View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Saskatchewan's Research Archive

A Problem-Solving Environment for the Numerical Solution of Boundary Value Problems

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
Jason J. Boisvert

©Jason J. Boisvert, January 2011. All rights reserved.

Permission to Use

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

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

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

Abstract

Boundary value problems (BVPs) are systems of ordinary differential equations (ODEs) with boundary conditions imposed at two or more distinct points. Such problems arise within mathematical models in a wide variety of applications. Numerically solving BVPs for ODEs generally requires the use of a series of complex numerical algorithms. Fortunately, when users are required to solve a BVP, they have a variety of BVP software packages from which to choose. However, all BVP software packages currently available implement a specific set of numerical algorithms and therefore function quite differently from each other. Users must often try multiple software packages on a BVP to find the one that solves their problem most effectively. This creates two problems for users. First, they must learn how to specify the BVP for each software package. Second, because each package solves a BVP with specific numerical algorithms, it becomes difficult to determine why one BVP package outperforms another. With that in mind, this thesis offers two contributions.

First, this thesis describes the development of the BVP component to the fully featured problem-solving environment (PSE) for the numerical solution of ODEs called `pythODE`. This software allows users to select between multiple numerical algorithms to solve BVPs. As a consequence, they are able to determine the numerical algorithms that are effective at each step of the solution process. Users are also able to easily add new numerical algorithms to the PSE. The effect of adding a new algorithm can be measured by making use of an automated test suite.

Second, the BVP component of `pythODE` is used to perform two research studies. In the first study, four known global-error estimation algorithms are compared in `pythODE`. These algorithms are based on the use of Richardson extrapolation, higher-order formulas, deferred corrections, and a conditioning constant. Through numerical experimentation, the algorithms based on higher-order formulas and deferred corrections are shown to be computationally faster than Richardson extrapolation while having similar accuracy. In the second study, `pythODE` is used to solve a newly developed one-dimensional model of the agglomerate in the catalyst layer of a proton exchange membrane fuel cell.

Acknowledgements

I thank Dr. Raymond J. Spiteri for giving me the opportunity to be a part of the Numerical Simulation Laboratory. His guidance, insight, and financial support made this thesis possible. I thank Dr. Paul H. Muir for his insight into the numerical solution of boundary value problems. His many contributions toward this thesis are greatly appreciated. I thank Dr. Marc Secanell for his contributions toward the multi-scale agglomerate model for proton exchange membrane fuel cells used in this thesis. I thank Dr. Dwight Makaroff, Dr. Kevin Stanley, and Dr. Uri Ascher for their contributions toward the final form of this thesis.

I thank the members of the Numerical Simulation Laboratory for sharing with me their enthusiasm for research and for the many insightful discussions about it.

Last but not least, I thank my partner Carla Gibson for her emotional support and patience. I thank my parents Luc and Diane Boisvert. Without the endless support and encouragement of my parents, this thesis would never have been written.

Contents

List of Tables

List of Figures

List of Abbreviations

BVP	Boundary Value Problem
DAE	Differential-Algebraic Equation
IVP	Initial Value Problem
GUI	Graphical User Interface
ODE	Ordinary Differential Equation
MIRK	Mono-Implicit Runge–Kutta
NAE	Nonlinear Algebraic Equations
PEMFC	Proton Exchange Membrane Fuel Cell
PSE	Problem-Solving Environment

Chapter 1

Introduction

Boundary value problems (BVPs) for ordinary differential equations (ODEs) are used as mathematical models in a wide variety of disciplines including biology, physics, and engineering. For example, suppose one wishes to determine the deflection of a uniformly loaded beam with variable stiffness and supported at both endpoints [4]. Letting x be the length of the beam, the deflection between $1 \leq x \leq 2$ can be described by the fourth-order ODE

$$x^3 y''''(x) + 6x^2 y'''(x) + 6xy''(x) = 1, \quad 1 < x < 2, \tag{1.1a}$$

where $y(x)$ is the deflection of the beam at position x. To ensure that the deflection at both endpoints is zero, the boundary conditions

$$y(1) = y''(1) = y(2) = y''(2) = 0, \tag{1.1b}$$

are imposed.

The process of solving BVP (1.1) involves finding a function $\mathbf{y}(x)$, $1 \leq x \leq 2$, that satisfies both the system of ODEs and the boundary conditions. In general, exact solutions to BVPs are typically not known. Therefore, researchers often apply numerical methods to a BVP in order to approximate the solution. Practical implementations of numerical methods for the solution of BVPs involve the employment of a sequence of complex numerical algorithms. A BVP software package usually begins with the discretization of a system of ODEs. This process approximates the ODEs by a system of (generally) nonlinear algebraic equations (NAEs). Next, a BVP software package typically uses a form of *Newton's method* to solve the NAEs; see Section 2.8 for a description of Newton's method. This results in solution approximations at discrete points, called *mesh points*, in the problem domain. The software package must then estimate and adaptively control some measure of the error in the numerical solution. Instead of implementing these numerical algorithms themselves, most researchers rely on existing software to numerically solve a BVP.

At present, there exist numerous high-quality BVP software packages from which to choose. Some of the more popular software packages include `COLSYS` [3], `COLNEW` [7], `BVP_SOLVER` [35], and `TWPBVPC` [13]. However, a dilemma arises when deciding which BVP software package to use. All the

BVP software packages mentioned above function quite differently from each other. For example, the manner in which a user specifies the problem differs between the BVP software packages. Both `COLSYS` and `COLNEW` allow ODEs to be specified as systems of m mixed-order ODEs in the form

$$\mathbf{y}^{(\mathbf{d})}(x) = \mathbf{f}(x, \mathbf{z}(\mathbf{y}(x))), \quad a < x < b, \tag{1.2a}$$

where

$$\mathbf{y}^{(\mathbf{d})}(x) = [y_1^{(d_1)}(x), \ldots, y_m^{(d_m)}(x)], \tag{1.2b}$$

$$\mathbf{f}(x, \mathbf{z}(\mathbf{y}(x))) = [f_1(x, \mathbf{z}(\mathbf{y}(x))), \ldots, f_m(x, \mathbf{z}(\mathbf{y}(x)))], \tag{1.2c}$$

and

$$\mathbf{z}(\mathbf{y}(x)) = [y_1(x), y_1^{(1)}(x), \ldots, y_1^{(d_1-1)}(x), \ldots, y_m(x), y_m^{(1)}(x), \ldots, y_m^{(d_m-1)}(x)], \tag{1.2d}$$

along with appropriate boundary conditions. In many cases, a user must re-formulate the system of ODEs so that it is consistent with (1.2). This often requires some algebraic manipulation. Using (1.1) as an example, let

$$\mathbf{z}(\mathbf{y}(x)) = \begin{pmatrix} y_1(x) \\ y_1'(x) \\ y_1''(x) \\ y_1'''(x) \end{pmatrix} = \begin{pmatrix} y(x) \\ y'(x) \\ y''(x) \\ y'''(x) \end{pmatrix}.$$

The fourth-order ODE can then be specified as

$$y_1'''' = f_1(x, \mathbf{z}(\mathbf{y}(x))) = \frac{1 - 6x^2 y_1'''(x) - 6x y_1''(x)}{x^3}, \quad 1 < x < 2.$$

In the case of other BVP software packages, such as `BVP_SOLVER` and `TWPBVPC`, the ODEs must be specified as a system of m first-order ODEs

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)), \quad a < x < b.$$

A system of mixed-order ODEs can be converted to a system of first-order ODEs. Using (1.1) as an example, let

$$\mathbf{y}(x) = \begin{pmatrix} y_1(x) \\ y_2(x) \\ y_3(x) \\ y_4(x) \end{pmatrix} = \begin{pmatrix} y(x) \\ y'(x) \\ y''(x) \\ y'''(x) \end{pmatrix}.$$

Next, the ODEs can be specified as a system of four first-order ODEs

$$y_1'(x) = f_1(x, \mathbf{y}(x)) = y_2(x), \tag{1.3}$$

$$y_2'(x) = f_2(x, \mathbf{y}(x)) = y_3(x), \tag{1.4}$$

$$y_3'(x) = f_3(x, \mathbf{y}(x)) = y_4(x), \tag{1.5}$$

$$y_4'(x) = f_4(x, \mathbf{y}(x)) = \frac{1 - 6x^2 y_4(x) - 6x y_3(x)}{x^3}. \tag{1.6}$$

As well as having different problem specifications, all four packages mentioned use different numerical algorithms to discretize and compute a numerical solution to the BVP. Both `COLSYS` and `COLNEW` use a spline-collocation algorithm to return a piecewise polynomial as a solution; see Section 2.6.2. However, the bases used to determine the piecewise polynomial are different. In particular, both `BVP_SOLVER` and `TWPBVPC` use an algorithm based on mono-implicit Runge–Kutta (MIRK) formulas to generate a discrete solution; see Section 2.6.1. However, `BVP_SOLVER` makes the discrete solution a basis for a continuous solution. In contrast, `TWPBVPC` returns only a discrete solution, and it couples the MIRK formulas with the use of deferred corrections; see Section 3.1.

Also, all the software packages mentioned above have different error-control strategies. An error-control strategy generally involves choosing the mesh points in the interval $a \leq x \leq b$ to compute a numerical solution so that the norm of an estimate of the error is less than a user-supplied tolerance. For example, both `COLSYS` and `COLNEW` choose the mesh points in order to minimize an estimate of the amount by which numerical solution differs from the exact solution, whereas `BVP_SOLVER` chooses the mesh points in order to minimize an estimate of the amount by which the numerical solution fails to satisfy the BVP. In contrast, `TWPBVPC` uses both an error estimate and a measure of the conditioning of a BVP to choose the mesh points. The error-control strategy of each code is described in Chapter 3.

Because each BVP software package functions so differently, users must often try multiple software packages on a BVP to find the one that solves their problem most effectively. Also, users may wish to use a BVP software package to verify a numerical solution obtained from another BVP software package. Unfortunately, problems arise when attempting to use multiple BVP packages to solve the same problem.

Using multiple BVP software packages requires the problem to be re-defined according to the specification of each package. As a consequence, users must write new code for each package they choose. This can prove to be a time-consuming task that requires users to learn the usage of each software package.

When using a BVP software package as a numerical research tool, a different problems arises. Each of the BVP software packages introduced above is built with the intention of using a single approach to numerically solve a BVP. In other words, each package forces users to adopt the same

discretization algorithm, NAEs solution algorithm, and error-control algorithm on every BVP they choose to solve. Therefore, if a user wishes to add new numerical algorithms for the purpose of comparing different algorithms, often a considerable amount of existing code must be modified.

Ideally, a BVP software package should allow users to add additional numerical algorithms without requiring them to modify a significant portion of existing code. Moreover, a BVP software package should also allow users to select among different numerical algorithms to solve BVPs. At present, there exist no known BVP software packages that have these features.

With that in mind, this thesis presents contributions in two forms:

1. This thesis describes the development of a BVP component to the problem-solving environment (PSE) called `pythODE`. This PSE offers users many features not presently available in other BVP software packages. For example, users can directly specify many of the steps used to numerically solve a BVP. If a numerical algorithm is not already present in the PSE, users can easily add the algorithm without significant modification of existing code. Easy extension is made possible by the use of well-known object-oriented design principles [42]. Once an algorithm is added, its performance can be easily compared against other similar algorithms by means of an automated test suite.

2. The BVP component of `pythODE` is used to make two research contributions. First, `pythODE` is used to compare the performance of four global-error estimation algorithms within a defect-control BVP solver. These algorithms are based on the use of Richardson extrapolation, higher-order formulas, deferred corrections, and a conditioning constant. Richardson extrapolation is a widely used global-error estimation algorithm. Despite the Richardson extrapolation algorithm having similar accuracy to both higher-order and deferred-correction algorithms, the study shows that the algorithms based on higher-order formulas and deferred corrections are computationally faster than Richardson extrapolation. Second, `pythODE` is used to solve a newly developed one-dimensional model of the agglomerate in the catalyst layer of a proton exchange membrane fuel cell (PEMFC). The method used to solve the model will be integrated in the two-dimensional PEMFC simulator `FCST` [30]. By solving this particular problem, the usefulness of `pythODE` for solving real-world problems is demonstrated.

1.1 Structure of thesis

The remainder of the thesis is divided into the following chapters. The theory behind the numerical solution of BVPs is described in Chapter 2. Existing BVP software packages and the newly developed PSE `pythODE` are described in Chapter 3. The research contributions are described in Chapter 4. Finally, conclusions and future work are described in Chapter 5.

Chapter 2

Numerical Methods For BVPs

This chapter offers a brief introduction to numerical methods for BVPs. Many of the concepts introduced in this chapter are used throughout the thesis.

The chapter begins by describing the types of BVPs that are well-suited for numerical methods. Section 2.1 describes well-known existence and uniqueness theorems for both linear and nonlinear BVPs. Section 2.2 describes the concept of conditioning for BVPs. The remainder of this chapter introduces various numerical methods used to approximate solutions to BVPs. The methods used to solve the BVPs fall into two categories: initial value methods and global methods. Section 2.3 introduces initial value methods. Section 2.4 introduces global methods. Section 2.5 discusses one-step methods that can be applied within the global-method framework. The final two sections introduce two important numerical methods used in implementations for BVP software packages. Section 2.7 introduces a common strategy used for mesh selection. Section 2.8 introduces Newton's method to solve systems of NAEs.

Throughout this chapter, first-order linear or nonlinear two-point BVPs are typically considered. A linear two-point BVP consists of a system of first-order linear ODEs

$$\mathbf{y}'(x) = \mathbf{A}(x)\mathbf{y}(x) + \mathbf{q}(x), \quad a < x < b, \tag{2.1a}$$

where $\mathbf{A} : \mathbb{R} \to \mathbb{R}^{m\times m}$ and $\mathbf{q} : \mathbb{R} \to \mathbb{R}^m$, accompanied by a system of m linear two-point boundary conditions

$$\mathbf{B}_a\mathbf{y}(a) + \mathbf{B}_b\mathbf{y}(b) = \boldsymbol{\beta}, \tag{2.1b}$$

where $\mathbf{B}_a, \mathbf{B}_b$ are constant $m \times m$ matrices and $\boldsymbol{\beta} \in \mathbb{R}^m$. A nonlinear BVP consists of a system of first-order nonlinear ODEs

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)), \quad a < x < b, \tag{2.2a}$$

where $\mathbf{f} : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$, accompanied by a system m nonlinear two-point boundary conditions

$$\mathbf{g}(\mathbf{y}(a), \mathbf{y}(b)) = \mathbf{0}, \tag{2.2b}$$

where $\mathbf{g} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m$.

2.1 Existence and uniqueness of BVP solutions

A discussion of the numerical solution of BVPs typically begins with a close look at the existence and uniqueness of exact solutions.

Compared to BVPs, the concepts of existence and uniqueness are much better understood for solutions to *initial value problems* (IVPs), i.e., a system of ODEs (2.1a) or (2.2a) subject to the initial condition

$$\mathbf{y}(a) = \mathbf{y}_a,$$

where $\boldsymbol{y}_a \in \mathbb{R}^m$. As a consequence, it is reasonable to attempt to extend the understanding of IVPs to the domain of BVPs. Therefore, an existence and uniqueness theorem for IVPs [4, Section 3.1] is first introduced. This theorem, as well as many other theorems presented in this chapter, assume that the ODE (2.2a) is *Lipschitz continuous* with respect to $\mathbf{y}$; i.e., there exists a constant L such that for all $(x, \mathbf{y})$ and $(x, \mathbf{z})$ in a given domain $\mathbf{D}$ such that

$$\|\mathbf{f}(x, \mathbf{y}) - \mathbf{f}(x, \mathbf{z})\| \leq L\|\mathbf{y} - \mathbf{z}\|,$$

for some norm $\|.\|$.

Theorem 2.1.1. *Suppose that $\boldsymbol{f}(x, \boldsymbol{y})$ is continuous on $\boldsymbol{D} = \{(x, \boldsymbol{y}) : \ a \leq x \leq b, \|\boldsymbol{y} - \boldsymbol{y}_a\| \leq \rho\}$ for some $\rho > 0$, and suppose that $\boldsymbol{f}(x, \boldsymbol{y})$ is Lipschitz continuous with respect to $\boldsymbol{y}$. If $\|\boldsymbol{f}(x, \boldsymbol{y})\| \leq M$ on $\boldsymbol{D}$ and $c = \min(b - a, \rho/M)$, then the IVP has a unique solution for $a \leq x \leq a + c$.*

Using Theorem 2.1.1, it may be possible to determine a value of $\mathbf{y}_a$ such that a solution to an IVP exists for $a \leq x \leq b$ that also satisfies the boundary conditions of a BVP with (2.1a) or (2.2a) as the ODE. However, a few issues may arise. For example, it may be possible to have an ODE for which the IVP has a solution, but the BVP does not. Also, there may be more than one IVP that satisfies the boundary conditions.

Consider the linear BVP [8]

$$y''(x) + y(x) = 0, \quad 0 < x < b, \tag{2.3a}$$

with boundary conditions

$$y(0) = 0, \tag{2.3b}$$

$$y(b) = B, \tag{2.3c}$$

for arbitrary $b, B \in \mathbb{R}$.

For $b \neq n\pi$, where n is an integer, a unique value of C can be found, and therefore a unique

solution

$$y(x) = C \sin x,$$

exists. The constant C is chosen to satisfy the right boundary condition

$$C \sin b = B. \tag{2.4}$$

For $b = n\pi$ and $B = 0$, any value of C provides a solution. As a consequence, infinitely many solutions exist. Finally, for $b = n\pi$ and $B \neq 0$, no solutions exist.

Even though a solution cannot be found that satisfies the boundary conditions for $b = n\pi$ and $B \neq 0$, an IVP that satisfies the ODEs can still be found. For example, let the problem have the initial conditions

$$y(0) = 0,$$

$$y'(0) = y'_a.$$

The solution (2.1) satisfies the ODEs with $C = y'_a$ satisfying the initial conditions. Therefore, a solution to the IVP exists for every y'_a. The problem, however, is that a solution for the IVP cannot be extended to the case where the boundary condition is $b = n\pi$ and $B \neq 0$.

The possibility of an infinite number of solutions suggests that the uniqueness part of Theorem 2.1.1 does not carry over to BVPs. The example (2.3) also shows that question of existence for BVPs is not simple. Therefore, it is worthwhile to take a closer look at the uniqueness of solutions for both linear and nonlinear BVPs.

Starting with linear BVPs (2.1), the ODE (2.1a) has a known general solution

$$\mathbf{y}(x) = \mathbf{Y}(x)\mathbf{s} + \mathbf{y}_p(x), \tag{2.6}$$

where $\mathbf{Y}(x) \in \mathbb{R}^{m \times m}$ is the fundamental solution to (2.1a) such that

$$\mathbf{Y}(a) = \mathbf{I},$$

and $\mathbf{y}_p(x) \in \mathbb{R}^m$ is the particular solution defined as

$$\mathbf{y}_p(x) = \mathbf{Y}(x) \int_a^x \mathbf{Y}^{-1}(t)\mathbf{q}(t)dt.$$

The constant, $\mathbf{s} \in \mathbb{R}^m$, must be chosen so that (2.6) satisfies the boundary conditions (2.1b) [4, Section 3.1.2]. In other words, $\mathbf{s}$ is a constant such that

$$\mathbf{B}_a\mathbf{y}(a) + \mathbf{B}_b\mathbf{y}(b) = \mathbf{B}_a[\mathbf{Y}(a)\mathbf{s} + \mathbf{y}_p(a)] + \mathbf{B}_b[\mathbf{Y}(b)\mathbf{s} + \mathbf{y}_p(b)] = \boldsymbol{\beta}.$$

Solving for $\mathbf{s}$ results in

$$\mathbf{s} = \mathbf{Q}^{-1}\left(\boldsymbol{\beta} - \mathbf{B}_b\mathbf{Y}(b)\int_a^b \mathbf{Y}^{-1}(t)\mathbf{q}(t)dt\right), \tag{2.7}$$

where

$$\mathbf{Q} = \mathbf{B}_a\mathbf{I} + \mathbf{B}_b\mathbf{Y}(b). \tag{2.8}$$

Satisfying the boundary conditions depends on determining a value for $\mathbf{s}$. Assuming that a solution for (2.1a) can be found, then a solution for $\mathbf{s}$ strictly depends on the invertibility of $\mathbf{Q}$. The following uniqueness theorem then holds.

Theorem 2.1.2. *Suppose $\boldsymbol{A}(x)$ and $\boldsymbol{q}(x)$ in the linear differential equations (2.1a) are continuous. Then the BVP (2.1) has a unique solution if and only if the matrix $\boldsymbol{Q}$ is non-singular.*

Unfortunately, solutions for first-order nonlinear BVPs (2.2) do not have a nicely defined general solution such as (2.6). However, some conclusions about the uniqueness of a solution for nonlinear BVPs can still be made.

