Numerical Solution of Nonlinear Boundary Value Problems for Ordinary Differential Equations in the Continuous Framework

Ásgeir Birkisson

Lincoln College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity Term 2013

Til mömmu, Ingu, Maríu Bjarkar og Sigrúnar

Acknowledgements

First of all, I would like to thank my supervisor, Nick Trefethen, for all his support during my time as a DPhil student. It has been both a privilege and a pleasure to work with him, and I consider myself extremely lucky to have had the chance to do so. His encouragement and guidance has been invaluable to me, from the first day as his student to the last.

Secondly, I would like to thank Toby Driscoll at the University of Delaware. A substantial amount of work presented in this thesis originates in our collaboration, and I am grateful for having had the chance to work with him and spend time at his home institution. Without him sharing with me his knowledge of numerical methods, or teaching me skills for implementing them in practical software, I doubt this thesis would have been in its present state.

Thirdly, I would like to thank Prof. Mike Giles and Prof. Folkmar Bornemann for agreeing to act as my examineers, for the very interesting discussion we had during my viva, and the numerous corrections, comments and suggestions they made which considerably improved this work.

Next, I would like to thank past and present fellow members of the Chebfun team. It has been a great experience to work with all of them. Particular mention must go to two people: Alex Townsend and Nick Hale. Alex generously offered to read most of the thesis. His careful attention to details resulted in a number of improvements to this work. Nick has been an incredible collaborator on the software side of Chebfun. It is largely due to his managing of the project and contributions that Chebfun is a developed enough platform to implement the algorithms of this thesis. Also, I am indebted to Lotti Ekert for all her assistance through my time in Oxford, and her abilities to make everything run so smoothly.

The Numerical Analysis group at Oxford is an inspiring environment to be at, and I have had the fortune to make many friends there, who are not only great mathematicians, but also great people. To name but a few, Iain Smears has been an inexhaustible well of knowledge about functional analysis, and

a great person to converse with about the Icelandic Sagas. I also learned a lot from Stefan Güttel, and it is always a pleasure to meet him. And last but not least, special mention goes to Pedro Gonnet, John Pearson and Mark Richardson. Thanks to them, I always looked forward to coming into work, and every minute in their company was an enjoyment, whether in the Gonnet room, or outside of the office.

I would not have been able to read for a DPhil at Oxford without the help of the funding bodies that supported me. Therefore, I am extremely grateful to the Sloane Robinson Foundation for sponsoring my award at Lincoln College, as well as the Engineering and Physical Sciences Research Council and the European Research Council.

Finally, I would like to thank all my family back home in Iceland. The most difficult thing about studying in Oxford has been to be away from them. And most of all, thanks to my fiancée, Sigrún, for all the years we've been together, and all the years we have ahead.

Abstract

Ordinary differential equations (ODEs) play an important role in mathematics. Although intrinsically, the setting for describing ODEs is the continuous framework, where differential operators are considered as maps from one function space to another, common numerical algorithms for ODEs discretise problems early on in the solution process. This thesis is about continuous analogues of such discrete algorithms for the numerical solution of ODEs.

This thesis shows how Newton's method for finite dimensional system can be generalised to function spaces, where it is known as Newton–Kantorovich iteration. It presents affine invariant damping strategies for increasing the chance of convergence for the Newton–Kantorovich iteration.

The derivatives required in this continuous setting are Fréchet derivatives, the continuous analogue of Jacobian matrices. In this work, we present how automatic differentiation techniques can be applied to compute Fréchet derivatives. We introduce `chebop`, a MATLAB solver for nonlinear boundary-value problems, which combines damped Newton iteration in function space and automatic Fréchet differentiation.

By proving that affine operators have constant Fréchet derivatives, it is demonstrated how automatic linearity detection of computed quantities can be implemented. This is valuable for black-box solvers, which can use the information to determine whether an iteration scheme has to be employed for solving a problem.

Like nonlinear systems of equations, nonlinear boundary-value problems can have multiple solutions. This thesis present two techniques for obtaining multiple solutions of operator equations: deflation and path-following. An algorithm combining the two techniques is proposed.

Contents

1 Introduction **1**

1.1 Chebyshev technology and Chebfun 4

1.1.1 Polynomial interpolation 4

1.1.2 Interpolation in Chebyshev nodes 7

1.1.3 Chebfun 8

1.2 Spectral methods 11

1.2.1 Background 11

1.2.2 Spectral collocation methods with a Chebyshev basis 12

1.2.3 Linear differential operators in Chebfun 17

1.3 Automatic differentiation 20

1.3.1 Computing derivatives in scientific computing 20

1.3.2 The evaluation trace, and how it can be augmented to yield derivative values 21

1.3.3 An example of forward mode AD 25

1.3.4 An example of reverse mode AD 26

1.3.5 The `advar` class for demonstrating operator overloaded, forward mode AD 29

2 Theory and Algorithms for Numerical Solution of Nonlinear ODEs by Newton Iteration in Function Space **33**

2.1 Ordinary differential equations 33

2.2 Theoretical background 36

2.2.1 Banach and Hilbert spaces 36

2.2.2 Operators between Banach spaces 39

2.2.3 Choice of function spaces 40

2.2.4 Derivatives of operators 41

2.2.5 Big O and small o notation 43

2.3 Classical Newton's method in Banach spaces 44

2.3.1 Historical background 44
2.3.2 Newton–Kantorovich theory 45
2.3.3 An example of Newton iteration in function space 47
2.4 Affine invariant Newton methods 50
2.4.1 Affine transformations 51
2.4.2 The algorithmic paradigm 52
2.4.3 Affine covariance for an error oriented algorithm 53
2.4.4 Affine contravariance for a residual oriented algorithm 58
2.5 Global Newton methods 62
2.5.1 Error-oriented globalised Newton's method 65
2.5.2 Residual based globalised Newton's method 69
2.6 Discussion 73
2.6.1 Demonstrating the importance of affine invariance 73
2.6.2 Effect of discretisation errors on Newton's method in function space 74
2.6.3 Ghost solutions 75

3 Fréchet Derivatives by Automatic Differentiation and Automatic Solutions of Nonlinear BVPs 77
3.1 A delayed, recursive AD implementation 79
3.1.1 Recursive evaluation of derivatives 81
3.1.2 The `anon` class for overcoming computational inefficiencies . . 84
3.2 Run-time AD implementation 86
3.3 Examples of automatic Fréchet differentiation 88
3.4 Automatic differentiation of functionals 92
3.5 Taylor testing for correctness of derivatives 94
3.6 Automatic numerical solution of ODE BVPs and the `chebop` class . . 96
3.6.1 Comparison with existing software 103
3.6.2 A graphical user interface to Chebfun – chebgui 109

4 Automatic Linearity Detection 111
4.1 Linearity of computed quantities 113
4.1.1 Definitions of linear and affine operators 113
4.1.2 Properties of affine operators 116
4.1.3 Limitations of automatic linearity detection 121
4.1.4 Boolean algebra notation 121
4.1.5 Propagation of information for linearity detection 122

4.2 Linearity detection in AD software 138
4.2.1 Examples 141

5 Multiple Solutions of Nonlinear BVPs **145**
5.1 Deflation for multiple solutions of nonlinear BVPs 147
5.1.1 Deflation for polynomial rootfinding 147
5.1.2 Deflation for nonlinear systems of equations 152
5.1.3 Deflation for operator equations 168
5.2 Path following 177
5.2.1 Path following for boundary value problems 178
5.3 Comparing and combining deflation and path following 190

6 Conclusions and Further Work **193**
6.1 Conclusions 193
6.2 Further work 197

Bibliography **199**

List of Figures

1.1 Plots of a `chebfun` `f` and its Chebyshev coefficients. 10
1.2 The Chebfun solution `u` to the BVP (1.25). 19

2.1 The solution of the BVP (2.23) by eight steps of Newton iteration. . . 50
2.2 Solving the ODE (2.51) using the algorithm `GENIB`. 74

3.1 Diagram showing recursive evaluation of a Fréchet derivative. 83
3.2 Comparison of execution for different implementations of delayed, recursive AD. 85
3.3 Taylor testing for correctness of derivatives of (3.6) 96
3.4 Taylor testing for correctness of derivatives of (3.7) 97
3.5 `chebop` solution of the BVP (3.8) 101
3.6 `chebop` solutions of the BVPs (3.9) and (3.10). 102
3.7 Solutions of the BVP for the motion of a pendulum, satisfying non-standard conditions. 104
3.8 Plot of the solutions of the six nonlinear BVPs we consider for comparing solvers. 106
3.9 Running time vs. accuracy obtained with different solvers for six nonlinear BVPs. 108
3.10 Screenshot of `chebgui`. 110

5.1 Two solutions of the BVP (5.1), describing the motion of a nonlinear pendulum. 146
5.2 MATLAB code implementing Algorithm 8. 151
5.3 Obtaining multiple roots of a polynomial via deflation in combination with Horner's method and Newton iteration. 152
5.4 Curves of $1/\tanh(\alpha(x))$ for different values of α. 163
5.5 Domains of attraction for each root of the nonlinear cubic-parabola system (5.26) 165

5.6 Grid-points for which norm deflation, without using inflation, successfully converged to a second root. 165
5.7 Grid-points for which norm deflation, using inflation, successfully converged to a second root. 166
5.8 Grid-points for which tanh-deflation, with $\alpha = 2$, successfully converged to a second root. 166
5.9 Grid-points for which gradient deflation successfully converged to a second root. 167
5.10 Grid-points for which gradient deflation successfully converged to a third root. 167
5.11 Two solutions of the Bratu problem. 174
5.12 Three solutions of the nonlinear pendulum BVP. 175
5.13 Two solutions of the nonlinear Painlevé I problem (5.35). 176
5.14 Six solutions of the simplified Carrier problem (5.36). 177
5.15 Solutions of the Bratu problem, depending on the value of λ. 179
5.16 Bifurcation diagram for the Bratu problem (5.47). 184
5.17 Bifurcation diagram for the Bratu problem (5.48). 185
5.18 Bifurcation diagram for the nonlinear pendulum (5.49). 186
5.19 Five solutions for the nonlinear pendulum (5.49). 187
5.20 Bifurcation diagram for the simplified Carrier problem (5.50). 188
5.21 Two more solutions to the simplified Carrier problem (5.50). 188
5.22 Bifurcation diagram for the Painlevé I problem (5.52). 189
5.23 Nine different solutions to the Painlevé I problem (5.52). 189

List of Tables

1.1 An evaluation trace for $y = (u_1 + u_2)\sin(u_2^2) + 4(u_1 + u_2^2)$. 22
1.2 A modified evaluation trace, using forward mode AD to compute a derivative at the same time. 26
1.3 A modified evaluation trace, using reverse mode AD to compute the sensitivity of the output with respect to all intermediate variables. . . 28

4.1 List of Boolean identities. 122
4.2 Definition of the the operators ζ_k and η_k. 127
4.3 An evaluation trace, showing the propagation of linearity information. 139

5.1 Performance of deflation schemes for the cubic-parabola system (5.26). 164

List of Algorithms

1 `NIB`: Newton iteration in Banach spaces. 46
2 `LENIB`: Local Error oriented Newton iteration in Banach spaces. . . . 59
3 `LRNIB`: Local residual oriented Newton iteration in Banach spaces. . . 62
4 `GENIB`: Global error Oriented Newton iteration in Banach spaces. . . 70
4 `GENIB` (continued) 71
5 `GRNIB`: Global residual based Newton iteration in Banach spaces. . . 72
5 `GRNIB` (continued) 73
6 Recursive evaluation of Fréchet derivatives. 82
7 An error-oriented damped Newton iteration for nonlinear BVPs. . . . 98
8 Deflation in combination with Horner's method and Newton iteration. 150
9 Deflation for finite dimensional systems. 162
10 Path-following for nonlinear BVPs. 183
11 Algorithm for finding multiple roots of nonlinear BVPs, combining deflation and path-following. 192

Chapter 1

Introduction

Ordinary differential equations (ODEs) play an important role in mathematics. They arise in many areas of science, such as physics, chemistry, biology, and engineering [BD97, DB02]. Whereas analytical solutions can be found for some linear problems, for nonlinear problems, most problems cannot be solved analytically in closed form, and numerical methods have to be applied.

With numerical methods, a common approach is to obtain the solution as a linear combination of basis functions $\{\varphi_k\}_{k=0}^{N}$, that is,

$$u_N(x) = \sum_{k=0}^{N} a_k \varphi_k(x), \tag{1.1}$$

where the a_ks are the coefficients that determine the solution. The numerical solution u_N is then considered to be a approximation to the true solution u.

The aim of this thesis is present a collection of contributions to the numerical solution of nonlinear ODEs. Intrinsically, the setting for describing ODEs is the continuous framework, where differential operators are considered as maps from one function space to another. The standard approach in numerical analysis is to discretise problems early on in the solution process and work with matrices and functions operating on vectors, which represent the discretised values of solutions. The work in this thesis, however, is unusual in that it stays in the continuous framework for as long as possible, only discretising at the very end when required for computations. Thus, to a considerable degree, this thesis is about *continuous analogues* of discrete algorithms for the numerical solution of ODEs. The closest in the literature to our point of view is the work of Deuflhard [Deu06], but we carry it much further by staying more fully in the continuous context in formulating our algorithms, while yet implementing everything in practical software.

Within ODEs, our focus will be on boundary-value problems (BVPs), where constraints are imposed on the solution of the differential equation at both endpoints of the interval of interest. We will also consider problems where constraints are imposed at interior points of the interval, or where constraints are applied on the solution over the whole interval.

The key mathematical and algorithmic contributions of this thesis are as follows:

- How to numerically perform Newton iteration in function space, where the derivatives are Fréchet derivatives — linear operators between Banach spaces (Chapter 2). The Newton iteration is damped to increase its chance of converging. This has led to the publication [BD12].

- How automatic differentiation can be implemented in a delayed (also known as lazy) manner, to only compute derivatives when required at runtime (Section 3.1).

- How linearity of computed quantities can be determined automatically, and how automatic linearity detection can be incorporated into implementations of automatic differentiation (Chapter 4). This is done via propagating information about linearity as computations are carried out. This work is discussed in [BD13].

- How to define deflation for operator equations and path-following in the continuous framework to obtain multiple solutions of nonlinear BVPs (Chapter 5). We propose an algorithm combining deflation and path-following for obtaining multiple solutions of nonlinear problems. We plan to submit a journal article describing this work.

The contributions listed above were developed in the context of a substantial amount of software development for the Chebfun project [T+11] (discussed below), which we introduce as we go along. Another major activity of the author in the past few years has been the creation of *chebgui* — a graphical user interface to the differential equations capabilities of Chebfun. We believe `chebgui` will make the research presented in this thesis available to a wider audience, and increase its impact. However, this thesis will not discuss `chebgui` apart from an example in Section 3.6.2.

Throughout this thesis, the basis functions used to represent the solution in Equation (1.1), $\varphi_k(x)$, are the *Chebyshev polynomials (of the first kind)*

$$T_k(x) = \cos(k \operatorname{acos}(x)), \tag{1.2}$$

suitably rescaled to an interval $[a, b]$ on which problems are defined. Chebyshev polynomials have very desirable properties for various areas of numerical analysis, such as interpolation and approximation as is described in [Tre13]. Furthermore, they can be used for the numerical solution of linear differential equations via spectral methods, as is described in [For96, Boy00, Tre00]. These observations led to the development of Chebfun [T$^+$11], an open-source software system, which aims to allow users to compute with functions rather than numbers, using methods collectively known as *Chebyshev technology*. Chebfun is used throughout this thesis to demonstrate concepts for functions and operators, and for implementing algorithms. We often allow ourselves to skip describing details of what goes on "under the hood" in Chebfun when they are not necessary for our discussions. Section 1.1 gives a background discussion on some of the useful properties of Chebyshev polynomials, as well as describing Chebfun. In Section 1.2, we give an overview of spectral methods.

A common approach to solving nonlinear problems in scientific computing, such as nonlinear systems of equations and nonlinear differential equations, is to apply iterative methods. Such iterative methods often require the derivatives of the functions or operators involved. A powerful technique for obtaining the derivatives required in a robust and accurate way is known as *algorithmic* or *automatic differentiation* (AD). AD is based on the observation that complicated functions are commonly composed of simpler operations, such as cos, log, + and $\times$, each of which can be differentiated easily. The simpler derivatives can then be combined via the chain rule to yield the derivative of the original function. Section 1.3 gives a background discussion on AD.

A widely used iterative method for solving an algebraic system of nonlinear equations

$$\mathbf{f}(\mathbf{s}) = \mathbf{0},$$

where $\mathbf{s} \in \mathbb{R}^n$ and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, is Newton's method, where the original nonlinear problem is solved via succession of linear problems. Newton's method can be extended to the continuous framework of differential equations, which will in general be our method of solving nonlinear ODEs. In this context, Newton's method is known as *Newton's method in function space*, *Newton–Kantorovich iteration* or *quasilinearisation*. It requires the so-called Fréchet derivatives of the differential operators of the problem; the continuous analogues of Jacobian matrices. Chapter 2 discusses the mathematical background of solving nonlinear ODEs numerically in the continuous framework (some of which we allow ourselves to use in this introductory chapter before the systematic introduction). Furthermore, it describes how damping can be used to increase the domain of convergence for Newton iteration in function space.

In Chapter 3, we describe algorithms for obtaining derivatives via AD in a delayed manner, which can be employed for supplying the derivatives required for the Newton–Kantorovich iteration. We present an algorithm that combines Newton iteration in function space and automatic differentiation for solving nonlinear ODEs numerically. We then introduce the `chebop` class which implements that algorithm in Chebfun. The `chebop` class works for a wide class of problems, offers an intuitive and easy syntax for setting up and solving problems, and allows the imposition of both boundary and interior point conditions on the solution.

Often in scientific computing, it is desirable to offer a black-box solver for a wide variety of problems, such as for differential equations or problems in optimisation. Whereas such general solvers need to be able to treat nonlinear problems, and thus set up an iteration scheme to obtain a solution, such iterations are wasteful in the case of linear problems. For example, if a general ODE solver gets passed a linear problem, it should not need to set up a Newton iteration, as the solution should be obtained in one step. However, detecting linearity in an automatic and robust way is not a trivial task. In Chapter 4, we propose a framework for detecting linearity of computed quantities in a reliable way. We then demonstrate implementations of automatic linearity detection in combination with automatic differentiation, both for a general scientific computing environment, as well as in the Chebfun context.

Just like nonlinear systems of equations, nonlinear boundary value problems can have multiple solutions. In Chapter 5, we investigate two techniques for obtaining multiple solutions, starting from the same initial guess. The first technique is *deflation*, which works by modifying the original problem, so that previously found solutions are no longer solutions of the problem being solved. We discuss deflation both in the finite and infite dimensional context. The second technique is path-following, where we trace curves in the "$\infty + 1$" dimensional hyperspace, where each point consists of a function and a scalar. We then propose an algorithm combining deflation and path-following for obtaining multiple solutions of nonlinear BVPs.

Lastly, we summarise our conclusions from this thesis in Chapter 6, and list potential areas of further work.

1.1 Chebyshev technology and Chebfun

1.1.1 Polynomial interpolation

Let $C[a, b]$ denote the space of continuous functions on the closed interval $[a, b]$, equipped with the ∞-norm, denoted by $\|\cdot\|$. Without loss of generality, we can take

$a = -1$ and $b = 1$, so that we focus on the unit interval $[-1, 1]$.

Let Π_n denote the space of polynomials of degree n or less, defined on $[-1, 1]$. Let $f \in C[-1, 1]$ be given. Furthermore, let the set of grid points $X_n = \{x_j\}_{i=0}^n$ satisfying

$$-1 \leq x_0 < x_1 < \cdots < x_n \leq 1$$

and data $\{v_j\}_{i=0}^n$ be given, such that

$$v_j = f(x_j), \quad 0 \leq j \leq n.$$

The points in X_n can be arbitrarily spaced, but are assumed to be distinct. The problem of polynomial interpolation is to find the polynomial $p^{X_n} \in \Pi_n$ that satisfies the constraints

$$p^{X_n}(x_j) = v_j, \quad 0 \leq j \leq n. \tag{1.3}$$

In the setting of polynomial interpolation, the points in X_n are known as *nodes*. The polynomial p^{X_n} is known as the *interpolant* through the nodes X_n.

It is well known to numerical analysts that if the nodes are taken to be equispaced points, that is,

$$x_i = -1 + j\frac{2}{n}, \quad 0 \leq j \leq n,$$

the results can be disastrous. Even though the interpolant fits the data, the error $\|f(x) - p^{X_n}(x)\|$ can be huge. Furthermore, as shown by Runge in [Run01], the error can grow as n is increased, even if the function f is analytic on $[-1, 1]$ (but with singularities in the complex plane). As described in [Tre13], this effect led to polynomial interpolation having a bad reputation in scientific computing for years. However, if one makes a better selection of nodes, in particular, points which cluster near the end points of the interval, the interpolation problem is well behaved. The quality of a set of nodes for use in polynomial interpolation is determined by its *Lebesgue constant*, which we now describe. For this, we need to know how the polynomial interpolation problem (1.3) can be solved.

Lagrange interpolation is a well known method for solving the problem of polynomial interpolation. Following [SV90], begin by defining the jth Lagrange polynomial through the nodes X_n as

$$l_j^{X_n}(x) := \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}.$$

We see that $l_j^{X_n}(x)$ is clearly a polynomial of degree n, and

$$l_j^{X_n}(x_k) = \begin{cases} 1, & \text{if } k = j, \\ 0, & \text{if } k \neq j. \end{cases} \tag{1.4}$$

The *Lagrange interpolating projection*, associated with the nodes X_n, is the operator

$$L^{X_n} : C \to \Pi_n, \tag{1.5}$$

such that

$$L^{X_n} f = \sum_{i=0}^{n} v_j \, l_j^{X_n}(x). \tag{1.6}$$

The interpolant p^{X_n} is thus given by $p^{X_n} = L^{X_n} f$. Combining Equations (1.4) and (1.6), we see that the constraints (1.3) are satisfied. The norm of L^{X_n} has the value [Pow81, Thm. 4.3]

$$\|L^{X_n}\| = \max_{-1 \le x \le 1} \sum_{j=0}^{n} |l_j^{X_n}(x)|. \tag{1.7}$$

We define the *Lebesgue function* of L^{X_n} as the function

$$\lambda^{X_n}(x) = \sum_{j=0}^{n} |l_j^{X_n}(x)|.$$

The maximum value of $\lambda^{X_n}(x)$ over the interval is known as the *Lebesgue constant* of L^{X_n} and is denoted by $\Lambda(X_n)$, i.e.

$$\Lambda(X_n) = \max_{-1 \le x \le 1} \lambda^{X_n}(x). \tag{1.8}$$

Comparing Equations (1.7) and (1.8), we see that $\|L^{X_n}\| = \Lambda(X_n)$. We note that $\Lambda(X_n)$ only depends on the set of nodes chosen, X_n, but not on the data, $\{v_j\}_{j=0}^n$.

The efficiency of an approximation by any linear projection P onto a subspace Y of C is governed by the inequality [Pow81, Thm. 3.1]

$$\|f - Pf\| \le (1 + \|P\|) d(f, Y). \tag{1.9}$$

In the case of polynomial interpolation of nth degree, we have $Y = \Pi_n$, $P = L^{X_n}$, $\|P\| = \Lambda(X_n)$ and $d(f, Y) = \|f - p_n^*\|$ where p_n^* is the best polynomial approximation of f of degree n in the ∞-norm sense. Equation (1.9) thus reads

$$\|f - p_n\| \le (1 + \Lambda(X_n)) \|f - p_n^*\|. \tag{1.10}$$

It is clear then, that it is desirable to minimise the Lebesgue constant, since a small Lebesgue constant will guarantee that the interpolant p_n will be close to the best one.

The set of nodes which minimises $\Lambda(X_n)$ for a given n is called the *optimal interpolation nodes*. We denote such sets by X_n^*. Finding a closed form expression for the points in X_n^* for a general n remains an open problem. However, the following results are known for the values of the Lebesgue constant for interpolation nodes in any set of nodes, providing a lower bound for the value of $\Lambda(X_n^*)$:

Theorem 1.1.1. *[Tre13, Thm. 15.2] The Lebesgue constants $\Lambda(X_n)$ for degree $n \geq 0$ polynomial interpolation in any set X_n of $n+1$ distinct nodes in $[-1,1]$ satisfy*

$$\Lambda(X_n) \geq \frac{2}{\pi}\log(n+1) + c, \tag{1.11}$$

where the constant c has the numerical value $c \approx 0.52125$.

Equispaced nodes suffer from exponential growth of the Lebesgue constant, as the following theorem describes. This is the major reason why polynomial interpolation in equispaced nodes is a bad idea:

Theorem 1.1.2. *[Tre13, Thm. 15.2] The Lebesgue constants $\Lambda(X_n)$ for degree $n \geq 0$ polynomial interpolation in a set X_n of $n+1$ equispaced nodes in $[-1,1]$ satisfy*

$$\Lambda_n > 1 + \frac{2^{n-2}}{n^2} \qquad \textit{and} \qquad \Lambda_n \sim \frac{2^{n+1}}{en\log(n)},\ n \to \infty,$$

with the inequality applying for $n \geq 1$.

1.1.2 Interpolation in Chebyshev nodes

While the optimal interpolation nodes X_n^* are unknown, there exist known sets of nodes with similar behaviour of Lebesgue constants, meaning that they are close to optimal for polynomial interpolation. One particular set of such nodes are *Chebyshev nodes* (of the second kind), defined by

$$x_j = -\cos\left(\frac{j\pi}{n}\right), \quad 0 \leq j \leq n,$$

for a given n. Chebyshev nodes are the points where the nth Chebyshev polynomial, given in Equation (1.2), takes extreme values. The values for Lebesgue constants of Chebyshev nodes are given in the following theorem.

Theorem 1.1.3. *[Tre13, Thm. 15.2] The Lebesgue constants $\Lambda(X_n)$ for degree $n \geq 0$ polynomial interpolation in a set X_n of $n+1$ Chebyshev nodes in $[-1,1]$ satisfy*

$$\Lambda_n \leq 1 + \frac{2}{\pi}\log(n+1) \quad \textit{and} \quad \Lambda_n \sim \frac{2}{\pi}\log(n),\ n \to \infty. \tag{1.12}$$

Comparing Equations (1.11) and (1.12), we see that even though Chebyshev nodes are not the optimal interpolation nodes, their Lebesgue constant has the same asymptotic behaviour as the Lebesgue constant for the best possible set of nodes. This means that by using Chebyshev nodes, it is possible to work with polynomials of a large degree without the interpolants deteriorating as their degree gets larger.

The following theorems, stated here without proofs, play a crucial role for interpolation in Chebyshev nodes.

Theorem 1.1.4. *[Tre13, Thm. 3.1] If f is Lipschitz continuous on $[-1,1]$, it has a unique representation as a Chebyshev series,*

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x),$$

which is absolutely and uniformly convergent. The coefficients are given for $k \geq 1$ by the formula

$$a_k = \frac{2}{\pi} \int_{-1}^{1} \frac{f(x)T_k(x)}{\sqrt{1-x^2}} dx,$$

and for $k = 0$ by the same formula with the factor $2/\pi$ changed to $1/\pi$.

Theorem 1.1.5. *[Tre13, Thm. 7.1] For an integer $\nu \geq 0$, let f and its derivatives through $f^{(\nu-1)}$ be absolutely continuous on $[-1,1]$, and suppose the νth derivative $f^{(\nu)}$ is of bounded variation V. Then for $k \geq \nu+1$, the Chebyshev coefficients of f satisfy*

$$|a_k| \leq \frac{2V}{\pi k(k-1)\cdots(k-\nu)} = \frac{2V}{\pi(k-\nu)^{\nu+1}}.$$

For the next theorem, we need to introduce *Bernstein ellipses*. A Bernstein ellipse is an ellipse in the complex plane with foci at -1 and 1. We let ρ denote the sum of the lengths the semi-major and semi-minor axes of the ellipse, and E_ρ denote the open region bounded by the ellipse. The following result is originally due to Bernstein in 1912 [Ber12].

Theorem 1.1.6. *[Tre13, Thm. 8.1] Let a function f analytic in $[-1,1]$ be analytically continuable to the open Bernstein ellipse E_ρ, where it satisfies $|f(x)| \leq M$ for some M. Then its Chebyshev coefficients satisfy $|a_0| \leq M$ and*

$$|a_k| \leq 2M\rho^{-k}, \quad k \geq 1.$$

1.1.3 Chebfun

The basic idea of Chebfun is to make use of Theorem 1.1.4 to represent a general function on an interval $[a,b]$ (translating between $[-1,1]$ and $[a,b]$ by a map) as a polynomial interpolant on a Chebyshev grid. Theorem 1.1.3 assures us that we can work with polynomials of a high degree without worrying about the effects of growing Lebesgue constants. Such a polynomial representation to f on $[a,b]$ is known in Chebfun as an object of type `chebfun`. A `chebfun` can either be *global*, meaning that it is continuous and smooth over the whole interval, or *piecewise*, so that it consists of smooth pieces on subintervals of $[a,b]$.

