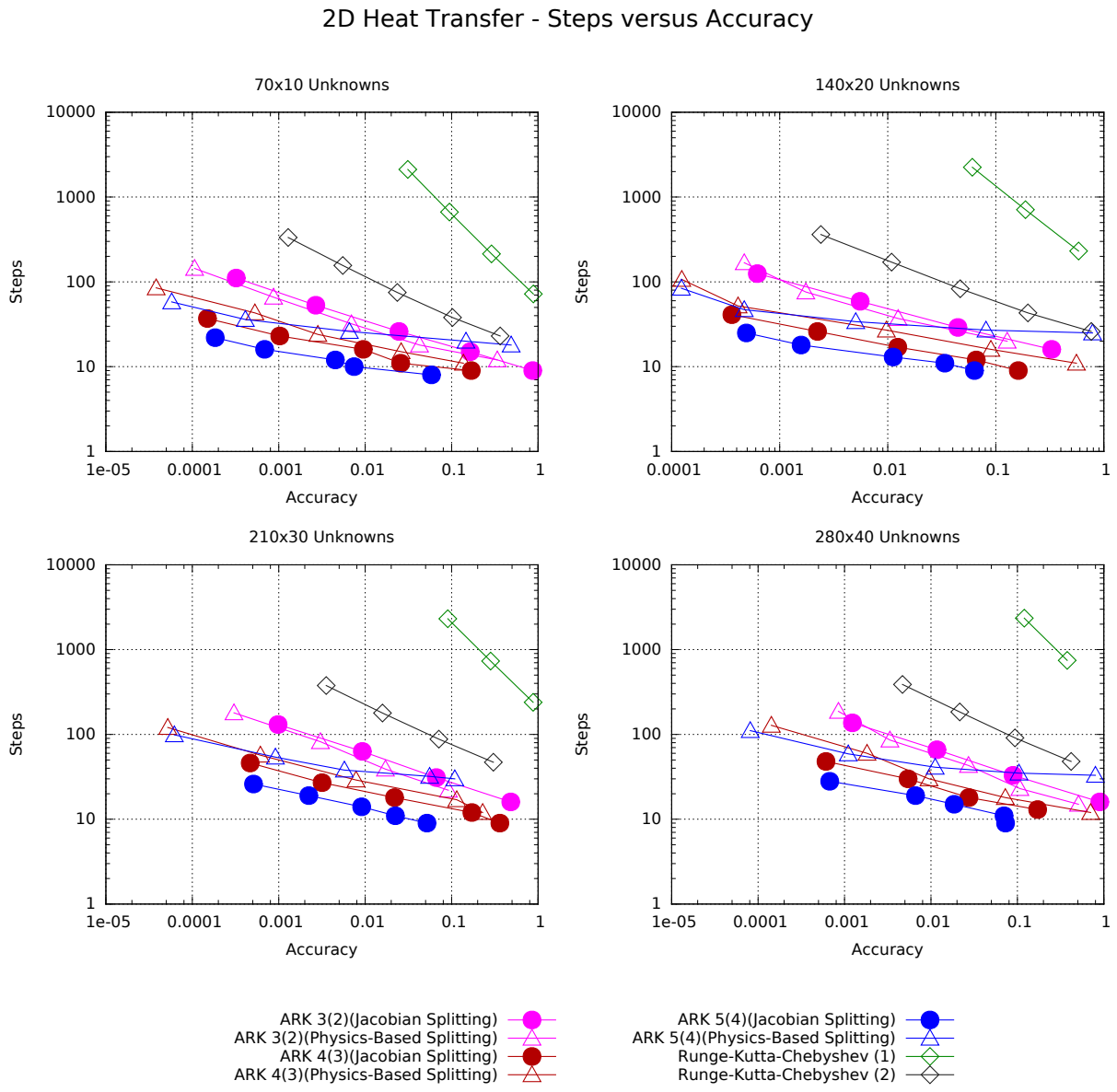

Figure E.2: The number of steps versus accuracy of IMEX and RKC methods applied to the heat transfer problem. Tolerances range from 10^{-4} to 10^{-8}.