Suppose there exists an IVP

$$\mathbf{w}' = \mathbf{f}(x, \mathbf{w}), \quad x > a,$$

with the initial condition,

$$\mathbf{w}(a) = \mathbf{s},$$

that satisfies the nonlinear boundary conditions (2.2b),

$$\mathbf{G}(\mathbf{s}) = \mathbf{g}(\mathbf{s}, \mathbf{w}(b; \mathbf{s})) = \mathbf{0}. \tag{2.10}$$

The system of equations in (2.10) consists of m NAEs for m boundary conditions. As is the case for systems of NAEs, there may be no solution, one solution, or many solutions. Therefore, the number of solutions to (2.10) is consistent with the number of solutions to the nonlinear BVP. The following theorem is a consequence.

Theorem 2.1.3. *Suppose that $\boldsymbol{f}(x, \boldsymbol{y})$ is continuous on $\boldsymbol{D} = \{(x, \boldsymbol{y}) : a \leq x \leq b, \|\boldsymbol{y}\| < \infty\}$ and satisfies a uniform Lipschitz condition in $\boldsymbol{y}$. Then the BVP (2.2) has as many solutions as there are distinct roots $\boldsymbol{s}^*$ of (2.10). For each $\boldsymbol{s}^*$, a solution of the BVP is given by*

$$\boldsymbol{y}(x) = \boldsymbol{w}(x; \boldsymbol{s}^*).$$

Theorem 2.1.3 states that uniqueness of a solution cannot be guaranteed for a given nonlinear BVP. However, it turns out that a strict uniqueness condition does not prevent the use of numerical methods from finding solutions to BVPs. In fact, the solutions to BVPs need only be *locally unique*.

Geometrically, local uniqueness can be seen as meaning that there exists a region around a solution $\mathbf{y}(x)$ to a BVP such that no other solution exists in that region [4]. For a solution $\mathbf{y}(x)$ to a BVP, local uniqueness means that there exists a $\rho > 0$ such that

$$\mathbf{D} = \{\mathbf{z} : \mathbf{z} \in \mathbf{C}[a,b], \sup_{a \le x \le b} \|\mathbf{z}(x) - \mathbf{y}(x)\| \le \rho\}, \tag{2.11}$$

and $\mathbf{y}$ is the only member of $\mathbf{D}$ that is also a solution to the BVP. In (2.11), $\mathbf{z} \in \mathbf{C}[a,b]$ denotes that $\mathbf{z}(x)$ is continuous throughout $[a,b]$.

Local uniqueness of a solution $\mathbf{y}(x)$ for a nonlinear BVP can be demonstrated by considering another solution $\hat{\mathbf{y}}(x)$ that also satisfies the system of nonlinear ODEs (2.2a) [4, Section 3.3.4], i.e.,

$$\hat{\mathbf{y}}' = \mathbf{f}(x, \hat{\mathbf{y}}), \quad a < x, \tag{2.12}$$

such that $\hat{\mathbf{y}}(a) = \mathbf{y}_a + \epsilon$, where ϵ is small. Using a Taylor series, $\mathbf{f}(x, \hat{\mathbf{y}})$ can be expanded about $\mathbf{y}$ to get

$$\mathbf{f}(x, \hat{\mathbf{y}}) = \mathbf{f}(x, \mathbf{y}) + \mathbf{A}(x; \mathbf{y})(\hat{\mathbf{y}} - \mathbf{y}) + O(\epsilon^2),$$

where $\mathbf{A}(x; \mathbf{y}) = \frac{\partial \mathbf{f}}{\partial \mathbf{y}}$ is the Jacobian matrix associated with the nonlinear ODE (2.12) evaluated at $(x; \mathbf{y})$ and $\epsilon = \|\hat{\mathbf{y}} - \mathbf{y}\|$. Ignoring the term $O(\epsilon^2)$, the system of nonlinear ODEs

$$\mathbf{z}' = \mathbf{A}(x; \mathbf{y})\mathbf{z}, \tag{2.13a}$$

where

$$\mathbf{z} = \hat{\mathbf{y}}(x) - \mathbf{y}(x),$$

can be defined.

Applying a similar treatment to the nonlinear boundary conditions (2.2b) results in

$$\mathbf{B}_a \mathbf{z}(a) + \mathbf{B}_b \mathbf{z}(b) = \mathbf{0}, \tag{2.13b}$$

where

$$\mathbf{B}_a = \frac{\partial \mathbf{g}(\mathbf{y}(a), \mathbf{y}(b))}{\partial \mathbf{y}(a)}, \quad \mathbf{B}_b = \frac{\partial \mathbf{g}(\mathbf{y}(a), \mathbf{y}(b))}{\partial \mathbf{y}(b)}.$$

The BVP (2.13) is referred to as the *variational problem*. If a unique solution $\mathbf{z}(x) \equiv \mathbf{0}$ exists for the variational problem, then the solution $\mathbf{y}(x)$ is said to be isolated. It can be shown that isolated solutions to the variational problem imply local uniqueness [4].

This section concludes with an example of a more explicit existence and uniqueness theorem for

second-order nonlinear BVPs of the form

$$y'' = f(x, y, y') \quad a < x < b, \tag{2.14a}$$

$$y(a) = B_a, \quad y(b) = B_b. \tag{2.14b}$$

Theorem 2.1.4. *Suppose that $f(x, y, y')$ is continuous on $\boldsymbol{D} = \{(x, y, y') : a \leq x \leq b,\ -\infty < y < \infty,\ -\infty < y' < \infty\}$ and satisfies a Lipschitz condition on $\boldsymbol{D}$ with respect to y and y', so that there exist constants, L, M, such that for any (x, y, y') and $(x, \hat{y}, \hat{y}')$ in $\boldsymbol{D}$,*

$$|f(x, y, y') - f(x, \hat{y}, \hat{y}')| \leq L|y - \hat{y}| + M|y' - \hat{y}'|.$$

If

$$b - a < 4 \begin{cases} \frac{1}{(4L - M^2)^{1/2}} \cos^{-1} \frac{M}{2\sqrt{L}}, & \text{if } 4L - M^2 > 0, \\ \frac{1}{(M^2 - 4L)^{1/2}} \cosh^{-1} \frac{M}{2\sqrt{L}}, & \text{if } 4L - M^2 < 0, \qquad L, M > 0, \\ \frac{1}{M}, & \text{if } 4L - M^2 = 0, \qquad M > 0, \\ \infty, & \text{otherwise}, \end{cases}$$

then the nonlinear BVP (2.14) has a unique solution.

A proof for this theorem can be found in Bailey *et al.* [8]. Interestingly enough, Theorem 2.1.4 implies that the uniqueness of a solution for many nonlinear BVPs depends on the size of the solution interval $[a, b]$.

2.2 Conditioning of BVPs

In this section, the type of BVPs that are well-suited for solution by numerical methods are described. In particular, BVPs for which a small change to the ODEs or boundary conditions results in a small change to the solution must be considered. A BVP that has this property is said to be *well-conditioned*. Otherwise, the BVP is said to be *ill-conditioned*.

This property is important due to the error associated with numerical solutions to BVPs. Depending on the numerical method, a numerical solution $\hat{\mathbf{y}}(x)$ to the linear BVP (2.1) may exactly satisfy the perturbed ODE

$$\hat{\mathbf{y}}' = \mathbf{A}(x)\hat{\mathbf{y}} + \mathbf{q}(x) + \mathbf{r}(x), \quad a < x < b, \tag{2.15a}$$

where $\mathbf{r} : \mathbb{R} \to \mathbb{R}^m$, and the linear boundary conditions

$$\mathbf{B}_a\hat{\mathbf{y}}(a) + \mathbf{B}_b\hat{\mathbf{y}}(b) = \boldsymbol{\beta} + \boldsymbol{\sigma}, \tag{2.15b}$$

where $\boldsymbol{\sigma} \in \mathbb{R}^m$. If $\hat{\mathbf{y}}(x)$ is a reasonably good approximate solution to (2.1), then $\|\mathbf{r}(x)\|$ and $\|\boldsymbol{\sigma}\|$ are small. However, this may not imply that $\hat{\mathbf{y}}(x)$ is close to the exact solution $\mathbf{y}(x)$. A measure of conditioning for linear BVPs that relates both $\|\mathbf{r}(x)\|$ and $\|\boldsymbol{\sigma}\|$ to the error in the numerical solution can be determined. The following discussion can be extended to nonlinear BVPs by considering the variational problem on small subdomains of the nonlinear BVP [4, Section 3.4].

Letting

$$\mathbf{e}(x) = \hat{\mathbf{y}}(x) - \mathbf{y}(x),$$

then subtracting the original BVP (2.1) from the perturbed BVP (2.15) results in

$$\mathbf{e}'(x) = \hat{\mathbf{y}}'(x) - \mathbf{y}'(x) = \mathbf{A}(x)\mathbf{e}(x) + \mathbf{r}(x), \quad a < x < b, \tag{2.16a}$$

with boundary conditions

$$\mathbf{B}_a\mathbf{e}(a) + \mathbf{B}_b\mathbf{e}(b) = \boldsymbol{\sigma}. \tag{2.16b}$$

Because (2.16) is linear, the general solution for the linear BVPs (2.6), with $\mathbf{s}$ defined in (2.7), can be applied. However, the form of the solution can be furthered simplified by letting

$$\boldsymbol{\Theta}(x) = \mathbf{Y}(x)\mathbf{Q}^{-1},$$

where $\mathbf{Y}(x)$ is the fundamental solution and $\mathbf{Q}$ is defined in (2.8). Then the general solution can be written as

$$\mathbf{e}(x) = \boldsymbol{\Theta}(x)\boldsymbol{\sigma} + \int_a^b \mathbf{G}(x,t)\mathbf{r}(t)dt, \tag{2.17}$$

where $\mathbf{G}(x,t)$ is Green's function [4],

$$\mathbf{G}(x,t) = \begin{cases} \boldsymbol{\Theta}(x)\mathbf{B}_a\boldsymbol{\Theta}(a)\boldsymbol{\Theta}^{-1}(t), & t \le x, \\ -\boldsymbol{\Theta}(x)\mathbf{B}_b\boldsymbol{\Theta}(b)\boldsymbol{\Theta}^{-1}(t), & t > x. \end{cases}$$

Taking norms of both sides of (2.17) and using the Cauchy–Schwartz inequality [4] results in

$$\|\mathbf{e}(x)\|_\infty \le \kappa_1\|\boldsymbol{\sigma}\|_\infty + \kappa_2\|\mathbf{r}(x)\|_\infty, \tag{2.18}$$

where

$$\kappa_1 = \|\mathbf{Y}(x)\mathbf{Q}^{-1}\|_\infty,$$

and

$$\kappa_2 = \sup_{a\le x\le b}\int_a^b \|\mathbf{G}(x,t)\|_\infty dt.$$

In (2.18), the L_∞ norm, sometimes called a *maximum norm*, is used due to the common use of this

norm in numerical BVP software. For any vector $\mathbf{v} \in \mathbb{R}^N$, the ℓ_∞ norm is defined as

$$\|\mathbf{v}\|_\infty = \max_{1 \le i \le N} |v_i| .$$

The measure of conditioning is called the *conditioning constant* κ, and it is given by

$$\kappa = \max(\kappa_1, \kappa_2). \tag{2.20}$$

When the conditioning constant is of moderate size, then the BVP is said to be well-conditioned.

Referring again to (2.18), the constant κ thus provides an upper bound for the norm of the error associated with the perturbed solution,

$$\|\mathbf{e}(x)\|_\infty \le \kappa \left[\|\boldsymbol{\sigma}\|_\infty + \|\mathbf{r}(x)\|_\infty \right]. \tag{2.21}$$

It is important to note that the conditioning constant only depends on the original BVP and not the perturbed BVP. As a result, the conditioning constant provides a good measure of conditioning that is independent of any numerical technique that may cause such perturbations. The well-conditioned nature of a BVP and the local uniqueness of its desired solution are assumed in order to numerically solve the problem.

2.3 Initial value methods

Initial value methods for BVPs are based on common techniques to numerically solve IVPs. Chapter 2 of Shampine *et al.* [33] describes numerical methods for IVPs in detail. Such methods can be used to solve BVPs. The two most common algorithms that employ numerical methods for IVPs are called *simple shooting* and *multiple shooting*.

The simplest way to solve BVPs with initial value methods is by simple shooting. For the BVP (2.2), simple shooting involves finding a vector of initial values $\mathbf{s}$ such that

$$\mathbf{y}(a; \mathbf{s}) = \mathbf{s},$$

and

$$\mathbf{g}(\mathbf{s}, \mathbf{y}(b; \mathbf{s})) = \mathbf{0}. \tag{2.22}$$

In order to evaluate (2.22), the associated IVP must be numerically solved from $x = a$ to $x = b$. To determine $\mathbf{s}$ for nonlinear boundary conditions (2.22), practical implementations of simple shooting usually combine Newton's method with some numerical IVP method. In that case, an initial guess for $\mathbf{s}$ is required.

From a theoretical viewpoint, the simplicity of simple shooting is attractive. For example, the theory for solving IVPs is much better understood than for BVPs. As a consequence, it is convenient to use simple shooting to extend theoretical results for IVPs to BVPs. In fact, the uniqueness theorem for nonlinear BVPs described in Section 2.1 is an application of simple shooting.

However, simple shooting is not commonly used in practice. This is due to two major factors. First, there is a possibility of encountering unstable IVPs during the use of simple shooting. An IVP is unstable if a small change in the initial data, e.g., the initial condition, produces a large change in the solution. Typically, IVPs with exponentially increasing solution components are considered unstable [4, Section 3.3]. On the other hand, a well-conditioned BVP can have an exponentially increasing solution component provided that an appropriate boundary condition at the right endpoint is present. When using simple shooting on BVPs, unstable IVPs can occur even when the BVP is well-conditioned [4, Section 4.1.3]. Second, the IVP arising from the application of simple shooting may only be integrable on some domain $[a, c]$, where $c < b$ [4, Section 4.1.3].

Multiple shooting method attempts to address these issues. In this approach, the solution domain is subdivided into smaller subdomains

$$a = x_0 < \cdots < x_N = b. \tag{2.23}$$

Then, a numerical method for IVPs can be used to solve an IVP on each subinterval

$$\mathbf{y}_i' = \mathbf{f}(x, \mathbf{y}_i), \quad x_{i-1} < x < x_i,$$

$$\mathbf{y}_i(x_{i-1}) = \mathbf{s}_{i-1},$$

for $i = 1, 2, \ldots, N$, where $\mathbf{s}_{i-1}$ is the initial condition for the IVP on the interval $x_{i-1} < x < x_i$. The result is a solution for each subinterval

$$\mathbf{y}(x) = \mathbf{y}_i(x; \mathbf{s}_{i-1}), \quad x_{i-1} \leq x \leq x_i, \quad i = 1, 2, \ldots, N.$$

The solutions of these systems of IVPs, one for each subinterval, must match at the shared points of each subinterval and must satisfy the boundary conditions. Thus, a series of *patching conditions* must be satisfied,

$$\mathbf{y}_i(x_i; \mathbf{s}_{i-1}) = \mathbf{s}_i, \quad x_{i-1} \leq x \leq x_i, \quad i = 1, 2, \ldots, N-1,$$

and

$$\mathbf{g}(\mathbf{s}_0, \mathbf{y}_N(b; \mathbf{s}_{N-1})).$$

The shooting vectors, $\mathbf{s}_i$, for each subinterval are determined by solving a system of generally

NAEs consisting of the patching equations and boundary conditions. Similar to simple shooting, if the BVP is nonlinear, then an IVP method is often combined with Newton's method to determine the final solution [33, Section 3.4].

Although multiple shooting attempts to address many of the problems of simple shooting, it is still faced with the task of integrating unstable ODEs. As a consequence, multiple shooting requires a large number of subintervals for BVPs that have exponentially increasing solution components [4]. As a consequence, the method becomes inefficient when compared to global methods, which form the topic of the next section.

2.4 Global methods

Global methods for solving BVPs generally consist of three steps. Each step is introduced by using a simple global method, called a *finite-difference* method, to solve a specific second-order linear BVP [4, Section 5.1.2]

$$y''(x) + \sin(x)\, y'(x) + y(x) = x, \quad 0 < x < 1, \tag{2.26a}$$

with separated two-point linear boundary conditions,

$$y(0) = 1, \quad y(1) = 1. \tag{2.26b}$$

The first step involves choosing a mesh

$$0 = x_0 < x_1 < \cdots < x_N = 1.$$

For simplicity, a uniform mesh is chosen; i.e., $x_i = ih$, $i = 0, 1, \ldots, N$, and $h = 1/N$.

The second step involves setting up a system of equations for which the unknowns are discrete solution values $\mathbf{y}_\pi = \{y_i\}_{i=0}^N$ at the mesh points, i.e., $y_i \approx y(x_i)$. The finite-difference method discretizes the ODE by replacing the derivatives of (2.26a) with finite-difference approximations. These approximations can be derived by recalling that $y(x_i + h)$ can be expanded by the use of the Taylor series

$$y(x_i + h) = y(x_i) + hy'(x_i) + \frac{h^2}{2}y''(x_i) + \frac{h^3}{6}y'''(x_i) + O(h^4). \tag{2.27}$$

In much the same way, $y(x_i - h)$, can be expanded to

$$y(x_i - h) = y(x_i) - hy'(x_i) + \frac{h^2}{2}y''(x_i) - \frac{h^3}{6}y'''(x_i) + O(h^4). \tag{2.28}$$

A finite-difference approximation for the first-order derivative can now be determined by subtracting

(2.28) from (2.27) and re-arranging to get

$$y'(x_i) = \frac{y(x_{i+1}) - y(x_{i-1})}{2h} + O(h^2), \tag{2.29a}$$

where $x_{i\pm1} = x_i \pm h$. A finite-difference approximation for second-order derivative can be determined by substituting (2.29a) into (2.27) to get

$$y''(x_i) = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{h^2} + O(h^2). \tag{2.29b}$$

To obtain the system of equations for the unknowns, the derivatives of (2.26a) are replaced with finite-difference approximations (2.29) at the internal mesh points

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} + \sin(x_i)\frac{y_{i+1} - y_{i-1}}{2h} + y_i = x_i, \quad i = 1, 2, \ldots, N-1,$$

and the boundary conditions yield

$$y_0 = 1, \quad y_N = 1.$$

The result is a system of $N+1$ linear equations

$$\boldsymbol{\mathcal{A}}\mathbf{y}_\pi = \mathfrak{b}, \tag{2.30}$$

where

$$\boldsymbol{\mathcal{A}} = \begin{pmatrix} \mathsf{a}_0 & \mathsf{c}_0 & 0 & 0 & \ldots & \ldots & 0 \\ \mathsf{b}_1 & \mathsf{a}_1 & \mathsf{c}_1 & 0 & \ldots & \ldots & 0 \\ 0 & \mathsf{b}_2 & \mathsf{a}_2 & \mathsf{c}_2 & 0 & \ldots & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ldots & \ldots & 0 & \mathsf{b}_{N-1} & \mathsf{a}_{N-1} & \mathsf{c}_{N-1} \\ 0 & \ldots & \ldots & \ldots & 0 & \mathsf{b}_N & \mathsf{a}_N \end{pmatrix},$$

$$\mathsf{a}_i = 1 - \frac{2}{h^2}, \quad \mathsf{b}_i = \frac{1}{h^2} - \frac{\sin(x_i)}{2h}, \quad \mathsf{c}_i = \frac{1}{h^2} + \frac{\sin(x_i)}{2h}, \quad i = 1, 2, \ldots, N-1, \tag{2.31}$$

$$\mathsf{a}_0 = \mathsf{a}_N = 1, \quad \mathsf{c}_0 = \mathsf{b}_N = 0,$$

and

$$\mathbf{y}_\pi = (y_0, y_1, \ldots, y_N)^T, \quad \mathfrak{b} = (1, x_1, \ldots, x_{N-1}, 1)^T.$$

The final step involves solving (2.30) to determine a discrete numerical solution.

Although it has been shown that this method can be used to numerically solve the BVP (2.26),

little has been said about the performance of the method. Ideally, the size of the global error

$$|e_i| = |y(x_i) - y_i|, \quad i = 0, 1, \ldots, N,$$

must approach zero as h approaches zero. This property is called *convergence*. This section is concluded by showing that the finite-difference method described in this section is indeed convergent. Convergence depends on two conditions.

First, the *local truncation error* must approach zero as h approaches zero. The local truncation error is defined as

$$\boldsymbol{\tau}_i[y] = \mathbf{L}_\pi \mathbf{y}(x_i),$$

where $\mathbf{L}_\pi$ is the differential operator. For this particular problem

$$\mathbf{L}_\pi y(x) = \boldsymbol{\Psi}(y(x)) - x,$$

where $\boldsymbol{\Psi}(\mathbf{y}(x))$ represents the method used to numerically solve the BVP. In this case,

$$\boldsymbol{\Psi}(y(x)) = \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} + \sin(x)\frac{y(x+h) - y(x-h)}{2h} + y(x).$$

Considering the finite-difference methods used for the derivative, the maximum local truncation error is

$$\boldsymbol{\tau}[y] = \max_{i=1,2,\ldots,N-1} \|\boldsymbol{\tau_i}[y]\| \leq ch^2, \tag{2.32}$$

where C is some constant. From the inequality (2.32), it is clear that $\boldsymbol{\tau}[y]$ approaches zero as h approaches zero. As a consequence, this method is said to be *consistent* and of order two. In general, a method is said to be of order p if the local truncation error is proportional to h^p.