We make a remark about the notational distinction between when we write Chebfun in upper-case, by which we mean the software system and project Chebfun, and `chebfun` in lower-case, by which we mean an instance of the class `chebfun`.

The coefficients in the Chebyshev series (1.1.4) are obtained by sampling f at increasingly finer Chebyshev grids, and then using the Fast Fourier Transform (FFT) to convert the sampled values to Chebyshev coefficients. Of course, it is not possible to store the coefficients of the infinite series on a computer. Chebfun uses the result that the coefficients of the series decay as more terms are used — once the `chebfun` construction process detects that the Nth coefficient of the series falls below machine precision, it stops sampling the function.[1] The terms in the series with $k < N$ are then used as the representation of the function.

To work with `chebfun` objects, one typically starts by creating a variable representing the independent variable on the interval $[a, b]$. Here, we wish to work on the interval $[0, 2]$; we pass the `chebfun` constructor a MATLAB anonymous function to sample on the grid and the desired interval:

```
>> x = chebfun(@(x) x, [0 2]);
```

Using operator-overloading of mathematical operations in MATLAB, one can then create more complicated functions, for example, a `chebfun` representing

$$f(x) = x \sin\left(\pi e^{2(x-1)^2}\right)$$

via

```
>> f = x.*sin(pi*exp(2*(x-1).^2));
```

Note that here, in order to multiply two `chebfun` objects, we used the `Matlab` notation of "dot-multiplication" of vectors, `.*`. This corresponds to the infinite dimensional analogue of element-wise multiplication of vectors. (Similarly, we used the syntax `.^2` for squaring a `chebfun`.)

In Figure 1.1, we show a plot of the `chebfun f`, and a plot of its the Chebyshev coefficients on a semi-log scale (obtained using the `chebpolyplot` method). We see that the first 30 or so coefficients are of same order, but for coefficients beyond those, the exponential decay, as stated in Theorem 1.1.6, is clearly visible.

More MATLAB commands have been overloaded in Chebfun. For example, evaluation of a `chebfun` at a given value of x is possible as follows:

[1] The stopping criterion for the construction process is actually a bit more involved than this, but the description given is sufficient for our purposes.

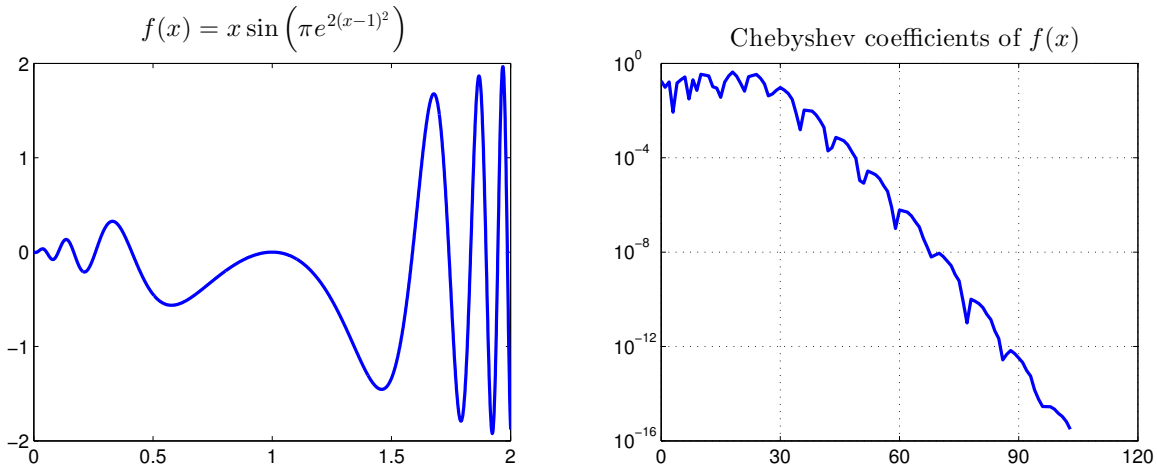

Figure 1.1: Plots of a `chebfun` `f` and its Chebyshev coefficients.

```
>> f(.5)
ans =
   −0.4464
```

Another example is the `length` command, which returns the number of Chebyshev points used to represent the function.

```
>> length(f)
ans =
   104
```

In this case, we needed 104 Chebyshev points, that is, a polynomial of degree 103, to represent f to machine precision.

To find the norm of the function, we call the `norm` method. By default, this returns the 2-norm of the function, but by passing a second argument, we can obtain its ∞-norm:

```
>> [norm(f) norm(f, inf)]
ans =
    1.0706    1.9675
```

Other major operations available include differentiation, integration, and rootfinding through the `diff`, `cumsum`, and `roots` methods respectively.

1.2 Spectral methods

1.2.1 Background

As mentioned above, a common approach to obtaining numerical solutions to differential equations is as a linear combination of basis functions,

$$u_N(x) = \sum_{k=0}^{N} a_k \varphi_k(x). \tag{1.13}$$

If the basis functions are global over the domain problem is defined on, such techniques are known as *spectral methods*. Spectral methods have a long and interesting history, dating back at least as far as the work of Lanczos [Lan38]. Here, we give a short overview of that history; for an excellent extended account, see [OT13].

The basic problems spectral methods are applied to are linear differential equations on $[-1, 1]$,

$$\mathcal{L}u = f, \quad \mathcal{B}u = c, \tag{1.14}$$

where $\mathcal{L}$ is a differential operator of order m, such that

$$\mathcal{L}u = \alpha^{(m)}(x)\frac{\mathrm{d}^m u}{\mathrm{d}x^m} + \cdots + \alpha^{(1)}(x)\frac{\mathrm{d}u}{\mathrm{d}x} + \alpha^{(0)}(x)u, \tag{1.15}$$

and $\mathcal{B}$ is an operator representing the k linear boundary conditions of the problem. The boundary conditions can be of Dirichlet, Neumann, or mixed type; homogeneous or non-homogeneous. The functions f and $\alpha^{(0)}, \ldots, \alpha^{(m)}$ are suitably smooth on $[-1, 1]$, and $c \in \mathbb{C}^k$.

Spectral methods can be categorised into collocation methods and coefficient methods. The former approach, also known as *pseudospectral* methods, obtains solutions via setting up matrices operating on the values of functions on the underlying grid, and solving the resulting linear system. The latter approach sets up matrices operating on the expansion coefficients in Equation (1.13), and solves the resulting linear system to obtain the coefficients directly.

A further division of spectral methods can be made by the way boundary conditions are imposed on the solution. One approach is *boundary bordering*, where k rows, representing the k boundary conditions of the problem, are added to the matrices arising before the linear system is solved. Another approach is to work in a problem-dependent basis, where each basis function satisfies the boundary conditions, so that the linear combination (1.13) will do so as well. However, as Orszag pointed out in [Ors71], such *basis recombination* techniques do not offer much benefit over the

more general boundary bordering technique, and in general, it is better to work with Chebyshev polynomials (given in Equation (1.2)), or Legendre polynomials. Furthermore, a general basis such as the Chebyshev basis is more convenient in practice, and enables constructing a black-box solver for a wide variety of problems.

The method Lanczos originally proposed in [Lan38] is known as the *tau method*. The tau method works on the coefficients of the solution in a Chebyshev basis, and uses boundary bordering for imposing boundary conditions. Further work on the tau method was for example later done by Ortiz [Ort69].

Traditionally, collocation methods were regarded as superior for linear differential equations with variable coefficients and nonlinear differential equations. Such methods thus became more popular in the 1970s and following decades. However, the recent work of Olver and Townsend [OT13], has reopened this question, and shown that it is possible to construct coefficient methods that in some cases are competitive or superior to collocation methods. This novel method has very desirable numerical properties, and works by setting the matrices up so that they map between different bases of ultraspherical functions.

Other approaches to spectral methods include Petrov-Galerkin methods [STW11], integral reformulation [Gre91, Dri10], and preconditioned iterative methods [Ors80, DM85, Elb06].

A classic treatment of spectral methods can be found in [GO77]. More recent, and computationally oriented textbooks include [For96, Boy00, Tre00].

1.2.2 Spectral collocation methods with a Chebyshev basis

Here, we focus on spectral collocation methods with a Chebyshev basis. The basic idea of such methods is as follows:

- Let a function f be given, and assume we want to obtain its derivative

$$f' = \mathcal{D}f, \tag{1.16}$$

  where $\mathcal{D}$ is the differentiation operator of the interval f is defined on.

- Construct the unique polynomial interpolant p_n of degree $\leq n$ to f on an $n+1$ point Chebyshev grid $X_n = \{x_j\}_{j=0}^n$, so that

$$p(x_j) = v_j = f(x_j), \quad 0 \leq j \leq n.$$

- Set

$$w_j = p'(x_j), \quad 0 \leq j \leq n, \tag{1.17}$$

and take the polynomial interpolant $w(x)$ of degree less than or equal to n through the data $\{w_j\}_{j=0}^n$ on X_n to be an approximation of $f'(x)$.

Just as differentiation is a linear operation on functions, computing the values w_j for $0 \leq j \leq n$ in (1.17) via differentiation of p_n is a linear operation on the $(n+1)$-vector v. Hence, it can be represented by a matrix multiplication. We denote that $(n+1) \times (n+1)$ matrix by D_{n+1}, so that

$$w = D_{n+1} v. \tag{1.18}$$

Comparing Equations (1.16) and (1.18), it is evident that the matrix D_{n+1} is the discretisation of the operator $\mathcal{D}$ on an $n+1$ point Chebyshev grid. Formulas for the entries of D_{n+1} are given for example in [WR00]. For higher order derivatives, one can either simply take powers of D_{n+1}, or construct them using explicit formulas [Pey02] or recurrences [Wel97, WR00].

Whereas differentiation matrices can be used to compute derivatives of functions, a more interesting application is to use them to solve differential equations of the form (1.14). The linear operator $\mathcal{L}$ from Equation (1.15) involves multiplication of the derivatives of u by the coefficient functions $\alpha^{(0)}, \ldots, \alpha^{(m)}$. Such multiplication corresponds to application of multiplication operators: linear operators $\mathcal{M}^{g(x)}$ defined by having the action

$$\mathcal{M}^{g(x)} : f(x) \mapsto g(x) f(x). \tag{1.19}$$

In pseudospectral methods, multiplication operators are the continuous analogues of diagonal matrices. Their discretisations, $M_{n+1}^{g(x)}$, will thus be diagonal matrices of dimensions $(n+1) \times (n+1)$, with the entries on the diagonal equal to the values of $g(x)$ on the $n+1$ gridpoints, obtained by sampling of g.

In order to obtain the discretisation of the operator $\mathcal{L}$ on an $(n+1)$-point Chebyshev grid, denoted by L_{n+1}, we now proceed as follows:

1. Compute the $(n+1)$-point discretisations

$$D_{n+1}^0,\, D_{n+1}^1, \ldots,\, D_{n+1}^m \qquad \text{and} \qquad M_{n+1}^{\alpha^{(0)}(x)},\, M_{n+1}^{\alpha^{(1)}(x)}, \ldots,\, M_{n+1}^{\alpha^{(m)}(x)}$$

of the operators

$$\mathcal{D}^0, \mathcal{D}^1, \ldots, \mathcal{D}^m \qquad \text{and} \qquad \mathcal{M}^{\alpha^{(0)}(x)}, \mathcal{M}^{\alpha^{(1)}(x)}, \ldots, \mathcal{M}^{\alpha^{(m)}(x)},$$

respectively. The construction of the discretisations of the multiplication operators require the sampling of the coefficient functions $\alpha^{(0)}, \ldots, \alpha^{(m)}$ on the Chebyshev grid. Note that $\mathcal{D}^0 = \mathcal{I}$, the identity operator, so D^0_{n+1} will simply be the $(n+1) \times (n+1)$ identity matrix.

2. Set $L_{n+1} = \sum_{j=0}^{m} M_{n+1}^{a^{(k)}(x)} D^k_{n+1}$.

When solving differential equations, it is necessary to impose conditions on the solutions, so that the problem is well-posed. We focus on the boundary bordering approach described above. The standard technique for boundary bordering is known as *row replacement*. Another technique for imposing conditions on the solution, known as *rectangular discretisation* of differential operator, has recently been introduced by Hale and Driscoll in [DH14]. It offers important benefits compared with row replacement, and removes ambiguity inherent in the standard method. We now describe with an example how row replacement proceeds, before discussing rectangular discretisation, which is the preferred technique in Chebfun.

Row replacement

Assume we have a standard second order, two-point BVP, with one Dirichlet condition imposed at each endpoint of $[-1, 1]$, that is

$$\alpha^{(2)}(x)u''(x) + \alpha^{(1)}(x)u'(x) + \alpha^{(0)}(x)u = f(x), \quad u(-1) = c_l,\, u(1) = c_r. \tag{1.20}$$

The method of row replacement also works for other type of boundary conditions (Neumann and mixed conditions), and can be extended to work for higher order problems, and problems where more than one condition is imposed at each boundary. Using row replacement, computations are carried out as follows:

1. We note that when discretised, the differential equation $\mathcal{L}u = f$ gives rise to the linear system

$$\begin{pmatrix} l_{0,0} & l_{0,1} & \cdots & l_{0,n-1} & l_{0,n} \\ l_{1,0} & l_{1,1} & \cdots & l_{1,n-1} & l_{1,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n-1,0} & l_{n-1,1} & \cdots & l_{n-1,n-1} & l_{n-1,n} \\ l_{n,0} & l_{n,1} & \cdots & l_{n,n-1} & l_{n,n} \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix}, \tag{1.21}$$

with $u_j = u(x_j)$ and $f_j = f(x_j)$. Another way to write (1.21) is as

$$L_{n+1}\mathbf{u} = \mathbf{f},$$

where L_{n+1} is an $(n+1) \times (n+1)$-matrix, and $\mathbf{u}$ and $\mathbf{f}$ are $(n+1)$-vectors.

2. However, the first and last rows of the linear system (1.21) are redundant, since the solution of the differential equation, u, is not governed by the values of f at $x = \pm 1$, but rather, the values of the boundary conditions, that is, c_l and c_r.

3. We thus remove the first and last row of L_{n+1} and the vector on the right-hand side:

$$\begin{pmatrix} \cancel{l_{0,0}} & \cancel{l_{0,1}} & \cancel{\cdots} & \cancel{l_{0,n-1}} & \cancel{l_{0,n}} \\ l_{1,0} & l_{1,1} & \cdots & l_{1,n-1} & l_{1,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n-1,0} & l_{n-1,1} & \cdots & l_{n-1,n-1} & l_{n-1,n} \\ \cancel{l_{n,0}} & \cancel{l_{n,1}} & \cancel{\cdots} & \cancel{l_{n,n-1}} & \cancel{l_{n,n}} \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} \cancel{f_0} \\ f_1 \\ \vdots \\ f_{n-1} \\ \cancel{f_n} \end{pmatrix}. \tag{1.22}$$

4. To impose the boundary conditions, described by the operator $\mathcal{B}$, we add rows to the linear system. The rows added are obtained by discretising the operator $\mathcal{B}$ on an $(n+1)$-point Chebyshev grid, to obtain the $2 \times (n+1)$ matrix B_{n+1}. Each row of B_{n+1} corresponds to one of the boundary condition; we add rows at the top and bottom of the linear system

$$\begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} & b_{0,n} \\ l_{1,0} & l_{1,1} & \cdots & l_{1,n-1} & l_{1,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n-1,0} & l_{n-1,1} & \cdots & l_{n-1,n-1} & l_{n-1,n} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,n-1} & b_{1,n} \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} c_l \\ f_1 \\ \vdots \\ f_{n-1} \\ c_r \end{pmatrix}. \tag{1.23}$$

   In the case of the problem (1.20), the boundary conditions rows will be all zero, apart from the entries operating on u_0 and u_n, which are equal to 1. Thus, the final linear system that gets solved is

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ l_{1,0} & l_{1,1} & \cdots & l_{1,n-1} & l_{1,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n-1,0} & l_{n-1,1} & \cdots & l_{n-1,n-1} & l_{n-1,n} \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} c_l \\ f_1 \\ \vdots \\ f_{n-1} \\ c_r \end{pmatrix}. \tag{1.24}$$

5. The solution of the linear system, $\mathbf{u} = [u_0, \ldots, u_n]^T$, is then used to construct a polynomial approximation on the $(n+1)$-point Chebyshev grid to the true solution of the problem, u.

Rectangular discretisation

The straightforward method of row replacement described above works well for standard scalar problems, with conditions imposed at the endpoints of the interval. However, if problems are more complicated, such as coupled systems of differential equations, or problems where constraints are imposed at interior points, row replacement runs into trouble. In that case, it is often not clear what rows need to be replaced, and what rows have to be kept.

In [DH14], the technique of *rectangular discretisation* of differential operators is introduced. One of the observation leading to rectangular discretisation is that differentiation decreases the degree of the polynomials used to represent the solutions. Hence, if a discretisation of a function is given as a polynomial of degree n on an $(n+1)$-point grid, the result of applying a linear differential operator of order m to the polynomial should result in a polynomial of degree $n-m$, that is, a polynomial on an $(n+1-m)$-point grid.

Rectangular discretisation also has the benefit of removing the ambiguity of what rows need to be replaced by rows imposing boundary or interior point conditions. Let n denote the discretisation used for $\mathcal{L}$, and denote the $(n+1)\times(n+1)$ matrix representing the discretised operator by $L_{n+1,n+1}$. Row replacement would simply remove rows from the square matrix $L_{n+1,n+1}$, to make space for boundary conditions. On the other hand, rectangular discretisation premultiplies $L_{n+1,n+1}$ with a rectangular *projection matrix* to give a new rectangular discretisation of $\mathcal{L}$. The rectangular projection matrix has $n+1$ columns, but only $n+1-m$ rows, where m is the differential order of $\mathcal{L}$. We denote the projection matrix by $P_{n+1\to n+1-m}$. Its role is to map discretisations on an $(n+1)$-point Chebyshev grid to an $(n+1-m)$-point Chebyshev grid.

Let $\widetilde{L}_{n+1-m,n+1}$ denote the rectangular differential operator obtained after premultiplying $L_{n+1,n+1}$ with $P_{n+1\to n+1-m}$, that is,

$$\widetilde{L}_{n+1-m,n+1} = P_{n+1\to n+1-m} L_{n+1,n+1}.$$

Once $\widetilde{L}_{n+1-m,n+1}$ has been obtained, rows that enforce conditions on the solutions are added at the bottom of the matrix. If the number of conditions, k matches the total differential order of the operator, m, this yields a square matrix $\widehat{L}_{n+1,n+1}$.

Similar computations are carried out on the $(n+1)$-vector $\mathbf{f}$ that represents the degree n polynomial interpolant to the function f on the right-hand side of the differential equation. First, $\mathbf{f}$ is projected to the $(n+1-m)$-point grid to give the

$(n+1-m)$-vector $\widetilde{\mathbf{f}}$ via $\widetilde{\mathbf{f}} = P_{n+1\to n+1-m}\mathbf{f}$. Then, the values of the boundary conditions are added to the bottom of $\widetilde{\mathbf{f}}$ to give a new $n+1$-vector $\widehat{\mathbf{f}}$. The discretised linear system defined by $\widehat{L}_{n+1,n+1}\mathbf{u} = \widehat{\mathbf{f}}$ is then solved to obtain a discretised solution $\mathbf{u}$ to the original differential equation.

By counting the number of derivatives for each unknown function in a problem, and constructing the projection matrices accordingly, the method of rectangular discretisation can be extended to work with systems of coupled differential equations.

1.2.3 Linear differential operators in Chebfun

In a system like Chebfun, where functions are represented on a Chebyshev grid, incorporating spectral differential operators to solve boundary-value problem seems like a natural thing. However, one of the main ideas of Chebfun is to be adaptive, that is, to determine at runtime how many gridpoints are needed. Whereas the formulas for constructing the differentiation matrices work for an arbitrary discretisation, it is not straightforward to determine how fine a discretisation is required, in particular, doing it in a way that is simple for the user.

The idea of creating differential matrices in an adaptive way is the main goal of the `linop` class (originally named the `chebop` class), described in [DBT08]. This is achieved by using anonymous functions in MATLAB to enable delayed evaluation of the operators, so that they can be instantiated, that is, their discretisations computed, for arbitrary sized grids.

To create a `linop`, one has to begin by creating a `domain` object, which defines the interval of interest.[2] In Section 3.6, we introduce syntax which most users of the software will use, but for now, we work with the original method of constructing `linop` objects. Differentiation operators can be constructed by calling the `diff` method on the `domain` object. For example, here we construct the second derivative operator $\mathcal{D}^2$ on the interval $[0,2]$:

```
>> d = domain(0, 2);
>> D2 = diff(d, 2)
D2 = linop
   Linear operator operating on chebfuns defined on:
     interval [0,2]
   with functional representation:
     @(u)diff(u,m)
   with n = 6 realization:
     41.6000  −68.3607   40.8276  −23.6393   17.5724   −8.0000
     21.2859  −31.5331   12.6833   −3.6944    2.2111   −0.9528
```

[2] We mention that in future versions of Chebfun, it is likely that this syntax will change.

```
   -1.8472    7.3167  -10.0669    5.7889   -1.9056    0.7141
    0.7141   -1.9056    5.7889  -10.0669    7.3167   -1.8472
   -0.9528    2.2111   -3.6944   12.6833  -31.5331   21.2859
   -8.0000   17.5724  -23.6393   40.8276  -68.3607   41.6000
  and differential order 2
```

The 6×6 matrix displayed corresponds to the discretisation of $\mathcal{D}^2$ on a six-point Chebyshev grid on the interval $[0, 2]$. If we had a 6-vector `v` with the values of a function $f(x)$ on that grid, then `w = D2*f` would result in a 6-vector which contained an approximation to f'' at that grid. By passing an integer value to the `linop`, we can ask for its instantiation on a grid with different number of nodes. For example, here we find the discretisation of $\mathcal{D}^2$ on a four point Chebyshev grid on $[0, 2]$:

```
>> D2(4)
ans =
    5.3333   -9.3333    6.6667   -2.6667
    3.3333   -5.3333    2.6667   -0.6667
   -0.6667    2.6667   -5.3333    3.3333
   -2.6667    6.6667   -9.3333    5.3333
```

To create a multiplication operator as defined in Equation (1.19), one calls the `diag` method with a `chebfun` argument. The resulting `linop` operates on functions defined on the same interval as the `chebfun` argument passed. Here we construct the operator $\mathcal{M}^{x^2}$:

```
>> Mx2 = diag(x.^2)
Mx2 = linop
   Linear operator operating on chebfuns defined on:
     interval [0,2]
   with functional representation:
     @(u)times(f,u)
   with n = 6 realization:
         0         0         0         0         0         0
         0    0.0365         0         0         0         0
         0         0    0.4775         0         0         0
         0         0         0    1.7135         0         0
         0         0         0         0    3.2725         0
         0         0         0         0         0    4.0000
```

Again, the discretisation of the operator on a six-point Chebyshev grid is displayed. The values on the diagonal of `Mx2` correspond to the values of the function x^2 at the nodes.

Via overloaded methods, `linop` objects can be combined to obtain more complicated operators. Operator arithmetic is possible for operators that operate on functions defined on the same interval. For example, to construct a `linop` representation

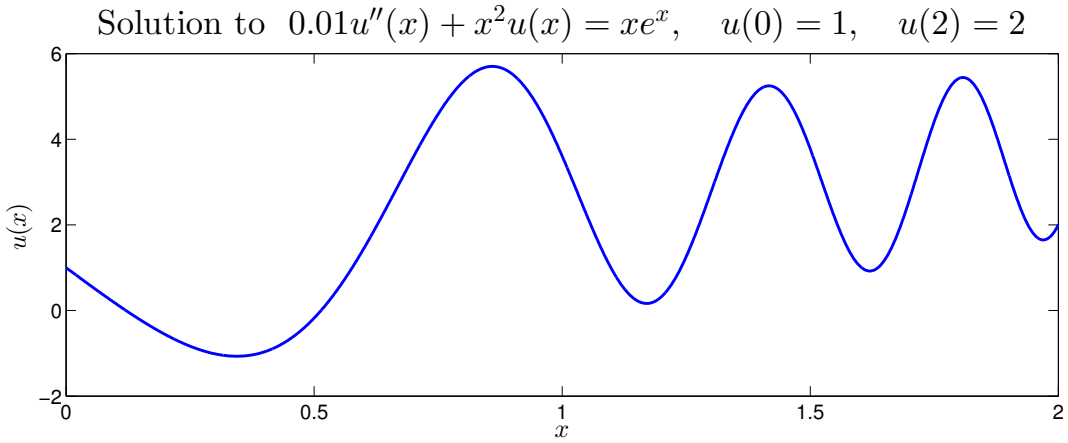

Figure 1.2: The Chebfun solution `u` to the BVP (1.25).

of the operator $\mathcal{L}$ such that

$$\mathcal{L} : u(x) \mapsto 0.01u''(x) + x^2u(x),$$

we call

```
>> L = 0.01*D2 + Mx2;
```

By assigning boundary conditions to `linop` objects, linear boundary-value problems of the form of (1.14), that is

$$\mathcal{L}u = f, \quad \mathcal{B}u = c,$$

where $\mathcal{L}$ represents a linear differential operator and $\mathcal{B}$ the linear boundary conditions, can be solved in few lines of MATLAB code. The problems can either be of scalar type, or coupled systems of ODEs. For example, to solve the linear BVP

$$0.01u''(x) + x^2u(x) = xe^x, \quad u(0) = 1,\ u(2) = 2, \tag{1.25}$$

we execute the following lines of code:

```
>> L.lbc = 1;
>> L.rbc = 2;
>> u = L\(x.*exp(x));
```

The solution returned, `u`, is a `chebfun` with `length(u) = 48`, and a residual 2-norm 7×10^{-13}. The resolution required to obtain this small residual was automatically determined by the system, and did not depend on the length of the `chebfun` on the right-hand side of the equation. The solution `u` is plotted in Figure 1.2.

As discussed in Section 1.2.2, rectangular discretisation is the technique currently used in Chebfun for linear differential operators. For example, here is the rectangularly discretised second order differentiation operator:

```
>> barymat(4,6)*D2(6)
ans =
   41.6000  −68.3607   40.8276  −23.6393   17.5724   −8.0000
    2.8000    0.4223   −8.1889    7.5777   −4.6111    2.0000
    2.0000   −4.6111    7.5777   −8.1889    0.4223    2.8000
   −8.0000   17.5724  −23.6393   40.8276  −68.3607   41.6000
```

Above, we employed the Chebfun method `barymat`, whose output is the projection matrix $P_{6\to 4}$, to multiply the square matrix `D2(6)`. The 4×6 matrix displayed corresponds to the rectangular discretisation of $\mathcal{D}^2$ on a six-point Chebyshev grid on the interval $[0, 2]$. Differently from the standard square discretisation of operators, now if we had a 6-vector `v` with the values of a function $f(x)$ on that grid, then `w = D2*v` would result in a 4-vector which contained an approximation to f'' on a four-point Chebyshev grid, rather than on a six-point grid.

1.3 Automatic differentiation

1.3.1 Computing derivatives in scientific computing

In various applications of scientific computing, derivatives play a central role. Given a function $f(x)$, the two best known methods for computing derivatives are:

- *Finite difference schemes.* Here, the standard idea is that, after having evaluated $f(x)$ on a grid $\{x_i\}_{i=0}^N$, the derivative(s) of f at a gridpoint x_k can be approximated using the values of f at x_k and nearby gridpoints. For example, on an equispaced grid with grid-spacing h, the centered approximation of the first derivative at $x = x_k$ is given by

$$f'(x_k) \approx \frac{-f(x_{k-1}) + f(x_{k+1})}{2h} = \frac{-f(x_k - h) + f(x_k + h)}{2h}. \tag{1.26}$$

  Finite difference schemes which only use information from one side of x_k can also be constructed, which is useful at the boundaries of intervals. Schemes which compute higher order derivatives can also be derived. There even exist schemes which work on grids with arbitrary spacing between gridpoints, and that can yield approximations of derivatives at a point $x = \xi$ which may or may not be a gridpoint. For details, see [For96].

- *Symbolic differentiation.* Here, a symbolic expression, such as $f(x) = \sin(x^2)$ is differentiated via lexical and grammatical analysis to yield a symbolic expression for the derivative, in this case, $f'(x) = 2x\cos(x^2)$. The symbolic expression obtained can then be evaluated for a given value of x to obtain the value of the derivative at that point.

However, both these methods suffer from drawbacks. Finite difference schemes suffer both from round-off errors and truncation errors (which usually dominate the round-off errors). For example, the centered approximation of the first derivative given above has truncation error $O(h^2)$ [For96]. On the other hand, symbolic differentiation depends on symbolic inputs and can experience a combinatorial explosion if the functions differentiated are more complicated than the simple example above. So while symbolic differentiation gives accurate derivatives, the time taken to obtain them might be very long. Also, memory requirements can be an issue.

Automatic, or algorithmic, differentiation (AD) aims at introducing techniques to overcome the drawbacks of the methods described above, while maintaining the desired properties. AD thus aims to compute accurate derivatives in a fast way, which only suffer from round-off errors from evaluation of functions but not truncation errors. Like finite difference schemes, AD works entirely at the numerical level. However, AD uses information about the functions it is differentiating to avoid truncation errors. Here we give a short introduction to AD, describing the essentials required for eventually implementing AD in Chebfun. For text-book references on AD, see [GW08, Nau12]. For a introductory article on the topic, see [Nei10], which describes how to implement standard AD techniques, in a similar way to what we do below.