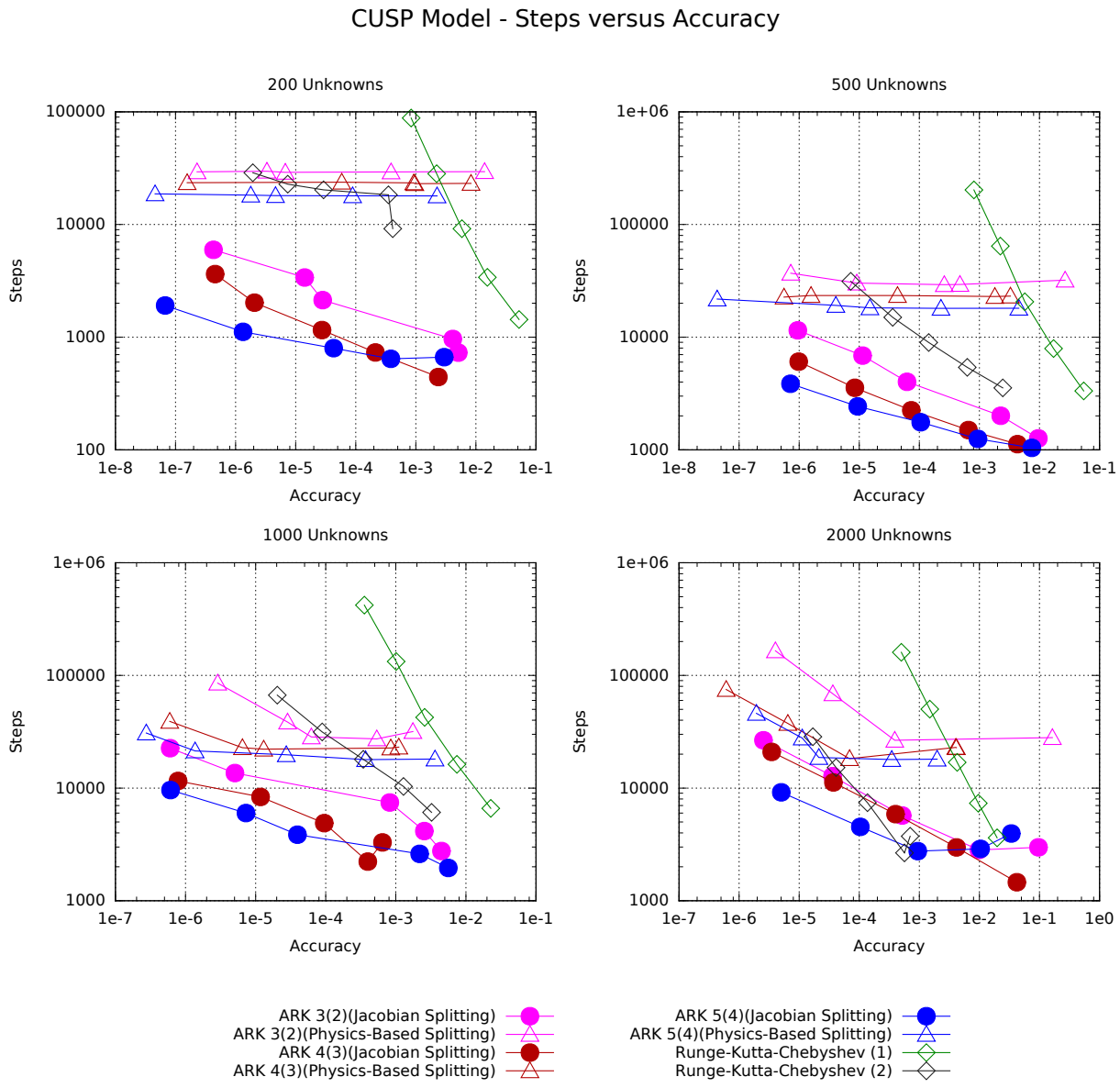

Figure E.3: The number of steps versus accuracy of IMEX and RKC methods applied to the `CUSP` problem. Tolerances range from 10^{-4} to 10^{-8}.