Second, the finite-difference method must be shown to be stable. A method is stable if for a given mesh, there exists a stepsize h_0, such that for all $h < h_0$

$$\|y_i\| \leq K \max\{\|\mathbf{y}_0\|, \|\mathbf{y}_N\|, \max_{i=1,2,\ldots,N-1} \|\boldsymbol{\Psi}(\mathbf{y}_i)\|\}, \quad i = 1, 2, \ldots, N-1, \tag{2.33}$$

and K is a constant [4]. In the case of this example, the inequality (2.33) holds as long as

$$\|\boldsymbol{\mathcal{A}}^{-1}\| \leq K. \tag{2.34}$$

Once consistency and stability are established for a numerical method, convergence follows [4, Page 190]. Convergence for this particular finite-difference method can be shown as follows.

Applying the method $\mathbf{\Psi}$ to the global error e_i, $i = 1, 2, \ldots, N-1$, results in

$$\mathbf{\Psi}(e_i) = \mathbf{L}_\pi y(x_i) = \boldsymbol{\tau}_i[y], \tag{2.35}$$

$$e_0 = e_N = 0.$$

Using (2.35), along with the inequalities (2.33) and (2.32), the inequality

$$|e_i| \leq K\tau[y] \leq Kch^2, \quad i = 1, 2, \ldots, N-1,$$

can be obtained. Therefore, as h approaches zero so does the global error, and thus the numerical method is convergent.

2.5 One-step methods

The finite-difference method from the previous section requires a tridiagonal system of linear equations to be solved. A one-step method has the form

$$\frac{\mathbf{y}_{i+1} - \mathbf{y}_i}{h_i} = \mathbf{\Psi}(\mathbf{y}_i, \mathbf{y}_{i+1}; x_i, h_i),$$

where $h_i = x_{i+1} - x_i$. These methods compute a discrete solution $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_N$ at the mesh points defined by (2.23). Because the discretization on subinterval i only depends on the unknowns i and $i+1$, implementations of one-step methods can take advantage of the structure of the corresponding matrix $\mathcal{A}$ to solve the system of equations more efficiently than those from the finite-difference method [5].

This section begins by using a one-step method, called the *trapezoidal rule*, to solve linear BVPs [5, Section 8.1]. Afterward, the trapezoidal rule is extended to nonlinear BVPs.

The trapezoidal rule is obtained by defining

$$\mathbf{\Psi}(\mathbf{y}_i, \mathbf{y}_{i+1}; x_i, h_i) = \frac{1}{2}[\mathbf{f}(x_i, \mathbf{y}_i) + \mathbf{f}(x_{i+1}, \mathbf{y}_{i+1})], \quad i = 0, 1, \ldots, N-1.$$

Applying the trapezoidal rule to the linear BVP (2.1) results in the system of linear equations

$$\left[-\frac{1}{h_i}\mathbf{I} - \frac{1}{2}\mathbf{A}(x_i)\right]\mathbf{y}_i + \left[\frac{1}{h_i}\mathbf{I} - \frac{1}{2}\mathbf{A}(x_{i+1})\right]\mathbf{y}_{i+1} = \frac{1}{2}\left[\mathbf{q}(x_i) + \mathbf{q}(x_{i+1})\right], \quad i = 0, 1, \ldots, N-1.$$

Re-writing the linear equations in matrix form results in

$$\begin{pmatrix} \mathbf{S}_0 & \mathbf{R}_0 & 0 & \dots & \dots & 0 \\ 0 & \mathbf{S}_1 & \mathbf{R}_1 & 0 & \dots & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & \mathbf{S}_{N-1} & \mathbf{R}_{N-1} \\ \mathbf{B}_a & 0 & \dots & \dots & 0 & \mathbf{B}_b \end{pmatrix} \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \vdots \\ \mathbf{y}_{N-1} \\ \mathbf{y}_N \end{pmatrix} = \begin{pmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \vdots \\ \vdots \\ \mathbf{v}_{N-1} \\ \boldsymbol{\beta} \end{pmatrix},$$

where

$$\mathbf{S}_i = -\frac{2}{h_i}\mathbf{I} - \mathbf{A}(x_i), \quad \mathbf{R}_i = \frac{2}{h_i} - \mathbf{A}(x_{i+1}), \quad \mathbf{v}_i = \mathbf{q}(x_i) + \mathbf{q}(x_{i+1}), \quad i = 0, 1, \dots, N-1.$$

The structure of the matrix is independent of the BVP and is referred to as a bordered almost-block-diagonal matrix [4].

Using the trapezoidal rule for nonlinear BVPs results in a system of NAEs. The system can be solved numerically by applying Newton's method.

2.6 Continuous solution methods

The methods described in the previous section return discrete numerical solutions to BVPs. In this section, methods that return a (continuous) piecewise polynomial as the numerical solution to the BVP (2.2) are described.

In particular, two higher-order one-step methods are described in this section. First, a continuous mono-implicit Runge–Kutta (MIRK) approach for first-order system of ODEs is described in Section 2.6.1. Second, spline-collocation methods for mixed-order ODE systems are described in Section 2.6.2.

2.6.1 An approach based on MIRK formulas

In this section, a method based on MIRK discretization formulas is described. This method can be applied to the BVP (2.2) [9]. This method determines numerical approximations $\mathbf{y}_i$ to the solution values $\mathbf{y}(x_i)$ at each of the points in the mesh (2.23).

The MIRK discretization formulas have the form

$$\boldsymbol{\varphi}_{i+1}(\mathbf{y}_i, \mathbf{y}_{i+1}) = \mathbf{y}_{i+1} - \mathbf{y}_i - h_i \sum_{j=1}^{s} b_j \mathbf{f}(x_i + c_j h_i, \mathbf{Y}_j) = \mathbf{0}, \quad i = 0, 1, \dots, N-1, \tag{2.36}$$

where

$$\mathbf{Y}_j = (1 - v_j)\mathbf{y}_i + v_j\mathbf{y}_{i+1} + h_i \sum_{k=1}^{j-1} a_{j,k}\mathbf{f}(x_i + c_k h_i, \mathbf{Y}_k), \quad j = 1, 2, \ldots, s, \tag{2.37}$$

are the *stages* of the MIRK method.

The coefficients, $v_j, b_j, a_{j,k}$, $j = 1, 2, \ldots, s$, $k = 1, 2, \ldots, j-1$, define the MIRK method, and $c_j = v_j + \sum_{k=1}^{j-1} a_{j,k}$. A system of NAEs is generated that consists of equations (2.36) and the boundary conditions (2.2b). The values $\mathbf{y}_i$ are determined by solving the system of NAEs.

Once the $\mathbf{y}_i$ values are determined, a piecewise polynomial, $\mathbf{S}(x) \in C^1[a, b]$, can be generated by using a continuous MIRK formula. On the subinterval $[x_i, x_{i+1}]$, $\mathbf{S}(x)$ takes the form

$$\mathbf{S}(x_i + \theta h_i) = \mathbf{y}_i + h_i \sum_{j=1}^{s^*} b_j(\theta)\mathbf{f}(x_i + c_j h_i, \mathbf{Y}_j), \quad 0 \le \theta \le 1,$$

where $s^* \ge s$. The polynomials $b_j(\theta)$ are defined by the particular continuous MIRK method used. Because $s^* \ge s$, additional stages may be required to determine $\mathbf{S}(x)$. The additional stages have the form of (2.37).

The coefficients for both discrete and continuous MIRK formulas of order two, four, and six can be found in Muir [24]. These particular MIRK formulas have an important property worth discussing.

The MIRK formulas [24] perform equally well when the solution modes of the ODEs are either increasing or decreasing. These formulas are called *symmetric formulas*. Unlike initial value methods, for which there is a well-defined direction of integration, global methods have no preferred direction of integration. This is particularly important for BVPs because information about the solution comes from more than one point, unlike IVPs [33, Section 3.4].

Also, MIRK methods have a distinguishing property when compared to other implicit Runge–Kutta methods. Unlike many implicit methods, MIRK methods allow the stage computations to be evaluated explicitly. On the other hand, general implicit methods require the stages to be evaluated implicitly. There are two standard approaches when dealing with this issue. The simplest approach is to determine the stages as part of the system of NAEs. An alternative approach is that the implicit stages can be expressed in terms of $\mathbf{y}_i$ and $\mathbf{y}_{i+1}$. This is often referred to as *parameter condensation* [5, Section 8.3].

2.6.2 Spline-collocation methods

In the previous section, a method that solves systems of first-order ODEs with two-point boundary conditions is presented. Recall that although these methods can be used to solve higher-order ODEs, the system of mixed-order ODEs must first be converted to a system of first-order ODEs. This results in a larger system of ODEs with additional dependent variables. It is possible to derive

a class of Runge–Kutta methods designed to handle higher-order ODEs directly. MIRK methods for second-order ODEs are an example of one such class [25].

In this section, a class of methods called spline-collocation methods [4, Section 5.6.3] that directly solve mixed-order ODEs (1.2a) with appropriate boundary conditions is described. A spline-collocation method produces an approximation to the solution of a BVP in the form of a piecewise polynomial

$$\mathbf{S}(x) = \sum_{j=1}^{M} \boldsymbol{\alpha}_j \boldsymbol{\phi}_j(x), \quad a \leq x \leq b, \tag{2.38}$$

where $\boldsymbol{\alpha}_j$ are unknown coefficients and $\boldsymbol{\phi}_j(x)$ are linearly independent basis functions. The parameter M is the number of free coefficients given by

$$M = Nkm + m^*,$$

where k is the number of collocation points in each subinterval, $m^* = \sum_{i=1}^{m} d_i$ with d_i defined in (1.2a), and N is the number of subintervals of the mesh that partitions $[a, b]$.

A suggested basis $\boldsymbol{\phi}_j(x)$ is the B-splines basis [4, Section 5.6.3]. However, it should be noted that other basis functions have shown improvements over B-splines [7].

For a given BVP, the parameters $\boldsymbol{\alpha}_j$ are determined by requiring $\mathbf{S}(x)$ to satisfy the m^* boundary conditions and $k \times N$ collocation conditions

$$\mathbf{S}^{(d)}(x_{ij}) - \mathbf{f}(x_{ij}, \mathbf{z}(\mathbf{y})) = \mathbf{0}, \quad i = 1, 2, \ldots, N, \quad j = 1, 2, \ldots, N, \tag{2.39}$$

where $x_{ij} = x_i + h_i c_j$ are the *collocation points* and $0 \leq c_1 \leq \cdots \leq c_k \leq 1$.

Software implementations of collocation methods, e.g., `COLSYS`, use *Gauss points* for $\{c_j\}_{j=1}^{k}$. For a method that has s stages, these points are chosen such that the order of the method satisfies $p = 2s$ [3, Section 2.6.1]. Similar to the MIRK methods presented in the previous section, methods that use Gauss points are symmetric.

If the BVP is linear, then the linear boundary conditions and (2.39) form a system of linear equations. The variables $\boldsymbol{\alpha}_j$ from (2.38) can then be determined by solving a linear system with a coefficient matrix $\mathbf{A} \in \mathbb{R}^{M \times M}$ called the *collocation matrix*. However, if the BVP is nonlinear, then Newton's method must be applied to the system of equations.

2.7 Mesh selection

Once a numerical solution is determined, a measure of defect, i.e., the amount by which the numerical solution fails to satisfy the original system of ODEs, or a measure of error can be associated

with each subinterval of a mesh (2.23). An example of a measure of error on each subinterval is

$$e_i = \max_{x_{i-1} \le x < x_i} \|\mathbf{y}(x) - \mathbf{S}(x)\|, \quad i = 1, 2, \ldots, N.$$

Often, the goal of a mesh selection strategy is to determine a mesh such that for each subinterval of the mesh, an estimate of e_i is less than a user-supplied tolerance. For the purpose of efficiency however, a mesh selection strategy often attempts to find a mesh such that an estimate of the error for each subinterval is as close to the user-supplied tolerance as possible. By doing so, the process of determining a numerical solution can use the least number of mesh points possible. In contrast, if a mesh selection strategy determines a mesh such that the estimated error for each subinterval is well below the user-supplied tolerance, a numerical solution would exceed the required accuracy at the cost of additional mesh points and therefore additional computational time. The mesh selection strategy can therefore greatly affect the overall performance of a BVP software implementation.

There exist a number of mesh selection strategies; one popular strategy is called *equidistribution* [3, Section 9.1.1]. The equidistribution algorithm requires an estimate of the error of a numerical solution for each subinterval of the mesh upon which the numerical solution is based. The estimate of the error is then used to suggest a new mesh such that the estimated error for each subinterval on the new mesh is approximately equal to the user-supplied tolerance. This may involve adding or deleting mesh points as well as redistributing the points already in the mesh. Because an estimate of the error is used, several attempts at equidistribution, along with finding a discrete solution for each attempt, take place before a satisfactory mesh is found.

2.8 Solving nonlinear algebraic equations

Once a mesh is chosen, a discrete numerical solution must be determined. In order to determine a discrete solution, many of the methods described in the previous sections require a system of NAEs to be solved.

Using the MIRK method as an example, a discrete solution is evaluated by solving the system of $(N+1)m$ equations

$$\mathbf{\Phi}(\mathbf{Y}) \equiv \begin{pmatrix} \boldsymbol{\varphi}_1(\mathbf{y}_0, \mathbf{y}_1) \\ \vdots \\ \boldsymbol{\varphi}_N(\mathbf{y}_{N-1}, \mathbf{y}_N) \\ \mathbf{g}(\mathbf{y}_0, \mathbf{y}_N) \end{pmatrix} = \mathbf{0}, \tag{2.40}$$

where $\mathbf{Y} = [\mathbf{y}_0, \ldots, \mathbf{y}_N]$ is the discrete solution vector and

$$\boldsymbol{\varphi}_i = \mathbf{y}_i - \mathbf{y}_{i+1} + h_i \mathbf{\Psi}(\mathbf{y}_i, \mathbf{y}_{i+1}; x_i, h_i),$$

where $\mathbf{\Psi}$ is based on a MIRK method. The function $\mathbf{\Phi}(\mathbf{Y})$ is often called the *residual function* [19].

Unlike the situation for systems of linear equations, there is no known method, even in principle, to determine an exact solution to a given system of NAEs. Instead, a numerical algorithm must be used to approximate a solution. Software implementations of global methods for BVPs often rely on a form of Newton's method for this task. In this section, Newton's method is briefly described.

Newton's method approximates a solution to (2.40) by iteratively evaluating

$$\mathbf{Y}^{\nu} = \mathbf{Y}^{\nu-1} - \mathbf{\Phi}'(\mathbf{Y}^{\nu-1})^{-1}\mathbf{\Phi}(\mathbf{Y}^{\nu-1}), \quad \nu = 1, 2, \ldots,$$

where $\mathbf{Y}^{\nu}$ is the solution after the νth Newton iteration, $\mathbf{Y}^{0}$ is a user-supplied initial guess, and

$$\mathbf{\Phi}'(\mathbf{Y}^{\nu-1}) = \left.\frac{\partial \mathbf{\Phi}}{\partial \mathbf{Y}}\right|_{\mathbf{Y}=\mathbf{Y}^{\nu-1}}$$

is the *Jacobian matrix* evaluated at $\mathbf{Y}^{\nu-1}$.

In practice, the inverse of the Jacobian matrix is not computed explicitly. Instead, the linear system

$$\mathbf{\Phi}'(\mathbf{Y}^{\nu-1})\boldsymbol{\delta}^{\nu} = -\mathbf{\Phi}(\mathbf{Y}^{\nu-1}), \tag{2.41}$$

is solved [19]. Then the next Newton iterate $\mathbf{Y}^{\nu}$ is determined from

$$\mathbf{Y}^{\nu} = \mathbf{Y}^{\nu-1} + \boldsymbol{\delta}^{\nu},$$

where $\boldsymbol{\delta}^{\nu}$ is often known as the *Newton direction* for the νth Newton iteration.

Newton iterations continue until a termination criterion is met. For a variety of applications, many different termination criteria exist [19]. BVP software packages often use the scaled termination criterion

$$\|\boldsymbol{\delta}^{\nu}\| \leq tol\, \|\mathbf{Y}^{\nu} + 1\|_{\infty}, \tag{2.42}$$

where *tol* is a user-supplied accuracy tolerance.

Newton's method is said to *converge* when condition (2.42) is satisfied. In practice, successful convergence is largely dependent on the user-supplied initial guess. In many cases, a poor initial guess leads to a Newton direction that *overshoots* the actual solution. For example, the function

$$F(x) = \tan^{-1}(x),$$

has a root $x = 0$. Applying Newton's method with an initial guess of $x = 10$, results in a sequence of values for x

$$10, -138, 2.9 \times 10^{4}, -1.5 \times 10^{9}, 9.9 \times 10^{17}.$$

It becomes quickly apparent that Newton's method is not converging to the solution [19].

The problem of overshooting can often be solved by only applying a fraction of a Newton direction, i.e.,

$$\hat{\boldsymbol{\delta}} = \lambda \boldsymbol{\delta}^{\nu}, \quad 0 \leq \lambda \leq 1,$$

where λ is known as the *damping factor*. A Newton iteration with $\lambda = 1$ is referred to as an *undamped* Newton iteration. The method used to determine the damping factor is referred to as the *global-convergence method*. Employing such a method reduces the importance of the quality of the initial guess on the overall success of Newton's method. Therefore, this section concludes with a description of one such global-convergence method, called *damped Newton's method*, often used by BVP software packages [4, Chapter 8].

The damped Newton's method determines a damping factor such that the *natural criterion function*

$$g(\lambda) = \frac{1}{2} \left\| \boldsymbol{\Phi}'(\mathbf{Y}^{\nu-1})^{-1} \boldsymbol{\Phi}(\mathbf{Y}^{\nu-1} + \lambda \boldsymbol{\delta}^{\nu}) \right\|^2,$$

satisfies the condition

$$g(\lambda) \leq (1 - 2\lambda\sigma)\, g_0, \tag{2.43}$$

where

$$g_0 = g(0) = \frac{1}{2} \|\boldsymbol{\delta}^{\nu}\|^2,$$

and $\sigma = 0.01$. Condition (2.43) ensures that any damping factor used to evaluate $\mathbf{Y}^{\nu}$ results in a reduction in the size of the residual.

A damping factor that satisfies condition (2.43) is determined iteratively, for each Newton iteration, by the use of a quadratic interpolating polynomial of the natural criterion function. The interpolating polynomial has a minimum at

$$\lambda_{\eta} = \frac{\lambda_{\eta-1}^2 g_0}{(2\lambda_{\eta-1} - 1) g_0 + g(\lambda_{\eta-1})}, \tag{2.44}$$

where λ_{η} is the ηth damping factor of the global-convergence method [4, Section 8.1.1]. In practice however, (2.44) may result in a damping factor that differs too much from the previous damping factor $\lambda_{\eta-1}$. With that in mind, the damping factor for global-convergence iteration η is chosen as

$$\lambda_{\eta} := \max\left(\tau \lambda_{\eta-1}, \lambda_{\eta}\right),$$

where $\tau = 0.1$. The global-convergence iterations continue until an appropriate damping factor is found or $\lambda_{\eta} < \lambda_{\min}$. The value $\lambda_{\min}$ is the smallest allowable damping factor. At that point, the global-convergence method is deemed to have failed. A suggested value for $\lambda_{\min}$ is 0.01 [4].

Overall, the damped Newton's method can be computationally expensive. A system of linear

equations must be solved every iteration of the global-convergence method. To reduce the number of these iterations, an estimate of the initial damping factor λ_0 can be made as close to the desired damping factor as possible. In order to do so, the information from previous Newton iterates can be used by letting

$$\tilde{\lambda}_\eta = \frac{\|\boldsymbol{\delta}^{\nu-1}\|}{\|\boldsymbol{\delta}^{\nu} - \boldsymbol{\Phi}'(\mathbf{Y}^{\nu-2})^{-1}\boldsymbol{\Phi}(\mathbf{Y}^{\nu-1})\|}\lambda_{\eta-1},$$

and set the initial damping factor to

$$\lambda_{\eta,0} = \max\left(\lambda_{\min}, \min\left(\tilde{\lambda}_\eta, 1\right)\right).$$

2.9 Summary

In Chapter 2, certain BVPs that are well-suited for solution with numerical methods are described. In particular, numerical methods should be only applied to BVPs that are well-conditioned and have a locally unique solution.