1.3.2 The evaluation trace, and how it can be augmented to yield derivative values

A key concept in AD is that of the *evaluation trace*. As described below, it is by working with the evaluation trace that AD is able to yield derivative values. The evaluation trace is essentially a sequence of elementary calculations taken to evaluate a program with input variables $u_1, \ldots, u_n$, to obtain output variables $y_1, \ldots, y_m$. In the mathematical context, a program often means a function or an operator. The evaluation trace is defined as follows:

> *An evaluation trace is basically a record of a particular run of a particular program, with particular specified values for the input variables, showing the*

Assignment	Value of v^k
$v_{-1} = u_1$	2.0000
$v_0 = u_2$	0.5000
$v_1 = v_{-1} + v_0$	$2.0000 + 0.5000 = 2.5000$
$v_2 = v_0^2$	$0.5000^2 = 0.2500$
$v_3 = \sin(v_2)$	$\sin(0.2500) = 0.2474$
$v_4 = v_1 v_3$	$2.5000 \times 0.2474 = 0.6185$
$v_5 = v_{-1} + v_2$	$2.0000 + 0.2500 = 2.2500$
$v_6 = 4v_5$	$4.0000 \times 2.2500 = 9.0000$
$v_7 = v_4 + v_6$	$0.6185 + 9.0000 = 9.6185$
$y = v_7$	9.6185

Table 1.1: An evaluation trace for $y = (u_1 + u_2)\sin(u_2^2) + 4(u_1 + u_2^2)$.

> *sequence of floating point values calculated by a (slightly idealized) processor and the operators that computed them.* [GW08, p. 4]

Each part of the sequence is associated with what is known in the AD literature as an *intermediate variable*. The final output of the program is obtained by performing series of computations, using elementary operators such as cos, log, +, ×, with the intermediate variables. All the elementary operators are assumed to be either unary or binary.

At the start of a computation, we initialise the intermediate variables v_k, for $k \leq 0$, to be the input variables to the program. We denote the kth elementary operator in the sequence by φ_k, and its result by the intermediate variable v_k, for $k > 0$. For example, if φ_k is a binary operator, we have $v_k = \varphi_k(v_i, v_j)$, where v_i and v_j are intermediate variables, with $i, j < k$.

As an example of an evaluation trace, let y be the function of two variables given by the formula

$$y = (u_1 + u_2)\sin(u_2^2) + 4(u_1 + u_2^2). \tag{1.27}$$

Suppose that we want to obtain the value of y with $u_1 = 2$ and $u_2 = 0.5$. The evaluation trace for one implementation of that computation is shown in Table 1.1.

It is by working with derivative information of the intermediate variables in the evaluation trace that AD can yield derivatives of the output variables $y_1, \ldots, y_m$ with respect to the input variables $u_1, \ldots, u_m$. There are two techniques that can be used to convert the original program to a program which can return derivatives:

- *Operator overloading.* This technique works in programming languages which support operator overloading in object-oriented programming, such as C++ and MATLAB. The idea is to introduce a new class, which in addition to storing the value of variables, also stores the value of the derivatives of variables (or information required to compute the derivatives at a later time). Then, when operations are carried out on variables, not only do the variable values get updated, but also the derivatives/derivative information, according to the chain rule of calculus.

- *Source transformation.* Here, the source code of the original program is analysed to return a new source code which computes derivatives as well (again using the chain rule). The new source code is then compiled, so that when the resulting program is executed, it computes both the values of the outputs, as well as their derivatives.

As described in [BB00], both techniques have their advantages and disadvantages. We will eventually be interested in implementing AD in Chebfun, which as described above uses operator overloading in MATLAB. In Chebfun, the operator overloading approach to AD is thus the most natural one, so that technique will be our focus.

Both the techniques listed above rely on the fact that the chain rule can be incorporated into the computation. The two most common approaches for incorporating the chain rule are

- *Forward mode.* Here, we *seed*, that is, initialise, the values of the derivatives of the input variables. In subsequent calculations, the derivatives of the intermediate variables with respect to the input variables are computed. Eventually, this gives the derivatives of the output variables with respect to the input variables. The forward mode can be thought of as a "bottom-up" approach to AD — start by seeding the derivatives of the inputs, and end by obtaining the derivatives of the outputs. The computational cost of the forward mode AD is proportional to the number of input variables to the program, n.

- *Reverse mode.* Here, we seed the value of the derivatives of the output variables at the end of the evaluation trace. We then go backwards in the evaluation trace, using information stored at each step of the original computation. As we go backwards in the evaluation trace, then for each intermediate variable, we compute its *adjoint*, which denotes the sensitivity of the output variable(s) on

the intermediate variable. Eventually, we obtain the adjoints of the input variables, which corresponds to the derivatives of the output variables with respect to the inputs. The reverse mode can be thought of as a "top-down" approach to AD — start by seeding the derivatives of the outputs, and eventually obtain how sensitive the outputs are with respect to each input. The computational cost of the reverse mode AD is proportional to the number of output variables of the program, m.

Forward mode AD is simpler both conceptually and in implementation. The choice between forward mode or reverse mode depends on the number of input variables versus the number of output variables of the program one wants to differentiate. Let m and n denote the number of outputs and inputs respectively. Then, as shown in [GW08], the cost of computing Jacobians using forward mode AD is given by a multiple ω_{for} of the cost of evaluating the original program, where

$$\omega_{for} \in [1 + n, 1 + 1.5n].$$

On the other hand, the cost of computing derivatives using reverse mode AD is given by a multiple ω_{rev} of the cost evaluating the original program, where

$$\omega_{rev} \in [1 + 2m, 1.5 + 2.5m].$$

Thus, if $n \ll m$, the forward mode is superior. For example, if $n = 1$ (so that the program has just one input, but multiple outputs), we can obtain the derivatives of all outputs going just once through the evaluation trace. If on the other hand, $m \ll n$, the reverse mode is preferable. In the common case of optimisation, where a program depends on multiple inputs, but only has one output (so that $m = 1$), we can obtain the sensitivity of the output with respect to all the inputs by going just once through the evaluation trace.

It is also possible to combine forward and reverse modes, going back and forth in the evaluation trace. It would be desirable to employ such a technique to obtain derivatives with a minimal number of arithmetic operations. However, as shown in [Nau07], that task is NP-complete.

There are several software packages available which implement the different techniques and modes of AD, in various programming languages. To name but a few:

- ADIFOR. This package implements forward mode, source transformation AD for Fortran 77 source code. See [BKMC96].

- ADiMAt. Uses a hybrid approach of source transformation and object orientied programming techniques for implementing forward mode AD in MATLAB. See [BBL+02].

- ADOL-C. Written in C++, this package implements both forward and reverse mode AD, using operator overloading. See [WG12].

- The MATLAB toolbox INTLAB [Rum99] incorporates operator overloaded forward mode AD.

- Tapenade. An AD tool that supports both C and Fortran code, and uses source transformation for computing derivatives of programs. See [HP13].

- TOMLAB/MAD (MATLAB AD). Implements forward mode, operator overloaded AD in object-oriented MATLAB [For06]. An important application of MAD is in the software `bvp4cAD` [SKF05]. `bvp4cAD` solves boundary-value problems in MATLAB using the `bvp4c` code [KS01], but uses accurate derivatives supplied by AD rather than a finite difference scheme. In Section 3.6, `bvp4cAD` is one of the codes we compare our `chebop` class with.

The eventual employment of AD we seek in Chebfun is in the context of differential equations, where in general, the number of input and output variables is small. Thus, we allow ourselves to select the forward mode approach to AD, which will be our main focus when discussing AD, although we give a short example of reverse mode AD below.

Before continuing, we mention a useful alternative to forward mode AD, described by Squire and Trapp in [ST98]. The method uses complex variables for estimating the derivatives of real functions. It proceeds by replacing h with ih (where i is the imaginary unit number) in the expression for the finite difference approximation of the derivative of an analytic function f in (1.26). Taking the imaginary part of the result, one obtains an approximation of f' with error $O(h^2)$, but importantly, it is free of the effect of subtractive cancellation errors (albeit at the expense of having to evaluate f at a complex argument).

1.3.3 An example of forward mode AD

As mentioned above, forward mode, operator overloaded AD works by injecting statements into the evaluation trace to compute derivatives at the same time as the elementary operations are carried out. Let us assume that we want to compute the

Intermediate variable			Tangent variable		
$v_{-1} =$	u_1	$= 2.0000$	$\dot{v}_{-1} =$	$\dot{u}_1$	$= 0.0000$
$v_0 =$	u_2	$= 0.5000$	$\dot{v}_0 =$	$\dot{u}_2$	$= 1.0000$
$v_1 =$	$v_{-1} + v_0$	$= 2.5000$	$\dot{v}_1 =$	$\dot{v}_{-1} + \dot{v}_0$	$= 1.0000$
$v_2 =$	v_0^2	$= 0.2500$	$\dot{v}_2 =$	$2v_0\dot{v}_0$	$= 1.0000$
$v_3 =$	$\sin(v_2)$	$= 0.2474$	$\dot{v}_3 =$	$\cos(v_2)\dot{v}_2$	$= 0.9689$
$v_4 =$	$v_1 v_3$	$= 0.6185$	$\dot{v}_4 =$	$\dot{v}_1 v_3 + v_1\dot{v}_3$	$= 2.6696$
$v_5 =$	$v_{-1} + v_2$	$= 2.2500$	$\dot{v}_5 =$	$\dot{v}_{-1} + \dot{v}_2$	$= 1.0000$
$v_6 =$	$4v_5$	$= 9.0000$	$\dot{v}_6 =$	$4\dot{v}_5$	$= 4.0500$
$v_7 =$	$v_4 + v_6$	$= 9.6185$	$\dot{v}_7 =$	$\dot{v}_4 + \dot{v}_6$	$= 6.6696$
$y =$	v_7	$= 9.6185$	$\dot{y} =$	$\dot{v}_7$	$= 6.6696$

Table 1.2: A modified evaluation trace for $y = (u_1 + u_2)\sin(u_2^2) + 4(u_1 + u_2^2)$, using forward mode AD to compute the derivative with respect to u_2 at the same time.

derivatives of the output variables with respect to the input variable u_i. Then for each intermediate variable v_k, we introduce the associated *tangent variable*

$$\dot{v}_k := \frac{\partial v_k}{\partial u_i}.$$

At the kth step in the evaluation trace, where we compute $v_k = \varphi_k(v_i)$ or $v_k = \varphi_k(v_i, v_j)$ (depending on whether φ_k is unary or binary), we apply the chain rule to compute the value of $\dot{v}_k$. That value is obtained by only using the values of previously computed intermediate and tangent variables.

As an example, suppose that in the expression above,

$$y = (u_1 + u_2)\sin(u_2^2) + 4(u_1 + u_2^2),$$

with $u_1 = 2$ and $u_2 = 0.5$, we want to obtain the derivative of y with respect to u_2. For this, we then seed the derivative of u_1 to be 0, and that of u_2 to be 1 (since $\frac{\partial u_1}{\partial u_2} = 0$ and $\frac{\partial u_2}{\partial u_2} = 1$). This immediately gives $\dot{v}_{-1} = 0$ and $\dot{v}_0 = 1$. Then for example, since $v_1 = v_{-1} + v_0$, we obtain

$$\dot{v}_1 = \dot{v}_{-1} + \dot{v}_0 = 0 + 1 = 1.$$

The rest of the computations for obtaining $\partial y/\partial u_2$ are shown in Table 1.2.

1.3.4 An example of reverse mode AD

Operator overloaded, reverse mode AD also works by injecting statements into the evaluation trace. However, in contrast with forward mode, where we selected the input

variable we wanted to obtain derivatives with respect to, and found the sensitivity of all the intermediate variables with respect to that input, in reverse mode AD, we select an output variable and compute the sensitivity of the output with respect to all intermediate variables. Such sensitivities must be computed backwards; we start by obtaining the output and then work our way back up the evaluation trace.

In the example we have considered in this section, the only output variable is y. We now associate with each intermediate variable v_i a new variable,

$$\bar{v}_i = \frac{\partial y}{\partial v_i},$$

which is called the *adjoint* variable. As noted in [GW08], this is actually a slight abuse of derivative notation; what we actually mean is that $\bar{v}_i = \frac{\partial y}{\partial \delta_i}$, where δ_i is a new independent variable added to the right-hand side of the equation defining v_i. The adjoint can be interpreted as the effect of what adding a small numerical value δ_i to v_i will have to the value of y; to first order, the value of y changes by $\bar{v}_i \delta_i$.

We now transform the original evaluation trace to compute derivatives using reverse mode. By definition, $\bar{y} = 1$, and since intermediate variables can only affect the value of y in statements where they appear on the right-hand side, we have for example that

$$\bar{v}_2 = \bar{v}_5 + \bar{v}_3 \cos(v_2),$$

since v_2 only appears on the right-hand side in the statements $v_3 = \sin(v_2)$ and $v_5 = v_{-1} + v_2$. Rather than listing all contributions to $\bar{v}_2$ in one line, it is customary to split the assignments into parts corresponding to the original statements that spawned them, so we write

$$\begin{aligned} \bar{v}_2 &= \bar{v}_5, \\ \bar{v}_2 &= \bar{v}_2 + \bar{v}_3 \cos(v_2). \end{aligned}$$

We show the modified evaluation trace for computing derivatives of y using reverse mode AD in Table 1.3.

It is easy to work out that the analytical derivatives of y with respect to u_1 and u_2 in (1.27) are

$$\frac{\partial y}{\partial u_1} = \sin(u_2^2) + 4,$$

and

$$\frac{\partial y}{\partial u_2} = \sin(u_2^2) + 2(u_1 + u_2)u_2 \cos(u_2^2) + 8u_2.$$

$$
\begin{aligned}
v_{-1} &= u_1 = 2.0000 \\
v_0 &= u_2 = 0.5000 \\
v_1 &= v_{-1} + v_0 = 2.0000 + 0.5000 = 2.5000 \\
v_2 &= v_0^2 = 0.5000^2 = 0.2500 \\
v_3 &= \sin(v_2) = \sin(0.2500) = 0.2474 \\
v_4 &= v_1 v_3 = 2.5000 \times 0.2474 = 0.6185 \\
v_5 &= v_{-1} + v_2 = 2.0000 + 0.2500 = 2.2500 \\
v_6 &= 4v_5 = 4 \times 2.2500 = 9.0000 \\
v_7 &= v_4 + v_6 = 0.6185 + 9.0000 = 9.6185 \\
y &= v_7 = 9.6185 \\
\bar{v}_7 &= \bar{y} = 1.0000 \\
\bar{v}_6 &= \bar{v}_7 = 1.0000 \\
\bar{v}_4 &= \bar{v}_7 = 1.0000 \\
\bar{v}_5 &= \bar{v}_6 4 = 1.0000 \times 4 = 4.0000 \\
\bar{v}_2 &= \bar{v}_5 = 4.0000 \\
\bar{v}_{-1} &= \bar{v}_5 = 4.0000 \\
\bar{v}_3 &= \bar{v}_4 v_1 = 1.0000 \times 2.5000 = 2.5000 \\
\bar{v}_1 &= \bar{v}_4 v_3 = 1.0000 \times 0.2474 = 0.2474 \\
\bar{v}_2 &= \bar{v}_2 + \bar{v}_3 \cos(v_2) = 4.0000 + 2.5000 \times 0.9689 = 6.4233 \\
\bar{v}_0 &= \bar{v}_2 2v_0 = 6.4233 \times 2 \times 0.50000 = 6.4233 \\
\bar{v}_0 &= \bar{v}_0 + \bar{v}_1 = 6.4233 + 0.2474 = 6.6707 \\
\bar{v}_{-1} &= \bar{v}_{-1} + \bar{v}_1 = 4.0000 + 0.2474 = 4.2474 \\
\bar{u}_2 &= \bar{v}_0 = 6.6707 \\
\bar{u}_1 &= \bar{v}_{-1} = 4.2474
\end{aligned}
$$

Table 1.3: A modified evaluation trace for $y = (u_1 + u_2)\sin(u_2^2) + 4(u_1 + u_2^2)$, using reverse mode AD to compute the sensitivity of the output y with respect to all intermediate variables v_i. Eventually, this yields derivatives of y with respect to both u_1 and u_2 at the same time. We show the computation of adjoint variables directly below the lines in the original evaluation trace that spawned them.

If we evaluate the derivatives at $u_1 = 2$, $u_2 = 0.5$, we obtain

$$\left.\frac{\partial y}{\partial u_1}\right|_{u_1=2,\, u_2=0.5} = 4.2474, \qquad \left.\frac{\partial y}{\partial u_2}\right|_{u_1=2,\, u_2=0.5} = 6.6697.$$

We see that by using AD, we were able to obtain the derivatives accurately (up to rounding errors). However, at no point in neither forward nor reverse modes was the expression for the derivative constructed explicitly — we always worked at the numerical level.

We make a remark about the difference between forward and reverse mode AD that is evident from our example. In forward mode, we only obtained the derivative of y with respect to one of the chosen input variables, u_2. On the other hand, in reverse mode, we obtained the derivatives of y with respect to both u_1 and u_2 at the same time. This would also have been the case even if the number of input variables had been large. Then, we would still have been able to obtain the sensitivity of y with respect to all the inputs at the same time. Furthermore, that only would have required an additional number of operations that was a small additional multiple of the number of operations used to evaluate the original expression.

1.3.5 The `advar` class for demonstrating operator overloaded, forward mode AD

We refer to [GW08] for further discussion of reverse mode AD, and focus on forward mode AD for the rest of the thesis. To work with the operator overloaded, forward mode implementation of AD on a computer, we introduce the `advar` class, written in object-oriented MATLAB. It will be useful both for demonstrating the ideas of this standard implementation of AD, as well as for demonstrating how automatic linearity detection can be implemented in combination with AD in Chapter 4. We note that the `advar` class is not a part of Chebfun, and is thus not distributed with it. The source code of the `advar` class is available at [BD14].

In addition to storing the value of a variable, an `advar` object also stores the value of its derivative. Rather than only being able to compute derivatives with respect to one variable at a time as we did in Table 1.2, `advar` objects support derivatives on a vector form. Hence, by seeding the derivatives at the start of computation with

$$\dot{u}_i = [0,\, 0,\, \ldots,\, 1, \ldots,\, 0] \quad (i\text{th position}), \tag{1.28}$$

we can obtain gradients and Jacobians of the outputs.

Revisiting the example from above, we create `advar` objects to represent u_1 and u_2. The constructor of the `advar` class is called with two arguments. The first argument is the value of the variable, and the second argument is used to seed the derivative of the variable. We now create the `advar` objects `u1` and `u2` as follows (seeding derivatives according to Equation (1.28)):

```
>> u1 = advar(2, [1 0])
u1 =
      val: 2
      der: [1 0]
>> u2 = advar(.5, [0 1]);
```

The information printed about the variable `u1` shows both its value and its derivatives with respect to `u1` and `u2`.

By overloading MATLAB methods, we can carry out operations on `advar` objects. The overloaded methods will both compute the new value of the variables, as well as derivatives according to the chain rule. For example, the essential lines of the overloaded `cos` method are as follows:

```
function advout = cos( advin )
% Copy advin to obtain an object of advar type
advout = advin;
% Compute the value of the returned variable and its derivatives
advout.val = cos(advin.val);
advout.der = -sin(advin.val).*advin.der;
end
```

and the overloaded `plus` method is as follows:

```
function advOut = plus(adv1, adv2)
% Copy adv1 to obtain an object of the advar type
advOut = adv1;
% Compute the value of the returned variables, and its derivatives
advOut.val = adv1.val + adv2.val;
advOut.der = adv1.der + adv2.der;
```

We are now able to evaluate the example from above using `advar` objects. Rather than displaying information about the value and derivatives of the new `advar` variable `y`, we call the `getval` and `getder` methods, which return the value and derivatives respectively:

```
>> y = (u1+u2).*sin(u2.^2) + 4*(u1+u2.^2);
>> yval = getval(y)
yval =
    9.6185
>> yder = getder(y)
```

```
yder =
    4.2474    6.6697
```

The first and second entries of `yder` corresponds to $\partial y/\partial u_1$ and $\partial y/\partial u_2$ respectively. We see that we were able to compute both derivative values correctly at the same time. In fact, the derivatives computed with AD match the analytical ones exactly in machine arithmetic:

```
>> % Create variables to allow evaluation of analytical derivatives
>> v1 = 2; v2 = .5;
>> % Compute analytical derivatives
>> dydu = [sin(v2^2) + 4,  ...
           sin(v2^2) + 2*(v1+v2)*v2*cos(v2^2) + 8*v2];
>> % Compute error in derivative values
>> err = yder - dydu
err =
     0     0
```

These matches are exact because in this case, MATLAB internally used the same routines in the same order to evaluate the steps leading to the values of the vector `yder` and the values of `dydu1` and `dydu2`. In general, one should not expect this to happen. Rather, there will be a small discrepancy close to machine precision between the analytically and the AD computed derivates.

Chapter 2

Theory and Algorithms for Numerical Solution of Nonlinear ODEs by Newton Iteration in Function Space

This chapter describes the theory and algorithms behind the approach of using Newton iteration in function space for numerically solving boundary-value problems of nonlinear ordinary differential equations. In Section 2.1, we present the terminology used to describe ordinary differential equations. Section 2.2 gives the necessary background material from functional analysis for our treatment. Section 2.3 describes the standard Newton's method in Banach spaces, originally devised by Kantorovich, and shows an example of how it can be implemented in Chebfun (following an example in [BD12]). We then present the analysis and algorithms of affine invariant Newton methods devised by Deuflhard, described in [Deu06] and earlier papers. This is the main focus of this chapter, and the eventual algorithms we implement make significant use of the results of Deuflhard. Section 2.4 describes affine invariant local Newton methods, and Section 2.5 describes affine invariant global Newton methods, where damping is introduced to increase the domain of convergence. Finally, we give an example demonstrating the importance of affine invariance for algorithms based on Newton's method, and discuss the effects of discretisation errors and ghost solutions on Newton's method in function space in Section 2.6.

2.1 Ordinary differential equations

Before we discuss numerical methods for the solution of ordinary differential equations, we discuss some of the terminology used throughout this thesis. Our terminol-

ogy matches that widely used in the literature, for example, in [Fox57], [CP91] and [AMR95].

An *ordinary differential equation* (ODE) is an equation which involves the derivatives of one or more *dependent variables* with respect to a single *independent variable*. Often, the independent variable denotes time or space, and is denoted by t or x respectively. In general, we adopt x to denote the independent variable.

We call ODEs with only one dependent variable *scalar* ODEs, whereas ODEs with more than one dependent variable are known as *coupled* or *simultaneous* ODEs. In case of the former, we generally use $u = u(x)$ to denote the dependent variable. In the latter case, we use u_1, u_2, ... or u, v, ... to denote the dependent variables.

The *order* of a scalar ODE is that of the highest derivative appearing in it. Hence, a general scalar ODE of order m can be written in the form

$$\tilde{\phi}\left(x, u, u', \ldots, u^{(m-1)}, u^{(m)}\right) = r(x), \tag{2.1}$$

where $u^{(j)} = \frac{d^j u}{dx^j}$, and r is a function known as the *right-hand side* of the ODE. Often, we prefer to incorporate the right-hand side of the ODE into the operator, and thus rewrite (2.1) as

$$\phi\left(x, u, u', \ldots, u^{(m-1)}, u^{(m)}\right) = 0. \tag{2.2}$$

To further simplify our notation, we often omit derivatives of u from the arguments of ϕ, and describe the ODE as

$$\phi(x, u) = 0. \tag{2.3}$$

Equations that can be written in the form (cf. Equation (1.14))

$$a^m(x)u^{(m)} + a^{m-1}(x)u^{(m-1)} + \cdots + a^1(x)u' + a^0(x)u - r(x) = 0, \tag{2.4}$$

are known as *linear* ODEs. Any other type of equation is said to be *nonlinear*.

For coupled ODEs, there are usually the same number of equations as unknown dependent variables. As an example, a general coupled ODE with two dependent variables will be of the form

$$\Phi\left(x, u, u', \ldots, u^{(m)}, v, v', \ldots, v^{(m)}\right) = 0, \tag{2.5}$$

where

$$\Phi\left(x, u, u', \ldots, u^{(m)}, v, v', \ldots, v^{(m)}\right) = \begin{bmatrix} \phi_1\left(x, u, u', \ldots, u^{(m)}, v, v', \ldots, v^{(m)}\right) \\ \phi_2\left(x, u, u', \ldots, u^{(m)}, v, v', \ldots, v^{(m)}\right) \end{bmatrix}. \tag{2.6}$$

Making similar notational simplifications as described above, we arrive at the ODE description

$$\Phi(x,u,v) = \begin{bmatrix} \phi_1(x,u,v) \\ \phi_2(x,u,v) \end{bmatrix} = 0. \tag{2.7}$$

The equations in (2.7) do not necessarily have the same order. The sum of the orders of the highest derivative of both variables, appearing in either equations, is called the *total order* of the ODE. If both equations can be written in a form similar to Equation (2.4), that is, with coefficients which only depend on x, the coupled ODE is said to be linear. Otherwise, we say it is nonlinear. The above extends easily to coupled ODEs with more than two dependent variables.

Another important concept is the *degree* of and ODE. The degree of an ODE is the power of the highest derivative appearing in it. For example, the ODEs

$$u'' + (u')^2 + \cos(u) = 0 \qquad (u')^2 + u = 0$$

are of first and second degrees respectively.

The most general solution of an ODE of order n contains n arbitrary constants. This general solution is known is the *complete primitive*. It is by imposing conditions on the solution u that we obtain a *particular solution* to the ODE. Common conditions to be imposed on the solution are *linear boundary conditions*, a set of equations

$$\mathbf{B}(u) = \boldsymbol{\gamma}, \tag{2.8}$$

where each equation is in the form

$$\begin{aligned} B_i(u) &= \left[\alpha_i^0 u(x) + \alpha_i^1 u'(x) + \ldots + \alpha_i^{n-1} u^{(n-1)}(x)\right]\Big|_{x=a} \\ &+ \left[\beta_i^0 u(x) + \beta_i^1 u'(x) + \ldots + \beta_i^{n-1} u^{(n-1)}(x)\right]\Big|_{x=b} = \gamma_i. \end{aligned} \tag{2.9}$$

Here, $\alpha_i^0, \ldots, \alpha_i^{n-1}$, $\beta_i^0, \ldots, \beta_i^{n-1}$ and γ_i are given constants, and a and b are the left and right end-points respectively of the interval on which a solution is sought. Note that this formulation allows for non separated conditions, for example periodic boundary conditions, or a condition such as $u'(a) + 3u'(b) = 2$. More generally, we can impose conditions on the solution which do not only depend on its values (and the values of its derivatives) at the end-points of the interval, for example (assuming we are solving a problem on the interval $[a,b] = [0,1]$):

- Interior point conditions, such as $u\left(\frac{1}{2}\right) = 3$, or $u'\left(\frac{3}{4}\right) = -1$.
- Conditions on u applying over the whole domain, such as $\int_0^1 u(x)dx = 1$.

We note that such conditions correspond to functionals, mapping the function u to a real number.

The differential system, composed of the differential equation ϕ and the set of constraints $\mathbf{B}$, is then known as a *boundary-value problem*. Imposing boundary conditions extends in a straightforward way to coupled ODEs.

A common approach of algorithms for numerical solution of ODEs, some of which are discussed in Section 3.6.1, is to discretise equations (2.3) and (2.8), that is, introduce a grid $X = \{x\}_{i=0}^{n}$, and solve the algebraic system of equations that results. In the case of nonlinear ODEs, the algebraic system will also be nonlinear. However, the approach taken in this thesis is to stay in the function space framework, which is the natural setting given the mathematical description of the problem. Such an approach is often referred to as *quasilinearisation* in the literature. Eventually, any numerical algorithm will have to introduce discretisations, but we try to postpone the discretisation for as long as possible. Our approach will be to implement Newton's method in function space, discussed in Sections 2.3–2.5. In the next section, we discuss some concepts from functional analysis necessary for such a function space approach to numerical solution of ODEs.

2.2 Theoretical background

We now define some concepts necessary to describe our function space approach for numerical solution of ODEs. This enables us to discuss the Newton–Kantorovich theorem and related theorems about the convergence of Newton iteration in Banach spaces in Sections 2.3–2.5. We remind the reader of some important concepts, which are discussed extensively for example in [AH05, HPC05]. Other references on the functional analysis background include [KA64], [LL72], [Lax02] and [Cia13].

2.2.1 Banach and Hilbert spaces

A vector space V, equipped with a norm $\|\cdot\|_V$, is said to be a *Banach space* if it is complete; that is, if every Cauchy sequence $\{u_n\} \in V$ such that

$$\lim_{m,n\to\infty} \|u_m - u_n\|_V = 0$$

converges to an element of V. Throughout this thesis, we let the underlying field of V be $\mathbb{R}$, so that V will be a real Banach space. We call an element $v \in V$ a *point* in the space.