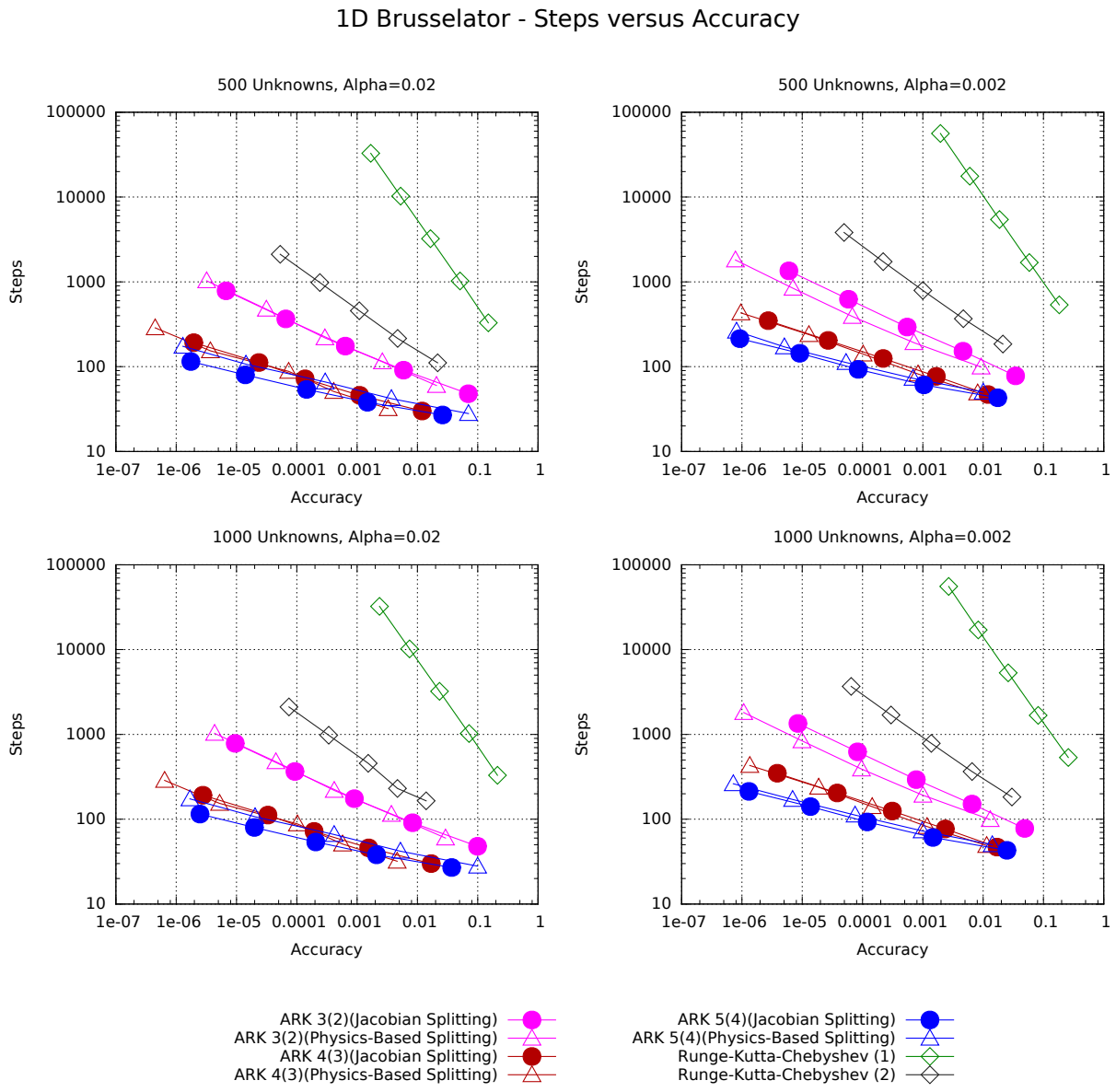

Figure E.4: The number of steps versus accuracy of IMEX and RKC methods applied to a one-dimensional Brusselator problems. Tolerances range from 10^{-4} to 10^{-8}.

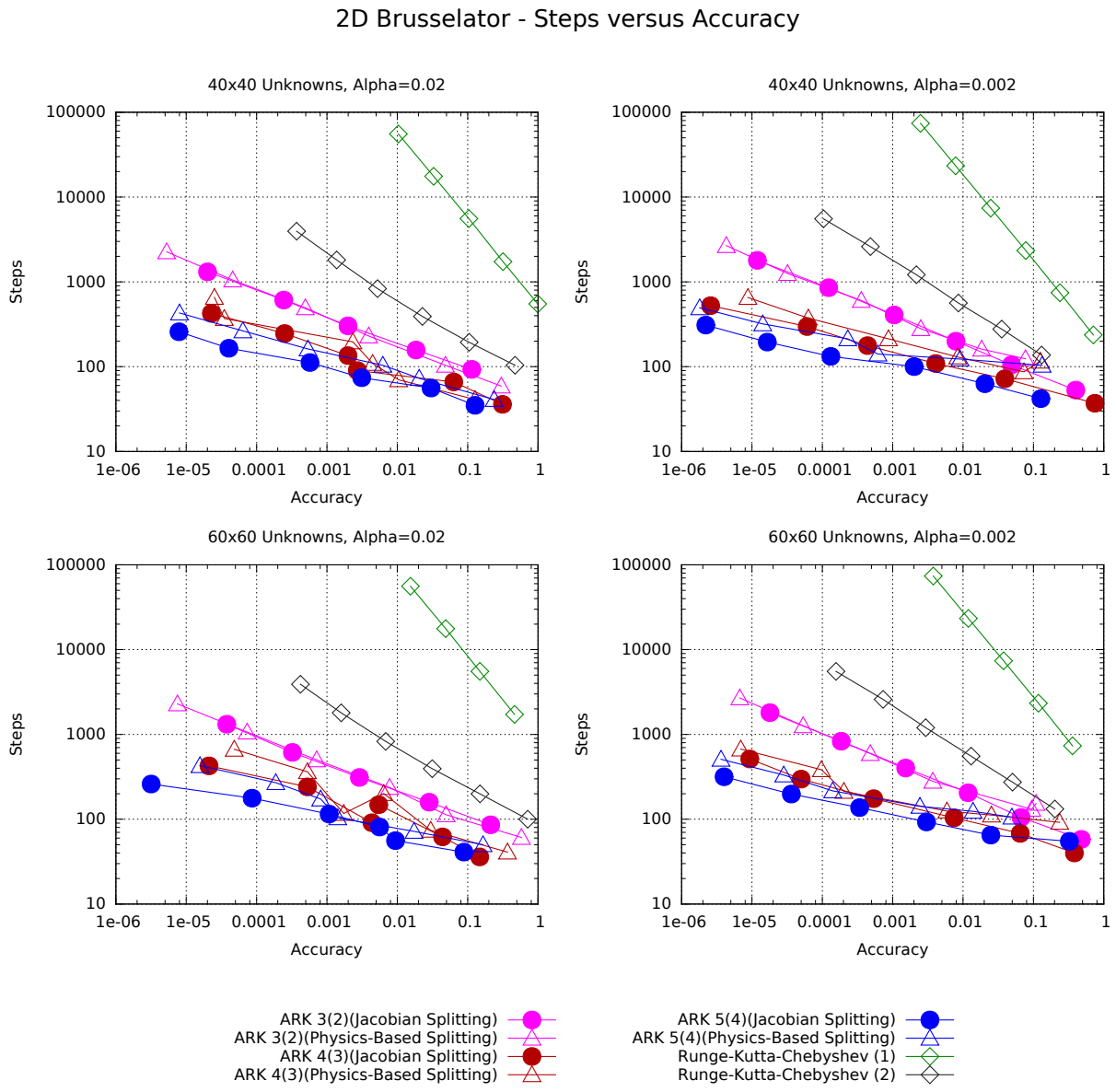

Figure E.5: The number of steps versus accuracy of IMEX and RKC methods applied to a two-dimensional Brusselator problems. Tolerances range from 10^{-4} to 10^{-8}.