In order to approximate a solution to a BVP, either an initial value method or a global method can be used. For initial value methods, simple shooting is desirable from a theoretical perspective. However, the method is rarely used in practice due to practical concerns. Multiple shooting overcomes many of these concerns. However, even multiple shooting can be problematic for certain BVPs. With that in mind, global methods are used for many numerical BVP software packages. A simple finite-difference scheme, introduced in Section 2.5, is an example of a global method. However, the scheme is not well-suited as a general BVP solver . In contrast, both MIRK schemes and collocation schemes, introduced in Section 2.6, are better-suited to handle a greater variety of BVPs. The chapter concludes with a discussion of mesh selection and Newton's method to solve NAEs. Both of these algorithms are vital to both the efficiency and robustness of a numerical BVP software package. Many of the numerical concepts introduced in this chapter are used in the BVP component of `pythODE`, which is the subject of the next chapter.

Chapter 3

A Problem-Solving Environment for BVPs

This chapter introduces a problem-solving environment dedicated to the numerical solution of ODEs called `pythODE`. The PSE consists of a BVP component and an IVP component.

The BVP component of `pythODE` is one of the primary contributions of this thesis. This component allows users to specify how each step of the numerical solution process of a BVP is performed. Therefore, the BVP component consists of a collection of numerical algorithms from which users can choose. Most of these algorithms are commonly found in other BVP software packages. However, they have been written in such a way to allow them to fit within the modularized framework of the PSE.

The IVP component is being developed in parallel to the BVP component of `pythODE`. However, because IVPs are not the focus of this thesis, the IVP component is not considered further.

The remainder of this chapter can be divided into the following sections. Section 3.1 describes existing BVP software packages. Section 3.2 introduces the features behind modern PSEs. Section 3.3 describes the architecture of `pythODE`. Section 3.4 describes the BVP component of `pythODE`. Section 3.5 demonstrates how to solve Bratu's problem with the BVP component of `pythODE`.

3.1 A review of BVP software packages based on global methods

In this section, some existing BVP software packages that use global methods to numerically solve BVPs are reviewed. The software packages are categorized according to the method of error control that is used.

This section begins with BVP software packages that attempt to return a continuous approximate solution, $\mathbf{S}(x)$, such that some norm of the global error

$$\mathbf{e}(x) = \mathbf{y}(x) - \mathbf{S}(x), \quad a \leq x \leq b, \tag{3.1}$$

where $\mathbf{y}(x)$ is the exact solution, is less than a user-specified tolerance. Such packages are said to employ *global-error control*. Of course, the exact solution of a given BVP is generally not known.

Therefore, these BVP software packages must estimate the global error. All the global-error BVP software packages mentioned in this section use Richardson extrapolation to estimate global error [4, Section 5.5.2]. Also, these software packages choose a mesh such that the error is equidistributed across the entire mesh.

The global-error control software package `COLSYS` is one of the earliest BVP software packages to use global methods to numerically solve BVPs. The software uses a B-spline collocation algorithm to produce a piecewise polynomial to represent the numerical solution of the BVP [3]. A later version of `COLSYS`, called `COLNEW`, replaces the B-Splines with a monomial representation [26]. This modification results in an improvement in the performance of `COLNEW` over `COLSYS` [7]. Additional modifications were made to `COLNEW` to extend the problem class of BVPs into the realm of boundary value differential-algebraic equations (DAEs) [6]. The resulting software package, called `COLDAE` [6], demonstrates the effectiveness of applying techniques for the numerical solution of BVPs to boundary value DAEs.

The language `Fortran 77` was used to create `COLSYS`, `COLNEW`, and `COLDAE`. This language lacks features such as user-defined data types, dynamic memory allocation, and default parameters for functions. As a result, the interfaces for all three BVP software packages are complex. They each require users to enter over 15 function parameters to use the primary solver routine.

The next two software packages attempt to return a numerical solution such that a measure of the local truncation error on each subinterval is less than a user-supplied tolerance. The BVP software packages `TWPBVP` and `TWPBVPC` [13] use MIRK discretization formulas to return a discrete numerical solution to the BVP. Both software packages use a deferred-correction approach based on the use of higher-order MIRK formulas [14] in order to return a more accurate solution than the solution obtained from using the discretization formulas alone. The deferred-correction approach also yields estimates of the local truncation error. These software packages return only a discrete solution approximation defined at the mesh points that partition the problem domain. The software package `TWPBVPC` uses a novel approach to mesh selection that involves estimating the conditioning of the BVP when selecting a new mesh [12]. The software then uses a combination of local truncation error and a conditioning constant estimate for mesh selection. As a consequence, `TWPBVPC` is able to solve challenging problems using fewer mesh points than `TWPBVP` [21]. Both `TWPBVP` and `TWPBVPC` use a local refinement algorithm to determine each new mesh [21].

The language `Fortran 77` was also used to create `TWPBVP` and `TWPBVPC`. The interface for both of these software packages are complex; users must enter 40 parameters to use the primary solver routine.

The remaining software packages mentioned in this section use a backwards-error approach to error control [34]. This involves estimating the maximum of a norm of the defect

$$\mathbf{r}(x) = \mathbf{S}'(x) - \mathbf{f}(x, \mathbf{S}(x)),$$

where $\mathbf{S}(x)$ is again the continuous numerical solution and where the ODEs is the first-order system $\mathbf{y}' = \mathbf{f}(x, \mathbf{y}(x))$. A C^1-continuous numerical solution is required in order for such BVP software packages to compute the defect at several points between mesh points [17] and thereby return an estimation of the defect. These BVP software packages are said to employ *defect-control*.

In a similar sense to the other types of BVP software packages, defect-control software packages return a solution only if the norm of the defect is less than a user-specified tolerance. There is a clear benefit to controlling the defect rather than the global error. Defect-control software packages are able to estimate the norm of the defect more directly, even under circumstances for which the global-error estimate is not valid. There is however a significant disadvantage to using defect control. The defect is only indirectly related to the global error. Therefore, it is possible for a defect-control software package to return a solution that satisfies the user-specified tolerance for the norm of the defect while the global-error norm remains large. In the most extreme cases, a defect-control software package may return a solution to a BVP that has no solution. These solutions have been called *pseudo-solutions* [34].

Both the defect-control software packages `MIRKDC` and `BVP_SOLVER` use a continuous MIRK approach. A discrete solution is determined by MIRK formulas and then forms the basis for the continuous numerical solution. One of the primary differences between the two software packages is the interface. The interface for `MIRKDC` is considerably more complicated than that of `BVP_SOLVER`. This is primarily a consequence of the implementation languages. The package `MIRKDC` is written in `Fortran` 77, whereas, `BVP_SOLVER` is written in `Fortran` 90/95 and therefore has a simpler interface due to the use of default parameters, dynamic memory allocation, and user-defined data types. In the case of `BVP_SOLVER`, users must use only a four-parameter initialization function and a three-parameter solver function. The software package `BVP_SOLVER` also includes several features not found in `MIRKDC`. For example, `BVP_SOLVER` provides users with an optional *a posteriori* global-error estimate through the use of Richardson extrapolation [4].

The final two BVP software packages discussed in this section are written in `Matlab`. The software package `bvp4c` also uses a fourth-order MIRK continuous approach to determine a discrete and continuous solution. Because `bvp4c` is written in `Matlab`, it comes bundled with a simple interface that can be used in conjunction with the many numerical algorithms found in `Matlab`. However, users are unable to choose which numerical algorithms are used within `bvp4c`.

The other `Matlab` BVP software package is called `bvp5c`; it attempts to control both the global error and the defect. The software `bvp5c` uses a four-point, fifth-order Labatto formula to determine

a continuous numerical solution to the BVP [20]. Similar to the other software packages mentioned, `bvp5c` controls the defect. However, as a consequence of the particular four-point Labatto formula, the scaled defect has the same order of convergence as the true error. Therefore, the true error asymptotically approaches the scaled defect [20]. As a consequence, when the norm of the defect is less than a user-supplied tolerance, so is an appropriately scaled norm of the global error.

3.2 Problem-solving environments

In order for a software package to be considered a PSE, it must have a specific set of features [28]. For example, a PSE should allow users to enter problems by using a language familiar to the problem domain. The language of the PSE is therefore said to be *domain-specific*. Also, a PSE should allow for the automatic selection of algorithms used to solve a problem, making the actual act of setting up to solve a problem as simple as possible. However, a PSE should still remain flexible by providing users with the ability to choose between different algorithms to solve a problem. Finally, a PSE should be expandable. Users should be able to easily add their own algorithms to the software package existing catalogue of numerical algorithms.

Software packages that are also PSEs are a powerful tool for a wide range of users. For example, users who have little programming experience are still able to use a PSE to solve problems. They simply enter the problem into the PSE in the domain-specific language with which they are already familiar. If a user has little knowledge of the methods used to solve the problem, they can use a PSE to determine a solution without being forced to extensively study the solution methods. Finally, users can easily develop and compare the performance of various solution methods on a given class of problems within a PSE.

Today, there exists a variety of PSEs for a wide range of problems. Widely used PSEs for general-purpose numerical computations include `MATLAB` [40], `Mathematica` [39], and `MAPLE` [40]. PSEs for a more specific problem class include `COMSOL` [38] for the numerical solution of PDEs and `pythNon` [37] for the numerical solution to systems of NAEs.

In regards to BVPs, none of the previously discussed BVP software packages can be considered a PSE. Ideally, a PSE dedicated to numerical solution of BVPs should allow users to select how each component of the solution procedure is performed. For example, users should be able to select which discretization algorithm, error-control algorithm, and nonlinear solution algorithm is used to solve a BVP. The remainder of this chapter describes a software package that offers the features of a PSE to the user.

3.3 The architecture of pythODE

This section describes the architecture of pythODE. The pythODE PSE is designed using a *layered architecture* [15]; see Figure 3.1. Software packages built with a layered architecture consist of components that can be neatly divided into layers. An individual layer depends only on itself or the layers below it [15, Chapter 4]. The two bottom layers of pythODE are described in this section. The BVP component of the top layer is the subject of the next section.

The bottom layer consists of Python, the language used to write the vast majority of the modules for pythODE. Python is a popular high-level language used in a wide variety of software applications. Similar to Java, software written in Python runs on top of a virtual machine that is available for many popular operating systems, including Windows, Mac OS X, and Linux. The virtual machine is used to execute source code written in Python in an interpreter-based manner. However, the code can be compiled before execution to increase performance. Despite the increase in speed, the code still runs more slowly than code written with fully compiled languages such as Fortran or C/C++. However, the use of a virtual machine greatly adds to the portability of software written in Python.

There are a variety of other languages from which to choose from to create a PSE. Recall from the previous section, most BVP software packages are written in Fortran and a few are written in Matlab. However, Python lacks high-quality BVP software written in the Python language itself. There have, however, been successful attempts at creating Python interfaces for existing Fortran BVP software [10]. However, although the user interfaces have been greatly improved, the BVP software packages are still used in the same manner as the original software. Therefore, they fall short of what a PSE should be. In other words, although the software usability has been greatly increased, the software still lacks flexibility and expandability. Instead, high-level features of the Python programming language are used to create a BVP software package that is consistent with the definition of a PSE, given in the previous section.

The middle layer contains Scipy [23] . The library Scipy consists of multiple routines and data types commonly used in scientific computing. Many of the data types, such as the multi-dimensional array object used throughout pythODE, were imported from Numpy [2], the package originally designed to allow Python to perform basic scientific computing. However, Scipy adds many additional modules that broaden Python's usefulness in scientific computing. Many of the Scipy modules consist of high-performance Fortran routines interfaced with Python through the use of F2py [27], a tool that automatically builds interfaces between the two languages. For example, the linear-algebra routines used in Scipy are originally from the Fortran high-performance linear-algebra library LAPACK [1].

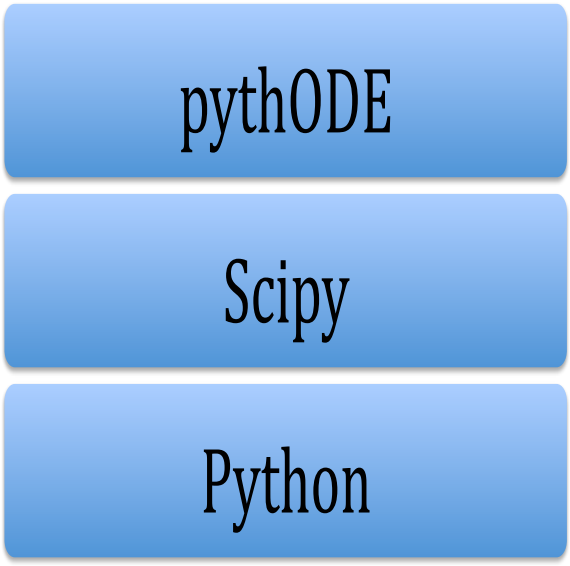

Figure 3.1: The layered architecture of pythODE.

3.4 Design and architecture of the BVP component of `pythODE`

At present, the BVP component of `pythODE` solves systems of first-order ODEs

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}(x)), \quad a < x < b, \tag{3.2a}$$

where $\mathbf{f} : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ and $\mathbf{y} : \mathbb{R} \to \mathbb{R}^m$. The software allows for non-separated, two-point boundary conditions

$$\mathbf{g}(\mathbf{y}(a), \mathbf{y}(b)) = \mathbf{0}, \tag{3.2b}$$

where $\mathbf{g} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m$.

The remainder of this section describes the design and architecture of the BVP component of `pythODE`.

The BVP component of `pythODE` is designed with the goal of completely modularizing the individual numerical algorithms used to solve a BVP. Fortunately, the computational flow of global methods can be neatly divided into individual numerical algorithms. See Figure 3.2 for the computational flow chart. The BVP component of `pythODE` allows user to select which algorithm they wish to use for each stage of the numerical solution process.

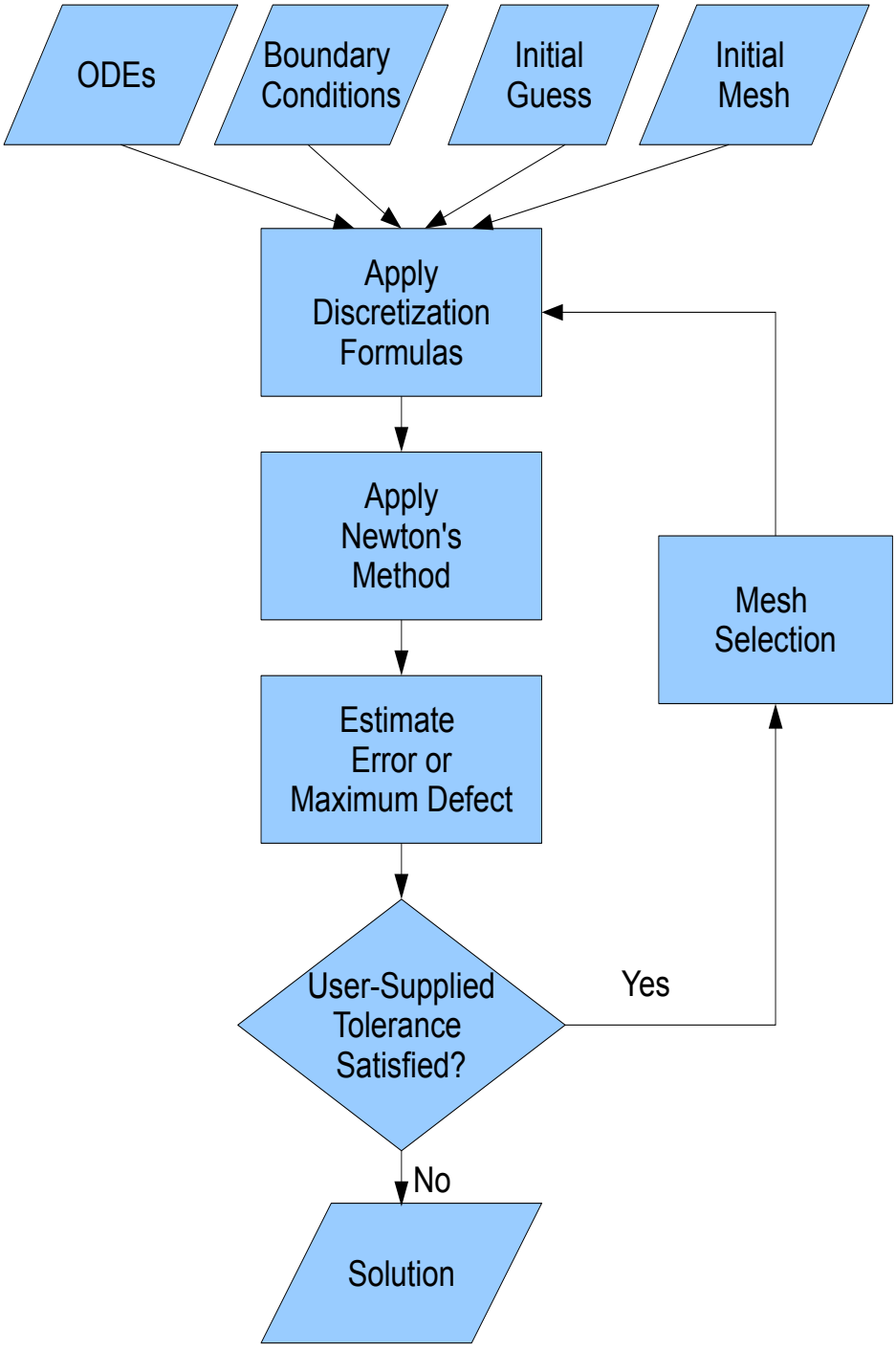

Figure 3.2: Computational flow chart of global methods for the numerical solution of BVPs.

Well-known object-oriented principles are used to achieve the goal of modularization. Each individual numerical algorithm is implemented as a separate class. Each class is required to implement class methods from an abstract class of the numerical algorithm category. By doing so, easy expansion of the PSE without modification of existing code is supported. For example, users who wish to add an error-estimation algorithm can create child class to an abstract error-estimation class. In Figure 3.3 , a BVP solver class loads class instances of all numerical algorithms selected by the user.

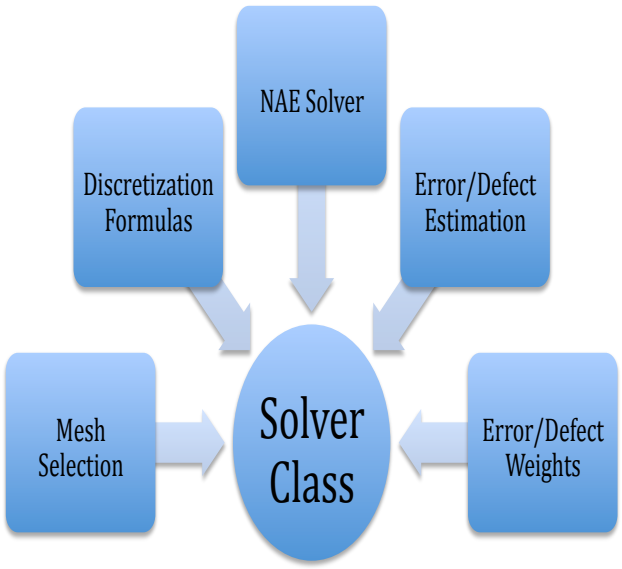

Figure 3.3: Instances of classes loaded by the primary solver class.

Of course, once users add a new numerical algorithm to the BVP component of `pythODE`, they may wish to compare the performance of their algorithm against several existing algorithms. Usually, this involves using the algorithm to solve several test problems. Determining which BVPs are good candidates for test problems often proves to be difficult. The problem must require enough computational time to provide a usable measure of performance. However, instead of forcing users to search for test problems, the BVP component of `pythODE` comes packaged with an automated test suite that consists of a large collection of well-known BVPs. As a consequence, once users implement a new algorithm, they are able to compare the performance of their numerical algorithm immediately against other existing algorithms.

This section concludes by describing each category of the numerical algorithms found in Figure 3.3 and how they relate to rest of the BVP component of `pythODE`. Users are able to choose which algorithms they wish to use for each category. If they do not, the BVP component of `pythODE` selects a default algorithm.

- **Mesh Selection:** Users have the option to choose from different mesh-selection algorithms. The default behaviour of the BVP component of `pythODE` uses two mesh-selection algorithms. The primary mesh-selection algorithm is one that uses information about either the error or the problem to determine a mesh that meets a user-supplied error tolerance. A secondary mesh-selection algorithm is used if adequate information is unavailable for the primary mesh-selection algorithm, e.g., the lack of a useful error estimate. By default, the BVP component of `pythODE` uses mesh equidistribution as the primary mesh-selection algorithm and simply doubles the mesh as the secondary mesh-selection algorithm.

- **Discretization Formulas:** Users are able to select the discretization algorithms used to numerically solve a BVP. In many cases, the discretization algorithm is part of a continuous algorithm, and therefore this algorithm also provides interpolation formulas required for the continuous solution. If the algorithm does not provide a continuous solution, the BVP component of `pythODE` provides users with interpolation options to make the solution continuous. At present, `pythODE` uses a continuous fourth-order MIRK approach as the default discretization algorithm to solve BVPs.