One of the benefits of working with Banach spaces is that we do not need to know the limit of the Cauchy sequence in order to be able to determine whether it is a Cauchy sequence. By contrast, to be able to determine whether a sequence converges by the standard definition, we need to know its limit. In general, when solving problems numerically, that limit will not be known.

Another important type of spaces are Hilbert spaces. A vector space V, equipped with an inner product function $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ is said to be a real *inner product space*. An inner product space V is said to be a *Hilbert space* if V is a Banach space under the norm induced by the inner product. In other words, Hilbert spaces are complete inner product spaces.

The problem of solving scalar ODEs, such as (2.3), is to find a point $u \in U$, where U is an open subset of a Banach space V specified by the problem, which satisfies the differential equation and the boundary conditions. In the case of coupled ODEs, such as (2.5), we seek solutions in an open subset of the *Cartesian product* $W := V_1 \times V_2$ of two Banach spaces V_1 and V_2. The Cartesian product of two Banach spaces is endowed with component-wise addition and scalar multiplication in the standard way.

We now state some standard results (found for example in [AH05, HN01]) about common function spaces and which ones of them are Banach/Hilbert spaces:

- The space $C[a, b]$ of continuous functions on $[a, b]$, equipped with the norm

$$\|u\|_p := \left[\int_a^b |u(x)|^p \mathrm{d}x \right]^{1/p},$$

  is not a Banach space, while $C[a, b]$ equipped with the ∞-norm

$$\|u\|_\infty := \max_{x \in [a,b]} |u(x)|,$$

  is.

- The space $C^m[a, b]$, consisting of functions on $[a, b]$ that are m times continuously differentiable, equipped with the norm

$$\|u\|_{C^m} := \left(\|u\|_\infty + \|u'\|_\infty + \ldots \left\|u^{(m)}\right\|_\infty \right) \tag{2.10}$$

  is also a Banach space.

- The linear space $L_2(a, b)$, consisting of measurable functions $u : (a, b) \to \mathbb{R}$ for which

$$\|u\|_2 = \sqrt{\int_a^b |u(x)|^2 \mathrm{d}x} < \infty,$$

is a Banach space. With the inner product

$$\langle u, v\rangle = \int_a^b u(x)v(x)\mathrm{d}x,$$

it is a Hilbert space. This space is called the *Lebesgue space* $L_2(a, b)$.

- The Sobolev space $H^p(a, b)$, which consists of functions whose weak derivatives (described below) of order less than or equal to p belong to $L_2(a, b)$, is a Hilbert space, with a norm

$$\|u\|_{H^p} := \left(\sum_{j=0}^{p}\int_a^b |u^{(j)}(x)|^2\mathrm{d}x\right)^{1/2}, \tag{2.11}$$

and an inner product

$$\langle u, v\rangle_{H^p} := \sum_{j=0}^{p}\int_a^b u^{(j)}(x)v^{(j)}(x)\mathrm{d}x.$$

For our definition of Sobolev spaces, we needed the notion of weak derivatives, which are defined as follows.

Definition 2.2.1. *[HN01, Def. 10.17] A function $u \in L_2(a, b)$ has a weak derivative $v = u' \in L_2(a, b)$ if*

$$\langle v, \varphi\rangle = -\langle u, \varphi'\rangle$$

for all $\varphi \in C_c^\infty(a, b)$, the set of test functions with compact support and continuous derivatives of all orders.

Sobolev spaces enjoy many useful properties. A property that will be important for us is that they can embedded into other spaces, which is defined as follows.

Definition 2.2.2. *[AH05, Def. 7.3.6] Let V and W be two Banach spaces with $V \subset W$. We say the space V is* continuously embedded *in W and write $V \hookrightarrow W$, if there exists a constant c such that*

$$\|v\|_W \leq c\,\|v\|_V \quad \forall v \in V.$$

The so-called Sobolev embedding theorems state what other spaces Sobolev spaces can be embedded in. In the one dimensional case, which is most relevant to us since we are considering ODEs, one obtains the result that

$$H^p(a, b) \hookrightarrow C[a, b] \tag{2.12}$$

for any $p \geq 1$. In other words, if u is a member of $H^p(a, b)$, derivatives of u up to degree $p - 1$ are continuous on $[a, b]$. For a further discussion, and the results of the other Sobolev embedding theorems (which depend on properties of the original Sobolev spaces), we refer to [AF03, Ch. 4] and [AH05, Ch. 7.3.3].

2.2.2 Operators between Banach spaces

Let two Banach spaces V and W be given. An *operator* or a *function* $\mathcal{T}$ from V to W is a rule which assigns to each element v in a subspace of V a unique element $w \in W$. This is denoted by $w = \mathcal{T}(v)$. Given an operator $\mathcal{T}$, its *domain*, $\mathrm{dom}(\mathcal{T})$, is the subset of V where $\mathcal{T}$ is defined:

$$\mathrm{dom}(\mathcal{T}) = \{v \in V \,|\, \mathcal{T}(v) \text{ is defined}\},$$

and its *range*, $\mathrm{ran}(\mathcal{T})$, is the set of all elements in W which are generated by $\mathcal{T}$:

$$\mathrm{ran}(\mathcal{T}) = \{w \in W \,|\, w = \mathcal{T}(v) \text{ for some } v \in \mathrm{dom}(\mathcal{T})\}.$$

Denoting the $\mathrm{dom}(\mathcal{T})$ by U, we write $\mathcal{T} : U \subset V \to W$.

$\mathcal{T}$ is said to be continuous at $v \in \mathrm{dom}(\mathcal{T})$ if

$$\{v_n\} \in \mathrm{dom}(\mathcal{T}) \text{ and } v_n \to v \text{ in } V \quad \Longrightarrow \quad \mathcal{T}(v_n) \to \mathcal{T}(v) \text{ in } W.$$

If $\mathcal{T}$ is continuous at every point in its domain, then $\mathcal{T}$ is said to be continuous. $\mathcal{T}$ is said to be bounded on its domain if there exists some finite $M > 0$ such that

$$\|\mathcal{T}v\|_W \leq M \|v\|_V \qquad \forall v \in \mathrm{dom}(\mathcal{T}).$$

Linear operators are an important type of operators. They satisfy the following property.

Definition 2.2.3. *[AH05, Def. 2.2.1] Let V and W be two linear spaces. An operator $\mathcal{L} : U \subset V \to W$ is said to be* linear *if*

$$\mathcal{L}(\mu u + \psi v) = \mu \mathcal{L}(u) + \psi \mathcal{L}(v), \quad \forall u, v \in U, \quad \mu, \psi \in \mathbb{R}. \tag{2.13}$$

In the case of linear operators, we often write $\mathcal{L}v$ rather than $\mathcal{L}(v)$. An operator not satisfying (2.13) is said to be nonlinear. Chapter 4 of this thesis focuses on how to implement a computational scheme to automatically detect whether an operator is linear.

We use $L(V, W)$ to denote the space of all continuous linear operators between the normed spaces V and W. An important special case is when $W = \mathbb{R}$, the underlying field of the real Banach space V. Elements of $L(V, \mathbb{R})$ are called *continuous linear functionals*, and the space $L(V, \mathbb{R})$ is called the *dual space* of V. The dual space of V is denoted by V', and is also a Banach space [HPC05].

It can be shown that for linear operators, continuity is equivalent to boundedness [HPC05, Thm. 3.4.3]. Thus, for an $\mathcal{L} \in L(V, W)$, we can define its *operator norm* as

$$\|\mathcal{L}\|_{V,W} = \sup_{0 \neq v \in V} \frac{\|\mathcal{L}v\|_W}{\|v\|_V}.$$

Bounded linear operators have many desirable properties. Whether an operator is bounded or not can depend on the domain it is defined on. For example, the differentiation operator

$$\mathcal{D} : C_\infty[a, b] \to C_\infty[a, b], \quad u \mapsto u',$$

where we equip C_∞ with the maximum norm, is unbounded, as can easily be seen with the choice $u(x) = \sin(\omega x)$. Therefore, it cannot be continuous. However, if we restrict ourselves to the space $C^1[a, b]$ or the Sobolev space $H^1(a, b)$, with norms given by (2.10) and (2.11) respectively, that is,

$$\mathcal{D} : C^1[a, b] \to C^0[a, b], \quad u \mapsto u',$$
$$\mathcal{D} : H^1(a, b) \to L_2(a, b), \quad u \mapsto u',$$

differentiation is a bounded operation. Similarly, the pth order differentiation operator $\mathcal{D}^p$, such that

$$\mathcal{D}^p : C^p[a, b] \to C^0[a, b], \quad u \mapsto u^{(p)},$$
$$\mathcal{D}^p : H^p(a, b) \to L_2(a, b), \quad u \mapsto u^{(p)},$$

is a bounded operator.

Finally, we remark that if $\mathcal{L} : U \subset V \to V$ is bijective, its inverse is defined as the operator $\mathcal{L}^{-1} : V \to V$ by $\mathcal{L}^{-1} f = u$, if and only if $\mathcal{L}u = f$. The range of $\mathcal{L}^{-1}$ will then be equal to the domain of $\mathcal{L}$.

2.2.3 Choice of function spaces

In this thesis, our focus is on two-point boundary value problems, where the differential equation is of the form

$$\mathcal{F}(u) = u'' + f(x, u, u') = 0,$$

where x is the independent space variable, and f is sufficiently smooth. As described below, implementing Newton's method in function space to seek a solution u^*, for which $\mathcal{F}(u^*) \approx 0$, requires the solution of a sequence of linear BVPs of the form

$$\delta u_k = -\mathcal{F}'(u_k)^{-1} \mathcal{F}(u_k),$$

where $\mathcal{F}'(u_k)$ is a linear operator known as the Fréchet derivative of $\mathcal{F}$ (defined in Section 2.2.4), and δu_k is the *continuous Newton update*.

Since the operators $\mathcal{F}'$ we encounter will be differential operators, their inverses will be integral operators. As asserted by a corollary of the Banach open mapping theorem, [Cia13, Thm. 5.6-2], if $\mathcal{F}'$ is bijective and bounded, $(\mathcal{F}')^{-1}$ will also be a bounded operator. Having $(\mathcal{F}')^{-1}$ bounded is a crucial assumption for many of the theorems about convergence of Newton's method presented below in Sections 2.3–2.5. Hence, for a second order ODE on the interval $[a, b]$, we restrict ourselves to finding solutions in the space $C^2[a, b]$ or the Sobolev space $H^2(a, b)$.

As will be evident below, the choice of the solution space and the residual space affects the error estimates, and therefore the convergence criteria of the Newton iteration. The approach of many collocation codes for solving nonlinear BVPs is to subdivide the interval $[a, b]$ into partitions, and obtain a solution that is globally continuous on $[a, b]$, but only piecewise sufficiently differentiable on each subinterval. This is for the example the situation analysed in [Deu06, Ch. 7.4], where the error is measured in the ∞-norm. In contrast, we will obtain solutions that are globally represented by `chebfun` objects, where the L_2 error is of more interest, see for example [Tre13, Ch. 16]. Hence, for a second order ODE, we will measure the errors in the solution space in the H^2 norm, and the residuals in the L_2 norm.

2.2.4 Derivatives of operators

Let a general operator $\mathcal{F} : U \subset V \to W$ be given, where V and W are Banach spaces, U is an open subset of V, and let u be a point in U. The notion of a Jacobian from ordinary vector calculus extends naturally to the setting of normed spaces, in which case it will be known as the *Fréchet derivative* of the operator $\mathcal{F}$ at the point u. It is defined as follows.

Definition 2.2.4. *[AH05, Def. 5.3.1][HPC05, Def. 4.4.1] Suppose that V and W are normed spaces, and let U be an open subset of V. Then $\mathcal{F} : U \subset V \to W$ is said to be* Fréchet differentiable *at $u \in U$ if and only if there exists an operator $\mathcal{L} \in L(V, W)$ such that*

$$\lim_{\|h\|_V \to 0} \frac{\|\mathcal{F}(u+h) - \mathcal{F}(u) - \mathcal{L}h\|_W}{\|h\|_V} = 0. \tag{2.14}$$

The limit is required to exist as $h \to 0$ in any manner, that is, given $\varepsilon > 0$, there exists $\delta > 0$ such that for $\|h\|_V < \delta$

$$\|\phi(u+h) - \phi(u) - Ah\|_W \leq \varepsilon \|h\|_V. \tag{2.15}$$

The operator $\mathcal{L}$ is then called the Fréchet derivative *of $\mathcal{F}$ at u, often denoted by $\mathcal{F}'(u)$. If $\mathcal{F}$ is Fréchet differentiable at all points in U, we say that $\mathcal{F}$ is Fréchet differentiable in U.*

From Equations (2.13) and (2.14), it is clear that if $\mathcal{F}$ is a member of $L(V, W)$, then it is its own Fréchet derivative. From (2.15), we see that if an operator is Fréchet differentiable at a point u, it is also continuous at u. Fréchet derivatives satisfy the usual properties one would expect of derivatives; that is, the addition, multiplication, and chain rules. This allows implementing Fréchet differentiation using the techniques of automatic differentiation as discussed in Chapter 3.

The notion of partial derivatives also extends to the setting of normed spaces, as discussed in [HN01, Sec. 13.3] and [AH05, Sec. 5.3]. If $\mathcal{F} : (U_1 \subset V_1) \times (U_2 \subset V_2) \to W$ is a differentiable map on the product of two Banach spaces, we have

$$\mathcal{F}(u_1 + h_1,\, u_2 + h_2) = \mathcal{F}(u_1, u_2) + \mathcal{L}_1 h_1 + \mathcal{L}_2 h_2 + o(h_1, h_2)$$

(see Section 2.2.5 for the definition of the small o notation) for suitable linear operators $\mathcal{L}_1 \in L(V_1,\, W)$ and $\mathcal{L}_2 \in L(V_2,\, W)$. We call $\mathcal{L}_1$ and $\mathcal{L}_2$ the partial derivatives of $\mathcal{F}$ with respect to u_1 and u_2 respectively, and denote them by

$$\mathcal{L}_1 = \frac{\partial \mathcal{F}}{\partial u_1}(u_1, u_2), \qquad \mathcal{L}_2 = \frac{\partial \mathcal{F}}{\partial u_2}(u_1, u_2).$$

Finally, the notion of higher order derivatives also extends to the setting of normed spaces, as discussed in [AH05, Sec. 5.3], [KA64, Sec. 17.2]. To be able to describe higher order derivatives, we need the notion of *bilinear maps*, which are defined as follows [KA64, Sec. 17.2]. Given two Banach spaces V and W, suppose that to each ordered pair of elements $u, v \in V$ there corresponds an element

$$w = \mathcal{P}(u, v) \in W.$$

Then $\mathcal{P} : V \times V \to W$ is said to be *bilinear* if the following two conditions are satisfied:

1. For all $u_1,\, u_2,\, v_1,\, v_2 \in V$ and numbers $\alpha,\, \beta$,

$$\mathcal{P}(au_1 + bu_2, v) = \alpha\mathcal{P}(u1, v) + \beta\mathcal{P}(u_2, v),$$
$$\mathcal{P}(u, av_1 + bv_2) = \alpha\mathcal{P}(u, v_1) + \beta\mathcal{P}(u, v_1),$$

   with $u, v \in V$.

2. There exists a real $M > 0$ such that, for all u, $v \in V$,

$$\|\mathcal{P}(u, v)\| \leq M \|u\| \|v\|.$$

The most elementary case of a bilinear map is when $V = W = \mathbb{R}$, where the general form of a bilinear operation will be

$$\mathcal{P}(u, v) = cuv, \qquad c \in R.$$

We observe that if an operator $\mathcal{F} : U \subset V \to W$ is Fréchet differentiable in U, where U is an open subset of the normed vector space V, the Fréchet derivative can be regarded as a mapping

$$\mathcal{F}' : U \subset V \to L(V, W).$$

If now the operator $\mathcal{F}'$ is Fréchet differentiable at a point $u_0 \in U$, we can speak of the derivative of $\mathcal{F}'$ at u_0, which is called the second derivative of $\mathcal{F}$, denoted by $\mathcal{F}''$. We then say that $\mathcal{F}$ is twice differentiable at u_0. If the second derivative exists at every point of U, we have that

$$\mathcal{F}'' : U \subset V \to L(V, L(V, W)). \tag{2.16}$$

As described in [KA64], $\mathcal{F}''$ can equivalently be regarded as an element of the space $L(V \times V, W)$, that is to say, $\mathcal{F}''$ is a bilinear map from $V \times V$ to W. We refer to [KA64] for further discussion of higher order Fréchet derivatives.

2.2.5 Big O and small o notation

Throughout this chapter, and in other sections of this thesis, we will make use of *big* O and *small* o notation, which we now define. We follow the standard definitions found in the literature, see for example [HN01].

Suppose that V and W are Banach spaces, and let $\mathcal{S} : U \subset V \to W$ be an operator, whose domain contains the origin of V. Then:

- We write

  $$\mathcal{S}(h) = o(h^n) \qquad \text{as } h \to 0,$$

  if

  $$\lim_{h \to 0} \frac{\|\mathcal{S}(h)\|}{\|h\|^n} = 0,$$

  meaning that $\|\mathcal{S}(h)\|$ approaches zero faster than $\|h\|^n$ as $h \to 0$.

- We write
$$\mathcal{S}(h) = O(h^n) \qquad \text{as } h \to 0,$$
if there are constants $\delta > 0$ and $C > 0$ such that
$$\|\mathcal{S}(h)\| \leq C\,\|h\|^n \qquad \text{when } \|h\| < \delta,$$
meaning that $\|\mathcal{S}(h)\|$ is bounded by a constant multiple of $\|h\|^n$ near 0.

If $\mathcal{S}, \mathcal{T} : U \subset V \to W$, we write
$$\mathcal{S}(h) = \mathcal{T}(h) + o(h^n) \quad \text{if} \quad \mathcal{S}(h) - \mathcal{T}(h) = o(h^n) \quad \text{as } h \to 0,$$
$$\mathcal{S}(h) = \mathcal{T}(h) + O(h^n) \quad \text{if} \quad \mathcal{S}(h) - \mathcal{T}(h) = O(h^n) \quad \text{as } h \to 0.$$

2.3 Classical Newton's method in Banach spaces

2.3.1 Historical background

Newton's method, and related techniques, are widely used approaches for solving nonlinear problems in mathematics. The basic idea behind Newton's method is linearisation. The history of Newton's method is interesting, and goes back centuries as described in historical accounts such as [Kol92], [Ypm95], [Deu06] and [Pol06].

Although Newton's method is is commonly traced back to Newton's *De analysi* published in 1669, already in the year 1600, Francois Vieta had designed a related perturbation technique for iteratively solving scalar polynomial equations (and there are accounts of related methods dating back to antiquity). The method proposed by Newton was mainly in the context of polynomial equations, although he also used it to find a solution x^* to Kepler's equation
$$x - e\sin(x) = M,$$
which describes the position of a planet moving in an elliptical orbit around the sun. (We refer to [Ypm95, Sec. 6] for further description of the problem, and the meaning of the variables e, M and x.) However, the procedure applied by Newton for this non-polynomial problem was based on geometrical arguments, and did not involve any calculus techniques.

In 1690, Raphson published a method in the context of polynomial equations, which used quotients of mathematical expression. He gave "recipes" for the required quotients of various expressions, but had not made the link to the derivatives of functions. An important advantage of Raphson's formulation was that it was possible

to reuse the quotient recipes for multiple iterations, while Newton had to recompute his at each iteration. Thus, Raphson's formulation improved the computational convenience greatly.

It was Simpson who in 1740 introduced the method now commonly known as Newton's method, where, given an equation

$$f(x) = 0, \tag{2.17}$$

and an initial guess of the solution x_0, successive approximations are computed via

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \tag{2.18}$$

where $f'(x_k)$ is the derivative of f at x_k. (The modern notation of Equation (2.18) was actually not used by Simpson, rather, it was introduced by Lagrange.) Simpson treated more general problems than his predecessors, including a square root function and a reciprocal. Fourier proved quadratic convergence of the method in a neighbourhood of a root, and Cauchy extended the method to multiple dimensions, where for an equation $F(x) = 0$, successive approximations are computed via

$$x_{k+1} = x_k - F'(x_k)^{-1} F(x_k),$$

where now, $F'(x_k)$ is the Jacobian of F at x_k.

2.3.2 Newton–Kantorovich theory

It was Kantorovich in 1948 who made the important contribution of generalising Newton's method to Banach spaces [Kan48]. He considered an equation similar to Equation (2.17),

$$\mathcal{F}(u) = 0,$$

but now, $\mathcal{F}$ is an operator, $\mathcal{F} : U \subset V \to W$, where V and W are Banach spaces, and U is an open subset of V. Kantorovich proposed the method

$$u_{k+1} = u_k - \mathcal{F}'(u_k)^{-1} \mathcal{F}(u_k).$$

Here, $\mathcal{F}'(u_k)$ is the Fréchet derivative of the nonlinear operator $\mathcal{F}$ at u_k, and $\mathcal{F}'(u_k)^{-1}$ is its inverse. In practice, problems are solved by computing the *Newton update* δu_k via

$$\mathcal{F}'(u_k)\delta u_k = -\mathcal{F}(u_k), \quad u_{k+1} = u_k + \delta u_k, \tag{2.19}$$

Algorithm 1 `NIB`: Newton iteration in Banach spaces.

Input: An operator, $\mathcal{F}$. An initial guess of a solution, u_0.
Output: A solution u, for which $\mathcal{F}(u) \approx 0$.

Evaluate and store $\mathcal{F}_0 = \mathcal{F}(u_0)$.
$k := 0$.
repeat
  Compute the Fréchet derivative of $\mathcal{F}$ at u_k, $\mathcal{F}'(u_k)$.
  Compute the Newton update δu_k via $\mathcal{F}'_k \delta u_k = -\mathcal{F}_k$.
  $u_{k+1} \leftarrow u_k + \delta u_k$.
  $k \leftarrow k + 1$.
  Evaluate and store $\mathcal{F}_k = \mathcal{F}(u_k)$.
until $\mathcal{F}_k \approx 0$
$u \leftarrow u_k$.

which avoids computing the inverse $\mathcal{F}'(u_k)^{-1}$. From Equation (2.19), it is clear that the original problem will be solved via solving a sequence of linear problems.

We can now describe our first algorithm, the basic Newton iteration in Banach spaces, listed as Algorithm 1. Its inputs are the operator $\mathcal{F}$ and an initial guess u_0. Its output, if it converges, is a solution u, for which $\mathcal{F}(u) \approx 0$ (as we cannot expect a numerical algorithm to give the exact solution u^* for which $\mathcal{F}(u^*) = 0$). Subsequent sections introduce convergence criteria which we use to determine whether a current guess of the solution is sufficiently good to be returned. For now, it suffices to think of the convergence criteria as being related to the norm of the updates, $\|\delta u_k\|$, or the norm of the residual, $\|\mathcal{F}(u_k)\|$.

It is well known that algorithms based on Newton iteration, such as Algorithm 1, will not necessarily converge to a solution. If the algorithm starts with a bad initial guess u_0, it can diverge. The following Theorem 2.3.1, about the local convergence of the Newton method, shows that if we start with an initial guess u_0 in a neighbourhood of a solution u^*, it convergences quadratically. For the theorem, let V and W denote Banach spaces, and let U be an open subset of V. A proof can be found in [AH05, p. 237].

Theorem 2.3.1. *[AH05, Thm. 5.4.1] Let be Fréchet differentiable, U open and convex. Assume that u^* is a solution of the equation $\mathcal{F}(u) = 0$ (that is, $\mathcal{F}(u^*) = 0$), such that $[\mathcal{F}'(u^*)]^{-1}$ exists and is continuous (thus bounded). Assume further that $\mathcal{F}'(u)$ is locally Lipschitz continuous at u^*,*

$$\|\mathcal{F}'(u) - \mathcal{F}'(v)\| \leq L \|u - v\|, \quad \forall\, u, v \in N(u^*),$$

where $N(u^*)$ is a neighbourhood of u^*, and $L > 0$ is a constant. Then there exists a $\delta > 0$ such that if $\|u_0 - u^*\| \leq \delta$, the Newton sequence $\{u_n\}$ is well-defined and converges to u^*. Furthermore, for some constant M, with $M\delta < 1$, we have the error bounds

$$\|u_{n+1} - u^*\| \leq M \|u_n - u^*\|^2 ,$$

and

$$\|u_n - u^*\| \leq (M\delta)^{2^n}/M.$$

This theorem guarantees that under the assumptions stated, Newton's method will converge quadratically to a solution. However, it has to assume the existence of a solution, and does not say anything about its uniqueness. Also, the assumptions depend on properties of $\mathcal{F}$ at the unknown u^*, rather than at the known initial guess u_0.

The *Newton–Kantorovich theorem* overcomes these problem. It states conditions which only depend on the Lipschitz constant of $\mathcal{F}'$, and the initial guess of the solution, u_0. Furthermore, it serves as an existence theorem about a solution of the original equation. In other words, based on properties which only depend on u_0, the existence of a solution can be guaranteed, and the Newton iteration will be guaranteed to converge. The theorem is originally due to Kantorovich in 1948 [Kan48]. Another proof can be found in [Zei85, p. 210]. Since the standard Newton–Kantorovich theorem is a corollary of the affine invariant version, Theorem 2.4.1, we do not present it here, but refer to Section 2.4.

While the Newton–Kantorovich theorem gives sufficient conditions for the convergence of Newton's method in Banach spaces, the values of the quantities required, such as the Lipschitz constant of the Fréchet derivative of $\mathcal{F}$, are difficult or impossible to obtain in practise. However, as described in the following section, by constructing computational estimates of the quantities, and modifying convergence theorems to incorporate those computational estimates, we will be able to construct algorithms which have a solid foundation in theoretical results.

2.3.3 An example of Newton iteration in function space

Before we discuss the ideas of affine invariant Newton methods in Sections 2.4 and 2.5, and describe more complicated algorithms than Algorithm 1, we show a typical example of how Newton's method in function space for solving a nonlinear BVP works.

Suppose we have a boundary-value problem of the form described in Section 2.1,

$$\phi(x,u) = 0, \quad \mathbf{B}(u) = \boldsymbol{\gamma}, \tag{2.20}$$

where $u = u(x)$ is a function, ϕ is a differential operator, and $\mathbf{B}$ is a set of boundary conditions to be imposed on u. We shall often refer to (2.20) simply as $\mathcal{F}(u) = 0$, that is, here we let $\mathcal{F}$ denote a BVP operator.

The Newton iteration in function space will produce a sequence of functions $u_1, u_2, \ldots$ such that if $u_k \to u^*$ as $k \to \infty$, then $\mathcal{F}(u^*) = 0$. The crucial component of the Newton iteration is the linearization of $\mathcal{F}$ about each u_k, which requires the Fréchet derivatives of ϕ and each functional comprising $\mathbf{B}$, as defined in Definition 2.2.4.

At the kth step of the iteration, we replace the operator $\mathcal{F}(u_k+\delta u)$ for a perturbation δu by the linearization $\mathcal{F}(u_k)+\mathcal{F}'(u_k)\delta u$, and find a root of this approximation to determine the *continuous Newton update*. Hence we obtain the linear boundary-value problem

$$\phi'(u_k)\delta u = -\phi(u_k), \tag{2.21}$$

$$\mathbf{B}'(u_k)\delta u = \boldsymbol{\gamma} - \mathbf{B}(u_k), \tag{2.22}$$

or using the notation introduced above,

$$\mathcal{F}'(u_k)\delta u = -\mathcal{F}(u_k).$$

Solving this problem for the function δu to obtain the update δu_k, we define $u_{k+1} = u_k + \delta u_k$ and iterate.