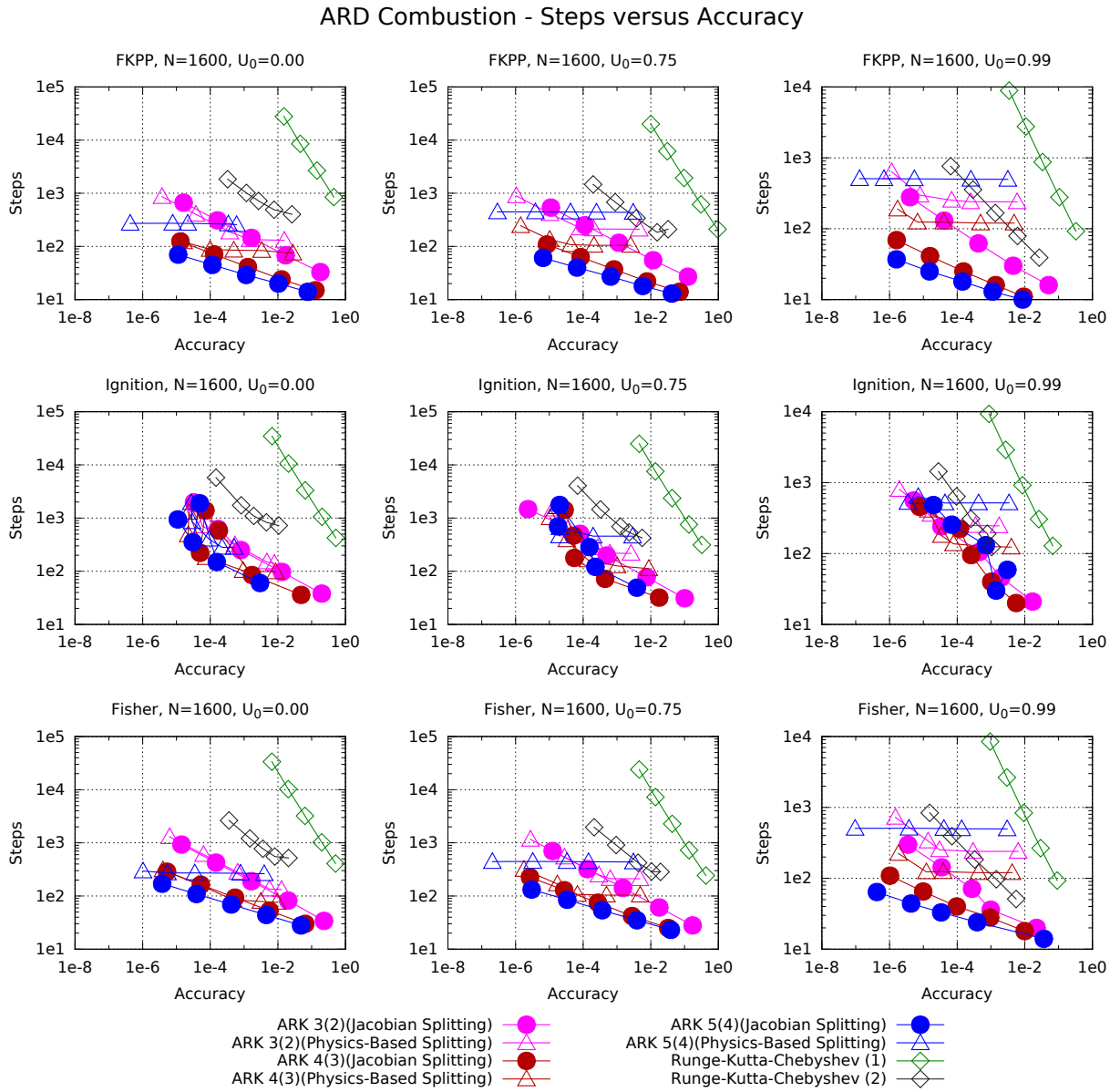

Figure E.6: The number of steps versus accuracy of IMEX and RKC methods applied to the set of one-dimensional combustion problems. Tolerances range from 10^{-4} to 10^{-8}.