- **NAE Solver:** The BVP component of `pythODE` uses `pythNon` to solve the systems of NAEs generated by the discretization formulas. Because `pythNon` modularizes the numerical routines used to solve systems of NAEs. Users have the ability to customize how the NAE solver functions within the BVP component of `pythODE`.

  For the purpose of this thesis, several modifications were made to `pythNon` to make the NAE solver more consistent with NAE solvers found in other BVP software packages. For example, a damped Newton's algorithm was added to `pythNon`. Also, because most discretization formulas result in almost-block-diagonal Jacobian matrices, an algorithm to evaluate and store these types of Jacobian matrices was added.

- **Error/Defect Estimation:** Users can either estimate the defect or global error. Currently, there are four different algorithms implemented to estimate the global error. These algorithms are based on Richardson extrapolation, higher-order formulas, deferred corrections, and a conditioning constant; see Section 4.1. Once the error is estimated, it can be used for both a termination criterion and mesh selection. By default, the BVP component of `pythODE` estimates a measure of the defect.

- **Error/Defect Weights:** Every known BVP software package reports a estimate of either the relative error or the maximum relative defect. The weights used to scale either the error or the maximum defect differ among BVP software. Therefore, the BVP component of `pythODE` allows users to select the weights they wish to use. The default weights depend on whether

an estimate of the error or maximum defect is being used. If the error is being estimated, then the BVP component of `pythODE` uses

$$\frac{\|\mathbf{y}(x) - \mathbf{S}(x)\|_\infty}{1 + \|\mathbf{S}(x)\|_\infty}, \quad a \leq x \leq b.$$

If the maximum defect is being estimated, then the BVP component of `pythODE` uses

$$\frac{\|\mathbf{S}'(x) - \mathbf{f}(x, \mathbf{S}(x))\|_\infty}{1 + \|\mathbf{f}(x, \mathbf{S}(x)\|_\infty}, \quad a \leq x \leq b.$$

The relative estimate of both the error and the maximum defect are slightly modified from the one used in `BVP_SOLVER` [35].

3.5 Using the BVP component of `pythODE` to solve Bratu's problem

In this section, the user interface of the BVP component of `pythODE` is demonstrated by solving a simple BVP.

The BVP component of `pythODE` is used to solve Bratu's problem,

$$y''(x) + \lambda \exp(y(x)) = 0, \quad 0 < x < 1,$$

where $\lambda = 1$, subject to the boundary conditions,

$$y(0) = y(1) = 0.$$

This particular problem occurs in a model of spontaneous combustion within a slab [33]. To be consistent with the BVP component of `pythODE` problem class (3.2), the problem must first be reformulated. This involves letting

$$y_1(x) = y(x),$$
$$y_2(x) = y'(x),$$

and then defining Bratu's problem as a system of first-order ODEs

$$f_1 = y_2(x),$$
$$f_2 = -\lambda \exp(y_1(x)).$$

The boundary conditions become

$$g_1 = y_1(0),$$
$$g_2 = y_1(1).$$

An initial guess

$$y_1 = x(1-x),$$
$$y_2 = 1 - 2x,$$

is also used [33]. After this quick reformulation, the problem can now be entered into a `Python` text file; see Figure 3.4. The parts of the text file are described below.

As with most `Python` text files, it begins with importing the required modules. In this case, the `Solver` module of the BVP component of `pythODE` is imported. In Figure 3.4, the `Solver` module is renamed `BVPSolver` for the purpose of clarity. If a user wishes to access a function or class located within the `Solver` module, the name `BVPSolver` must be placed in front of the function or class name, separated by a dot. Next, the functions for the ODEs, boundary conditions, and initial-guess are defined. Although the initial guess is defined as a function, the BVP component of `pythODE` allows it to be specified in several forms. For example, a vector of values for both y_1, y_2 can be used. Each function defined in Figure 3.4 returns a `Scipy` array. The index of the array begins at 0, therefore note that $f_1 \equiv$ `f[0]`, etc. The BVP component of `pythODE` imports `Scipy` functions, such as `exp` and `zeros`; see Figure 3.4. As a consequence, users are not required to import any `Scipy` modules. Instead, `Scipy` functions can be accessed in a similar way as other `pythODE` functions.

Up until this point, the functions for the ODES, boundary conditions, and initial guess have been described to the BVP component of `pythODE` in much the same way as many other BVP software packages. However, the function names have not been passed to the BVP component of `pythODE`. Also, none of the additional information about the BVP has been provided, e.g., the boundary points, the number of ODEs, etc. Most BVP software packages accomplish this task through function parameters. Instead, the BVP component of `pythODE` uses a `Python` dictionary to allow users to define the BVP; see Figure 3.4. A dictionary is an array that links keywords to various values of any data type, including function references. One benefit to using a dictionary is that dictionary keys can be in plain text. Therefore, information about the BVP can be sent to `pythODE` in as transparent a manner as possible. The `pythNon` PSE uses a similar interface.

Once the information about the BVP is entered into the dictionary, it can be passed to the `solver` class of the BVP component of `pythODE`; see Figure 3.4. This is accomplished by including the dictionary as the only required parameter for the solver constructor. Afterward, the primary solve method can be run without requiring any further information from the user.

The code in Figure 3.4 shows the minimum amount of information `pythODE` requires to numerically solve a BVP. It should be noted that the user is free to enter additional options into `pythODE`, e.g., the discretization formulas used to solve the BVP, an error tolerance, etc. If a user does not enter these options, the BVP component of `pythODE` uses default options. For example, because a user-supplied tolerance was not provided, the BVP component of `pythODE` uses a tolerance of 10^{-4}.

```
# Import the BVP component of pythODE
import pythODE.BVP.Solver as BVPSolver

# Define functions
def ODEFunction(x,y,f):
        # Define Lambda for Bratu's problem
        BratuLambda = 1.0
        # Define the ODE function
        f[0] = y[1]
        f[1] = -BratuLambda*BVPSolver.exp(y[0])
        return f

def BCFunction(ya,yb,g):
        # Define the boundary condition function
        g[0] = ya[0]
        g[1] = yb[0]
        return g

def guessFunction(x):
        # Define the initial-guess function
         y = BVPSolver.zeros(2) # Returns a size 2 array of zeros
         y[0] = x*(1.0-x)
         y[1] = 1.0-2.0*x
         return y

# Define a Python dictionary for pythODE
BVPinfo = {} # Initialize a python dictionary
BVPinfo['ODE'] = ODEFunction # Pass the name of the ODE function
BVPinfo['BC'] = BCFunction  # Pass the name of the boundary condition function
BVPinfo['Initial guess'] = guessFunction # Pass the name of the initial-guess function
BVPinfo['Number of ODEs'] = 2 # The number of ODEs
BVPinfo['Boundary points'] = [0.0,1.0] # The location of the boundary points

# Solve the BVP
sol = BVPSolver.solver(BVPinfo) # Pass the dictionary into the constructor
SolvedBVPinfo = sol.solve() # Solve the BVP
```

Figure 3.4: Using the BVP component of `pythODE` to solve Bratu's problem.

If a required item of information is not provided to `pythODE`, the PSE returns an error message alerting the user; see Figure 3.5.

```
bvp error --> problem dictionary error --> missing value --> Number of ODEs
```

Figure 3.5: `pythODE` is alerting the user that the dictionary entry 'Number of ODEs' has not been defined.

The PSE returns a `Python` dictionary, called `SolvedBVPinfo` in Figure 3.4, filled with information about the numerical solution. A variety of useful information about the numerical solution is stored in the solution dictionary, e.g., an estimate of the error associated with the numerical solution. The solution dictionary also provides a means for the user to access the continuous numerical solution through the dictionary entry `Evaluate`.

The continuous solution can be used to generate a plot of $y_1(x)$ for $0 \leq x \leq 1$. In Figure 3.6, the function `linspace` is used to generate 30 equally spaced discrete points between 0 and 1. Next, the function `eval` is used to generate a solution at those 30 points. The y_1 component of the solution is stored in an array called `yArray`. The `Python` graphing module `matplotlib` [41] can then be used to plot a graph of the solution for y_1.

The resulting graph is shown in Figure 3.7. It should be noted that this is one of two possible solutions to Bratu's problem. The other solution can be obtained by multiplying both components of the initial guess by a factor of five [33].

```
# Get the function for the continuous numerical solution
eval = SolvedBVPinfo['Evaluate']
# Get the values for x required for a plot, store in an array
xArray = BVPSolver.linspace(0,1,30)
yArray = BVPSolver.zeros(30) # Initialize a solution an array of zeros
for i in range(0,30):
        solution =  eval(mesh[i]) # eval function returns a numerical solution for y
        # Next, fill the solution vector with a numerical solution for y
        yArray[i] = solution[0]

# Use the solution to create a graph
import matplotlib.pyplot as plt # Import matplotlib
plt.plot(xArray,yArray) # Generate the plot of the solution y as a function of x
plt.show() # Show the plot
```

Figure 3.6: Creating a plot of the solution to Bratu's problem.

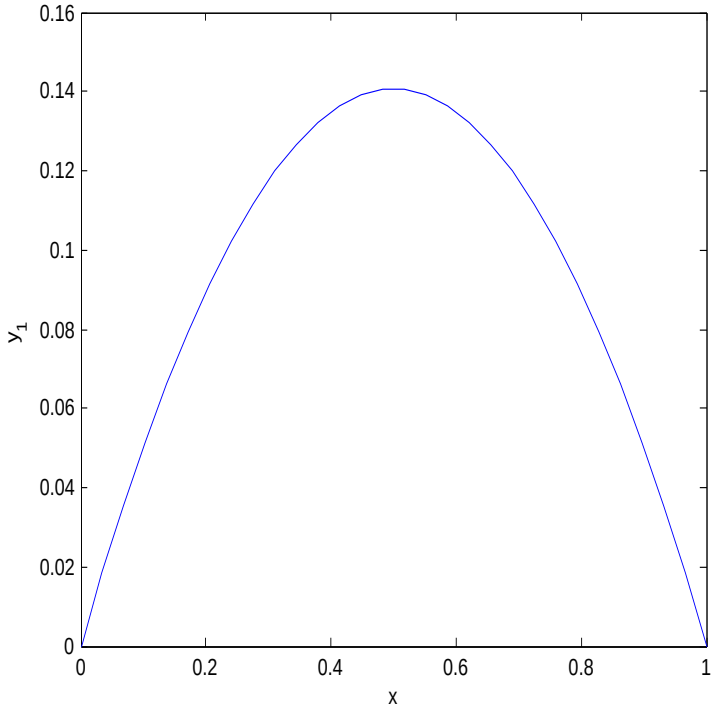

Figure 3.7: Solution y_1 to Bratu's problem.

3.6 Summary

In this chapter, the BVP component of `pythODE` is described. Through careful design decisions, the software package contains all the desired features of a PSE described in Section 3.2. For example, the PSE offers a considerable amount of flexibility to the users. This is accomplished by modularizing the individual numerical algorithms used to solve a BVP. As a consequence, users can essentially design their own global method. The PSE is also highly expandable. By using object-oriented programming principles, researchers can easily add new numerical algorithms to the software. Finally, the PSE is easy to use. The software takes advantage of the `Python` dictionary to allow users to describe the BVP in plain text. To further simplify the interface, the PSE offers a default numerical method for the solution of BVPs; see Section 3.4. In the end, users are only required to enter five items of information, all associated with the mathematical formulation of the problem, in order to numerically solve a BVP.

Chapter 4

Numerical Experiments and Applications

The BVP component of `pythODE` is used as a software platform for two investigations. In Section 4.1, the BVP component of `pythODE` is used to perform a comparison of the performance of four global-error estimation algorithms chosen from the literature [36]. In Section 4.2, `pythODE` is used to solve a newly developed model of the agglomerate of a proton exchange membrane fuel cell (PEMFC).

4.1 Global-error methods

For many BVP software packages, the estimation of the global error is a critical step in the numerical solution process. The software packages `COLSYS` and `COLNEW` use an estimation of the global error to assess the accuracy of the numerical solution. For many problems, BVP software packages must typically estimate the global error many times before an acceptable numerical solution is obtained.

Recall that the BVP software packages `MIRKDC`, `BVP_SOLVER`, and `bvp4c` estimate the defect instead of the global error. In general, users often have a better understanding of global error than the defect. Therefore, it is beneficial for defect-control BVP software packages to provide the users with an optional an *a posteriori* global-error estimate.

There are a number of different algorithms that can be used to estimate the global error effectively. Although there have been studies of different methods to estimate the error for global-error BVP software packages ([29], [36]), few studies have extended the results to defect-control BVP software. With that in mind, the BVP component of `pythODE` can be used as a platform to compare the performance of four known algorithms for estimating the global error in defect-control BVP software packages. These algorithms are based on the use of Richardson extrapolation, higher-order formulas, deferred corrections, and a conditioning constant. Each global-error estimation algorithm is described below.

4.1.1 Richardson extrapolation

Many BVP software packages use *Richardson extrapolation* to estimate the global error [4]. This algorithm starts with a discrete numerical solution $\mathbf{Y}_h$ for a given mesh. Next, the software deter-

mines a more accurate numerical solution $\mathbf{Y}_{h/2}$ by halving each subinterval of the original mesh. Then, an estimate of the norm of the global error, e_{RE}, is given by

$$e_{RE} = \left\| \frac{2^p}{2^p - 1} \left(\mathbf{Y}_h - \mathbf{Y}_{h/2} \right) \right\|_\infty ,$$

where p is the order of the discretization formula.

A system of NAEs must be solved to determine a solution to $\mathbf{Y}_{h/2}$. This requires the use of Newton's method. The original solution $\mathbf{Y}_h$, extended to the new mesh through a continuous MIRK approach, proves to be an effective initial guess. With that in mind, it is assumed that only one undamped Newton iteration is required to return a solution $\mathbf{Y}_{h/2}$ that is sufficiently close to the exact solution of the NAEs [9].

4.1.2 Higher-order formulas

Higher-order formulas can be used to determine a more accurate numerical solution with the same mesh as for the original solution. Specifically, the global error can be estimated by

$$e_{HO} = \| \mathbf{Y}_p - \mathbf{Y}_q \|_\infty ,$$

where $\mathbf{Y}_p$ is the original discrete solution of order p and $\mathbf{Y}_q$ is the more accurate discrete solution of order $q > p$. If symmetric MIRK formulas are used, $q = p + 2$.

A benefit to using a higher-order formula is that the Jacobian matrix evaluated for the original solution proves to be an adequate approximation of the Jacobian matrix required to solve the system of higher-order equations with Newton's method [9]. Therefore, the computationally expensive task of re-evaluating the Jacobian matrix for the error estimation can be avoided [9]. Also, similar to Richardson extrapolation, the original solution $\mathbf{Y}_p$ proves to be an effective initial guess for Newton's method. Therefore, one undamped Newton iteration is used to determine the higher-order solution.

4.1.3 Deferred corrections

The higher-order algorithm requires a system of NAEs generated by a higher-order discretization formula to be solved on the same mesh as the original solution. Alternatively, a more accurate solution can be obtained by solving a system of NAEs that corrects for some of the truncation error associated with the original discretization through the use of *deferred corrections*. A deferred-correction approach exists for MIRK formulas [14]. In this approach, the first correction is given by [14]

$$\mathbf{\Phi}_p(\mathbf{Y}_{p+2}) = -\mathbf{\Phi}_{p+2}(\mathbf{Y}_p), \tag{4.1}$$

where $\mathbf{\Phi}_p$ is the system of NAEs generated by a MIRK formula of order p. In (4.1), $\mathbf{\Phi}_{p+2}$ is the system of NAEs associated with the higher-order formula, $\mathbf{Y}_p$ is the discrete solution associated with $\mathbf{\Phi}_p$, and $\mathbf{Y}_{p+2}$ is the discrete numerical solution associated with $\mathbf{\Phi}_{p+2}$. In order to determine $\mathbf{Y}_{p+2}$, Newton's method is applied to the system

$$\mathbf{\Phi}_p(\mathbf{z}) + \mathbf{\Phi}_{p+2}(\mathbf{Y}_p) = \mathbf{0}, \tag{4.2}$$

where $\mathbf{z}$ is the unknown. Once $\mathbf{z}$ is determined, $\mathbf{Y}_{p+2}$ is assigned the value of $\mathbf{z}$. A global-error estimate of the form

$$e_{DC} = \|\mathbf{Y}_p - \mathbf{Y}_{p+2}\|_\infty,$$

can then be obtained.

An advantage of this approach is that the system of equations (4.2) has the same Jacobian matrix, evaluated at $\mathbf{Y}_p$, as the original solution. Therefore, the Jacobian matrix does not have to be re-evaluated in order to apply Newton's method on (4.2). Similar to the previous algorithms, $\mathbf{Y}_p$ proves to be an effective initial guess for Newton's method. Thus, one undamped Newton iteration can be used to determine the higher-order solution.

4.1.4 Conditioning constant based algorithm

In Chapter 2, an expression (2.17) for the global error $\mathbf{e}(x)$ of a linear BVP (2.1) is described. Recall from Section 2.2 that the general solution can be used to define a conditioning constant κ. In this section, a form of κ that can be easily computed for the purposes of global-error estimation is described.

Most BVP software packages use a scaled norm for global-error estimation. In order to be consistent with those software packages, diagonal weight matrices $\mathbf{W}_1(x)$, $\mathbf{W}_2$, $\mathbf{W}_3(x) \in \mathbb{R}^{N\times N}$ are applied to (2.17) resulting in

$$\mathbf{W}_3^{-1}(x)\mathbf{e}(x) = (\mathbf{W}_3^{-1}(x)\mathbf{\Theta}\mathbf{W}_2)(\mathbf{W}_2^{-1}\boldsymbol{\sigma}) + \int_a^b (\mathbf{W}_3^{-1}(x)\mathbf{G}(x,t)\mathbf{W}_1(t))(\mathbf{W}_1^{-1}(t)\mathbf{r}(t))dt.$$

Taking the norms of both sides results in

$$\|\mathbf{e}(x)\|_{\mathbf{W}_3} \leq \kappa \max(\|\|\boldsymbol{\sigma}\|_{\mathbf{W}_2}, \mathbf{r}(x)\|_{\mathbf{W}_1}), \tag{4.3}$$

where

$$\|\mathbf{e}(x)\|_{\mathbf{W}_3} = \max_{a\leq x\leq b} \|\mathbf{W}_3^{-1}(x)\mathbf{e}(x)\|_\infty, \quad \|\boldsymbol{\sigma}\|_{\mathbf{W}_2} = \|\mathbf{W}_2^{-1}\boldsymbol{\sigma}\|_\infty, \ \|\mathbf{r}(x)\|_{\mathbf{W}_1} = \max_{a\leq x\leq b} \|\mathbf{W}_1^{-1}(x)\mathbf{r}(x)\|_\infty,$$

and

$$\kappa = \max_{a \le x \le b} \left(\int_a^b \|\mathbf{W}_3^{-1}(x)\mathbf{G}(x,t)\mathbf{W}_1(t)\|_\infty dt + \|\mathbf{W}_3^{-1}(x)\mathbf{\Theta}\mathbf{W}_2\|_\infty \right).$$

When compared to the conditioning constant (2.20), this particular κ is better suited for numerical computations because for a sufficiently fine mesh,

$$\kappa \approx \left\| \mathbf{U}_3^{-1} \frac{\partial \mathbf{\Phi}(\mathbf{Y})}{\partial \mathbf{Y}}^{-1} \mathbf{U}_{12} \right\|_\infty,$$

where $\mathbf{U}_{12} = \text{diag}\{\mathbf{W}_1(x_1), \ldots, \mathbf{W}_1(x_N), \mathbf{W}_2\}$ and $\mathbf{U}_3 = \text{diag}\{\mathbf{W}_3(x_0), \ldots, \mathbf{W}_3(x_N)\}$ [34]. Therefore, κ can be quickly estimated by making use of the factored Jacobian matrix $\partial\mathbf{\Phi}(\mathbf{Y})/\partial\mathbf{Y}$ already evaluated by the numerical method used to determine a solution to the BVP. The computation of κ can be made even more efficient by making use of the Higham–Tisseur algorithm [18] for the estimation of the matrix norm.

After evaluating the conditioning constant, an estimate for the bound of the global error is given by

$$e_{CO} = \kappa \max(\|\boldsymbol{\sigma}\|_{\mathbf{W}_2}).$$

It is worth noting that this algorithm is especially convenient for defect-control BVP software packages. These packages determine the norm of the defect during the numerical solution process of the BVP. As a consequence, the only additional cost to this global-error estimation algorithm is estimating the conditioning constant.

4.1.5 Adding the global-error estimation algorithms to `pythODE`

For the purpose of this thesis, the global-error estimation algorithms based on Richardson extrapolation, higher-order formulas, and deferred corrections were implemented as `Python` modules and added to the BVP component of `pythODE`. The algorithms are similar to that of the `Fortran 95` implementation of all three algorithms used in `BVP_SOLVER` [9]. However, they have been slightly modified to fit within the modularized framework of `pythODE`.