Let us now be explicit about the process for solving the example nonlinear BVP

$$0.01u'' + uu' - xu = 0, \quad u(0) = -7/6, \quad u'(1) = 3/2. \tag{2.23}$$

Identifying with (2.20), we have $\phi(x,u) = 0.01u'' + uu' - xu$, and the boundary condition functionals are given by $B_1(u) = u(x)\big|_{x=a}$ and $B_2(u) = u'(x)\big|_{x=b}$. We obtain the relevant linearizations in standard perturbation fashion:

$$\begin{aligned}\phi(u+v) - \phi(u) &= 0.01v'' + (u+v)(u+v)' - xv - uu' \\ &= 0.01v'' + uv' + vu' - xv + O\big(\|v\|^2\big).\end{aligned}$$

Upon dropping high-order terms, this leads to the Fréchet derivative

$$\phi'(u) = 0.01\frac{d^2}{dx^2} + u\frac{d}{dx} + u' - x.$$

This linear operator consists of parts that perform differentiation and parts that perform pointwise multiplication, so that

$$\phi'(u) : v(x) \mapsto 0.01v''(x) + uv' + u'v - xv.$$

Proceeding similarly for the boundary conditions, we find that the Fréchet derivatives of B_1 and B_2 are the linear functionals

$$B_1'(u) : v \mapsto v(x)\big|_{x=0},$$
$$B_2'(u) : v \mapsto v'(x)\big|_{x=1}.$$

Note that since both boundary conditions are linear, $B_1'(u)$ and $B_2'(u)$ are independent of u. Also note that $B_2'(u)$ contains a differentiation term. Altogether, the Newton update δu_k is defined as the solution of the linear BVP

$$0.01\delta u'' + u\delta u' + u'\delta u - x\delta u = -\phi(u_k),$$
$$\delta u(0) = -7/6 - u_k(0), \qquad \delta u'(1) = 3/2 - u_k'(1).$$

We now use Chebfun to demonstrate how the BVP (2.23) is solved using Newton iteration in function space. We use the `linop` class discussed in Section 1.2.3 to represent the linear operators involved. The iteration stops when the norm of the Newton update falls below 10^{-10}, which takes 8 iterations with the initial guess $u_0(x) = \frac{3}{2}x - \frac{7}{6}$. In Figure 2.1 we plot the sequence of solution estimates, with an additional axis showing the cumulative 2-norms of the update functions. After the first three or so iterations, the quadratic convergence of the iteration can be readily confirmed from the numerical values of the residuals. Below, we show the code used for the Newton iteration.

```
dom = domain(0,1);                    % Problem domain
D = diff(dom); D2 = diff(dom, 2);     % Differentiation operators
x = chebfun(@(x) x, d);               % The chebfun 'x' on the domain
% MATLAB anonymous function for the ODE part of BVP
phi = @(x,u) .01*diff(u, 2) + u.*diff(u) - x.*u;
u = 3*x/2 - 7/6;                      % Initial guess
nrmdelu = 1;                          % Initialize variables
while nrmdelu > 1e-10                 % Newton iteration
  % linop representing the Frechet derivative at u
  L = 0.01*D2 + diag(u)*D + diag(diff(u) - x);
  Du = diff(u);                       % Needed to compute the rbc
  % Assign left and right boundary conditions to the linop L
  L.lbc = -7/6 - u(0);                % Dirichlet condition at x=0
  L.rbc = {'neumann', 3/2 - Du(1)};   % Neumann condition at x=1
  delu = L\(-phi(x, u));              % Solve the linearized BVP
  u = u + delu;                       % Take Newton step
  nrmdelu = norm(delu);               % 2-norm of Newton update
end
```

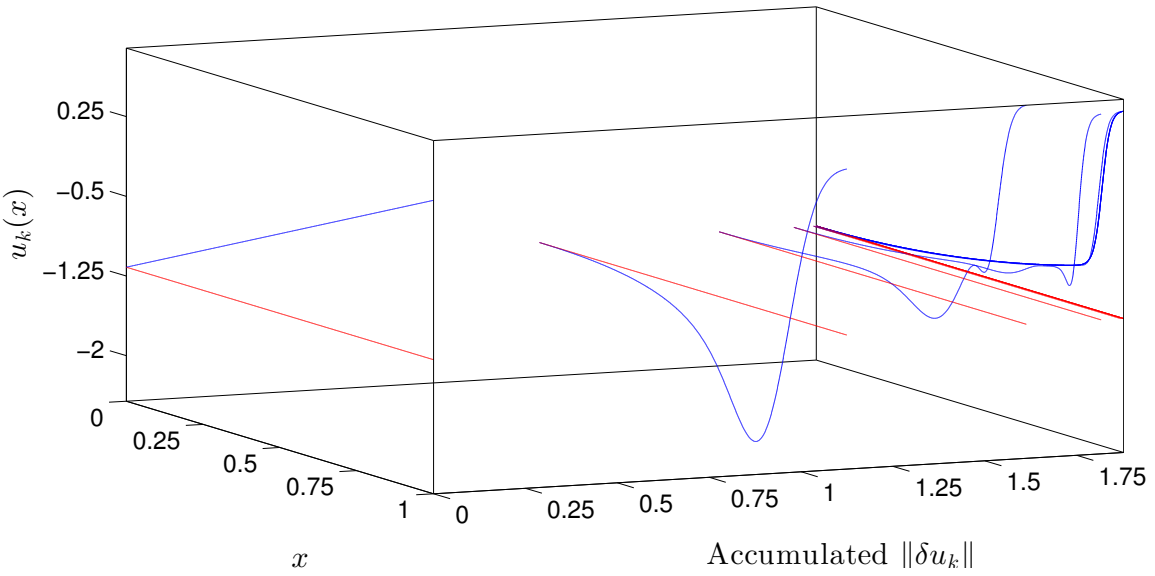

Figure 2.1: The solution of the BVP (2.23) by eight steps of Newton iteration in function space. The y axis shows the cumulative sum of $\|\delta u_k\|_2$ for the Newton update functions δu_k. Note that each red line corresponds to one iteration, and as we get closer to a solution, the lines cluster (indicating that the updates are getting smaller). After some initial large updates, the iteration eventually converges quadratically to the solution.

2.4 Affine invariant Newton methods

We now focus on the *affine invariant* framework developed by P. Deuflhard in the 1970s, treated comprehensively in [Deu06]. Our presentation will follow these steps:

- First, we discuss the effect that affine transformations of the domain and image spaces have on Newton's method.
- We then discuss the importance of Lipschitz constants and operator norms for theorems about the convergence of Newton methods, and how to obtain computational estimates for these quantities which are difficult to obtain analytically.
- By combining affine transformations and Lipschitz constants, we obtain Lipschitz constants which are invariant under affine transformations. This allows us to modify convergence theorems for Newton's method, resulting in affine invariant convergence results for Newton's method.
- The computational estimates of the affine invariant Lipschitz constants can then be used as a basis for algorithms implementing Newton's method in function

space, where they will allow us to monitor convergence and give error estimates. We will give both error-oriented and residual-oriented algorithms.

2.4.1 Affine transformations

Let $\mathcal{F} : U \subset V \to W$ be an operator, where V and W are Banach spaces, and U is an open subset of V. It is clear that the standard Newton iteration for solving nonlinear BVPs as described in Section 2.3,

$$\mathcal{F}'(u_k)\delta u_k = -\mathcal{F}(u_k), \quad u_{k+1} = u_k + \delta u_k, \tag{2.24}$$

is sensitive to scaling or re-gauging of variables. Given nonsingular *scaling operators* $\mathcal{M}_L$ and $\mathcal{M}_R$ for left and right scaling, (2.24) can be rewritten as

$$(\mathcal{M}_L\mathcal{F}'(u_k)\mathcal{M}_R)(\mathcal{M}_R^{-1}\delta u_k) = -\mathcal{M}_L\mathcal{F}(u_k). \tag{2.25}$$

Even though (2.24) and (2.25) are mathematically equivalent, all norms used in algorithms implementing Newton's method have to be replaced by their scaled counterparts, for example,

$$\|\delta u_k\| \longrightarrow \left\|\mathcal{M}_R^{-1}\delta u_k\right\|, \qquad \|\mathcal{F}(u_k)\| \longrightarrow \|\mathcal{M}_L\mathcal{F}(u_k)\|.$$

More generally, we consider affine invariant transformations of the nonlinear operator equation $\mathcal{F}(u) = 0$. Given arbitrary bounded linear bijective operators $\mathcal{A} : W \to Z$ (where Z is some Banach space) and $\mathcal{B} : U \to U$, we study affine transformations of the original operator equation as

$$\mathcal{G}(\hat{u}) = \mathcal{A}\mathcal{F}(\mathcal{B}\hat{u}) = 0, \quad u = \mathcal{B}\hat{u}. \tag{2.26}$$

Applying Newton's method to the equation $\mathcal{G}(\hat{u}) = 0$ then gives

$$\mathcal{G}'(\hat{u}_k)\delta\hat{u}_k = -\mathcal{G}(\hat{u}_k), \quad \hat{u}_{k+1} = \hat{u}_k + \delta\hat{u}_k.$$

Calculus with the Fréchet derivatives of the operators involved gives

$$\mathcal{G}'(\hat{u}_k) = \mathcal{A}\mathcal{F}'(\mathcal{B}\hat{u}_k)\mathcal{B} = \mathcal{A}\mathcal{F}'(u_k)\mathcal{B},$$

so with the starting guess $\hat{u}_0 = \mathcal{B}^{-1}u_0$, we find that

$$u_k = B\hat{u}_k, \quad k = 0,\, 1,\, \ldots. \tag{2.27}$$

From (2.27), it is clear that the iterates are invariant under the transformation of the image space, that is, the effect of $\mathcal{A}$. Such an invariance property is known in the

literature as *affine covariance*. Furthermore, they are transformed in the same way as the whole original domain space by $\mathcal{B}$. Such an invariance property is called *affine contravariance*. Ideally, we would want any convergence theorem for Newton iteration to maintain both invariance properties. However, maintaining the full invariance property turns out to be impossible [Deu06]. There is evidence in the literature that for the numerical solution of nonlinear BVPs for ODEs, error oriented Newton methods are preferable to residual based Newton methods, which can run into problems when problems are ill-conditioned [DNW02, Deu06]. Furthermore, as we will see below, the error oriented methods give an estimate of the error in the solution space norm, which is usually of greater interest than an estimate of the error in the image space norm. Since error oriented algorithms correspond to affine covariance while residual based algorithms correspond to affine contravariance, the former will be our main focus. Furthermore, the default algorithm we implement in Chebfun for solving nonlinear BVPs will be affine covariant.

2.4.2 The algorithmic paradigm

From Section 2.3, it is clear that the convergence of Newton's method depends on mathematical quantities such as the Lipschitz constant of $\mathcal{F}'$. Also, it requires that the derivatives $\mathcal{F}'(u)$ are invertible for all members of the Newton sequence, so a standard requirement is that $[\mathcal{F}'(u)]^{-1}$ exists and is bounded:

$$\left\|\mathcal{F}'(u)^{-1}\right\|_{U\subset V\to W} \leq \beta < \infty, \quad \forall u \in U.$$

In order to determine whether we should expect Newton's method to work for particular problems, it would thus be desirable if algorithms were able to acquire information about such operator norms and Lipschitz constants. However, it is virtually a computationally impossible task to obtain the operator norm of $\mathcal{F}'^{-1}$ over the whole $\mathrm{dom}(\mathcal{F})$. Even local estimates, such as

$$\left\|\mathcal{F}'(u_0)^{-1}\right\|_{U\subset V\to W} \leq \beta_0,$$

are difficult to obtain.

The idea advocated by proponents of affine invariant Newton methods to overcome these issues is known as the *algorithmic paradigm* [Deu06, p. 20]. The essential ingredient is that algorithms implementing Newton's method should depend on cheaply available affine invariant computational estimates of affine invariant Lipschitz constants.

To describe this further, assume that a convergence proof of Newton iteration depends on a theoretical Lipschitz constant ω, defined over a nonempty domain D, such that

$$\omega = \sup_{u,v,w \in D} g(u, v, w),$$

where g is a scalar expression, containing only affine invariant terms. If we have three members of D, $\hat{u}$, $\hat{v}$ and $\hat{w}$, we can obtain a computational local estimate of ω by

$$[\omega] = g(\hat{u}, \hat{v}, \hat{w}),$$

adopting the notation of $[\cdot]$ for computational estimates of theoretical quantities. It is clear that by construction, ω and $[\omega]$ share the same affine invariance property, and

$$[\omega] \leq \omega.$$

As an example, an affine covariant Lipschitz constant which appears in one of the convergence theorems to follow (Theorem 2.4.1) is given by rewriting (2.28) to give

$$\omega_0 = \sup_{u,v \in U} g(u, v, u_0) := \sup_{u,v \in U} \frac{\|\mathcal{F}'(u_0)^{-1}(\mathcal{F}'(u) - \mathcal{F}'(v))\|}{\|u - v\|}.$$

As a part of the Newton iteration, we compute the iterates u_0 and u_1, which can then be used to give an estimate of ω_0 by

$$[\omega_0] = g(u_1, u_0, u_0) = \frac{\|\mathcal{F}'(u_0)^{-1}(\mathcal{F}'(u_1) - \mathcal{F}'(u_0))\|}{\|u_1 - u_0\|}.$$

As shown by the various bit counting lemmas in [Deu06], efficient Newton algorithms can be constructed if $[\omega]$ catches at least one binary digit of ω. The importance of using affine invariant Lipschitz constants and estimates is that without adhering to those principles, distortion effects caused by affine transformations, for example scaling, could impact our computational estimates, causing them to be of little use for algorithms.

2.4.3 Affine covariance for an error oriented algorithm

We start by focusing on affine covariance, where we keep $\mathrm{dom}(\mathcal{F})$ fixed (so $\mathcal{B} = \mathcal{I}$), and look at problems

$$\mathcal{G}(u) = \mathcal{A}\mathcal{F}(u) = 0.$$

This will result in affine covariant theory about the local quadratic convergence of Newton's method, which will be used to construct algorithms with convergence criteria based on the iterates $\{u_k\}$, correction norms, $\|\delta u_k\|$, or error norms, $\|u_k - u^*\|$.

The assumptions of the standard Newton–Kantorovich theorem discussed in Section 2.3.2 can be telescoped to the requirement [Deu06]

$$\left\|\mathcal{F}'(u_0)^{-1}(\mathcal{F}'(u)-\mathcal{F}'(v))\right\| \le \omega_0 \left\|u-v\right\|, \quad u,v,u_0 \in U. \tag{2.28}$$

The Lipschitz constant ω_0 as defined above will be affine covariant, since

$$\begin{aligned}\mathcal{G}'(u_0)^{-1}(\mathcal{G}'(u)-\mathcal{G}'(v)) &= (\mathcal{A}\mathcal{F}'(u_0))^{-1}\mathcal{A}(\mathcal{F}'(u)-\mathcal{F}'(v)) \\ &= \mathcal{F}'(u_0)^{-1}(\mathcal{F}'(u)-\mathcal{F}'(v)),\end{aligned}$$

so both sides of (2.28) are independent of $\mathcal{A}$. This allows us to state the Newton–Kantorovich theorem in the affine covariant framework. For a proof, see [Deu06], or [Yam85], which also contains a number of upper and lower bounds for the error at the kth step of Newton iteration, $\|u_k - u^*\|$.

Theorem 2.4.1. *(Affine covariant Newton-Kantorovich theorem, [Deu06, Thm. 2.1]) Let $\mathcal{F}: U \subset V \to W$ be a continuously Fréchet differentiable mapping, with U open and convex. At a starting point $u_0 \in U$, let $\mathcal{F}'(u_0)$ be invertible. Assume that*

$$\begin{aligned}\left\|\mathcal{F}'(u_0)^{-1}\mathcal{F}(u_0)\right\| &\le \alpha < \infty, \\ \left\|\mathcal{F}'(u_0)^{-1}(\mathcal{F}'(v)-\mathcal{F}'(u))\right\| &\le \overline{\omega}_0 \left\|v-u\right\|, \quad u,v \in U, \\ h_0 := \alpha\overline{\omega}_0 &\le \tfrac{1}{2} \\ \bar{S}(u_0,\rho_-) := \{u \in U \mid \|u-u_0\| &\le \rho_-\} \subset U,\end{aligned}$$

where $\rho_- := (1-\sqrt{1-2h_0})/\overline{\omega}_0$. Then the sequence $\{u_k\}$ obtained from the ordinary Newton iteration is well-defined, remains in $\bar{S}(u_0,\rho_-)$, and converges to some u^* with $F(u^*)=0$. For $h_0 < \frac{1}{2}$, the convergence is quadratic. Moreover, with

$$t^{**} = 1+\sqrt{1-2h_0}, \qquad \rho_+ := t^{**}/\overline{\omega}_0,$$

and the assumption that $\bar{S}(u_0,\rho_+) \subset U$, the solution u^* is unique in $S(u_0,\rho_+)$.

Like the standard Newton–Kantorovich theorem, the affine covariant version also serves both as existence and uniqueness theorem for a solution. However, like the standard one, it incorporates a Lipschitz constant which depends on an operator norm, which is computationally difficult to obtain.

An alternative approach for studying the convergence properties of Newton's method, which avoids working with operator norms and only works with vector norms,

goes back to Mysovskikh in 1949 [Mys49]. The so-called Newton–Mysovskikh theorem can also be put in an affine covariant framework. Here, we state both the affine covariant Newton–Mysovskikh theorem, and the "refined Newton–Mysovskikh theorem", found in [DP92] and [Deu06]. For the non-affine invariant version of the Newton–Mysovskikh theorem, see for example [OR70].

We make a remark that in both theorems, the selection of the function space in which we seek solutions enters through the norm of the space. Thus, by specifying the norm on the solution space, then algorithmically, we have fixed the space of the solutions. The Newton method will then, if possible at all, implicitly be using the appropriate solution space.

Theorem 2.4.2. *(Affine covariant Newton–Mysovskikh theorem, [DP92, Thm. 1.1]) Let $\mathcal{F} : U \subset V \to W$ be a continuously Fréchet differentiable mapping, with $F'(u)$ invertible for all $u \in U$, U open and convex. Let $\bar{S}(u_0, \rho) := \{u \in U \mid \|u - u_0\| \leq \rho\}$ define a closed ball with radius ρ around u_0. Moreover, assume*

$$\left\|\mathcal{F}'(v)^{-1}[\mathcal{F}'(u + s(v-u)) - \mathcal{F}'(u)](v-u)\right\| \leq s\omega \|v-u\|^2 \tag{2.29}$$

for $u, v \in U$ and $s \in [0,1]$. Then for all starting points $u_0 \in U$ such that

$$\|\delta u_0\| = \left\|\mathcal{F}'(u_0)^{-1}\mathcal{F}(u_0)\right\| \leq \alpha_0 < \infty,$$

$$h_0 := \alpha_0\omega/2 < 1,$$

and

$$\bar{S}(u_0, \rho) \subset U \quad \textit{with} \quad \rho = \alpha_0 \sum_{j=0}^{\infty} h_0^{2^j - 1} \leq \frac{\alpha_0}{1 - h_0},$$

we have the following results:

(i) The sequence $\{u_k\}$ of Newton iterates remains in $\bar{S}(u_0, \rho)$ and converges to a solution $u^* \in \bar{S}(u_0, \rho)$.

(ii) The convergence rate can be estimated by

$$\|u_{k+1} - u_k\| \leq \tfrac{1}{2}\omega \|u_k - u_{k-1}\|^2.$$

(iii) The following error estimate holds

$$\|u_k - u^*\| \leq \varepsilon_k \|u_k - u_{k-1}\|^2,$$

where

$$\varepsilon_k := \tfrac{1}{2}\omega \sum_{j=0}^{\infty} (h_0^{2^k})^{2^j - 1} \leq \frac{\omega/2}{1 - h_0^{2^k}}.$$

The (affine covariant) Newton–Mysovskikh theorem only asserts the existence of a solution, but not its uniqueness. Following [DP92], we observe that the Lipschitz condition (2.29) is a special case of the stronger assumption

$$\left\|\mathcal{F}'(w)^{-1}[\mathcal{F}'(t)-\mathcal{F}'(u)](t-u)\right\| \leq \omega \left\|v-u\right\|^2, \quad \forall u,t,w \in U \text{ collinear} \tag{2.30}$$

if we take $w = v$ and $t = u + s(v - u)$. Note that (2.30) is clearly affine covariant, since for $\mathcal{G} = \mathcal{A}\mathcal{F}$,

$$\begin{aligned}\mathcal{G}'(u)^{-1}(\mathcal{G}'(v)-\mathcal{G}'(u))(v-u) &= (\mathcal{A}\mathcal{F}'(u))^{-1}\mathcal{A}(\mathcal{F}'(v)-\mathcal{F}'(u))(v-u)\\ &= \mathcal{F}'(u)^{-1}(\mathcal{F}'(v)-\mathcal{F}'(u))(v-u).\end{aligned}$$

As stated in [DP92, Lem. 1.4], if $\mathcal{F} : U \subset V \to W$ is twice continuously differentiable on U, U open and convex, (2.29) and (2.30) are actually equivalent. Even so, replacing the Lipschitz condition (2.29) with the stronger but practically equivalent condition (2.30) allows us to arrive at the refined Newton–Mysovskikh theorem. The refined Newton–Mysovskikh theorem states conditions about existence and uniqueness of solutions of the operator equation $\mathcal{F}(u) = 0$, and about how Newton's method will converge to that solution, and only uses vector norms in its assumptions.

Theorem 2.4.3. *(The refined Newton–Mysovskikh theorem, [DP92, Thm. 1.5]) Let $\mathcal{F} : U \subset V \to W$ be a continuously Fréchet differentiable mapping, with $F'(u)$ invertible for all $u \in U$, U open and convex. Assume that we can find a starting point $u_0 \in U$ and constants α_0, $\omega \geq 0$ such that*

$$\|\delta u_0\| = \left\|\mathcal{F}'(u_0)^{-1}\mathcal{F}(u_0)\right\| \leq \alpha_0 < \infty, \tag{2.31a}$$

$$\left\|\mathcal{F}'(w)^{-1}[\mathcal{F}'(v)-\mathcal{F}'(u)](v-u)\right\| \leq \omega \left\|v-u\right\|^2 \quad \forall\, u,v,w \in U \text{ collinear}, \tag{2.31b}$$

$$h_0 := \alpha_0\omega/2 < 1, \tag{2.31c}$$

$$\bar{S}(u_0,\rho) := \{u \in U \mid \|u-u_0\| \leq \rho_-\} \subset U, \tag{2.31d}$$

$$\text{with } \rho := \alpha_0 \sum_{j=0}^{\infty} h_0^{2^j-1} \leq \frac{\alpha_0}{1-h_0}. \tag{2.31e}$$

Then:

(i) The sequence $\{u_k\}$ of Newton iterates remains in $S(u_0,\rho)$ and converges to a solution $u^* \in \bar{S}(u_0,\rho)$.

(ii) u^* is the unique solution of $\mathcal{F}(u) = 0$ in the region

$$U^* := \bigcup_{k=0}^{\infty} S(u_k, 2/\omega) \cap U.$$

(iii) The following error estimates hold:

$$\|u_{k+1} - u_k\| \leq \tfrac{1}{2}\omega \|u_k - u_{k-1}\|^2, \tag{2.32}$$

$$\|u_k - u^*\| \leq \tfrac{1}{2}\omega \|u_{k-1} - u^*\|^2, \tag{2.33}$$

$$\|u_k - u^*\| \leq \frac{\|u_{k+1} - u_k\|}{1 - \omega/2 \|u_{k+1} - u_k\|}. \tag{2.34}$$

The affine covariant Newton–Mysovskikh theorems presented here will be most useful for constructing algorithms for solving nonlinear differential equations using Newton's method. What remains is to describe how we can obtain estimates of the Lipschitz constants in order to determine whether we should expect Newton's method to converge locally. Using the notation

$$h_k := \tfrac{1}{2}\omega \|\delta u_k\| = \tfrac{1}{2}\omega \|u_{k+1} - u_k\|, \tag{2.35}$$

we see that multiplying (2.32) by $\omega/2$ gives that

$$h_k \leq h_{k-1}^2. \tag{2.36}$$

If we define the contraction factor

$$\Theta_k := \frac{\|\delta u_{k+1}\|}{\|\delta u_k\|},$$

which is easily computable during the Newton iteration, we see from the definition of h_k in (2.35), and by applying (2.36), that Θ_k can be written in terms of the unknown theoretical quantities h_k as

$$\Theta_k = \frac{h_{k+1}}{h_k} \leq h_k < 1. \tag{2.37}$$

Thus, if we encounter $\Theta_k \geq 1$, we classify the Newton iteration as "not locally convergent", since the assumptions of the theorem have been violated.

The final ingredient required for a local Newton method is a convergence criterion. An intuitive termination criterion for an error oriented approach would be to stop the iteration when

$$\|u_k - u^*\| \leq \eta_{\text{err}},$$

where η_{err} is a user prescribed error tolerance. Since u^* is unknown, we have to find a way to estimate $\|u_k - u^*\|$. Following [Deu06], we use (2.37) to define the computationally available *a-posteriori* estimates

$$[h_k]_1 := \Theta_k \leq h_k, \quad k = 0, 1, \ldots.$$

Rewriting (2.37) as $h_{k+1} = \Theta_k h_k$, and shifting the index $k+1 \to k$, we obtain the *a-priori* estimates (using (2.36) and (2.37))

$$[h_k] := \Theta_{k-1}[h_{k-1}]_1 = \Theta_{k-1}^2 = \frac{h_k^2}{h_{k-1}^2} = h_k \frac{h_k}{h_{k-1}^2} \leq h_k, \quad k = 1, 2, \ldots.$$

Hence, rewriting (2.34) in terms of h_k as

$$\|u_k - u^*\| \leq \frac{\|\delta u_k\|}{1 - h_k},$$

and using the computational estimate $h_k \to [h_k] = \Theta_{k-1}^2$, we arrive at the easily computable termination criterion

$$\frac{\|\delta u_k\|}{1 - \Theta_{k-1}^2} \leq \eta_{\text{err}}.$$

We present the local Newton iteration algorithm `LENIB` as Algorithm 2. Compared to Algorithm 1, we now require the user to pass the algorithm an error tolerance η_{err}, and the maximum number of iteration the algorithm is allowed, $k_{\max}$. The algorithm will return either a solution, or a `flag` which indicates no convergence, either because the algorithm did not converge in $k_{\max}$ steps, or because we encounter a contraction factor $\Theta_k \geq 1$. In case of the latter, we have to resort to *global Newton algorithms*, which are the topic of Section 2.5. Note that Algorithm 2 requires the computation of contraction factors in order to determine whether it has converged. Thus, by construction, it will always perform at least two iterations, even for linear problems where we expect Newton's method to converge in one step. However, by applying the linearity detection technique discussed in Chapter 4, we can detect whether we can safely terminate the iteration after one Newton step, and avoid taking an unnecessary second step for linear problems.

2.4.4 Affine contravariance for a residual oriented algorithm

We now discuss affine contravariant Newton methods, where in Equation (2.26), we keep the image space of $\mathcal{F}$ fixed (so $\mathcal{A} = \mathcal{I}$), and look at problems

$$\mathcal{G}(v) = \mathcal{F}(\mathcal{B}v), \quad u = \mathcal{B}v.$$

Algorithm 2 `LENIB`: Local Error oriented Newton iteration in Banach spaces.

Input: An operator, $\mathcal{F}$. An initial guess of a solution, u_0. Error tolerance η_{err}. Maximum number of iterations, $k_{\max}$.

Output: A solution u, for which $\|u - u^*\| \leq \eta_{\text{err}}$ (where $\mathcal{F}(u^*) = 0$), or a `flag` which denotes no convergence.

for $k = 0, \ldots k_{\max}$ **do** ▷ Iterate up to $k_{\max}$ times.
  $\mathcal{F}_k \leftarrow \mathcal{F}(u_k)$ ▷ Evaluate and store.
  $\mathcal{F}'_k \leftarrow \mathcal{F}'(u_k)$ ▷ Compute the Fréchet derivative of $\mathcal{F}$ at u_k.
  $\delta u_k \leftarrow -(\mathcal{F}'_k)^{-1}\mathcal{F}_k$ ▷ Compute Newton update (via $\mathcal{F}'_k \delta u_k = -\mathcal{F}_k$).
  $u_{k+1} \leftarrow u_k + \delta u_k$. ▷ Next guess of the solution.
  if $k \geq 1$ **then**
    $\Theta_{k-1} \leftarrow \|\delta u_k\| \,/\, \|\delta u_{k-1}\|$ ▷ Contraction factor.
    if $\|\delta u_k\| \,/(1 - \Theta_{k-1}^2) \leq \eta_{\text{err}}$ **then** ▷ Termination criterion.
      return $u = u_{k+1}$ ▷ Return the solution found.
    else if $\Theta_{k-1} \geq 1$ **then** ▷ Divergence detected.
      return `flag`: Newton iteration is not locally convergent.
    end if
  end if
end for
return `flag`: Newton iteration not convergent within $k_{\max}$ iterations.

This approach to affine invariant Newton methods was introduced by A. Hohmann in [Hoh94]. It yields results in image space, that is, the results are based on information about the residuals $\mathcal{F}(u_k)$, which are independent of the choice of $\mathcal{B}$. Here, we use the fact that the assumptions of the Newton–Mysovskikh theorem can be telescoped to the requirement [Deu06]

$$\|(\mathcal{F}'(\bar{u}) - \mathcal{F}'(u))(\bar{u} - u)\| \leq \omega \|\mathcal{F}'(u)(\bar{u} - u)\|^2, \quad u, \bar{u} \in \operatorname{dom}(\mathcal{F}), \tag{2.38}$$

which defines a Lipschitz constant ω that is affine contravariant, since

$$(\mathcal{G}'(\bar{v}) - \mathcal{G}'(v))(\bar{v} - v) = (\mathcal{F}'(\bar{u}) - \mathcal{F}'(u))\mathcal{B}(\bar{v} - v) = (\mathcal{F}'(\bar{u}) - \mathcal{F}'(u))(\bar{u} - u),$$

and

$$\mathcal{G}'(v)(\bar{v} - v) = \mathcal{F}'(u)\mathcal{B}(\bar{v} - v) = \mathcal{F}'(u)(\bar{u} - u).$$

Here, we state the affine contravariant version of the Newton–Mysovskikh theorem. It only depends on vector norms, and states results in terms of residual norms, rather than error norms. The norms are assumed to be scaled relative to the norm of the residual of the initial guess u_0. Similarly to Theorems 2.4.2 and 2.4.3, the selection of the residual space to which the operator $\mathcal{F}$ maps enters through the norm of the

residuals. Thus, by specifying the norm on the residual space, then algorithmically, we have fixed the space of the residuals. The Newton method will then, if possible at all, implicitly be using the appropriate residual space.