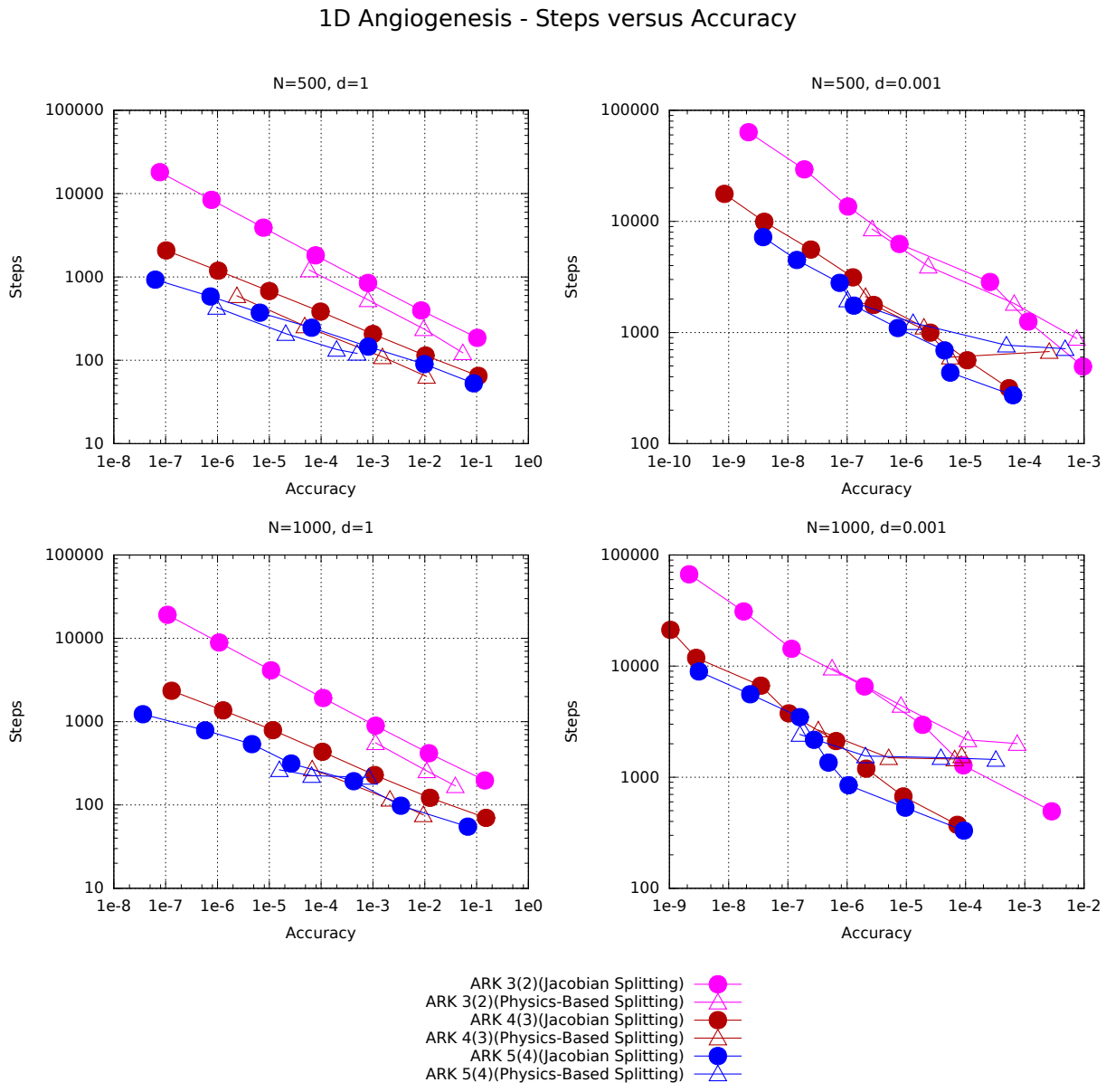

Figure E.7: The number of steps versus accuracy of IMEX and RKC methods applied to the tumour angiogenesis model. Tolerances range from 10^{-5} to 10^{-11}.

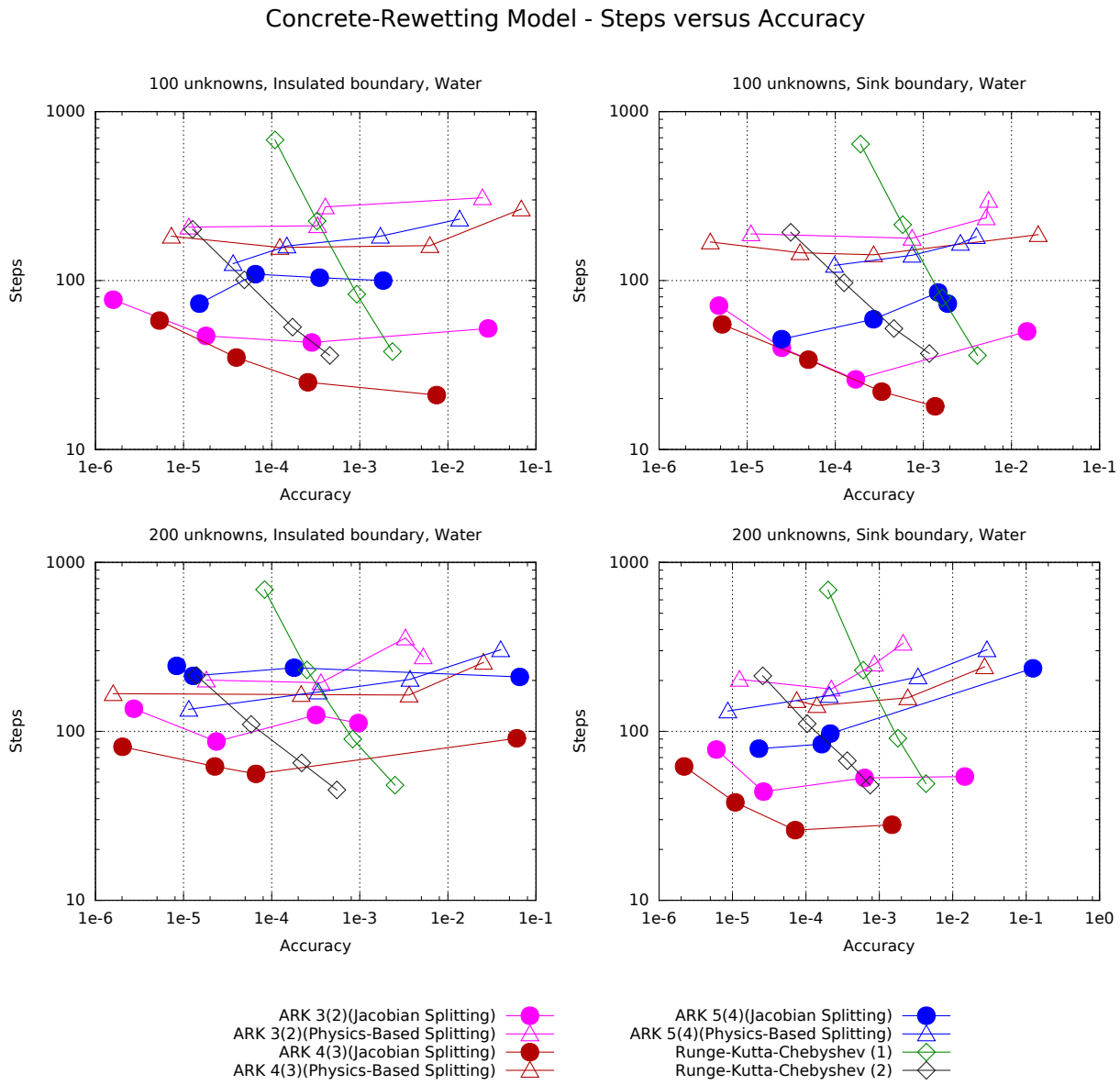

Figure E.8: The number of steps versus accuracy of IMEX and RKC methods applied to the concrete rewetting problem for both insulated and sink boundaries. Tolerances range from 10^{-4} to 10^{-9}.