On the other hand, the global-error estimation algorithm based on a conditioning constant was added to the BVP component of `pythODE` by creating a `Python` to `Fortran` interface for an existing `Fortran` implementation of the algorithm [34]. As a consequence, it is believed that the module performs better than one created purely in `Python` [10]. However, when using the `Fortran` module within `pythODE`, additional computational time must be spent converting a `Scipy` data type that holds the factored Jacobian matrix into a `Fortran` array.

4.1.6 Test problems

This section describes three test problems used to compare of both the accuracy and runtime of the different global-error estimation algorithms. Each test problems is solved by using a MIRK formula of order two, four, and six with defect control. The global-error estimate is used as an *a posteriori* error estimate. A range of tolerance values 10^{-4}, 10^{-5}, ..., 10^{-8} is used to solve the problem.

1. The first problem [11] is

$$\epsilon y''(x) + (y'(x))^2 = 1, \quad 0 < x < 1, \tag{4.4a}$$

subject to the boundary conditions

$$y(0) = 1 + \epsilon \ln \cosh\left(\frac{-0.745}{\epsilon}\right), \quad y(1) = 1 + \epsilon \ln \cosh\left(\frac{0.255}{\epsilon}\right), \tag{4.4b}$$

with the an exact solution

$$y(x) = 1 + \epsilon \ln \cosh\left(\frac{x - 0.745}{\epsilon}\right).$$

An initial guess of $y(x) \equiv 1$, $y'(x) \equiv 0$ is used. To help achieve measurable timings, the value of ϵ for each MIRK order used was varied. For MIRK order two, four and six, $\epsilon = 0.08$, 0.03, and 0.025 respectively. To further help achieve measurable timings, the problem was solved 20 times.

2. The second problem [11] is

$$\epsilon y''(x) = y(x) + y(x)^2 - \exp\left(\frac{-2x}{\sqrt{x}}\right), \quad 0 < x < 1, \tag{4.5a}$$

subject to the boundary conditions

$$y(0) = 1, \quad y(1) = \exp\left(\frac{-1}{\sqrt{\epsilon}}\right), \tag{4.5b}$$

with an exact solution

$$y(x) = \exp\left(\frac{-x}{\sqrt{\epsilon}}\right).$$

An initial guess of $y(x) \equiv 1/2$ and $y'(x) \equiv 0$ is used. Similar to the first problem, the value of ϵ for each MIRK order was varied to help achieve measurable timings. For MIRK order two, $\epsilon = 10^{-2}$. For MIRK order four and six, $\epsilon = 10^{-5}$. The problem is solved 10 times.

3. The third problem [4, Example 1.20] is

$$\epsilon f''''(x) + f(x)f'''(x) + g(x)g'(x) = 0, \quad 0 < x < 1, \tag{4.6a}$$

subject to the boundary conditions

$$f(0) = f(1) = f'(0) = f'(1) = 0, \quad g(0) = \Omega_0, \quad g(1) = \Omega_1. \tag{4.6b}$$

An exact solution for this problem is not known. A reference solution is generated by the BVP component of `pythODE` using a sixth-order MIRK formula and a tolerance of 10^{-12}. For the numerical experiment, $\Omega_0 = -1$ and $\Omega_1 = 1$. The initial guess $g(x) = 2x - 1$, $g'(x) = 2$, and $f(x) \equiv f'(x) \equiv f''(x) \equiv f'''(x) \equiv 0$ is used. The value of ϵ is allowed to vary with the order of MIRK formula to help achieve measurable timings. For MIRK order two, $\epsilon = 0.1$. For MIRK order four and six, $\epsilon = 0.01$. This problem is solved 20 times.

4.1.7 Numerical results

This section describes the results of the numerical experiments. All computations were performed using a 2.6 GHz Intel Core 2 Duo with 4 GB DDR2 RAM running at 667 MHz. The operating system was `Mac OS X` 10.5. The `pythODE` PSE was run using `python` 2.5. The times, in seconds, reported are cumulative over all runs. Below, the results are described based on the order of the MIRK formula used to solve the test problems. For select test problems, relative execution time of the global-error estimation algorithms for MIRK order two, four, and six are shown in Figures 4.2, 4.2, and 4.3 respectively. Relative execution time refers to the percent of the overall solution time dedicated to the global-error estimation algorithm. Detailed tables of the results can be found in the Appendix.

Results for second-order MIRK formula

When using a second-order MIRK formula, the results for all test problems show excellent agreement between the true global error and e_{RE}, e_{HO}, and e_{DC}. There is only a negligible difference between e_{HO} and e_{DC}. The algorithm based on a conditioning constant, however, gives a substantial over-estimate of the global error by several orders of magnitude.

For test problem (4.4), Richardson extrapolation costs between 25% and 28%. In contrast, both the higher-order and deferred-correction algorithms cost between 4% and 6%. The cost of the algorithm based on the conditioning constant is around 3%. For test problem (4.5), Richardson extrapolation costs between 24% and 28%; see Figure 4.1. Both the higher-order and deferred-correction algorithms cost between 4% and 8%. The cost of the algorithm based on a conditioning constant is around 3%. Finally, for test problem (4.6), Richardson extrapolation costs between

37% and 42%. In contrast, both the higher-order and deferred-correction algorithms cost between 1% and 3%. The algorithm based on the conditioning constant cost around 6%. In this case, the algorithm based on conditioning constant is slower than both the higher-order and deferred-correction algorithms due to the additional computational time required to convert a large `Scipy` data structure, which holds a factored Jacobian matrix, into a `Fortran` array.

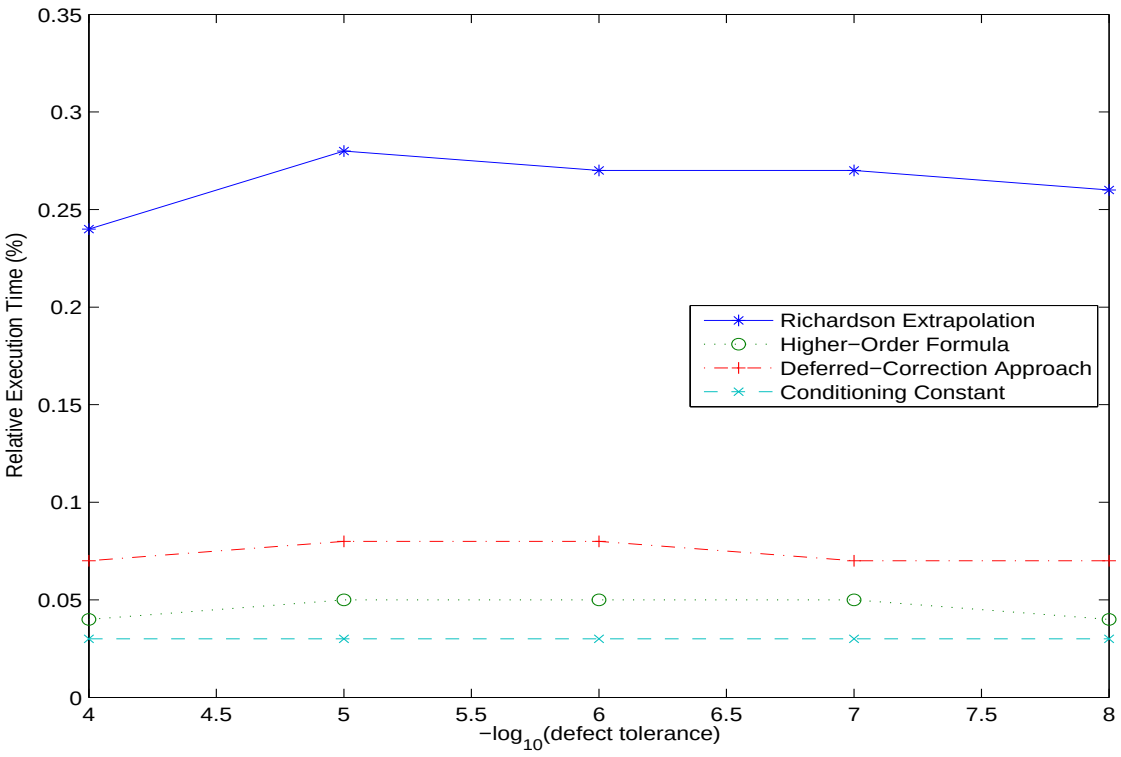

Figure 4.1: Relative execution time of the global error estimates as a function of tolerance for problem (4.5) when using a second-order MIRK formula.

Results for fourth-order MIRK formula

Similar accuracy results are achieved when using a fourth-order MIRK formula. There is excellent agreement between the true global error and e_{RE}, e_{HO}, and e_{DC}. There is only a negligible difference between e_{HO} and e_{DC}. The algorithm based on a conditioning constant gives a substantial over-estimate of the global error by several orders of magnitude.

For test problem (4.4), Richardson extrapolation costs between 11% and 23%; see Figure 4.2. In contrast, both the higher-order and deferred-correction algorithms cost between 3% and 8%. The cost of the algorithm based on a conditioning constant is around 1%. For test problem (4.5), Richardson extrapolation cost between 6% and 12%. Both the higher-order and deferred-correction algorithms cost between 2% and 4%. The cost of the algorithm based on a conditioning constant

is around 1%. Finally, for test problem (4.6), Richardson extrapolation costs between 31% and 39%. In contrast, both higher-order and deferred-correction algorithms cost between 2% and 3%. In this case, the algorithm based on a conditioning constant cost around 5% due to the additional computational time required to convert a large `Scipy` data structure to a `Fortran` array.

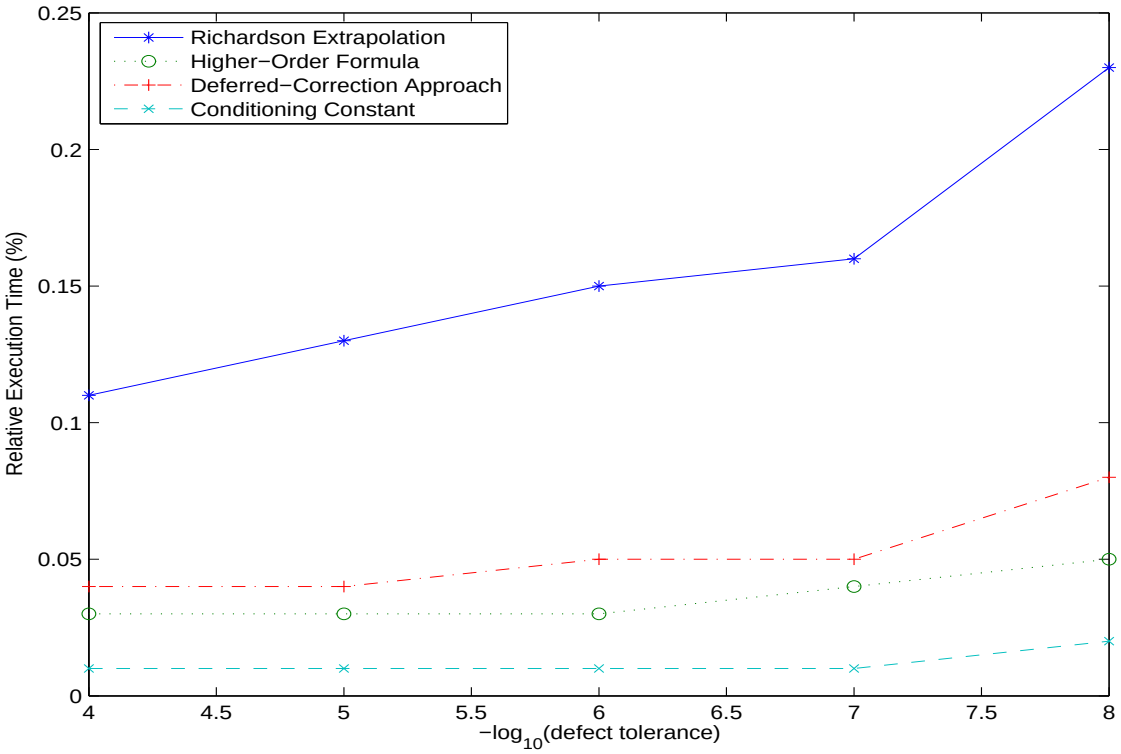

Figure 4.2: Relative execution time of the global error estimates as a function of tolerance for problem (4.4) when using a fourth-order MIRK formula.

Results for sixth-order MIRK formula

Similar accuracy results are achieved when using a sixth-order MIRK formula. There is excellent agreement between the true error and e_{RE}, e_{HO}, and e_{DC}. There is only a negligible difference between e_{HO} and e_{DC}. The algorithm based on a conditioning constant gives a substantial over-estimate of the global error by several orders of magnitude.

For test problem (4.4), Richardson extrapolation costs between 7% and 14%. Both the higher-order and deferred-correction algorithms cost between 3% and 6%. The cost of the algorithm based on a conditioning constant is around 1%. For test problem (4.5), Richardson extrapolation costs between 6% and 8%. Both the higher-order and deferred-correction algorithms cost between 2% and 3%. The cost of the algorithm based on a conditioning constant is negligible. Finally, for test problem (4.6), Richardson extrapolation costs between 25% and 29%; see Figure 4.3. In contrast,

both the higher-order and deferred-correction algorithms cost between 2% and 4%. The algorithm based on the conditioning constant costs around 3%.

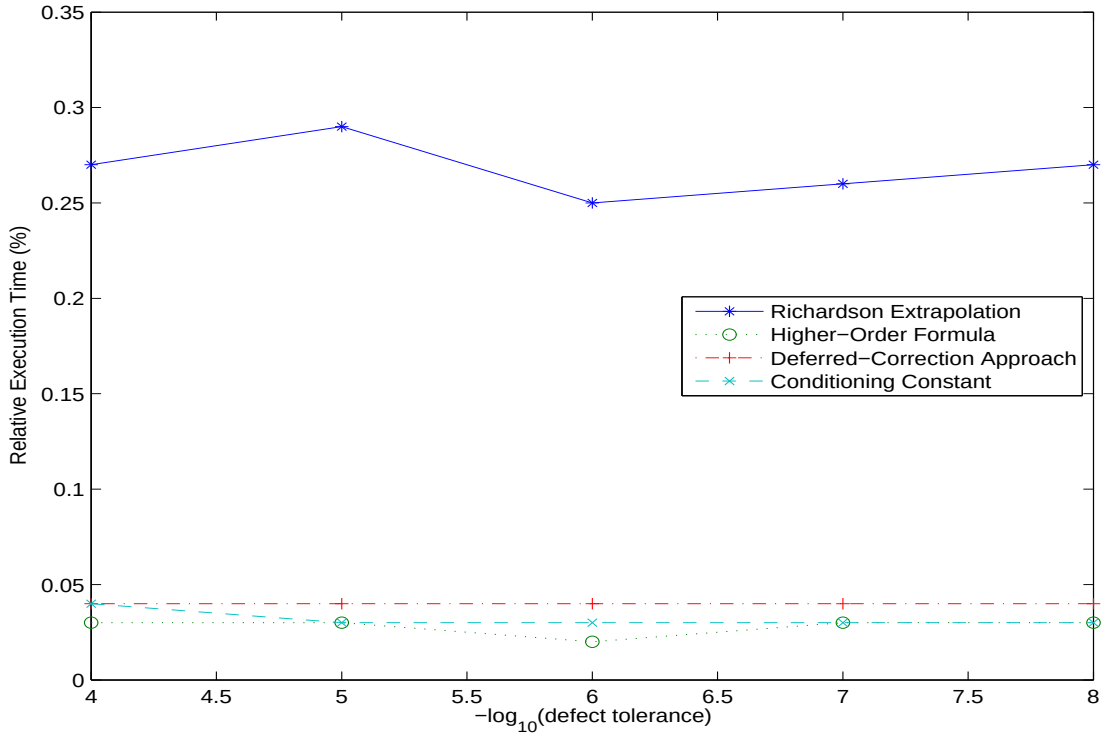

Figure 4.3: Relative execution time of the global error estimates as a function of tolerance for problem (4.6) when using a sixth-order MIRK formula.

4.1.8 Conclusions

Several conclusions can be made from the results of the numerical experiments:

1. If a factored Jacobian is available, an *a posteriori* global-error estimate for defect control BVP software packages should be based on either a higher-order or a deferred-correction algorithm as opposed to Richardson extrapolation. In fact, our results indicate that global-error estimation based on higher-order formulas slightly outperforms deferred corrections.

2. These results may also be applicable to global-error control software packages that use Richardson extrapolation, e.g., `COLSYS`. Instead of using Richardson extrapolation, an improvement in the performance of these software packages may be obtained by either implementing a higher-order or deferred-correction approach to global-error estimation.

3. The algorithm based on the conditioning constant provides a poor estimate of the global error because the estimate of the conditioning constant does not provide a tight upper bound in

(4.3) [9]. However, if desired a better estimate of the conditioning constant may be obtained by using

$$\frac{\|\mathbf{y}(x) - \mathbf{S}(x)\|_{\mathbf{W}_3}}{\max(\|\mathbf{r}(x)\|_{\mathbf{W}_1}, \|\boldsymbol{\delta}\|_{\mathbf{W}_2})} \leq \kappa,$$

where either e_{HO} or e_{DC} is used to estimate $\|\mathbf{y}(x) - \mathbf{S}(x)\|_{\mathbf{W}_3}$.

Similar conclusions are made for a `BVP_SOLVER` implementation of the same global-error estimation algorithms [9].

4.2 Multi-scale agglomerate model for PEMFCs

In this section, the ability of the BVP component of `pythODE` to solve a complex real-world BVP is demonstrated. The problem consists of a newly developed model of the agglomerate located inside of a proton exchange membrane fuel cell (PEMFC).

A PEMFC uses hydrogen to produce electrical energy with water vapour as the primary by-product. The low environmental impact makes PEMFCs a desirable alternative to conventional combustion engines for automobiles [22].

The design of a PEMFC facilitates a series of electrochemical reactions. In the simplest sense, a PEMFC can be divided into an anode side and a cathode side separated by a membrane; see Figure 4.4. Within the anode side, hydrogen is separated into electrons and protons. The electrons provide electrical energy, and the protons travel through a membrane and into a catalyst layer located within the cathode side. In the catalyst layer, protons combine with electrons and oxygen to produce water vapour. In order for oxygen to reach the catalyst layer, the oxygen travels through a gas diffusion layer, also located within the cathode side, composed of carbon fibres.

For this particular model, it is assumed that the catalyst layer is composed of ionomer-filled spherical agglomerates [30]. Each agglomerate is composed of carbon and platinum particles. Multiple agglomerates are surrounded and bonded by electrolyte composed mostly of Nafion. Oxygen that reaches the catalyst layer dissolves into the electrolyte surrounding the agglomerate. Oxygen is then transported to a reaction site within the agglomerate by diffusion; see Figure 4.5.

This section is mostly concerned with a model of the agglomerate. It can be used to determine the concentration of oxygen $[O_2]$ and the ionic potential ϕ_m throughout the agglomerate. The model consists of a variety of constants and design parameters, brief descriptions of which can be found in Table 4.1 [30]. The values for the design parameters can be found in Section 4.2.2.