Theorem 2.4.4. *[Deu06, Thm. 2.12](Affine contravariant Newton–Mysovskikh theorem.) Let $\mathcal{F} : U \subset V \to W$ be a Fréchet differentiable mapping, with $F'(u)$ invertible for all $u \in U$, U open and convex. Assume that the following affine contravariant Lipschitz condition holds:*

$$\|[\mathcal{F}'(v) - \mathcal{F}'(u)](v-u)\| \leq \omega \|\mathcal{F}'(u)(v-x)\|^2 \quad \forall\, u, v \in U. \tag{2.39}$$

Define the open level set $\Lambda_\omega = \left\{u \in U \mid \|\mathcal{F}(u)\| < \frac{2}{\omega}\right\}$ and let $\bar{\Lambda}_\omega \subset U$ be bounded. For a given initial guess u_0 of an unknown solution u^* for which $\mathcal{F}(u^*) = 0$, let

$$h_0 := \frac{1}{2}\omega \|\mathcal{F}(u_0)\| < 1 \quad (\textit{that is, } u_0 \in \Lambda_\omega). \tag{2.40}$$

Then the ordinary Newton iterates $\{u_k\}$ defined by

$$\mathcal{F}'(u_k)\delta u_k = -\mathcal{F}(u_k), \quad u_{k+1} = u_k + \delta u_k,$$

remain in Λ_ω and converge to some solution point $u^* \in \Lambda_\omega$ with $\mathcal{F}(u^*) = 0$. The iterative residuals $\{\mathcal{F}(u_k)\}$ converge to zero at an estimated rate

$$\|\mathcal{F}(u_{k+1})\| \leq \tfrac{1}{2}\omega \|\mathcal{F}(u_k)\|^2. \tag{2.41}$$

Note that this affine contravariant version of the Newton–Mysovskikh theorem only proves that the Newton residuals $\mathcal{F}(u_k)$ converge quadratically to zero; it does not prove anything about the convergence of the Newton iterates u_k. Also, it does not prove any results about the uniqueness of the solution u^*. This is in agreement with the affine contravariant theory only stating results in image space.

As in Section 2.4.3, to be able to construct an algorithm based on the affine contravariant version of the Newton–Mysovskikh theorem, we need to identify computationally available estimates for the Lipschitz constants and a termination criterion. Defining the residual associated Kantorovich quantity

$$h_k := \tfrac{1}{2}\omega \|\mathcal{F}(u_k)\|, \tag{2.42}$$

we obtain by multiplying (2.41) by $\frac{1}{2}\omega$ that

$$h_{k+1} \leq h_k^2. \tag{2.43}$$

Defining the easily computable contraction factor Θ_k as

$$\Theta_k = \frac{\|\mathcal{F}(u_{k+1})\|}{\|\mathcal{F}(u_k)\|},$$

we have from (2.42) and (2.43) that

$$\Theta_k = \frac{h_{k+1}}{h_k} \leq h_k. \tag{2.44}$$

Keeping the condition (2.40) in mind, we see that encountering $\Theta_0 \geq 1$ indicates that the initial guess u_0 was not close enough to u^* for Theorem 2.4.4 to guarantee convergence of the Newton iteration. For constructing a convergence monitor for $k > 0$, following [Deu06], we use (2.44) to define the computationally available *a-posteriori* estimates

$$[h_k]_1 := \Theta_k \leq h_k, \quad k = 0, 1, \ldots.$$

Equation (2.44) states that $h_{k+1} = \Theta_k h_k$, which yields the *a-priori* estimate

$$[h_{k+1}] = \Theta_k [h_k] = \Theta_k^2 = \frac{h_{k+1}^2}{h_k^2} = h_{k+1}\frac{h_{k+1}}{h_k^2} \leq h_{k+1}, \quad k = 0, 1, \ldots.$$

Assuming that $[h_{k+1}] \approx [h_{k+1}]_1$, we obtain the recurrence relationship

$$\Theta_{k+1} \approx \Theta_k^2 \leq \Theta_0 < 1.$$

Thus, if we encounter a violation of this recursion,

$$\Theta_{k+1} > \Theta_0 \qquad \text{or} \qquad \Theta_{k+1} > \alpha\Theta_k^2, \quad \alpha \gtrsim 1 \tag{2.45}$$

we identify the Newton iteration as "not locally convergent".

The final ingredient required for a local Newton method is a convergence criterion. An intuitive termination criterion for the residual oriented approach is to stop the iteration when

$$\frac{\|\mathcal{F}(u_k)\|}{\|\mathcal{F}(u_0)\|} \leq \eta_{\text{res}},$$

where η_{res} is a user prescribed error tolerance. Such a termination criterion is easily computable during the course of Newton iteration.

We present the affine contravariant local Newton iteration algorithm `LRNIB` as Algorithm 3. As in the algorithm `LENIB`, we also require the user to pass the maximum number of iterations the algorithm is allowed, $k_{\max}$. However, now the tolerance required is for the norm of the residual, η_{res}, rather than the norm of the error, η_{err}. The algorithm will either return a solution, or a `flag` which indicates no convergence, either because the algorithm did not converge in $k_{\max}$ steps, or because we encounter a violation of (2.45), that is, a contraction factor $\Theta_k \geq \Theta_0$ (or in case of stricter contraction factor requirement, $\Theta_k \geq 2\Theta_{k-1}^2$). As for the algorithm `LENIB`, in case of the latter, we have to resort to global Newton algorithms.

Algorithm 3 `LRNIB`: Local residual oriented Newton iteration in Banach spaces.

Input: An operator, $\mathcal{F}$. An initial guess of a solution, u_0. Error tolerance η_{res}. Maximum number of iterations, $k_{\max}$.

Output: A solution u, for which $\|\mathcal{F}(u)\| \,/\, \|\mathcal{F}(u_0)\| \leq \eta_{\text{res}}$, or a `flag` which denotes no convergence.

$\mathcal{F}_0 \leftarrow \mathcal{F}(u_0)$ ▷ Initial evaluation and storage.
$\rho_0 \leftarrow 1$ ▷ "Scaled residual" of initial guess, i.e. $\|\mathcal{F}_0\| \,/\, \|\mathcal{F}_0\|$.
for $k = 0, \ldots k_{\max}$ **do** ▷ Iterate up to $k_{\max}$ times.
  $\mathcal{F}'_k \leftarrow \mathcal{F}'(u_k)$ ▷ Compute the Fréchet derivative of $\mathcal{F}$ at u_k.
  $\delta u_k \leftarrow -(\mathcal{F}'_k)^{-1}\mathcal{F}_k$ ▷ Compute Newton update (via $\mathcal{F}'_k \delta u_k = -\mathcal{F}_k$).
  $u_{k+1} \leftarrow u_k + \delta u_k$ ▷ Next guess of the solution.
  $\mathcal{F}_{k+1} \leftarrow \mathcal{F}(u_{k+1})$ ▷ Evaluate and store for next guess.
  $\rho_{k+1} \leftarrow \|\mathcal{F}_{k+1}\| \,/\, \|\mathcal{F}_0\|$ ▷ Scaled residual for next guess.
  if $\rho_{k+1} \leq \eta_{\text{res}}$ **then** ▷ Termination criterion.
    return $u = u_{k+1}$ ▷ Return the solution found.
  end if
  $\Theta_k \leftarrow \rho_{k+1}/\rho_k$ ▷ Contraction factor.
  if $k = 0$ and $\Theta_0 \geq 1$ **then** ▷ Check whether initial guess sufficiently good.
    return `flag`: Newton iteration is not locally convergent.
  else if $k \geq 1$ and $\Theta_k \geq \Theta_0$ (or $\Theta_k \geq 2\Theta_{k-1}^2$) **then** ▷ Divergence detected.
    return `flag`: Newton iteration is not locally convergent.
  end if
end for
return `flag`: Newton iteration not convergent within $k_{\max}$ iterations.

2.5 Global Newton methods

The algorithms described above have excellent convergence properties once the current approximation gets sufficiently close to a solution. However, it is evident from the assumptions of the theorems that such behaviour can only be guaranteed if we start the algorithm with a good enough initial guess. In the error-oriented framework, the requirement for quadratic convergence of Newton iteration was that

$$\|\delta u_0\| \, \omega/2 < 1,$$

where ω is as defined in (2.31b). In the residual-oriented framework, the requirement was

$$\frac{1}{2}\omega \, \|\mathcal{F}(u_0)\| < 1,$$

where here, ω is as defined in (2.39). This dependence of the Newton iteration on a good initial guess is a well known fact in scientific computing. In this section,

we present algorithms which converge for a wider range of initial guesses than the algorithms above; in other words, algorithms which expand the domain of initial guesses for which Newton iteration will converge.

One approach to globalising the local Newton methods is to introduce a continuation parameter λ, so as to embed the original problem $\mathcal{F}(u) = 0$ into a family of problems

$$\tilde{\mathcal{F}}(0, \lambda) = 0, \quad \lambda \in [0, 1],$$

so that the initial guess u_0 is the solution for $\lambda = 0$, and the solution for $\lambda = 1$ is the solution of the original problem. Such *homotopy* or *path-following* methods are discussed further in Chapter 5. Here, we instead focus on how the algorithms above can be modified to increase their domain of convergence by adjusting the steplength, or *damping* the Newton iteration.

For an iterative method such as Newton's method, one would hope that the iterates u_k successively approach the solution u^* as k increases, that is, that

$$\|u_{k+1} - u^*\| < \|u_k - u^*\| \quad (\text{unless } u_k = u^*).$$

In cases where the assumptions of the affine covariant convergence theorems in Section 2.4.3 are satisfied, this requirement will be satisfied by default due to the results of the theorems. Otherwise, we have to introduce a criterion that allows us to monitor whether we are converging to a solution. One approach is to introduce the residual level function

$$T(u) := \tfrac{1}{2} \|\mathcal{F}(u)\|_2^2 .$$

T has the property that

$$\begin{aligned} T(u) = 0 &\Leftrightarrow u = u^*, \\ T(u) > 0 &\Leftrightarrow u \neq u^*. \end{aligned} \tag{2.46}$$

For the Newton iteration, we can thus require the monotonicity criterion

$$T(u_{k+1}) < T(u_k), \quad \text{if } T(u_k) \neq 0. \tag{2.47}$$

Associated with the level function are the level sets

$$G(w) := \{u \in U | T(u) \leq T(w)\} .$$

In the spirit of the affine covariant framework discussed above, we introduce the generalised level function

$$T(u|\mathcal{A}) := \tfrac{1}{2} \|\mathcal{A}\mathcal{F}(u)\|_2^2 , \tag{2.48}$$

for a bounded bijective operator $\mathcal{A}$, whose domain is included in the range of $\mathcal{F}$. Restricting ourselves to a Hilbert space W, so that $\mathcal{A}$ will have associated with it an adjoint operator $\mathcal{A}^*$, defined with the property that

$$\langle v, \mathcal{A}w\rangle = \langle \mathcal{A}^* v, w\rangle$$

for any $v, w \in W$, and that $\mathcal{F}(u)^T$ can by the Riesz representation theorem be interpreted as the linear functional with the action that

$$\mathcal{F}(u)^T v = \langle \mathcal{F}(u), v\rangle \qquad \text{for all } v \in W,$$

we rewrite (2.48) as

$$T(u|\mathcal{A}) = \tfrac{1}{2}\langle \mathcal{A}\mathcal{F}(u), \mathcal{A}\mathcal{F}(u)\rangle = \tfrac{1}{2}\mathcal{F}(u)^T \mathcal{A}^*\mathcal{A}\mathcal{F}(u).$$

The Newton direction, $\delta u_k = -\mathcal{F}'(u_k)^{-1}\mathcal{F}(u_k)$, has the important property that for any $\mathcal{A}$, there exists a $\lambda_k > 0$ such that $T(u+\lambda_k \delta u_k|\mathcal{A}) < T(u|\mathcal{A})$; that is, any level function will decrease if we take small enough step in the Newton direction. This is formally shown in the following lemma, leading to the same conclusion as Theorem 3.6 in [Deu06].

Lemma 2.5.1. *Let $\mathcal{F}'_k = \mathcal{F}'(u_k)$ be nonsingular, $\mathcal{F}'_k = \mathcal{F}(u_k) \neq 0$, and $\mathcal{A}$ a bijective operator. With $\delta u_k = -\mathcal{F}'_k \mathcal{F}_k$ as the Newton direction, there exists some $\mu > 0$ such that*

$$T(u_k + \lambda \delta u_k|\mathcal{A}) < T(u_k|\mathcal{A}), \quad \textit{for all } 0 < \lambda < \mu.$$

Proof. Define $\vartheta(\lambda) := T(u_k + \lambda\delta u_k|\mathcal{A})$. Then,

$$\vartheta(0) = T(u_k|\mathcal{A}).$$

Also,

$$\begin{aligned}\vartheta'(\lambda) &= \frac{\partial T(u_k + \lambda\delta u_k|\mathcal{A})}{\partial\lambda}\delta u_k \\ &= \mathcal{F}(u_k + \lambda\delta u_k)\mathcal{A}^*\mathcal{A}\mathcal{F}'(u_k + \lambda\delta u_k)\delta u_k,\end{aligned}$$

so

$$\begin{aligned}\vartheta'(0) &= \mathcal{F}_k^T \mathcal{A}^*\mathcal{A}\mathcal{F}'_k \delta u_k \\ &= -\mathcal{F}_k^T \mathcal{A}^*\mathcal{A}\mathcal{F}_k = -2T(u_k|\mathcal{A}) < 0.\end{aligned}$$

□

In other words, the Newton direction δu_k is always a good direction to take, but it might be that taking the full Newton step via $u_{k+1} = u_k + \delta u_k$ is too optimistic. Instead, we might need to take a damped Newton step via $u_{k+1} = u_k + \lambda_k \delta u_k$. The question thus becomes what a good value of λ_k to take is. In the algorithms we discuss, we make the choice of λ_k based on the results in [Deu06], which give theoretically optimal damping factors under some assumptions about the problem listed below.

2.5.1 Error-oriented globalised Newton's method

In this section, we study how the results of Section 2.4.3 about affine covariant Newton methods for error oriented algorithms can be extended to devise algorithms with globalised convergence properties. Our approach is to describe selected important findings of Section 3.3 in [Deu06], which allows us to construct the algorithm `GENIB` — a globalised error oriented Newton iteration in Banach spaces, at the end of this section.

We seek to find a damping parameter λ_k, to be used in the *damped Newton iteration*

$$\mathcal{F}'(u_k)\delta u_k = -\mathcal{F}(u_k), \quad u_{k+1} = u_k + \lambda_k \delta u_k, \quad \lambda_k \in (0, 1],$$

which give the greatest decrease for the generalised level sets as defined from above

$$T(u|\mathcal{A}) := \tfrac{1}{2} \left\| \mathcal{A}\mathcal{F}(u) \right\|_2^2 .$$

We should expect that the value of λ_k depends on the value of $\mathcal{A}$ chosen. To see what value of λ_k to choose, then using the notation from above, we start with an affine invariant Lipschitz condition (compare to Theorem 2.4.3)

$$\left\| \mathcal{F}'(u)^{-1}(\mathcal{F}'(v) - \mathcal{F}'(u))(v - u) \right\| \leq \omega \left\| v - u \right\|^2$$

for $u, v \in U$ and $\mathcal{F}'$ nonsingular for all $u \in D$. Following the proof of Theorem 3.12 in [Deu06], at the kth iterate, we have that (dropping the subscripts for u_k and δu_k)

$$\begin{aligned}
\left\| \mathcal{A}\mathcal{F}(u + \lambda\delta u) \right\| &= \left\| \mathcal{A}(\mathcal{F}(u + \lambda\delta u) - \mathcal{F}(u) - \mathcal{F}'(u)\delta u) \right\| \\
&= \left\| \mathcal{A} \left(\int_{s=0}^{\lambda} (\mathcal{F}'(u + s\delta u) - \mathcal{F}'(u))\delta u \mathrm{d}s - (1 - \lambda)\mathcal{F}'(u)\delta u \right) \right\| \\
&\leq (1 - \lambda) \left\| \mathcal{A}\mathcal{F}(u) \right\| + O(\lambda^2) \quad \text{for } \lambda \in [0, 1].
\end{aligned}$$

We can estimate the $O(\lambda^2)$ term as follows:

$$\begin{aligned}
&\left\| \mathcal{A}\mathcal{F}'(u) \int_{s=0}^{\lambda} \mathcal{F}'(u)^{-1}(\mathcal{F}'(u+s\delta u) - \mathcal{F}'(u))\delta u \mathrm{d}s \right\| \\
&= \|\mathcal{A}\mathcal{F}'(u)\| \int_{s=0}^{\lambda} \omega s \left\|\delta u\right\|^2 \mathrm{d}s \qquad \text{(using Lipschitz condition inside integral)} \\
&= \|\mathcal{A}\mathcal{F}'(u)\| \, \omega \tfrac{1}{2}\lambda^2 \left\|\delta u\right\|^2 \\
&= \tfrac{1}{2}\lambda^2 \|\mathcal{A}\mathcal{F}'(u)\| \, h_k \left\|(\mathcal{A}\mathcal{F}'(u))^{-1}\mathcal{A}\mathcal{F}(u)\right\| \qquad \text{(with } h_k := \omega \left\|\delta u\right\|\text{)} \\
&\leq \tfrac{1}{2}\lambda^2 h_k \|\mathcal{A}\mathcal{F}'(u)\| \left\|(\mathcal{A}\mathcal{F}'(u))^{-1}\right\| \|\mathcal{A}\mathcal{F}(u)\| \\
&= \tfrac{1}{2}\lambda^2 h_k \|\mathcal{A}\mathcal{F}'(u)\| \, \kappa(\mathcal{A}\mathcal{F}'(u)),
\end{aligned}$$

where we defined

$$\kappa(\mathcal{A}\mathcal{F}'(u)) := \left\|(\mathcal{A}\mathcal{F}'(u))^{-1}\right\| \|\mathcal{A}\mathcal{F}(u)\|$$

in the last line. Further defining $\bar{h}_k := h_k\kappa(\mathcal{A}\mathcal{F}'(u_k))$, the estimate for the $O(\lambda^2)$ term can be written as

$$\tfrac{1}{2}\lambda^2\bar{h}_k \|\mathcal{A}\mathcal{F}'(u)\| ,$$

and we have that

$$\|\mathcal{A}\mathcal{F}(u_k + \lambda\delta u_k)\| \leq t_k(\lambda|\mathcal{A}) \|\mathcal{A}\mathcal{F}(u_k)\|$$

where $t_k(\lambda|\mathcal{A})$ is the parabola in λ:

$$1 - \lambda + \tfrac{1}{2}\lambda^2\bar{h}_k.$$

Minimising t_k with respect to λ, we arrive at the locally estimated optimal choice of λ_k as

$$\bar{\lambda}_k(\mathcal{A}) := \min(1, 1/\bar{h}_k).$$

Up to now, we have not made any choice of what $\mathcal{A}$ to use. This is where affine invariance becomes truly important, as demonstrated by an example in Section 2.6.1. The choice of simply requiring the residual to decrease amounts to the choice $\mathcal{A} = \mathcal{I}$, the identity operator. This corresponds to the residual monotonicity criterion

$$T(u_{k+1}|\mathcal{I}) \leq T(u_k|\mathcal{I}). \tag{2.49}$$

In the case of Newton iteration in the finite dimensional setting, problems arise if $\mathcal{F}'(u_k)$ is ill-conditioned, as then, the optimal damping parameter has to satisfy

$$\bar{\lambda}_k(\mathcal{I}) = (h_k \mathrm{cond}(\mathcal{F}'(u_k)))^{-1} \ll 1.$$

Thus, with $u_{k+1} = u_k + \lambda_k \delta u_k$, the Newton iteration will stop making progress, as it gets stuck with too small step sizes. It might also happen that we obtain a value of λ_k smaller than the mininum allowed step size of our algorithms.

In the case of $\mathcal{F}$ being a differential operator, then with the choice $\mathcal{A} = \mathcal{I}$, we have that $\kappa(\mathcal{A}\mathcal{F}'(u)) = \|\mathcal{F}'(u)^{-1}\| \, \|\mathcal{F}(u)\|$, which will always be greater than or equal to 1. The importance of $\kappa(\mathcal{A}\mathcal{F}'(u))$ indicates that a good choice of $\mathcal{A}$ at the kth iteration is

$$\mathcal{A}_k := \mathcal{F}'(u_k)^{-1},$$

so that $\kappa(\mathcal{A}\mathcal{F}'(u_k)) = \kappa(\mathcal{I}) = 1$. The associated level function $T(u|\mathcal{F}'(u_k)^{-1})$ is known as the *natural level function*, and the associated monotonicity requirement

$$T(u_{k+1}|\mathcal{F}'(u_k)^{-1}) \leq T(u_k|\mathcal{F}'(u_k)^{-1}),$$

is known as the *natural monotonicity criterion*. Defining the *simplified Newton correction* $\overline{\delta u}_{k+1}$ by

$$\overline{\delta u}_{k+1}(\lambda_k) := -\mathcal{F}'(u_k)^{-1}\mathcal{F}(u_k + \lambda_k + \delta u_k) = -\mathcal{F}'(u_k)^{-1}\mathcal{F}(u_{k+1}),$$

(so that we are solving using the previous derivative, but with the next residual if the step $u_{k+1} = \lambda_k + \delta u_k$ is accepted), the natural monotonicity criterion can be written as

$$\begin{aligned} & T(u_{k+1}|\mathcal{F}'(u_k)^{-1}) \leq T(u_k|\mathcal{F}'(u_k)^{-1}) \\ \Leftrightarrow \; & \left\|\mathcal{F}'(u_k)^{-1}F(u_{k+1})\right\|_2 \leq \left\|\mathcal{F}'(u_k)^{-1}F(u_k)\right\|_2 \\ \Leftrightarrow \; & \left\|\overline{\delta u}_{k+1}\right\|_2 \leq \|\delta u_k\|_2 . \end{aligned}$$

As described in [Deu06], for an equation $\mathcal{F}(u) = 0$, $\mathcal{F} : U \subset V \to W$, under assumptions of nonsingularity of $\mathcal{F}'$, compactness of the path-connected component $U_0 \subseteq U$ of $G(u_0|\mathcal{F}'(u^*)^{-1})$, and that a local affine covariant Lipschitz condition

$$\left\|\mathcal{F}'(u_k)^{-1}(\mathcal{F}'(u) - \mathcal{F}'(u_k))(u - u_k)\right\| \leq \omega_k \|u - u_k\|^2, \quad \text{for } u, u_k \in U_0, \tag{2.50}$$

holds, the natural monotonicity tests leads to the theoretical optimal damping strategy for Newton's method

$$u_{k+1} = u_k + \bar{\lambda}_k \delta u_k, \quad \bar{\lambda}_k := \min(1, 1/h_k), \quad h_k = \omega_k \|\delta_k\| .$$

We are thus now in the situation that finding the value of the Lipschitz constant ω_k, and thus $h_k = \omega_k \|\delta u_k\|$, gives us the optimal damping factor $\bar{\lambda}_k$. Since ω_k

is defined over U_0, we are in a similar situation as in Section 2.4, where we seek easily available computational estimates, denoted by $[\omega_k]$ and $[h_k]$. This gives a computational estimate of the steplength, $[\bar{\lambda}_k] := \min(1, 1/[h_k])$.

Since our estimates will have to be based on pointwise sampling of the value of ω_k, we have that

$$[\omega_k] \leq \omega_k \Rightarrow [h_k] \leq h_k.$$

This implies that

$$[\bar{\lambda}_k] \geq \bar{\lambda}_k,$$

in other words, we could take too large steps in the Newton direction. Therefore, it might be necessary to repeatedly reduce the step length as we gain better computational estimates, that is, apply a *prediction-correction* strategy.

In [Deu06], Deuflhard arrives at the following correction strategy at the kth step of Newton iteration, using adaptive trust region methods:

$$\lambda_k^{j+1} := \min(\tfrac{1}{2}\lambda_k^j, 1/[h_k^j]), \quad \text{for } j = 0,\, 1,\, \ldots,$$

where

$$[h_k^j] := \frac{2\left\|\overline{\delta u}_{k+1}(\lambda_k^j) - (1-\lambda_k^j)\delta u_k\right\|}{(\lambda_k^j)^2 \left\|\delta u_k\right\|}.$$

In order to be able to apply the prediction strategy, we need a good initial estimate of λ at the kth iteration, that is, λ_j^0. Using a Lipschitz condition slightly different from (2.50) in order to be able to use the most recent information,

$$\left\|\mathcal{F}'(u_k)^{-1}(\mathcal{F}'(u) - \mathcal{F}'(u_k))v\right\| \leq \bar{\omega}_k \left\|u - u_k\right\| \left\|v\right\|,$$

and observing that

$$\begin{aligned}
\left\|\overline{\delta u}_k - \delta u_k\right\| &= \left\|(\mathcal{F}'(u_{k-1})^{-1} - \mathcal{F}'(u_k)^{-1})\mathcal{F}(u_k)\right\| \\
&= \left\|(\mathcal{F}'(u_{k-1})^{-1} - \mathcal{F}'(u_k)^{-1})\mathcal{F}'(u_{k-1})\overline{\delta u}_k\right\| \\
&= \left\|(\mathcal{I} - \mathcal{F}'(u_k)^{-1}\mathcal{F}'(u_{k-1}))\overline{\delta u}_k\right\| \\
&= \left\|\mathcal{F}'(u_k)^{-1}(\mathcal{F}'(u_k) - \mathcal{F}'(u_{k-1}))\overline{\delta u}_k\right\| \\
&\leq \bar{\omega}_k \left\|u_k - u_{k-1}\right\| \left\|\overline{\delta u}_k\right\| = \bar{\omega}_k \lambda_{k-1} \left\|\delta u_{k-1}\right\| \left\|\overline{\delta u}_k\right\|
\end{aligned}$$

we obtain the estimate

$$[\bar{\omega}_k] := \frac{\left\|\overline{\delta u}_k - \delta u_k\right\|}{\lambda_{k-1} \left\|\delta u_{k-1}\right\| \left\|\overline{\delta u}_k\right\|},$$

which leads to the initial prediction value λ_k^0 for $k > 0$:

$$\lambda_k^0 := \min(1, \mu_k), \quad \mu_k := \frac{\left\|\overline{\delta u}_k\right\|}{\left\|\overline{\delta u}_k - \delta u_k\right\|} \frac{\|\delta u_{k-1}\|}{\|\delta u_k\|} \lambda_{k-1}.$$

It only remains to select the starting value λ_0^0. For our algorithms, we select the value $\lambda_0^0 = 1$, although the literature also recommends the option of making a choice $\lambda_0^0 \ll 1$ for problems which are known to be "highly nonlinear".

Throughout the damped Newton iteration, we monitor the contraction factor associated with the natural monotonicity test,

$$\Theta_k := \frac{\left\|\overline{\delta u}_{k+1}\right\|}{\|\delta u_k\|}.$$

Encountering $\Theta_k \geq 1$ indicates that we are trying to take too large a Newton step, so a reduction of λ_k is necessary. Finding $\Theta_k < 1$ for $\lambda_k = 1$ indicates that we have reached the domain of quadratic convergence of Newton's method, so we can switch to the simpler algorithm of local Newton iteration `LENIB` from Section 2.4.3.

We can now describe the algorithm `GENIB` for a global error-oriented Newton iteration in Banach spaces, shown as Algorithm 4. Its corresponding algorithm in [Deu06] is called `NLEQ-ERR`. The algorithm starts by using the prediction-correction strategy described above for finding the steplength of the Newton iteration, but once it detects that it has reached the domain of quadratic convergence, it switches to the algorithm `LENIB`. Note that it is necessary to pass the algorithm a minimum allowed stepsize $\lambda_{\min}$, as the algorithm will stagnate (that is, $u_{k+1} \approx u_k$) if it starts repeatedly taking too small steps. However, rather than giving up straight away if we encounter $\lambda_k^j \leq \lambda_{\min}$, we try to "give the iteration a kick" by taking a full Newton step. This heuristic step amounts to the algorithm moving to a different region in the solution space, in the hope that it will put the iteration in a position to converge. We use the Boolean variable χ in the algorithm to monitor whether such a full step has been taken — if successive iterations start to converge, χ will be reset.

2.5.2 Residual based globalised Newton's method

The global error-oriented Newton method described above will eventually be our preferred method for solving nonlinear ODEs. However, it is also interesting to describe a global residual-based Newton method and compare the resulting algorithms. In a residual-based Newton method, we study the damped Newton iteration

$$\mathcal{F}'(u_k)\delta u_k = -\mathcal{F}(u_k), \quad u_{k+1} = u_k + \lambda_k \delta u_k,$$

Algorithm 4 `GENIB`: Global error Oriented Newton iteration in Banach spaces.

Input: An operator, $\mathcal{F}$. An initial guess of a solution, u_0. Error tolerance η_{err}. Maximum number of iterations, $k_{\max}$. Minimum allowed step length $\lambda_{\min}$.