Appendix F

Eigenvalue Plots

The following plots show the eigenvalues of the Jacobian for the RHSs of the IVPs studied in this thesis. Plots are shown for the Jacobian of the entire RHS and for the Jacobian of the implicit part of the physics-based splitting. These eigenvalues are evaluated at five subintervals of the solution. The distribution of the eigenvalues of the Jacobian can be useful in explaining why certain methods perform better on certain IVPs; see Section 2.3.4 for a discussion of linear stability analysis.

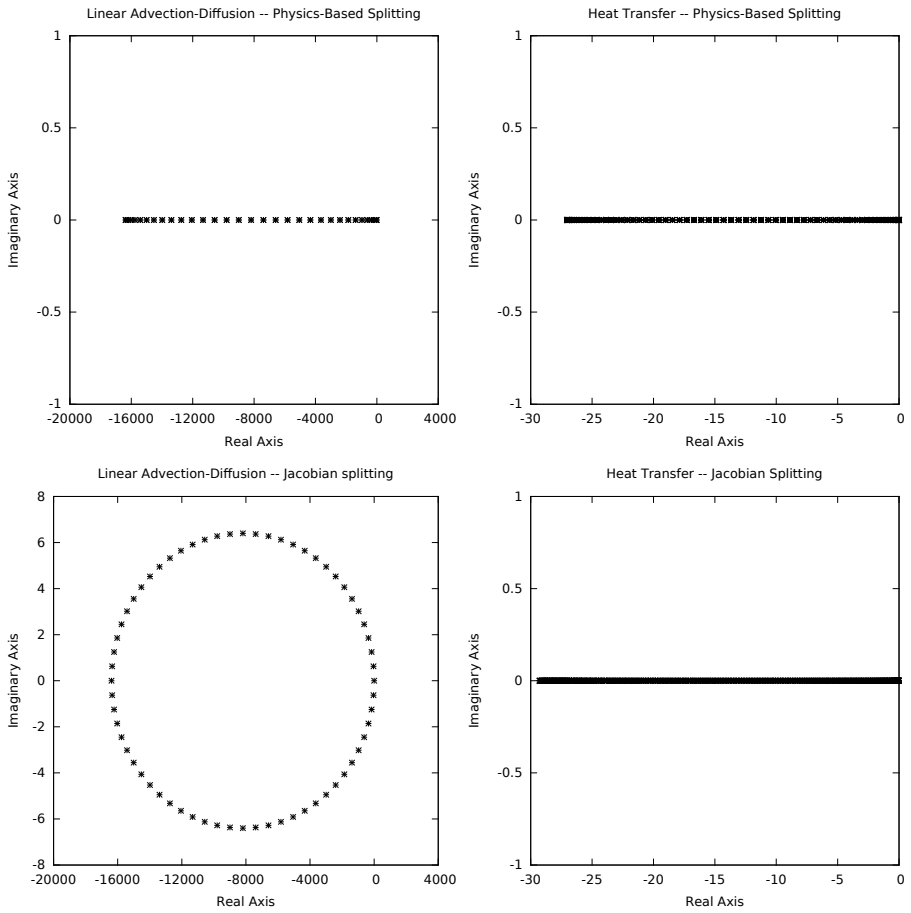

Figure F.1: Distribution of eigenvalues for the linear advection-diffusion ($d = 1$, $a = 1/10$ and 64 unknowns) and the heat transfer problem (70×10 unknowns).