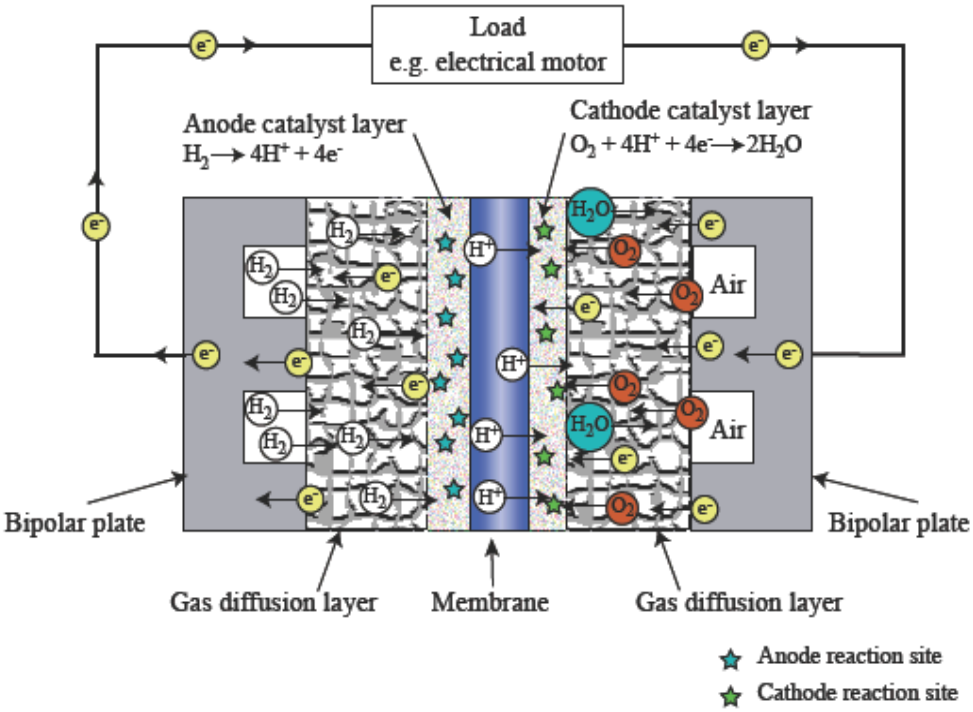

Figure 4.4: A two-dimensional cross-sectional view of a PEMFC. [30]

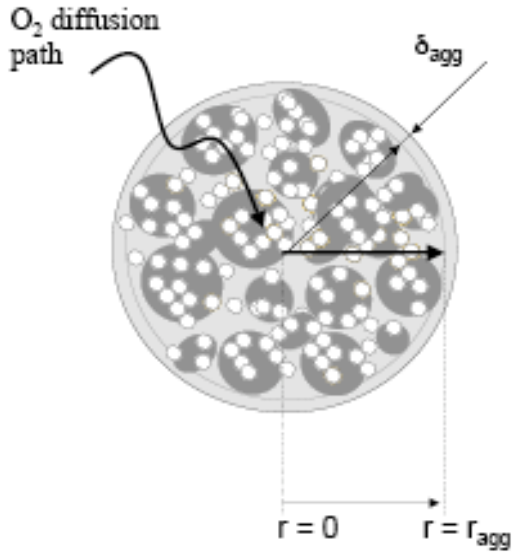

Figure 4.5: Agglomerate of the catalyst layer of a PEMFC. [31]

Constants	
R	gas constant, 8.214 J/(mol $\cdot$ K)
F	Faraday constant, 96500 C/mol
$H_{O_2,N}$	Henry's law coefficient for oxygen in Nafion, 0.3125 atm $\cdot$ m^3/mol
$D_{O_2,N}$	Diffusion of oxygen through Nafion, 8.75×10^{-10} m^2/s
Operating Conditions	
T	operating temperature
$P_{O_2,0}$	partial pressure of oxygen (assumed constant throughout agglomerate)
ϕ_s	solid phase potential
E_{th}	theoretical cell voltage
$\phi_{m,0}$	exchange ionic potential
Electrochemical Parameters	
γ	reaction order for oxygen
α_c	cathodic reaction transfer coefficient
A_v	specific reaction surface area per volume of catalyst layer
i_0^{ref}	reference exchange current density
$[O_2]^{\text{ref}}$	reference concentration of oxygen
σ_m^{eff}	effective electrolyte conductivity
Structural Parameters	
ϵ_V^{CL}	void volume fraction in the catalyst layer
r_{agg}	radius of the agglomerate
δ_{agg}	electrolyte film surrounding the agglomerate
ϵ_{agg}	volume fraction of the ionomer inside the agglomerate

Table 4.1: Constants and design parameters for agglomerate model.

A detailed derivation of the model is beyond the scope of this thesis; see Secanell [31] and Secanell *et al.* [32] for details. Assuming azimuthal symmetry, the model consists of a system of two second-order ODEs

$$D^{\text{eff}}(r)\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d}{dr}[O_2](r)\right) = \frac{j(r)}{4F}, \tag{4.7a}$$

$$\sigma_m^{\text{eff}}\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d}{dr}\phi_m(r)\right) = j(r), \quad 0 < r < r_{\text{agg}} + \delta_{\text{agg}}, \tag{4.7b}$$

where

$$j(r) = \begin{cases} \frac{A_v}{1-\epsilon_V^{CL}} i_0^{\text{ref}} \left(\frac{[O_2](r)}{[O_2]^{\text{ref}}}\right)^{\gamma} \exp\left(\frac{\alpha_c F}{RT}\left(E_{\text{th}} - (\phi_s - \phi_m(r))\right)\right), & \text{if } r \in [0, r_{\text{agg}}), \\ 0, & \text{otherwise,} \end{cases}$$

and

$$D^{\text{eff}}(r) = \begin{cases} \epsilon_{\text{agg}} \cdot D_{O_2,N}, & \text{if } r \in [0, r_{\text{agg}}), \\ D_{O_2,N}, & \text{otherwise.} \end{cases}$$

The system of ODEs is subject to four boundary conditions

$$[O_2](r_{\text{agg}} + \delta_{\text{agg}}) = \frac{p_{O_2,0}}{H_{O_2,N}},$$

$$\frac{d}{dr}[O_2](0) = 0,$$

$$\phi_m(r_{\text{agg}} + \delta_{\text{agg}}) = \phi_{m,0},$$

$$\frac{d\phi_m}{dr}(0) = 0.$$

4.2.1 Problem reformulation

The model presents several issues that may prevent numerical software from successfully solving the BVP. For example, both r_{agg} and δ_{agg} are typically small, e.g., r_{agg} is typically around 2.5×10^{-7}m. As a consequence, the problem must be re-scaled to avoid potential problems caused by the small solution domain. At $r = r_{\text{agg}}$, the definitions of both j and D^{eff} change. Also, (4.7) has a $2/r$ coefficient that is unbounded as r approaches zero. In this section, the problem is reformulated to allow numerical software, including the BVP component of `pythODE`, to solve this problem despite these issues.

First, the problem is divided into two systems of ODEs in order to avoid the lack of smoothness at $r = r_{\text{agg}}$ [16, Example 3.5.8]. One system is defined on the domain $[0, r_{\text{agg}}]$ and the other is defined on the domain $[r_{\text{agg}}, r_{\text{agg}} + \delta_{\text{agg}}]$.

The first system of ODEs is defined for $r \in [0, r_{\text{agg}}]$. The domain is also re-scaled to $[0, 1]$ by

letting

$$r(x) = x \cdot r_{\text{agg}}. \tag{4.8}$$

Using (4.8) in (4.7) and performing some algebraic manipulation results in

$$\frac{d^2}{dx^2}[O_2]_1(x) = \frac{j(x) r_{\text{agg}}^2}{4FD^{\text{eff}}(x)} - \frac{2}{x}\frac{d}{dx}[O_2]_1(x), \tag{4.9a}$$

$$\frac{d^2}{dx^2}\phi_{m,1}(x) = \frac{j(x) r_{\text{agg}}^2}{\sigma_m^{\text{eff}}} - \frac{2}{x}\frac{d}{dx}\phi_{m,1}(x), \tag{4.9b}$$

where $[O_2]_1$ is the concentration of oxygen on $[0, r_{\text{agg}}]$ and $\phi_{m,1}$ is the ionic potential on $[0, r_{\text{agg}}]$.

The second system of ODEs is defined for $r \in [r_{\text{agg}}, r_{\text{agg}} + \delta_{\text{agg}}]$. The domain is also re-scaled to $[0, 1]$. Therefore, let

$$r(x) = x \cdot \delta_{\text{agg}} + r_{\text{agg}}. \tag{4.10}$$

Using (4.10) in (4.7) and performing some algebraic manipulation results in

$$\frac{d^2}{dx^2}[O_2]_2(x) = \frac{j(x)\delta_{\text{agg}}^2}{4FD^{\text{eff}}(x)} - \frac{2\delta_{\text{agg}}}{x \cdot \delta_{\text{agg}} + r_{\text{agg}}}\frac{d}{dx}[O_2]_2(x), \tag{4.11a}$$

$$\frac{d^2}{dx^2}\phi_{m,2}(x) = \frac{j(x)\delta_{\text{agg}}^2}{\sigma_m^{\text{eff}}} - \frac{2\delta_{\text{agg}}}{x \cdot \delta_{\text{agg}} + r_{\text{agg}}}\frac{d}{dx}\phi_{m,2}(x), \tag{4.11b}$$

where $[O_2]_2$ is the concentration of oxygen on $[r_{\text{agg}}, r_{\text{agg}} + \delta_{\text{agg}}]$ and $\phi_{m,2}$ is the ionic potential on $[r_{\text{agg}}, r_{\text{agg}} + \delta_{\text{agg}}]$.

To ensure continuity at $r = r_{\text{agg}}$ between the systems of ODEs (4.9) and (4.11), the ODEs are solved as one large system with the added non-separated boundary conditions

$$[O_2]_1(1) = [O_2]_2(0), \tag{4.12a}$$

$$\frac{1}{r_{\text{agg}}}\frac{d}{dx}[O_2]_1(1) = \frac{1}{\delta_{\text{agg}}}\frac{d}{dx}[O_2]_2(0), \tag{4.12b}$$

$$\phi_{m,1}(1) = \phi_{m,2}(0), \tag{4.12c}$$

$$\frac{1}{r_{\text{agg}}}\frac{d}{dx}\phi_{m,1}(1) = \frac{1}{\delta_{\text{agg}}}\frac{d}{dx}\phi_{m,2}(0). \tag{4.12d}$$

Conditions (4.12) ensures C^1 continuity through the internal point r_{agg}.

The problem has been reformulated to deal with both the small scale and the lack of smoothness at $r = r_{\text{agg}}$ This section is concluded by addressing the singularity at $r = 0$. Note that as r approaches zero [4, Example 1.5],

$$\frac{2}{r}\frac{d}{dr}[O_2]_1 \to 2\frac{d^2}{dr^2}[O_2]_1,$$
$$\frac{2}{r}\frac{d}{dr}\phi_{m,1} \to 2\frac{d^2}{dr^2}\phi_{m,1}.$$

Applying the transformation to (4.9) when $x = 0$ results in the following ODEs

$$\frac{d^2}{dx^2}[O_2]_1 = r_{\text{agg}}^2 \frac{j(x)}{12FD^{\text{eff}}(x)}, \tag{4.13a}$$

$$\frac{d^2}{dx^2}\phi_{m,1} = r_{\text{agg}}^2 \frac{j(x)}{3\sigma_m^{\text{eff}}}. \tag{4.13b}$$

The system of ODEs (4.13) is used when $x = 0$. When $x \neq 0$, the systems of ODEs (4.9) and (4.11) are used.

4.2.2 Solving the agglomerate model with `pythODE`

The BVP component of `pythODE` is used to solve the agglomerate model. In order to solve the BVP, the following algorithms are used. For the mesh selection algorithm, mesh equidistribution is used. A numerical solution is obtained by using a sixth-order continuous MIRK approach. The software package only returns the solution when an estimate of the global error is less than the user-supplied tolerance of 10^{-6}. The global error is estimated by deferred corrections. An initial guess based on experimental data is not presently available. Instead, a constant initial guess that agrees with the boundary conditions is used,

$$[O_2]_1 \equiv \frac{P_{O_2,0}}{H_{O_2,N}}, \qquad \frac{d}{dx}[O_2]_1 \equiv 0, \qquad \phi_{m,1} \equiv \phi_{m,0}, \qquad \frac{d}{dx}\phi_{m,1} \equiv 0,$$

$$[O_2]_2 \equiv \frac{P_{O_2,0}}{H_{O_2,N}}, \qquad \frac{d}{dx}[O_2]_2 \equiv 0, \qquad \phi_{m,2} \equiv \phi_{m,0}, \qquad \frac{d}{dx}\phi_{m,2} \equiv 0,$$

for $0 < x < 1$.

In order to solve the problem, typical PEMFC design parameters are used [31]. The value of each parameter can be found in Table 4.2.

Figure 4.6 shows the resulting concentration of oxygen throughout the agglomerate. As the centre of the agglomerate is approached, the concentration of oxygen drops. This is likely caused by an increase in the number of reactions that take place as oxygen diffuses towards the centre of the agglomerate. Figure 4.7 shows the resulting ionic potential. The ionic potential σ_m shows a slight increase as r approaches $r_{\text{agg}} + \delta_{\text{agg}}$.

The agglomerate model, with the design parameters indicated in Table 4.2, was also solved with the BVP software packages `COLNEW`, `BVP_SOLVER`, and `bvp4c`. The solutions from all three BVP software packages agree with the solution obtained from the BVP component of `pythODE`. It should also be noted that none of the mentioned BVP software packages, including the BVP component of `pythODE`, are able to obtain a solution for $\phi_s < 0.4$ with the initial guess as described.

Operating Conditions	
T	353 K
P_{O_2}	0.5 atm
ϕ_s	0.6 V
E_{th}	1.23 V
$\phi_{m,0}$	-0.2 V
Electrochemical Parameters	
γ	0.5
α_c	1.0
A_v	0.2 Pt/m
i_0^{ref}	2.47×10^{-4} A/m$^2 \cdot$ Pt
$[O_2]^{\text{ref}}$	34.52 mol/m^3
σ_m^{eff}	1.0 S/m
Structural Parameters	
ϵ_V^{CL}	0.6
r_{agg}	2.5×10^{-7} m
δ_{agg}	2.5×10^{-8} m
ϵ_{agg}	0.5

Table 4.2: Parameter values of the agglomerate model.

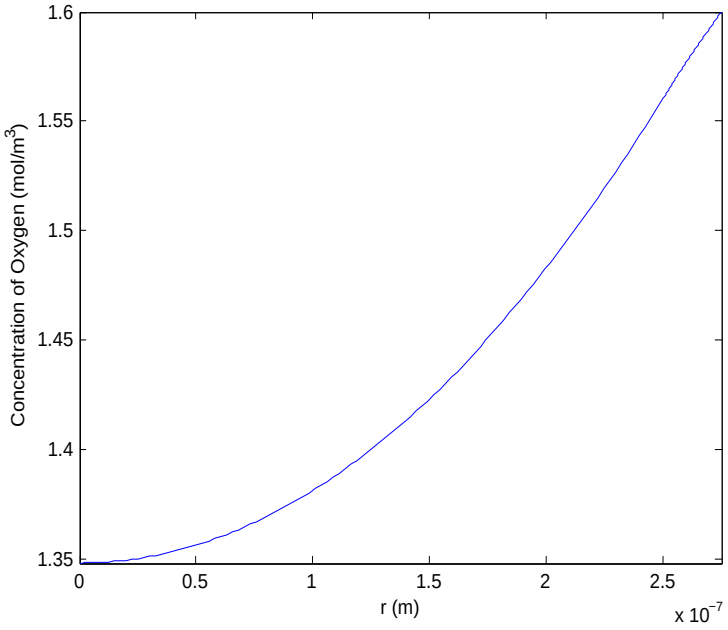

Figure 4.6: Concentration of oxygen $[O_2]$ in the agglomerate.

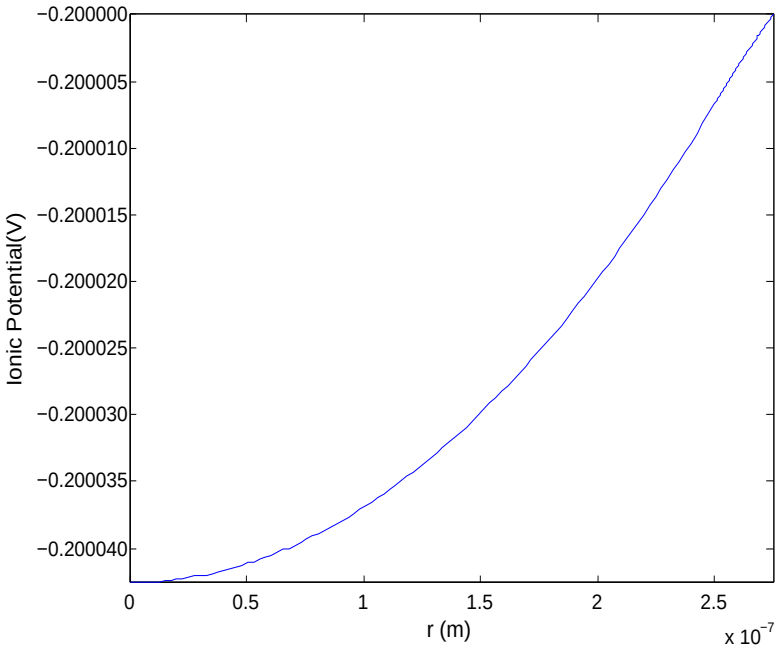

Figure 4.7: Ionic potential ϕ_m in the agglomerate.

4.2.3 Summary

The BVP component of `pythODE` is used to successfully solve a complex real-world problem. Ultimately, the agglomerate model will be added to the PEMFC simulator `FCST` [30]. The simulator itself would be responsible for supplying the parameters used for the model. To solve the agglomerate model within the simulator, a BVP software package capable of solving the model for a wide range of parameters must be added to `FCST`.

Chapter 5

Conclusions And Future Work

Numerically solving a BVP is a complex process that involves the combination of a series of numerical methods. As seen in Chapter 2, there are a variety of choices for each numerical method used to solve a BVP.

In this thesis, a BVP component to the PSE `pythODE` is presented. This component is the first known BVP software package that completely modularizes the solution process of a BVP. As a direct consequence, researchers have the flexibility to create their own solution process from a catalogue of numerical algorithms. The modularization of the numerical methods to solve BVPs is largely achieved by the use of well-known object-oriented programming principles. By doing so, researchers can easily add new numerical algorithms into the PSE without changing existing code. Once a numerical algorithm is added to the PSE, the effects can be immediately evaluated with an automated test suite. As a consequence, the BVP component of `pythODE` has the potential to become a powerful research tool.

The BVP component of `pythODE` is also used to perform two research investigations. First, the PSE is used to compare both the runtime and accuracy of four known global-error estimation algorithms. One algorithm, based on Richardson extrapolation, is used in existing BVP software packages. The three other algorithms are based on the use of higher-order formulas, deferred corrections, and a conditioning constant. Through numerical experimentation, it can be shown that an approach based on higher-order formulas and deferred corrections are computationally faster than Richardson extrapolation while having similar accuracy. Second, the BVP component of `pythODE` is used to solve a newly developed model of a agglomerate in a PEMFC.

This thesis is concluded with several suggestions of future work:

1. At present, users of `pythODE` require a certain amount of programming knowledge to use the code interface. To greatly reduce this requirement, a graphical user interface (GUI) could be added to the PSE. By design, the layered architecture of `pythODE` allows for the easy implementation of a GUI without the need to modify the existing code of the PSE.

2. The BVP component of `pythODE` may also be used to experiment with methods to preserve the known *positivity property* of a solution to a BVP, i.e., ensure that $y(x) \geq 0$ for $a \leq x \leq b$. It is possible for BVP software packages to return a solution that has negative values despite

a known positivity property. The error from the Newton's method used to solve the NAEs generated during the solution process is one possible cause of this. Therefore, different variants of Newton's methods that preserve the known positivity of numerical solutions may be able to help avoid this issue. Because `pythODE` easily allows for the addition of new variants of Newton's methods, it is an ideal platform for this form of research.

3. The problem class of `pythODE` may be increased to include boundary value DAEs. Systems of DAEs are composed of ODEs and generally nonlinear algebraic constraints. Boundary value DAEs are also subject to a system of boundary conditions. In the past, BVP software packages have been successfully adapted to solve boundary value DAEs, e.g., `COLDAE` [6]. Using similar techniques, it is possible that the BVP component of `pythODE` can also be modified to numerically solve these problems. In addition, `pythODE` is an ideal platform to study new numerical algorithms that aid in both the effectiveness and efficiency of numerical methods to solve boundary value DAEs.

Appendix: Tables of Global-Error Results for MIRK Formulas of Orders Two, Four, and Six

Tables 1, 2, and 3 contain the results from a second-order MIRK formula applied to problems (4.4), (4.5), and (4.6) respectively. Tables 4, 5, and 6 contain the results from a fourth-order MIRK formula applied to problems (4.4), (4.5), and (4.6) respectively. Tables 7, 8, and 9 contain the results from a sixth-order MIRK formula applied to problems (4.4), (4.5), and (4.6) respectively.