Output: A solution u, for which $\|u - u^*\| \leq \eta_{\text{err}}$ (where $\mathcal{F}(u^*) = 0$), or a `flag` which denotes no convergence.

$\lambda_0 \leftarrow 1$ ▷ Initial value of step length.
$\chi \leftarrow$ `FALSE` ▷ Monitors whether a "kick" has been taken.
for $k = 0, \ldots k_{\max}$ **do** ▷ Iterate up to $k_{\max}$ times.
  $\mathcal{F}_k \leftarrow \mathcal{F}(u_k)$ ▷ Evaluate and store.
  $\mathcal{F}'_k \leftarrow \mathcal{F}'(u_k)$ ▷ Compute the Fréchet derivative of $\mathcal{F}$ at u_k.
  $\delta u_k \leftarrow -(\mathcal{F}'_k)^{-1}\mathcal{F}_k$ ▷ Compute Newton update (via $\mathcal{F}'_k \delta u_k = -\mathcal{F}_k$).
  if $k \geq 1$ **then** ▷ Compute a prediction value for λ_k

$$\mu_k \leftarrow \frac{\left\|\overline{\delta u}_k\right\|}{\left\|\overline{\delta u}_k - \delta u_k\right\|} \frac{\|\delta u_{k-1}\|}{\|\delta u_k\|} \lambda_{k-1}, \quad \lambda_k \leftarrow \min(1, \mu_k)$$

  end if
  `acceptStep` $\leftarrow$ `FALSE`
  while $\neg$ `acceptStep` **do** ▷ Iterate until we are happy with the step
    if $\lambda_k < \lambda_{\min}$ **then** ▷ Regularity test
      if $\chi =$ `TRUE` **then** ▷ We have already tried the "kick"
        return `flag`: Regularity test failed.
      else ▷ Try taking a full step
        $u_{k+1} \leftarrow u_k + \delta u_k, \quad \chi \leftarrow$ `TRUE`, `acceptStep` $\leftarrow$ `TRUE`
      end if
    else
      $u_{k+1} \leftarrow u_k + \lambda_k \delta u_k$ ▷ Compute trial iterate
      $\overline{\delta u}_{k+1} \leftarrow -(\mathcal{F}'_k)^{-1}\mathcal{F}(u_{k+1})$ ▷ Compute simplified Newton update
      $\Theta_k \leftarrow \dfrac{\left\|\overline{\delta u}_{k+1}\right\|}{\|\delta u_k\|}$ ▷ Monitoring quantity
      $\mu'_k \leftarrow \dfrac{\|\delta u_k\| \, \lambda_k^2}{2\left\|\overline{\delta u}_{k+1}\lambda_k - (1-\lambda_k)\delta u_k\right\|}$ ▷ Value used for correction strategy
      if $\Theta_k \geq 1$ **then** ▷ Step length too big
        $\lambda_k \leftarrow \lambda'_k := \min(\mu'_k, \frac{1}{2}\lambda_k)$. ▷ Try new λ_k and go to regularity test
      else ▷ Natural monotonicity criterion satisfied
        $\lambda'_k \leftarrow \min(1, \mu'_k)$ ▷ New potential value for λ_k
        if $\lambda'_k = \lambda_k = 1$ **then** ▷ Strategy is telling us to take a full step
          Switch to local algorithm `LENIB`, with u_{k+1} as initial guess.
          return $u \leftarrow$ solution `LENIB` finds.
          (Algorithm continued on next page)

Algorithm 4 `GENIB` (continued)

```
            else if λ'_k ≥ 4λ_k then
                λ_k ← λ'_k  ▷ Try whether a larger step gives greater improvement,
                                ▷ and compute a new simplified Newton update
            else
                accept ← TRUE                                   ▷ Accept λ_k
            end if
        end if
    end if
  end while
end for
return flag: Newton iteration not convergent within k_max iterations.
```

where we impose the residual contraction condition

$$\|\mathcal{F}(u_{k+1})\| < \|\mathcal{F}(u_k)\| .$$

With similar arguments as above, the search for the step length λ_k has to be split into a prediction step and correction steps, based on computationally available estimates of Lipschitz constants. The strategy derived in [Deu06] is to use the correction strategy

$$\lambda_k^{j+1} = \min(\tfrac{1}{2}\lambda_k^j, 1/[h_k^{j+1}]), \quad \text{with } [h_k^{j+1}] := \frac{2\left\|\mathcal{F}(u_k + \lambda_k^j \delta u_k) - (1-\lambda_k^j)\mathcal{F}(u_k)\right\|}{(\lambda_k^j)^2 \|\mathcal{F}(u_k)\|},$$

and the prediction value

$$\lambda_k^0 = \min(1, \mu_k), \quad \text{with } \mu_k := \frac{\|\mathcal{F}(u_{k-1})\|}{\|\mathcal{F}(u_k)\|} \frac{\frac{1}{2}\|F(u_{k-1})\|(\lambda_{k-1})^2}{\|\mathcal{F}(u_k) - (1-\lambda_k)\mathcal{F}(u_{k-1})\|},$$

for $k > 0$, where λ_{k-1} is the step length choosen by the correction strategy at the previous Newton step. What remains is to select the starting value λ_0^0. As before, we choose $\lambda_0^0 = 1$, although for problems which are known to be "highly-nonlinear", a starting value $\lambda_0^0 \ll 1$ is recommended.

With these considerations, we can describe `GRNIB` — a global residual based Newton iteration in Banach spaces, shown as Algorithm 5. Its corresponding algorithm in [Deu06] is called `NLEQ-RES`. Similarly as for the algorithm `GENIB`, we want to switch to a local residual based Newton method once the algorithm detects it is in the domain of quadratic convergence. For that, we define the residual monotonicity test monitor

$$\Theta_k = \frac{\|\mathcal{F}(u_{k+1})\|}{\|\mathcal{F}(u_k)\|}.$$

If we encounter $\Theta_k < 1$ with $\lambda_k = 1$, we switch to the algoritm `LRNIB`.

Algorithm 5 `GRNIB`: Global residual based Newton iteration in Banach spaces.

Input: An operator, $\mathcal{F}$. An initial guess of a solution, u_0. Error tolerance η_{res}. Maximum number of iterations, $k_{\max}$. Minimum allowed step length $\lambda_{\min}$.

Output: A solution u, for which $\|\mathcal{F}(u)\| \,/\, \|\mathcal{F}(u_0)\| \leq \eta_{\text{res}}$, or a `flag` which denotes no convergence.

$\mathcal{F}_0 \leftarrow \mathcal{F}(u_0)$ ▷ Initial evaluation and storage.
$\lambda_0 \leftarrow 1$ ▷ Initial value of step length.
$\chi \leftarrow$ `FALSE` ▷ Monitors whether a "kick" has been taken.
for $k = 0, \ldots k_{\max}$ **do** ▷ Iterate up to $k_{\max}$ times.
  if $\|\mathcal{F}_k\| \,/\, \|\mathcal{F}_0\| \leq \eta_{\text{err}}$ **then** ▷ Solution found
    return u_k.
  end if
  $\mathcal{F}'_k \leftarrow \mathcal{F}'(u_k)$ ▷ Compute the Fréchet derivative of $\mathcal{F}$ at u_k.
  $\delta u_k \leftarrow -(\mathcal{F}'_k)^{-1}\mathcal{F}_k$ ▷ Compute Newton update (via $\mathcal{F}'_k \delta u_k = -\mathcal{F}_k$).
  if $k \geq 1$ **then** ▷ Compute a prediction value for λ_k
    $\mu_k \leftarrow \dfrac{\|\mathcal{F}(u_{k-1})\|}{\|\mathcal{F}(u_k)\|}\mu'_{k-1}, \quad \lambda_k \leftarrow \min(1, \mu_k)$
  end if
  `acceptStep` $\leftarrow 0$
  while $\neg$ `acceptStep` **do** ▷ Iterate until we are happy with the step
    if $\lambda_k < \lambda_{\min}$ **then** ▷ Regularity test
      if $\chi =$ `TRUE` **then** ▷ We have already tried the "kick"
        return `flag`: Regularity test failed.
      else ▷ Try taking a full step
        $u_{k+1} \leftarrow u_k + \delta u_k, \quad \chi \leftarrow$ `TRUE`, `acceptStep` $\leftarrow 1$
      end if
    else
      $u_{k+1} \leftarrow u_k + \lambda_k \delta u_k$ ▷ Compute trial iterate
      $\mathcal{F}_{k+1} \leftarrow \mathcal{F}(u_{k+1})$ ▷ Evaluate residual at trial iterate
      $\Theta_k \leftarrow \dfrac{\|\mathcal{F}(u_{k+1})\|}{\|\mathcal{F}(u_k)\|}$ ▷ Monitoring quantity
      $\mu'_k \leftarrow \dfrac{\|\mathcal{F}(u_k)\|\,\lambda_k^2}{2\,\|\mathcal{F}(u_{k+1}) - (1-\lambda_k)\mathcal{F}(u_k)\|}$ ▷ Value used for correction strategy
      if $\Theta_k \geq 1$ **then** ▷ Step length too big
        $\lambda_k \leftarrow \lambda'_k := \min(\mu'_k, \frac{1}{2}\lambda_k)$. ▷ Try new λ_k and go to regularity test
      else ▷ Natural monotonicity criterion satisfied
        $\lambda'_k \leftarrow \min(1, \mu'_k)$ ▷ New potential value for λ_k
        if $\lambda'_k = \lambda_k = 1$ **then** ▷ Strategy is telling us to take a full step
          Switch to local algorithm `LRNIB`, with u_{k+1} as initial guess.
          return $u \leftarrow$ solution `LRNIB` finds.
          (Algorithm continued on next page)

Algorithm 5 `GRNIB` (continued)

```
            else if λ'_k ≥ 4λ_k then
                λ_k ← λ'_k  ▷ Try whether a larger step gives greater improvement,
                                ▷ and compute a new simplified Newton update
            else
                accept ← TRUE                                   ▷ Accept λ_k
            end if
        end if
    end if
  end while
end for
return flag: Newton iteration not convergent within k_max iterations.
```

2.6 Discussion

In this chapter, we have described a number of results related to Newton's method in Banach spaces, and the various algorithms that different affine invariance criteria lead to. Based on this, we now summarize the preferred techniques implemented in Chebfun, which we will present in more details in Chapter 3. We have implemented all the algorithms `LENIB`, `LRNIB`, `GENIB` and `GRNIB`, and future versions of Chebfun will include all of them. Usually, an estimate of the error in the solution space norm is of greater interest than an estimate of the error in the image space norm. Furthermore, the algorithm `GENIB` enjoys nice global convergence properties. Therefore, we have selected `GENIB` as the default algorithm when solving nonlinear BVPs in Chebfun. As discussed above, selecting the norm on the solution space implicitly fixes the space in which solutions are sought. For a second order ODE, the default norm used in the error oriented algorithms is the Sobolev H^2, while in the residual based algorithms, we use the L_2 norm to ensure that the Fréchet derivative of the differential operator has a bounded inverse. We plan to allow users to pass in their selected norm as an input to our solver, thereby giving them the option to fix the solution space themselves.

2.6.1 Demonstrating the importance of affine invariance

As an example of an ODE where affine covariance becomes important, consider this more difficult version of the Carrier problem taken from [BO78, Sec. 9.7]:

$$0.0005u'' + 2(1 - x^2)u + u^2 = 1, \quad u(-1) = u(1) = 0, \tag{2.51}$$

where we take

$$u_0(x) = 2(x^2 - 1)\left(1 - \frac{2}{1 + 20x^2}\right)$$

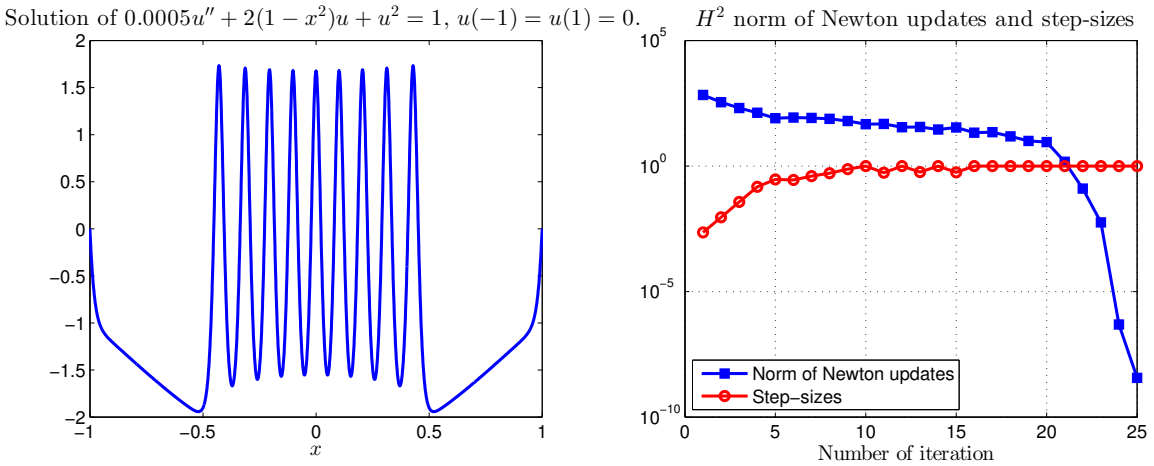

Figure 2.2: The solution of the BVP (2.51), showing the norm of the Newton updates and the step-sizes used in the Newton iteration.

as our initial guess. Here, the global affine covariant Newton algorithm `GENIB` takes 25 steps to return the solution, which is a `chebfun` of length 879, with an H^2 error estimate $5.5 \cdot 10^{-11}$. We plot the solution, the H^2 norms of the updates and the step-sizes used in Figure 2.2. On the other hand, if we try to solve this problem with a global Newton that is not affine covariant, we find that the solver requires 96 Newton steps, as it gets stuck for too long using small step-sizes.

2.6.2 Effect of discretisation errors on Newton's method in function space

So far, all our discussion of Newton iteration in function space and the algorithms we implement have assumed that we are able to solve the operator equation

$$\mathcal{F}'(u_k)\delta u_k = -\mathcal{F}(u_k)$$

exactly, to determine the exact Newton correction δu_k. However, when we eventually solve problems on a computer, discretisation errors are bound to occur, as we have to approximate δu_k. In our context, the discretisation error will arise from deciding when we have solved the linearised problem at each Newton step to sufficient precision. Such *inexact Newton methods* are for example discussed in [DES82]. As pointed out in [Deu06], inexact Newton methods are indeed the correct framework when describing the numerical solution of operator equations. Following [Deu06, Ch. 7.4], starting from the current approximation u_k at the kth Newton iteration, we define two Newton corrections in function space

$$\mathcal{F}'(u_k)\delta u_k = -\mathcal{F}(u_k) + r_k, \qquad \mathcal{F}'(u_k)\delta y_k = -\mathcal{F}(u_k).$$

Here, the residual r_k is generated by discretisation error such that

$$\mathcal{F}'(u_k)(\delta u_k - \delta y_k) = r_k.$$

The preceding results about error estimates and steplength strategy will be perturbed, since we cannot compute the exact update. The perturbations at the kth step are described by the quantity

$$\delta_k := \frac{\|\delta u_k - \delta y_k\|}{\|\delta u_k\|},$$

which quantifies the discretisation error. Once δ_k reaches a threshold value, one should refine the mesh and solve for a new Newton update.

The perturbations will enter both the step-size selection in the damped phase, as well as affect convergence in the quadratic phase. Since we apply spectral methods for solving the linearised problems, which achieve high accuracy, the quotient will be very small in the damped phase, and the perturbation effects become negligible. In the quadratic case, the adaptive spectral method we apply actually suffers more from ill-conditioning of the discretised operators rather than the mesh not being fine enough. We would therefore not make gains from refining our grid, as that would not improve or Newton update. We observe in the quadratic phase that the quadratic convergence of the Newton iteration tails off a few orders above machine precision due to this ill-conditioning. Quantifying how this affects the termination criterion of our solver and the error estimates returned is a future research topic.

2.6.3 Ghost solutions

Finally, we address the possibility of the so-called "ghost solutions" of BVPs, which can occur when problems are solved numerically. Assume we are solving a BVP, which has a locally unique solution. As described in [DB02, Deu06], if the problem is solved numerically using multiple shooting techniques, the discrete system we end up solving also has a locally unique solution. On the other hand, if the BVP is discretised using global discretisation methods, spurious discrete solutions may occur, which have nothing to do with the unique continuous solution of the original problem.

Such behaviour has indeed been experienced in practice, as for example described in [SM04]. There, the authors consider the *Bratu equation*

$$u'' + \lambda e^u = 0, \qquad u(0) = u(1) = 1, \tag{2.52}$$

which is known not to have any solutions for values of λ greater than the critical value $\lambda_c \approx 3.513830719125$. Even so, they report that the well-known MATLAB code

for solving BVPs, `bvp4c`, converges to a ghost solution of the problem with $\lambda = 3.55$. We encounter the BVP (2.52) again in Chapters 3 and 5.

This should sound alarm bells for our eventual numerical implementation of the algorithms described in this chapter, as we solve the linear problems arising in Newton's method with pseudospectral methods, which are global. However, as derived in [Deu06, Sec. 7.4.2], the quasilinearisation approach which we employ, where the mesh can be refined as the Newton iteration progresses, is not cursed by ghost solutions. As an example, using the Chebfun implementation of the algorithms of this chapter, discussed in Chapter 3, we successfully find a solution of (2.52) with $\lambda = 3.513830719125$ using tolerance 10^{-10}, while if we take $\lambda = 3.513830719126$, our solver returns a 'no-convergence' flag.

Chapter 3

Fréchet Derivatives by Automatic Differentiation and Automatic Solutions of Nonlinear BVPs

In the preceding chapter, we gave the mathematical background for implementing Newton's method in function space and listed algorithms for solving problems numerically with the method. In order to be able to implement those algorithms on a computer, we have to have a means of obtaining the Fréchet derivatives of nonlinear operators — a crucial component for the Newton iteration.

As established in Section 2.2, Fréchet derivatives are linear operators. Since they satisfy the same differential rules as derivatives from ordinary calculus — the sum rule, product rule, chain rule, etc. — we should be able to compute them using the technique of automatic differentiation (AD), introduced in Section 1.3. This chapter describes such *automatic Fréchet differentiation* (AFD).

In Section 3.6, we propose an algorithm combining Newton's method in function space and automatic differentiation. To demonstrate how the algorithms of this chapter can be implemented, we use Chebfun as our framework, where the Fréchet derivatives are represented by the `linop` class. To encapsulate Newton's method and AD, we present the the `chebop` class, a black-box solver for nonlinear BVPs in Chebfun.

The origin of the work presented in this chapter is as follows: The original idea of using anonymous functions and unique IDs to enable recursive, lazy evaluation of Fréchet derivatives is due to Tobin Driscoll; the implementation and analysis were done jointly by Driscoll and the author. The discussion in Section 3.1 is an extension and an update of the treatment found in the joint paper [BD12]. The `anon` class presented in Section 3.1.2 and the accompanying writing was devised and implemented

by the author. The run-time AD implementation and the `ADchebfun` class discussed in Section 3.2 is due to the author. The examples of automatic Fréchet differentiation and automatic differentiation of functionals, discussed in Sections 3.3 and 3.4 respectively, are due to the author, as well as the writing about Taylor testing for correctness of derivatives in Section 3.5. The `chebop` class (and the writing about it) is a work of the author, as is the majority of work behind `chebgui` presented in Section 3.6.2 (with contributions from Nick Hale).

When solving practical BVPs, we can expect the number of variables involved (the unknown functions we are trying to determine) to be fairly small, of order tens or hundreds, but not in the thousands. In particular, problems are inherently square, since the number of unknown variables is usually the same as the number of differential equations. Hence, for AFD, we can choose to implement the more straightforward forward mode AD. As described in Section 1.3, there are two approaches of implementing AD: operator overloading and source transformation. Since we are interested in implementing AD in Chebfun, which involves overloading MATLAB methods, we focus on operator overloading in this chapter.

Here, we discuss two implementations of automatic Fréchet differentiation:

- A delayed, recursive evaluation of derivatives, described in Section 3.1. This approach is discussed in [BD12].
- A run-time evaluation of derivatives, described in Section 3.2.

Both implementations of AFD offer the novelty that the derivatives are linear operators, not matrices or vectors, so that the resolutions needed for the derivatives can be determined automatically. Moreover, the delayed, recursive implementation has several unique features compared with traditional AD implementations:

- Using anonymous functions and numbering each intermediate variable, we enable a recursive, delayed evaluation of derivatives with forward mode AD. On the other hand, in standard forward mode AD, the derivatives are computed at the same time as the functions are evaluated.
- There is no need to declare variables to be "AD variables" (known as *active variables* in the AD literature), or to initialise derivative fields explicitly before calculations are performed. This is in contrast to standard AD implementations, where the user has to decide at the start of calculations what variables he or she wants to find derivatives with respect to (the "base variables"), and seed derivative values accordingly.

- It enables computing the derivative of any variable with respect to any other variable.

We make a remark about terminology. Let f and g be two functions, and assume that g depends on f (and possibly other functions), that is, there exists an operator $\mathcal{G}$,

$$\mathcal{G} : f \mapsto \mathcal{G}(f), \tag{3.1}$$

such that $g = \mathcal{G}(f)$. Then, when we speak of the "derivative of g with respect to f", we mean the Fréchet derivative of the operator $\mathcal{G}$ at the point (in function space) f, as defined by Definition 2.2.4.

3.1 A delayed, recursive AD implementation

Here, we describe delayed, recursive evaluation of Fréchet derivatives. Delayed evaluation is helpful in a software system such as Chebfun to avoid computational overhead for tasks that do not require AD. The idea of delayed evaluation is similar to the one used in the implementation of the `linop` class for linear operators in Chebfun. As described in [DBT08], one way to achieve such delayed or lazy evaluation in MATLAB is to use anonymous functions. Although presented here in the AFD context, we report that we have also been successful in implementing a delayed and recursive approach to AD where the derivatives are the standard gradient vectors and Jacobian matrices.

As described in Section 1.3, where we introduced the `advar` class for working with forward mode AD, the basic idea behind AD is to introduce a new data structure (or modify a current one) which is able to work with derivative information. The `advar` class we presented contained two floating point numbers, corresponding to the calculated values of a variable and its derivative. One can think of this as an act of associating a new field to the standard double precision number to enable calculations of derivatives. Thanks to the ease of working with functions and linear operators in Chebfun, we use it as the framework for demonstrating how automatic Fréchet differentiation can be implemented. To enable AD for `chebfun` objects, we introduce a new field in the `chebfun` object named `jac`, as an abbreviation for Jacobian.[1] When we call the `chebfun` constructor, we initialise the `jac` field of the output to be empty, which allows the recursive evaluation of derivatives later on.

As an example, take the overloaded `tan` method of the `chebfun` class. The essential lines are:

[1]The term *Jacobian* only describes finite-dimensional operators, not our continuous setting of Fréchet derivatives. However, we chose it due to its familiarity.

```
function fout = tan(fin)
% Compose the input chebfun with the tan function
fout = comp(fin, @(x) tan(x));
% Assign a Jacobian (anon. func.) and ID (integers) to fout
fout.jac = @(u) diag(sec(fin).^2)*diff(fin, u);
fout.ID = newIDnum;
```

The call to the `comp` method in the second line is the mechanism of Chebfun to compose the input `fin` with the tan function (informally speaking, "take the tan of a `chebfun`"). In the third line, we assign to the `jac` field of the output, `fout`, the MATLAB anonymous function handle

```
@(u) diag(sec(fin).^2)*diff(fin, u)
```

Here, the argument `u` stands for the variable we ultimately differentiate with respect to. The use of an anonymous function allows delayed evaluation; the value of `fin.jac` does not need to be known at construction time (in the sense that we only need to know the anonymous function in the `jac` field of `fin`, not the actual derivative). Without the anonymous function wrapper, we would have a typical forward-mode implementation as is described in Sections 1.3 and 3.2. Here, the `diag` method corresponds to creating a multiplication operator as defined in (1.19), in this case,

$$\mathcal{M}^{\sec^2(f(x))} : g(x) \mapsto \sec^2(f(x))g(x). \tag{3.2}$$

The process of the eventual evaluation of the `.jac` field to obtain a linear operator is described in Section 3.1.1.

In the last line of the code above, we assign to the `ID` field of `fout` the output from the method `newIDnum`. The function `newIDnum` returns to every `chebfun` created a unique ID, which consists of two integers. One of the integers is a timestamp initialised the first time the `newIDnum` method is called in a MATLAB session, and the other is a counter which starts at 1 in each session and is incremented each time `newIDnum` is called.[2] We emphasize that each intermediate variable created when a function evaluation takes place gets a unique ID. This is analogous to assigning a unique intermediate variable to each elementary operation in the evaluation trace (discussed in Section 1.3). This will be important when eventually evaluating the derivatives.

[2]It is necessary to make the ID consist of two integers to avoid problems when saving and reloading MATLAB workspaces.

3.1.1 Recursive evaluation of derivatives

We now describe how recursion and ID fields are used to evaluate derivatives. The main result is that the recursive evaluation is made possible by giving each intermediate variable a unique ID, and making each intermediate variable store the evaluation trace leading to its construction. To a reader familiar with AD, our implementation might seem like "forward-mode AD with a bit of reverse feel to it".

Assume that we have two variables `f` and `g`, representing the functions $f = f(x)$ and $g = g(x)$, and that g depends on f in the sense (3.1). To obtain `g`, we had to start with `f` and perform a sequence of elementary operations, each of which created temporary intermediate variables. Every intermediate variable had a `jac` field, containing a recipe for a derivative that depends on intermediate variables previously created.

Our method for obtaining the derivative of g with respect to f is to call `diff(g, f)`. The `diff` method starts by using the `ID` fields to see whether f and g are the same function. If so, the derivative is $\mathcal{I}$, the identity operator on the interval that f and g are defined on. Otherwise, the anonymous function in the `jac` field of `g` is evaluated with `f` as its argument. This will recurse through all intermediate variables in the evaluation trace between `f` and `g`, until `f` is reached and $\mathcal{I}$ forms the bottom of the recursion.

To put this information into context with the example shown previously of the overloaded tan method, the fifth line of the method read

```
fout.jac = @(u) diag(sec(fin).^2)*diff(fin, u);
```

The `diag` part corresponds to the multiplication operator given by (3.2). The second part, `diff(fin, u)`, corresponds to the linear operator that is the derivative of f_{in} with respect to u. Comparing the anonymous function to standard results of calculus, we see that it represents the chain rule. However, note that since `diff(fin, u)` is a part of the anonymous function, no actual function call is made at the time the `jac` field gets assigned a value. If we make the assignment

```
g = tan(f);
```

`g` corresponds to `fout` and `f` corresponds to `fin`. Then, when we call

```
dgdf = diff(g, f);   % Differentiate g with respect to f
```

Algorithm 6 Recursive evaluation of Fréchet derivatives.

Input: Variables `f` and `g`, representing the functions f and g. Both `f` and `g` are assumed to have a field `jac`, which stores a recipe for computing derivative with respect to a given function. They are also assumed to have a field `ID`, a unique identifier assigned to every instance of instances of the object class of `f` and `g`.

Output: The derivative of g with respect to f.

`fID` $\leftarrow$ `f.ID`, `gID` $\leftarrow$ `g.ID` ▷ Extract the IDs of `f` and `g`

if `g.jac` $= \emptyset$ **then**

If the `jac` field of `g` is empty, g must be the independent function.
By definition, g has the zero operator as its derivative.
return $\mathcal{Z}$

else if `fID = gID` **then**

We have reached the bottom of recursion, and $f = g$.
Initialise the derivative to be the identity operator.
return $\mathcal{I}$

else

We are still working our way back up the evaluation trace.

differentiate Recursively compute the derivatives — with respect to `f` — of the intermediate variables used to create `g`. Let $w_1, \ldots, w_n$ denote these intermediate variables, and $\partial w_1/\partial f, \ldots, \partial w_n/\partial f$ their deriatives with respect to f.

return The result of combining $\partial w_1/\partial f, \ldots, \partial w_n/\partial f$ according to the chain rule.

end if

`u` gets assigned the value `f`, and the result will be the linear operator

$$\mathcal{M}^{\sec^2(f(x))}\mathcal{I}.$$

The evaluation process is further explained in the listing of Algorithm 6 and Figure 3.1, where we show an example of the evaluation of a Fréchet derivative. In both the algorithm and the figure, we let $\mathcal{Z}$ denote the zero operator on the interval that the functions f and g are defined on. As before $\mathcal{D}$, $\mathcal{I}$ and $\mathcal{M}$ stand for differentiation, identity, and multiplication operators respectively.