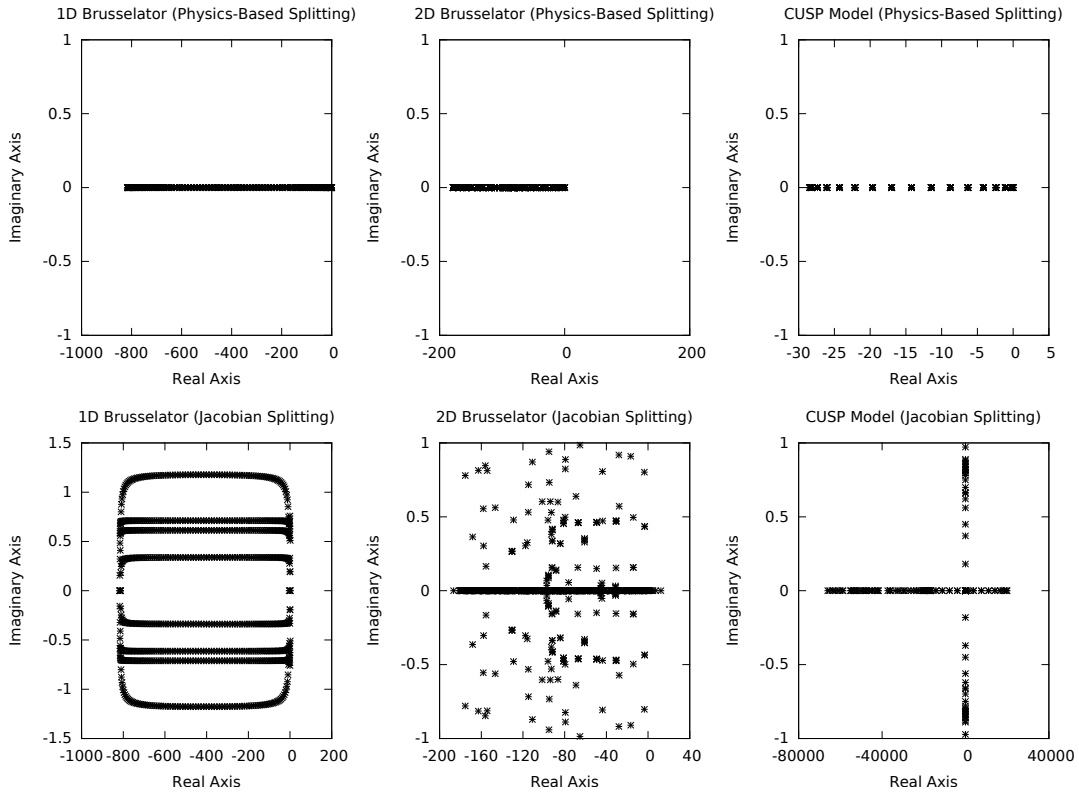

Figure F.2: Distribution of eigenvalues for the one-dimensional Brusselator ($\alpha = 1/50$ and 500 unknowns), the two-dimensional Brusselator ($\alpha = 1/50$ and 2500 unknowns), and the `CUSP` model ($\sigma = 1/144$ and 500 unknowns).

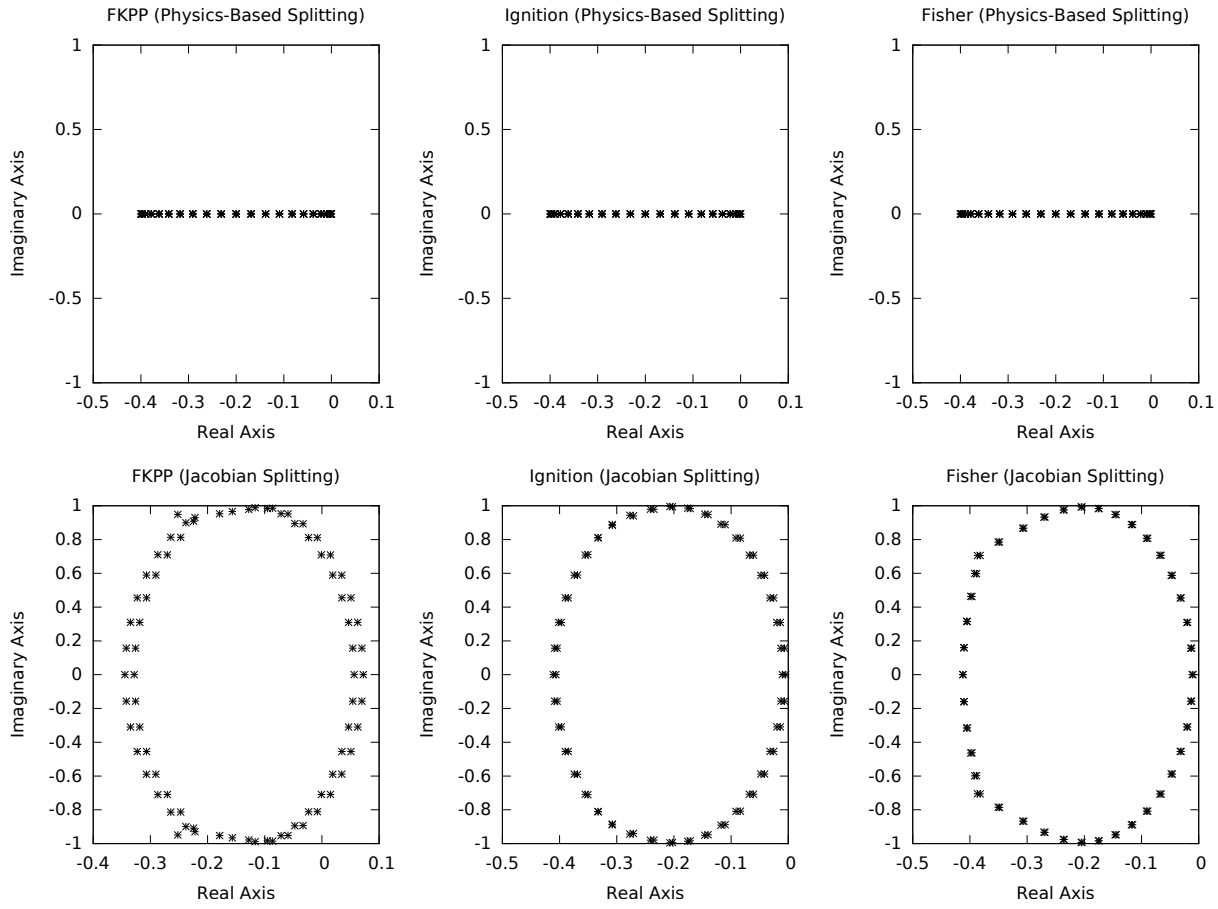

Figure F.3: Distribution of eigenvalues for the combustion model, for $U_0 = 0.99$ and 40 unknowns.