The entries of each of the tables are organized by columns. The first column, Algorithm, is the algorithm used to estimate the global error. These algorithms are based on Richardson extrapolation (RE), higher-order formulas (HO), deferred corrections (DC), and a conditioning constant (CO). The second column, Tol, is the user-supplied tolerance for the defect. The third column, Time, is the runtime in seconds for the global-error algorithm. The fourth column, % Total Time, is the percent of the total solution time used for the global-error algorithm. The fifth column, Global Error, is the exact global error. The sixth column, Estimated Error, is the estimated global error provided by the global-error algorithm. The seventh column, τ, is the absolute value of the difference between the exact global error and its estimate; it measures the quality of the global-error estimate.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	8.52	0.26	2.4E-05	2.4E-05	5.8E-08
	1E-5	29.57	0.27	2.0E-06	1.9E-06	5.0E-09
	1E-6	100.55	0.28	1.8E-07	1.8E-07	1.9E-10
	1E-7	357.17	0.28	1.7E-08	1.7E-08	1.9E-12
	1E-8	995.88	0.25	1.6E-09	1.6E-09	2.7E-14
HO	1E-4	1.33	0.04	2.4E-05	2.4E-05	4.9E-08
	1E-5	4.55	0.04	2.0E-06	1.9E-06	6.3E-09
	1E-6	14.97	0.04	1.8E-07	1.8E-07	2.1E-10
	1E-7	51.90	0.04	1.7E-08	1.7E-08	8.1E-13
	1E-8	161.85	0.04	1.6E-09	1.6E-09	1.9E-14
DC	1E-4	2.12	0.07	2.4E-05	2.4E-05	4.9E-08
	1E-5	7.25	0.07	2.0E-06	1.9E-06	6.3E-09
	1E-6	23.88	0.07	1.8E-07	1.8E-07	2.1E-10
	1E-7	80.82	0.06	1.7E-08	1.7E-08	8.1E-13
	1E-8	256.02	0.06	1.6E-09	1.6E-09	1.9E-14
CO	1E-4	0.98	0.03	2.4E-05	1.5E-01	1.5E-01
	1E-5	3.22	0.03	2.0E-06	4.3E-02	4.3E-02
	1E-6	10.52	0.03	1.8E-07	1.4E-02	1.4E-02
	1E-7	34.25	0.03	1.7E-08	4.2E-03	4.2E-03
	1E-8	112.55	0.03	1.6E-09	1.2E-03	1.2E-03

Table 1: Results for problem (4.4), MIRK order two.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	4.22	0.24	5.5E-05	5.5E-05	5.1E-10
	1E-5	10.57	0.28	8.5E-06	8.5E-06	1.2E-11
	1E-6	36.60	0.27	7.2E-07	7.2E-07	8.4E-14
	1E-7	123.38	0.27	6.4E-08	6.4E-08	1.3E-16
	1E-8	403.77	0.26	6.1E-09	6.1E-09	7.9E-17
HO	1E-4	0.72	0.04	5.5E-05	5.5E-05	4.3E-10
	1E-5	1.83	0.05	8.5E-06	8.5E-06	3.3E-12
	1E-6	6.33	0.05	7.2E-07	7.2E-07	3.0E-14
	1E-7	21.32	0.05	6.4E-08	6.4E-08	9.0E-16
	1E-8	68.88	0.04	6.1E-09	6.1E-09	0.0E+00
DC	1E-4	1.17	0.07	5.5E-05	5.5E-05	4.3E-10
	1E-5	2.98	0.08	8.5E-06	8.5E-06	3.3E-12
	1E-6	10.23	0.08	7.2E-07	7.2E-07	3.0E-14
	1E-7	34.33	0.07	6.4E-08	6.4E-08	9.0E-16
	1E-8	111.10	0.07	6.1E-09	6.1E-09	0.0E+00
CO	1E-4	0.47	0.03	5.5E-05	1.7E-02	1.7E-02
	1E-5	1.18	0.03	8.5E-06	6.6E-03	6.6E-03
	1E-6	4.12	0.03	7.2E-07	1.9E-03	1.9E-03
	1E-7	13.90	0.03	6.4E-08	5.7E-04	5.7E-04
	1E-8	45.38	0.03	6.1E-09	1.8E-04	1.8E-04

Table 2: Results for problem (4.5), MIRK order two.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	28.77	0.39	1.8E-04	1.8E-04	2.1E-10
	1E-5	92.82	0.38	1.8E-05	1.8E-05	2.2E-12
	1E-6	301.38	0.37	1.7E-06	1.7E-06	1.3E-13
	1E-7	763.73	0.42	2.7E-07	2.7E-07	1.1E-13
	1E-8	2648.28	0.38	2.3E-08	2.3E-08	1.1E-13
HO	1E-4	1.10	0.01	1.8E-04	1.8E-04	1.3E-09
	1E-5	3.53	0.01	1.8E-05	1.8E-05	1.1E-11
	1E-6	11.55	0.01	1.7E-06	1.7E-06	6.3E-15
	1E-7	29.07	0.02	2.7E-07	2.7E-07	1.1E-13
	1E-8	99.65	0.01	2.3E-08	2.3E-08	1.1E-13
DC	1E-4	1.75	0.02	1.8E-04	1.8E-04	1.3E-09
	1E-5	5.67	0.02	1.8E-05	1.8E-05	1.1E-11
	1E-6	18.38	0.02	1.7E-06	1.7E-06	6.3E-15
	1E-7	46.45	0.03	2.7E-07	2.7E-07	1.1E-13
	1E-8	159.08	0.02	2.3E-08	2.3E-08	1.1E-13
CO	1E-4	4.48	0.06	1.8E-04	2.5E-01	2.5E-01
	1E-5	14.45	0.06	1.8E-05	7.3E-02	7.3E-02
	1E-6	47.07	0.06	1.7E-06	2.2E-02	2.2E-02
	1E-7	118.78	0.07	2.7E-07	8.7E-03	8.7E-03
	1E-8	408.87	0.06	2.3E-08	2.5E-03	2.5E-03

Table 3: Results for problem (4.6), MIRK order two.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	1.98	0.11	4.6E-06	4.6E-06	2.6E-09
	1E-5	3.25	0.13	5.2E-07	5.2E-07	8.1E-11
	1E-6	5.67	0.15	5.4E-08	5.4E-08	4.9E-12
	1E-7	9.70	0.16	5.9E-09	5.9E-09	9.4E-14
	1E-8	14.28	0.23	1.4E-09	1.4E-09	2.9E-14
HO	1E-4	0.53	0.03	4.6E-06	4.6E-06	1.5E-08
	1E-5	0.75	0.03	5.2E-07	5.2E-07	2.4E-09
	1E-6	1.28	0.03	5.4E-08	5.5E-08	2.4E-10
	1E-7	2.18	0.04	5.9E-09	5.9E-09	1.0E-12
	1E-8	3.03	0.05	1.4E-09	1.4E-09	3.4E-12
DC	1E-4	0.70	0.04	4.6E-06	4.6E-06	1.5E-08
	1E-5	1.02	0.04	5.2E-07	5.2E-07	2.4E-09
	1E-6	1.87	0.05	5.4E-08	5.5E-08	2.4E-10
	1E-7	3.27	0.05	5.9E-09	5.9E-09	1.0E-12
	1E-8	4.75	0.08	1.4E-09	1.4E-09	3.4E-12
CO	1E-4	0.10	0.01	4.6E-06	3.5E-02	3.5E-02
	1E-5	0.33	0.01	5.2E-07	6.2E-03	6.2E-03
	1E-6	0.52	0.01	5.4E-08	1.1E-03	1.1E-03
	1E-7	0.92	0.01	5.9E-09	1.8E-04	1.8E-04
	1E-8	1.27	0.02	1.4E-09	4.7E-05	4.7E-05

Table 4: Results for problem (4.4), MIRK order four.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	1.70	0.07	1.2E-05	1.2E-05	2.6E-09
	1E-5	2.25	0.06	4.2E-06	4.2E-06	4.4E-10
	1E-6	3.62	0.08	5.1E-07	5.1E-07	1.9E-11
	1E-7	6.63	0.10	4.7E-08	4.7E-08	5.3E-13
	1E-8	12.83	0.12	5.0E-09	5.0E-09	1.8E-14
HO	1E-4	0.38	0.02	1.2E-05	1.2E-05	1.2E-07
	1E-5	0.57	0.02	4.2E-06	4.2E-06	3.7E-09
	1E-6	0.83	0.02	5.1E-07	5.1E-07	1.6E-10
	1E-7	1.48	0.02	4.7E-08	4.7E-08	4.5E-12
	1E-8	2.82	0.03	5.0E-09	5.0E-09	1.5E-13
DC	1E-4	0.62	0.03	1.2E-05	1.2E-05	1.2E-07
	1E-5	0.75	0.02	4.2E-06	4.2E-06	3.7E-09
	1E-6	1.28	0.03	5.1E-07	5.1E-07	1.6E-10
	1E-7	2.22	0.03	4.7E-08	4.7E-08	4.5E-12
	1E-8	4.20	0.04	5.0E-09	5.0E-09	1.5E-13
CO	1E-4	0.17	0.01	1.2E-05	1.6E-02	1.6E-02
	1E-5	0.22	0.01	4.2E-06	1.3E-02	1.3E-02
	1E-6	0.33	0.01	5.1E-07	2.1E-03	2.1E-03
	1E-7	0.52	0.01	4.7E-08	1.6E-04	1.6E-04
	1E-8	1.00	0.01	5.0E-09	5.7E-05	5.7E-05

Table 5: Results for problem (4.5), MIRK order four.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	4.28	0.32	4.3E-05	4.3E-05	8.5E-08
	1E-5	7.33	0.34	4.9E-06	4.9E-06	4.7E-09
	1E-6	14.15	0.31	4.2E-07	4.2E-07	1.2E-10
	1E-7	26.38	0.35	4.8E-08	4.8E-08	5.1E-12
	1E-8	47.78	0.39	6.0E-09	6.0E-09	6.2E-13
HO	1E-4	0.27	0.02	4.3E-05	4.4E-05	5.2E-07
	1E-5	0.42	0.02	4.9E-06	5.0E-06	2.9E-08
	1E-6	0.83	0.02	4.2E-07	4.2E-07	7.2E-10
	1E-7	1.60	0.02	4.8E-08	4.8E-08	3.6E-11
	1E-8	2.57	0.02	6.0E-09	6.0E-09	6.9E-13
DC	1E-4	0.38	0.03	4.3E-05	4.4E-05	5.2E-07
	1E-5	0.67	0.03	4.9E-06	5.0E-06	2.9E-08
	1E-6	1.33	0.03	4.2E-07	4.2E-07	7.2E-10
	1E-7	2.32	0.03	4.8E-08	4.8E-08	3.6E-11
	1E-8	3.90	0.03	6.0E-09	6.0E-09	6.9E-13
CO	1E-4	0.65	0.05	4.3E-05	6.6E-02	6.6E-02
	1E-5	1.03	0.05	4.9E-06	1.4E-02	1.4E-02
	1E-6	2.00	0.04	4.2E-07	1.8E-03	1.8E-03
	1E-7	3.37	0.04	4.8E-08	3.9E-04	3.9E-04
	1E-8	5.67	0.05	6.0E-09	9.2E-05	9.2E-05

Table 6: Results for problem (4.6), MIRK order four.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	1.67	0.07	1.5E-06	1.5E-06	2.1E-10
	1E-5	2.35	0.08	1.6E-07	1.6E-07	4.6E-11
	1E-6	3.22	0.09	1.6E-08	1.6E-08	3.2E-12
	1E-7	4.28	0.12	2.8E-09	2.8E-09	1.9E-13
	1E-8	6.28	0.14	2.8E-10	2.8E-10	4.2E-15
HO	1E-4	0.58	0.03	1.5E-06	1.5E-06	1.5E-08
	1E-5	0.72	0.03	1.6E-07	1.6E-07	4.3E-10
	1E-6	1.00	0.03	1.6E-08	1.6E-08	1.8E-11
	1E-7	1.28	0.04	2.8E-09	2.8E-09	7.3E-12
	1E-8	1.87	0.04	2.8E-10	2.8E-10	9.6E-14
DC	1E-4	0.72	0.03	1.5E-06	1.5E-06	1.5E-08
	1E-5	1.03	0.04	1.6E-07	1.6E-07	4.3E-10
	1E-6	1.53	0.04	1.6E-08	1.6E-08	1.8E-11
	1E-7	1.87	0.05	2.8E-09	2.8E-09	7.3E-12
	1E-8	2.68	0.06	2.8E-10	2.8E-10	9.6E-14
CO	1E-4	0.03	0.00	1.5E-06	9.4E-03	9.4E-03
	1E-5	0.12	0.00	1.6E-07	1.5E-03	1.5E-03
	1E-6	0.10	0.00	1.6E-08	1.7E-04	1.7E-04
	1E-7	0.25	0.01	2.8E-09	1.1E-04	1.1E-04
	1E-8	0.38	0.01	2.8E-10	1.2E-05	1.2E-05

Table 7: Results for problem (4.4), MIRK order six.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	2.75	0.06	1.5E-08	1.5E-08	1.9E-13
	1E-5	2.78	0.06	1.3E-08	1.3E-08	1.4E-13
	1E-6	2.70	0.05	1.1E-08	1.1E-08	1.3E-13
	1E-7	6.07	0.08	8.0E-11	8.0E-11	4.4E-16
	1E-8	6.80	0.08	3.9E-11	3.9E-11	8.2E-17
HO	1E-4	0.83	0.02	1.5E-08	1.5E-08	1.2E-11
	1E-5	0.85	0.02	1.3E-08	1.3E-08	1.0E-11
	1E-6	0.88	0.02	1.1E-08	1.1E-08	9.0E-12
	1E-7	1.82	0.02	8.0E-11	8.0E-11	1.5E-14
	1E-8	2.03	0.02	3.9E-11	3.9E-11	5.7E-15
DC	1E-4	1.20	0.03	1.5E-08	1.5E-08	1.1E-11
	1E-5	1.20	0.02	1.3E-08	1.3E-08	8.8E-12
	1E-6	1.18	0.02	1.1E-08	1.1E-08	7.7E-12
	1E-7	2.65	0.03	8.0E-11	8.0E-11	1.5E-14
	1E-8	2.93	0.03	3.9E-11	3.9E-11	5.7E-15
CO	1E-4	0.17	0.00	1.5E-08	2.8E-05	2.8E-05
	1E-5	0.17	0.00	1.3E-08	4.2E-05	4.2E-05
	1E-6	0.17	0.00	1.1E-08	1.5E-04	1.5E-04
	1E-7	0.33	0.00	8.0E-11	3.4E-07	3.4E-07
	1E-8	0.40	0.00	3.9E-11	1.2E-06	1.2E-06

Table 8: Results for problem (4.5), MIRK order six.

Algorithm	Tol	Time	% Total Time	Global Error	Estimated Error	τ
RE	1E-4	2.28	0.27	3.9E-05	3.9E-05	7.5E-09
	1E-5	2.93	0.29	4.9E-06	4.9E-06	7.5E-10
	1E-6	4.88	0.25	1.7E-07	1.7E-07	7.9E-12
	1E-7	6.80	0.26	2.8E-08	2.8E-08	1.1E-12
	1E-8	10.43	0.27	3.5E-09	3.5E-09	7.9E-13
HO	1E-4	0.22	0.03	3.9E-05	3.9E-05	1.9E-07
	1E-5	0.32	0.03	4.9E-06	4.9E-06	1.7E-08
	1E-6	0.43	0.02	1.7E-07	1.7E-07	2.0E-10
	1E-7	0.83	0.03	2.8E-08	2.8E-08	1.8E-11
	1E-8	1.12	0.03	3.5E-09	3.5E-09	1.8E-12
DC	1E-4	0.37	0.04	3.9E-05	3.9E-05	1.9E-07
	1E-5	0.45	0.04	4.9E-06	4.9E-06	1.7E-08
	1E-6	0.80	0.04	1.7E-07	1.7E-07	2.0E-10
	1E-7	1.05	0.04	2.8E-08	2.8E-08	1.8E-11
	1E-8	1.65	0.04	3.5E-09	3.5E-09	1.8E-12
CO	1E-4	0.32	0.04	3.9E-05	8.4E-03	8.4E-03
	1E-5	0.35	0.03	4.9E-06	3.7E-03	3.7E-03
	1E-6	0.53	0.03	1.7E-07	1.3E-04	1.3E-04
	1E-7	0.77	0.03	2.8E-08	3.2E-05	3.2E-05
	1E-8	1.23	0.03	3.5E-09	6.7E-06	6.7E-06

Table 9: Results for problem (4.6), MIRK order six.

Bibliography

[1] LAPACK. http://www.netlib.org/lapack/.

[2] Numpy, Feburary 2010. http://www.numpy.org/.

[3] Ascher, U., Christiansen, J., and Russell, R. D. A collocation solver for mixed order systems of boundary value problems. *Math. Comp. 33*, 146 (1979), 659–679.

[4] Ascher, U. M., Mattheij, R. M. M., and Russell, R. D. *Numerical solution of boundary value problems for ordinary differential equations*, vol. 13 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995. Corrected reprint of the 1988 original.

[5] Ascher, U. M., and Petzold, L. R. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1998.

[6] Ascher, U. M., and Spiteri, R. J. Collocation software for boundary value differential-algebraic equations. *SIAM J. Sci. Comput. 15*, 4 (1994), 938–952.

[7] Bader, G., and Ascher, U. A new basis implementation for a mixed order boundary value ODE solver. *SIAM J. Sci. Statist. Comput. 8*, 4 (1987), 483–500.

[8] Bailey, P. B., Shampine, L. F., and Waltman, P. E. *Nonlinear two point boundary value problems*. Mathematics in Science and Engineering, Vol. 44. Academic Press, New York, 1968.

[9] Boisvert, J. J., Muir, P. H., and Spiteri, R. J. A numerical study of global error estimation schemes for defect control BVODE codes. *Saint Mary's University, Dept. of Math. and Comp. Sci. Technical Report Series,cs.smu.ca/tech_reports/* (2009).

[10] Boisvert, J. J., Muir, P. H., and Spiteri, R. J. py_bvp: A universal python interface for BVP codes. In *Proceedings of the 2010 Spring Simulation Multiconference* (New York, NY, USA, 2010), SpringSim '10, ACM, pp. 95:1–95:9.

[11] Cash, J. R. http://www2.imperial.ac.uk/∼jcash/BVP_software/readme.php.

[12] Cash, J. R., and Mazzia, F. A new mesh selection algorithm, based on conditioning, for two-point boundary value codes. *J. Comput. Appl. Math. 184*, 2 (2005), 362–381.

[13] Cash, J. R., and Mazzia, F. Hybrid mesh selection algorithms based on conditioning for two-point boundary value problems. *J. Numer. Anal. Ind. Appl. Math. 1*, 1 (2006), 81–90.

[14] Cash, J. R., and Wright, M. H. A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation. *SIAM J. Sci. Statist. Comput. 12*, 4 (1991), 971–989.

[15] Evans, E. *Domain-Driven Design: Tacking Complexity in the Heart of Software.* Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[16] Gladwell, I., Shampine, L., and Thompson, S. *Solving ODEs with MATLAB.* Cambridge University Press, New York, NY, USA, 2003.

[17] Hanson, P. M., and Enright, W. H. Controlling the defect in existing variable-order Adams codes for initial value problems. *ACM Trans. Math. Software 9*, 1 (1983), 71–97.

[18] Higham, N. J. FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Trans. Math. Software 14*, 4 (1988), 381–396 (1989).

[19] Kelley, C. T. *Solving nonlinear equations with Newton's method.* Fundamentals of Algorithms. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2003.

[20] Kierzenka, J., and Shampine, L. F. A BVP solver that controls residual and error. *JNAIAM J. Numer. Anal. Ind. Appl. Math. 3*, 1-2 (2008), 27–41.

[21] Mazzia, F., and Trigiante, D. A hybrid mesh selection strategy based on conditioning for boundary value ODE problems. *Numer. Algorithms 36*, 2 (2004), 169–187.

[22] Mench, M. M. *Fuel Cell Engines.* Wiley, 2004.

[23] Millman, J. `Scipy`, Feburary 2010. `http://www.scipy.org/`.

[24] Muir, P. H. Optimal discrete and continuous mono-implicit Runge-Kutta schemes for BVODEs. *Adv. Comput. Math. 10*, 2 (1999), 135–167.

[25] Muir, P. H., and Adams, M. Mono-implicit Runge–Kutta Nyström methods with application to boundary value ordinary differential equations. *BIT 41*, 4 (2001), 776–799.

[26] Osborne, M. R. Collocation, difference equations, and stitched function representations. In *Topics in numerical analysis, II (Proc. Roy. Irish Acad. Conf., Univ. College, Dublin, 1974).* Academic Press, London, 1975, pp. 121–132.

[27] Peterson, P. `F2PY`: Fortran to Python Interface Generator, July 2007. `http://cens.ioc.ee/projects/f2py2e/`.

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

[29] Russell, R. D., and Christiansen, J. Adaptive mesh selection strategies for solving boundary value problems. *SIAM J. Numer. Anal. 15*, 1 (1978), 59–80.

[30] Secanell, M. *Computational Modeling and Optimization of Proton Exchange Membrane Fuel Cells*. Ph.D. dissertation, University of Victoria, Department of Mechanical Engineering, 2007.

[31] Secanell, M., 2010. Multi-Scale Agglomerate Model (Research Proposal).

[32] Secanell, M., Karan, K., Suleman, A., and Djilali, N. Multi-variable optimization of pemfc cathodes using an agglomerate model. *Electrochimica Acta 52*, 22 (2007), 6318 – 6337.

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

[34] Shampine, L. F., and Muir, P. H. Estimating conditioning of BVPs for ODEs. *Math. Comput. Modelling 40*, 11-12 (2004), 1309–1321.

[35] Shampine, L. F., Muir, P. H., and Xu, H. A user-friendly Fortran BVP solver. *J. Numer. Anal. Ind. Appl. Math. 1*, 2 (2006), 201–217.

[36] Skeel, R. D. Thirteen ways to estimate global error. *Numer. Math. 48*, 1 (1986), 1–20.

[37] Ter, T. A problem-solving environment for the numerical solution of nonlinear algebraic equations. Master's thesis, University of Saskatchewan, Department of Computer Science, 2007.

[38] `COMSOL`. `http://www.comsol.com/`.

[39] `Mathematica`. `www.wolfram.com/products/mathematica/index.html`.

[40] `MATLAB`. `www.mathworks.com/products/matlab/`.

[41] `matplotlib`. `http://matplotlib.sourceforge.net/`.

[42] Yang, D. *C++ and object-oriented numeric computing for scientists and engineers*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.