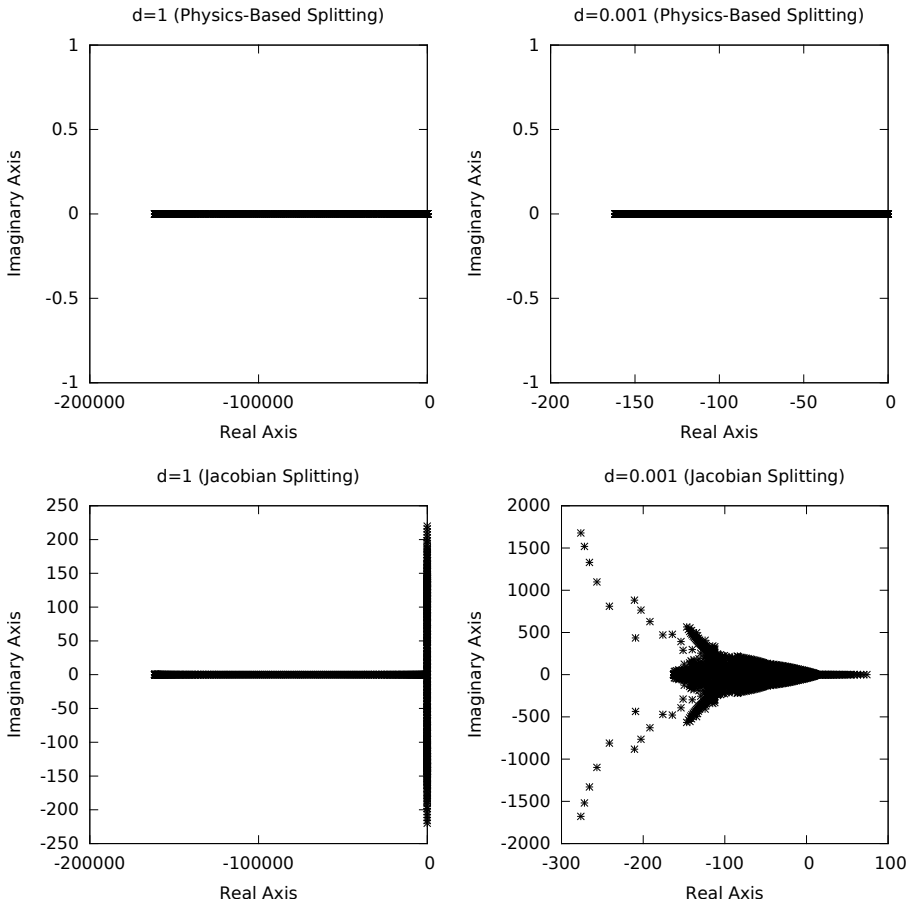

Figure F.4: Distribution of eigenvalues for the tumour angiogenesis problem for $d = 1$ and with 200 unknowns.

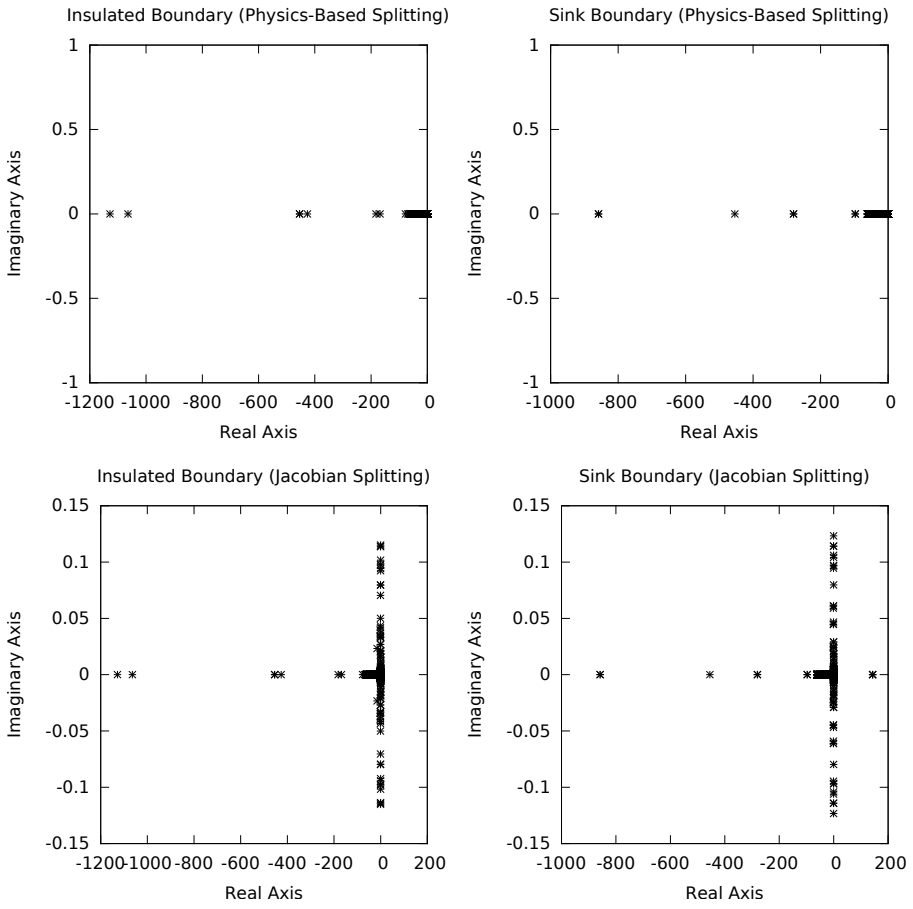

Figure F.5: Distribution of eigenvalues for the concrete-rewetting problem using a either sink or insulated boundary condition, both with 100 unknowns.