We make a remark that our current implementation of recursive evaluation of derivatives does not use graph recombination. Thus, the cost of evaluating derivatives could grow much faster than for standard forward mode AD. As an example, assume we executed the lines

```
u = cos(f);
```

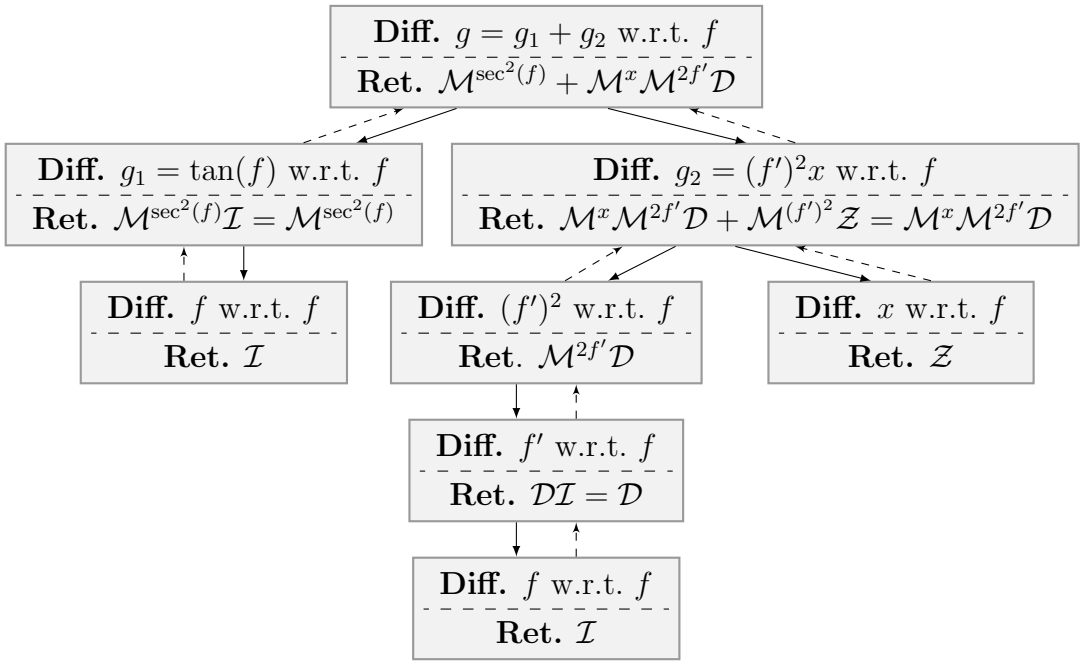

Figure 3.1: Recursive evaluation of the Fréchet derivative of g with respect to f, where $g_1 = \tan(f)$, $g_2 = (f')^2 x$ and $g = g_1 + g_2$. The evaluation works recursively down the solid arrows, then returns derivatives up through the dashed arrows.

```
g = u + u.*u;
```

where `f` is some `chebfun` we want to find derivatives with respect to. If we then computed the derivative of `g` with respect to `f`, our algorithm would first compute the derivative of `u` with respect to `f` (the left argument of the `+` operator), but then not make use of the information created in that process when computing the derivative of `u.*u` with respect to `f` (the right argument of the `+` operator). However, this is unlikely to be a major concern in the ODE application we intend for AFD. Furthermore, by using hashing to create a look-up table for storing and enabling access to previously computed derivatives, this problem could be avoided.

Note that in Chebfun, the `diff` method plays two roles. Here, we have described how calling `diff` with two `chebfun` inputs computes the Fréchet derivative of the first input with respect to the second input. In this case, the output is a `linop`. If `diff` is called with one `chebfun` argument, the result is another `chebfun` that corresponds to the derivative of the polynomial representation of the input `chebfun`. Thus, when `diff` is called with one argument, the derivative returned is calculated with traditional numerical differentiation, and no information from the evaluation trace is used.

3.1.2 The anon class for overcoming computational inefficiencies

While the straightforward use of anonymous functions above is effective, it leads to a serious inefficiency for certain client codes. When operations are applied to a variable repeatedly, the anonymous functions effectively create a stack of those operations; in particular, the variable workspaces in context at the creation of the `jac` fields. Unfortunately, MATLAB internally creates new copies of the entire stack of workspaces upon each anonymous function creation, which can lead to a quadratic growth in memory, as discussed below.[3] Thus, we found it necessary to create a new service class named `anon` to store these workspaces more efficiently, while offering similar functionality to anonymous functions.

An `anon` object has four fields:

- `function`: A string that defines the function that the `anon` represents.
- `variablesName`: A cell array of strings with the names of the variables in the `function` string.
- `workspace`: A cell array of variables (doubles and `chebfun` objects) with the values of these variables.
- `depth`: A double used for memory control, further discussed below.

(The reader who has studied the `functions` method of normal anonymous functions in MATLAB will spot the similarity.)

By overloading the appropriate MATLAB methods for the `anon` class, one can work with `anon` objects in a similar way as one would with anonymous functions. Using the new `anon` class, the overloaded `chebfun` method `tan` appears follows:

```
function fout = tan(fin)
% Compose the input chebfun with the tan function
fout = comp(fin, @(x) tan(x));
% Assign a Jacobian (anon) and ID (integers) to the chebfun returned
fout.jac = anon('@(u) diag(sec(fin).^2)*diff(fin, u)', {'fin'}, {fin});
fout.ID  = newIDnum;
```

[3] In short, the MATLAB constructor of anonymous functions does not make an efficient usage of objects already created.

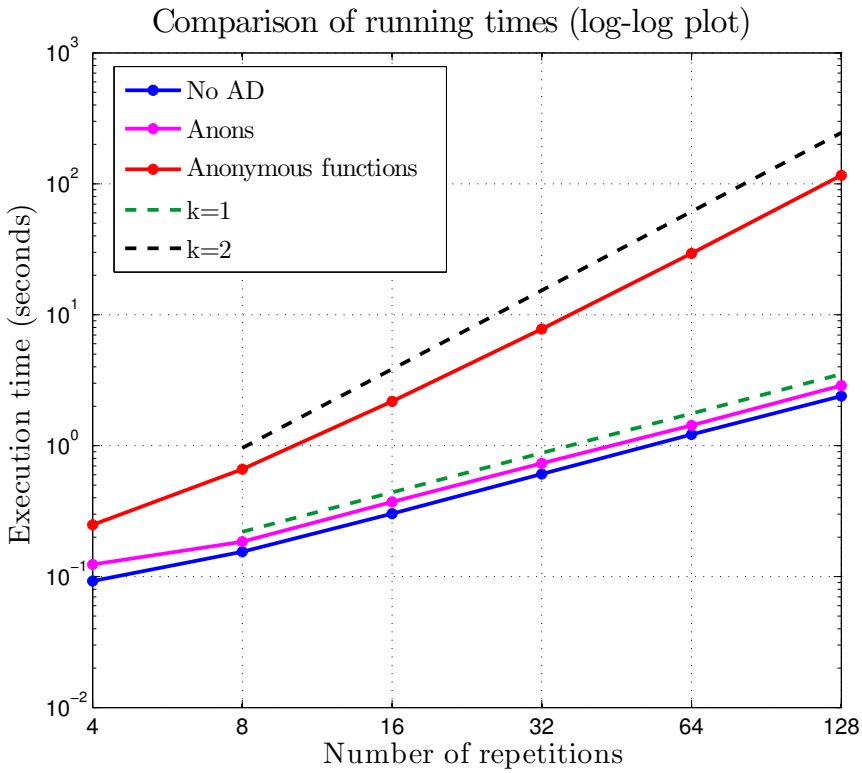

Figure 3.2: Comparison of execution time of repeatedly calling `f = cos(f) + sin(f)` where `f` is a `chebfun`, depending on whether AD information is stored via anonymous functions, the `anon` class, or disabled.

Demonstrating the benefits of the `anon` class

As a simple example demonstrating the benefit of the `anon` class over anonymous functions, consider repeatedly executing the statement

```
f = cos(f) + sin(f);
```

where `f` is a `chebfun`. In Figure 3.2, we show different running times obtained by executing the statement increasing number of times, using the `anon` class or anonymous functions to enable the lazy evaluation of derivatives. As a reference, we also show running times when no AD calculation is involved at all (after removing the `jac` field of the `chebfun` objects and the corresponding lines of code). The quadratic growth when using anonymous functions is evident, while one obtains linear growth when using no AD or the `anon` class.

Controlling memory usage

Whereas the `anon` class described above removes certain inefficiencies of implementing delayed and recursive AD in MATLAB, another issue arises for certain client codes, especially ones based on iterative processes, such as GMRES [SS86]. As the number of iterations grows, the computational graph which starts at the base AD variable (in the Chebfun case, the independent variable `x`) becomes huge, and memory requirements increase. In fact, memory control is a very important issue for AD in general [GW08].

In order to manage the use of memory, we monitor the `depth` property of the `anon` object. The `depth` of an `anon` indicates how many elementary operations were required to construct the variable it is attached to, starting from the AD base variable. By definition, the `depth` of the base variable is 0, and the `depth` of other variables is given by the maximum of the `depth` of the variables it is constructed from plus 1. Once the `depth` exceeds a maximum value, a dummy `anon` with an empty workspace is returned, and future AD calculations with the resulting variables are disabled.[4]

3.2 Run-time AD implementation

The `advar` class presented in Section 1.3 implements the arguably simplest, and in many cases most efficient approach to AD, where the values of derivatives are computed at the same time as the values of variables. We now discuss such a run-time implementation of automatic Fréchet differentiation, using Chebfun as the framework. For this, we introduce the `ADchebfun` class, which has two fields:

- `chebfun`: For storing the `chebfun` representation of the function of interest.
- `der`: For storing the Fréchet derivative of the function with respect to the base variable (described below).

We see that the structure of the `ADchebfun` class is the continuous analogue of the `advar` class.

The run-time AD implementation lacks some of the nice properties of the delayed evaluation discussed in Section 3.1, in particular the option of being able to differentiate any function with respect to any other function. However, it avoids having to introduce lines of code to other parts of the system. As an added benefit, it avoids all overhead for other computations where derivative information is not required.

[4] In Chebfun, the default maximum allowed `depth` of `anon` objects is 25. It can be adjusted using `chebfunpref('addepth', newvalue)`. This allows users to switch AD off completely by setting the value to be 0, or to allow as large a tree as possible before memory runs out by setting it to `inf`.

In Section 1.3, we saw that to compute derivatives using the run-time implementation, we had to start the computation by seeding the derivatives. Correspondingly, with `ADchebfun`, we designate a base variable and seed its Fréchet derivative, that is, assign a variable to have the identity operator $\mathcal{I}$ as its derivative.[5] All derivatives subsequently computed will then be the derivatives of variables with respect to that base variable. The derivative seeding is similar to the second argument we passed to the `advar` constructor in Section 1.3. However, when the `ADchebfun` class is used for supplying derivatives for Newton iteration in function space, such seeding is automatically taken care of internally. Thus, when solving BVPs in Chebfun, no effort is required from the user for seeding derivatives. Due to its simplicity, and the fact that it does not require introducing additional lines in core Chebfun code, we believe that the run-time AD implementation is the best approach for Chebfun, and future versions will include that implementation.

As an example of `ADchebfun` code, take the overloaded `tan` method. The essential lines are:

```
function fout = tan(fin)
% Initialise fout to be an ADchebfun
fout = fin;
% Assign the chebfun of fout to be tan of the chebfun of the input fin
fout.chebfun = tan(fin.chebfun);
% Compute the derivative of fout according to chain rule
fout.der = diag(sec(fin.chebfun).^2)*fin.der;
end
```

Another method is the overloaded `times` method. Here, the essential lines are:

```
function h = times(f, g)
% Initialise h to be an ADchebfun
h = f;
% Assign the chebfun of h to be the chebfun of f times the chebfun of g
h.chebfun = f.chebfun .* g.chebfun;
% Compute the derivative of h according to the derivative product rule
h.der = diag(g.chebfun)*f.der + diag(f.chebfun)*g.der;
end
```

Note that here, we have omitted showing lines of code that handle the cases when either of the inputs `f` or `g` are a `double` or a regular `chebfun`.

In both cases, the `tan` and `times` methods start by initialising the output to be an `ADchebfun`. This is to ensure that the outputs will have the required fields. The meth-

[5] In the case of coupled systems of ODEs, we specify each solution component as a base variable. Their derivatives will then consist of blocks that are all the zero operator $\mathcal{Z}$, except for one block, which is the identity operator $\mathcal{I}$. This represents that each base variable has the derivative $\mathcal{I}$ with respect to itself, and $\mathcal{Z}$ with respect to the other base variables.

ods then carry out the expected operations on the `chebfun` components of the inputs — taking the tangent or multiplying the two inputs together respectively. Finally, the Fréchet derivative of the output with respect to the base variable is computed. As described above, commands such as `diag(sec(fin.chebfun).^2)` create linear multiplication operators. These multiplication operators are then composed with the Fréchet derivatives of the inputs via the multiplication operator `*`, as prescribed by the rules of calculus.

3.3 Examples of automatic Fréchet differentiation

We now show examples of how Fréchet derivatives are computed in Chebfun. Although above we described two different implementations of AD in Chebfun, with different calling sequences for computing derivatives, the resulting derivatives have the same form. We highlight the differences where appropriate. We use the abbreviations DR-AD for the delayed, recursive AD implementation, and RT-AD for the run-time implementation.

We begin our exploration of automatic Fréchet differentiation in Chebfun by creating a standard `chebfun` representing the linear function x on the interval of interest. Other functions on the same interval can then be defined as follows:

```
>> % Create a linear chebfun on [0, 4.125]
>> x = chebfun('x', [0, 4.125]);
>> % Create another chebfun by operating on x
>> f = sin(4*pi/x);
```

Here, we have created a chebfun `f` to represent the function $f(x) = \sin(4\pi x)$ on the interval $[0, \frac{17}{4}]$. No action is required for DR-AD, but for RT-AD, we now need to convert `f` to an `ADchebfun` as follows:

```
>> f = ADchebfun(f)
f =  ADchebfun with
   chebfun column (1 smooth piece) of vertical scale 1
       interval       length   endpoint values
[       0,     4.1]        60        0        1
   and a derivative of block-size [1x1].
```

The output above indicates that now, `f` is an `ADchebfun`, which includes both a `chebfun` and a derivative. We can obtain that derivative using the `getder` method:

```
>> getder(f)
ans = linop
   Linear operator operating on chebfuns defined on:
```

```
  interval [0,4.125]
with functional representation:
  @(u)u
with n = 6 realization:
  1.0000         0         0         0         0         0
       0    1.0000         0         0         0         0
       0         0    1.0000         0         0         0
       0         0         0    1.0000         0         0
       0         0         0         0    1.0000         0
       0         0         0         0         0    1.0000
```

The corresponding command for DR-AD is `diff(f, f)`. As expected, the derivative of `f` is the identity operator, indicating that the Fréchet derivative of f with respect to itself is $\mathcal{I}$.

We now define the functions g and h on the interval $[0, \frac{17}{4}]$ by

$$\begin{aligned} g = &\quad x + f^3, \\ h = &\quad \exp(f) + g', \end{aligned}$$

where $'$ denotes the derivative of a function with respect to the independent function x. By operating on `f`, we create chebfun objects `g` and `h` to represent g and h as follows:

```
>> g = x + f.^3;
>> h = exp(f) + diff(g);
```

Before showing the derivative of g with respect to f as computed by AD, we work out what the analytical answer should be. Suppose that we perturb the function f by an infinitesimal function δf. What effect will this have on g? Using total derivatives to denote the Fréchet derivatives, the answer is that the first-order variation takes the form

$$\frac{\mathrm{d}g}{\mathrm{d}f} = \frac{\partial g}{\partial f} + \frac{\partial g}{\partial x}\cancelto{0}{\frac{\mathrm{d}x}{\mathrm{d}f}} = \frac{\partial}{\partial f}\left(x + f^3\right) = 0 + 3f^2 = 3f^2.$$

It is evident that $\frac{\mathrm{d}g}{\mathrm{d}f}$ is the linear operator that multiplies a function on $[0, \frac{17}{4}]$ by $3f^2$, i.e.

$$\frac{\mathrm{d}g}{\mathrm{d}f} : k(x) \mapsto 3f(x)^2 k(x). \tag{3.3}$$

We obtain the `linop` representation of the operator (3.3) by calling the `getder` method with `g` as an argument:

```
>> dgdf = getder(g)        % Obtain the derivative of g w.r.t. f.
dgdf = linop
   Linear operator operating on chebfuns defined on:
```

```
  interval [0,4.125]
with n = 6 realization:
       0         0         0         0         0         0
       0    2.8339         0         0         0         0
       0         0    1.9581         0         0         0
       0         0         0    1.0419         0         0
       0         0         0         0    0.1661         0
       0         0         0         0         0    3.0000
```

The corresponding Chebfun command for DR-AD is `diff(g, f)`. The entries shown on the diagonal of the operator correspond to the values of $3f^2$ on a six-point Chebyshev grid on $[0, \frac{17}{4}]$. We verify that the AD derivative matches the analytical one by letting it operate on the function x on the interval and measuring the norm between the resulting `chebfun` and the analytical answer from (3.3):

```
>> % Measure the difference between the two functions in the 2-norm
>> norm(dgdf*x - 3*f.^2.*x)
ans =
   3.2832e-15
```

Importantly, not all Fréchet derivatives are multiplication operators. Now suppose we want to obtain $\frac{\mathrm{d}h}{\mathrm{d}f}$, the derivative of h with respect to f. We have that

$$\frac{\mathrm{d}h}{\mathrm{d}f} = \frac{\partial h}{\partial f} + \frac{\partial h}{\partial g}\frac{\mathrm{d}g}{\mathrm{d}f} + \frac{\partial h}{\partial x}\frac{\mathrm{d}\overbrace{x}^{0}}{\mathrm{d}f} = \exp(f) + \frac{\partial h}{\partial g}3f^2. \tag{3.4}$$

Thus, we are left with finding an expression for $\frac{\partial h}{\partial g}$. Since

$$h = \exp(f) + g',$$

we have

$$h + \delta h = \exp(f) + (g + \delta g)',$$

so $\frac{\partial h}{\partial g}$ must be equal to $\mathcal{D}$ — the differentiation operator on the interval. Equation (3.4) becomes

$$\frac{\mathrm{d}h}{\mathrm{d}f} = \exp(f) + \mathcal{D} \cdot 3f^2,$$

and we see that $\frac{\mathrm{d}h}{\mathrm{d}f}$ is a linear operator, which has the following action on a function:

$$\frac{\mathrm{d}h}{\mathrm{d}f} : k(x) \mapsto \exp(f(x))k(x) + [3f(x)^2 k(x)]'. \tag{3.5}$$

We obtain the `linop` representation of the operator (3.5) via

```
>> dhdf = getder(h)        % Obtain the derivative of h w.r.t. f.
dhdf = linop
   Linear operator operating on chebfuns defined on:
     interval [0,4.125]
   with functional representation:
     @(u) feval(a,u)+feval(b,u)
   with n = 6 realization:
     1.0000   14.3888   -2.7479    0.7719   -0.0890    0.7273
          0    1.9871    1.8987   -0.4519    0.0498   -0.4020
          0   -2.7480    0.6080    0.8174   -0.0720    0.5556
          0    1.2289   -1.5361    1.7165    0.1611   -1.0525
          0   -0.8492    0.8491   -1.0104    1.1710    3.8080
          0    1.5191   -1.4505    1.4622   -0.8434   15.0819
   and differential order 1
```

The corresponding Chebfun command for DR-AD is `diff(h, f)`. If we let $\frac{\mathrm{d}h}{\mathrm{d}f}$ operate on the function x^2, we expect the result to be

$$\frac{\mathrm{d}h}{\mathrm{d}f}x^2 = \exp(f(x))x^2 + [3f(x)^2x^2]'.$$

We can thus confirm that the AD result is the correct operator by letting $\frac{\mathrm{d}h}{\mathrm{d}f}$ operate on the function x^2 on the interval and measure the norm between the resulting `chebfun` and the analytical derivative:

```
>> % Measure the difference between the computed and analytical answer
>> norm(dhdf*x.^2 - (exp(f).*x.^2 + diff(3*f.^2.*x.^2)))
ans =
   8.1365e-13
```

We now describe a major difference between DR-AD and RT-AD. By design, RT-AD is only able to compute derivatives of h with respect to f. However, DR-AD is able to compute the derivative of any function with respect to any other function. Hence, it can also compute the derivative of h with respect to g. We obtain the `linop` representation of that operator by calling `diff(h, g)`, and confirm its correctness (from above, we expect it to be $\mathcal{D}$) by letting it operate on the function $\sin(x^2)$:

```
>> dhdg = diff(h, g);
>> % Measure the difference between the computed and analytical answer
>> norm(dhdg*sin(x.^2) - 2*x.*cos(x.^2))
ans =
   3.6599e-14
```

After discussing both DR-AD and RT-AD, our conclusion is that if we need to be able to compute the derivative of any function with respect to any other function, DR-AD is the best option. If we know *a priori* with respect to what variables we

want to obtain derivatives, RT-AD is both simpler to implement and avoids the computational inefficiencies related to DR-AD. The main application of AD in Chebfun is to obtain Fréchet derivatives of nonlinear operators. We know beforehand these will be derivatives with respect to the unknown functions u, v, ... that appear in the differential equations. Therefore, we plan for future version of Chebfun to contain the RT-AD implementation.

3.4 Automatic differentiation of functionals

So far, we have been focusing on Fréchet derivatives of operators mapping from open subsets of one function space to another, that is

$$\mathcal{F} : U \subset V \to W.$$

However, we are also interested in obtaining Fréchet derivatives of real functionals, that is

$$\mathcal{J} : U \subset V \to \mathbb{R}.$$

As for operators, the Fréchet derivatives of functionals satisfy Definition 2.2.4.

If $\mathcal{J}$ is a linear functional, that is, it satisfies

$$\mathcal{J}(\alpha u + \beta v) = \alpha \mathcal{J}(u) + \beta \mathcal{J}(v), \qquad \forall \alpha, \beta \in \mathbb{R}, \quad u, v \in U,$$

then $\mathcal{J}$ is its own Fréchet derivative. Moreover, the derivatives of nonlinear functionals will be linear functionals.

We need to be careful about the choice function spaces when working with functionals. For example, since elements of the linear space $L_2(a, b)$ are not defined pointwise, an evaluation functional cannot act on that space. Likewise, the functional

$$\mathcal{J} : u(x) \mapsto \int_a^b u(x)^2 \mathrm{d}x$$

cannot be differentiated at a function $u \notin L_2(a, b)$.

For the first example of AD of functionals in Chebfun, assume that our function space of interest is that of continuous functions on the interval $[0, 5]$. Now consider the *evaluation functional* at the midpoint of the domain:

$$\mathcal{E} : u(x) \mapsto u(2.5).$$

Since $\mathcal{E}$ is clearly a linear functional, it is its own derivative. In the Chebfun framework, a linear functional is represented by a "row-`linop`", an object of dimension $1 \times \infty$, which maps `chebfun` objects to scalars. We obtain the `linop` representation of $\mathcal{E}$ through AD as follows:

```
>> x  = chebfun('x', [0 5]);
>> u  = ADchebfun(sin(x));
>> Eu = u(2.5);                % Evaluate u at x = 2.5
>> dEdu = getder(Eu);          % Obtain the Frechet derivative of E
```

Looking at the five-point discretisation of the operator `dEdu`, we have:

```
>> dEdu(5)
ans =
     0     0     1     0     0
```

As expected, if this discretisation acts on a 5 point discretisation of a `chebfun`, it picks-out the value at $x = 2.5$. We confirm this is the case by letting it act on the function x^2:

```
>> dEdu*x.^2 - 2.5^2
ans =
     0
```

Another linear functional is the definite integral operator,

$$\mathcal{S} : u(x) \mapsto \int_0^5 u(x)\mathrm{d}x.$$

In Chebfun, the command `sum(u)` returns the definite integral of a `chebfun` `u`. We can thus obtain the `linop` representation of $\mathcal{S}$ as follows:

```
>> Su = sum(u);              % Compute definite integral of u
>> dSdu = getder(Su)         % Obtain the Frechet derivative of S
dSdu = linop
   Linear operator operating on chebfuns defined on:
     interval [0,5]
   with n = 6 realization:
     0.1000    0.9019    1.4981    1.4981    0.9019    0.1000
```

The values shown correspond to the Clenshaw-Curtis weights on a six-point Chebyshev grid on $[0, 5]$.

As an example of a nonlinear functional, consider the "half norm-squared" functional on $[0, 5]$,

$$\mathcal{J} : L_2[0, 5] \to \mathbb{R}, \qquad \mathcal{J} : u(x) \mapsto \tfrac{1}{2}\left\|u(x)\right\|_2^2 = \tfrac{1}{2}\int_0^5 u(x)^2\mathrm{d}x.$$

We can obtain its analytical Fréchet derivative in the standard perturbation fashion:

$$\begin{aligned}\mathcal{J}(u+v)-\mathcal{J}(u) &= \tfrac{1}{2}\left[\int_0^5 (u(x)+v(x))^2\mathrm{d}x - \int_0^5 u(x)^2\mathrm{d}x\right]\\ &= \tfrac{1}{2}\left[\int_0^5 (u(x)^2+2u(x)v(x)+v(x)^2)\mathrm{d}x - \int_0^5 u(x)^2\mathrm{d}x\right]\\ &= \int_0^5 u(x)v(x)\mathrm{d}x + O\big(\|v\|^2\big).\end{aligned}$$

Hence, the Fréchet derivative of $\mathcal{J}$ is the linear operator

$$\mathcal{J}: L_2[0,5]\to\mathbb{R}, \qquad \mathcal{J}': v(x)\mapsto \int_0^5 u(x)v(x)\mathrm{d}x.$$

We now proceed to linearise $\mathcal{J}$ around $u(x)=\sin(x)$, and check the correctness of the AD computed derivative by letting it operate on the function e^x as follows:

```
>> Ju = .5*sum(u.^2);        % Evaluate J at u
>> dJdu = getder(Ju);        % Obtain the Frechet derivative
>> dJdu*exp(x) - innerProduct(sin(x), exp(x))
ans =
   2.8422e-14
```

Above, we called the `innerProduct` method of Chebfun, which returns the L_2 inner product of two `chebfun` objects.

3.5 Taylor testing for correctness of derivatives

A useful technique for confirming the correctness of derivatives computed with AD is known as *Taylor remainder convergence testing* or simply *Taylor testing*. For a discussion in the literature, see [FHFR13]. The major benefit of Taylor testing, as will be evident below, is that it works as a black-box method for checking correctness of derivatives; there is no need to work out analytical derivatives to compare against, nor compare against derivatives computed via finite differences.

Taylor testing is based on the observation that given an operator $\mathcal{F}: U\subset V\to W$, an argument u and a perturbation to the argument δu, then by Taylor's theorem,

$$T_1(h) := \|\mathcal{F}(u+h\delta u)-\mathcal{F}(u)\|_W \to 0 \quad \text{at } O(h).$$

However, if additionally, one has the Fréchet derivative of $\mathcal{F}$ at u, $\mathcal{F}'(u)$, then Taylor's theorem dictates that

$$T_2(h) := \|\mathcal{F}(u+h\delta u)-\mathcal{F}(u)-h\mathcal{F}'(u)\delta u\|_W \to 0 \quad \text{at } O(h^2).$$

T_1 and T_2 are known as *first-* and *second-order Taylor remainders*, respectively.

The procedure of Taylor testing is as follows. Let an operator $\mathcal{F}$ be given. Assume that we have another operator $\mathcal{A}$, and want to test whether $\mathcal{A}$ is the Fréchet derivative of $\mathcal{F}$. Then, to test whether the relation $\mathcal{A} = \mathcal{F}'$ holds, we repeatedly halve h, and check whether

$$\frac{T_2(h/2)}{T_2(h)} \cong \frac{1}{4}.$$

As an example, consider the operator

$$\phi(x,u) : u \mapsto (u')^3 + \sin(u^2) - 1. \tag{3.6}$$

The following code performs Taylor testing to confirm that AD in Chebfun computes the correct Fréchet derivative, ϕ'. The call to the `norm` method in the code returns the L_2 norm of the `chebfun` passed. In Figure 3.3, we show a log-log plot demonstrating the results. The code shown shrinks the value of h by a factor of 5 at each step (to avoid too many points cluttering the plot). Using the AD-supplied derivative, we observe the second-order convergence, whereas without using the derivative, we achieve first-order convergence.

```
% Operator to be differentiated
phi = @(u) diff(u).^3 + sin(u.^2) - 1;
% Function we are differentiating around
u = chebfun(@(x) exp(x) + x.^2);
% Seed random generator to ensure same values
rng(1311);
% Create a random chebfun perturbation
del = chebfun(rand(8,1));

for hCounter = 1:18
    % Repeatedly shrink the value of h, and compute perturbation
    h = 5^-hCounter;          pert = h*del;
    % Evaluate phi at u and u+pert, and compute Frechet derivative
    phi_u = phi(u);           phi_up = phi(u + pert);
    dphidu = diff(phi_u, u);
    % Evaluate Taylor remainders
    T1(hCounter) = norm(phi_up - phi_u);
    T2(hCounter) = norm(phi_up - phi_u - dphidu*pert);
end
```

The idea of Taylor testing turns out to be useful in Section 5.1, where we want to confirm that we are using the correct derivatives of complicated functions involving norms. Note that Taylor testing relies on smoothness assumptions, for example, with the operator

$$\phi(x,u) : u \mapsto \sin((u')^3) + \sin(u^2) - 1. \tag{3.7}$$

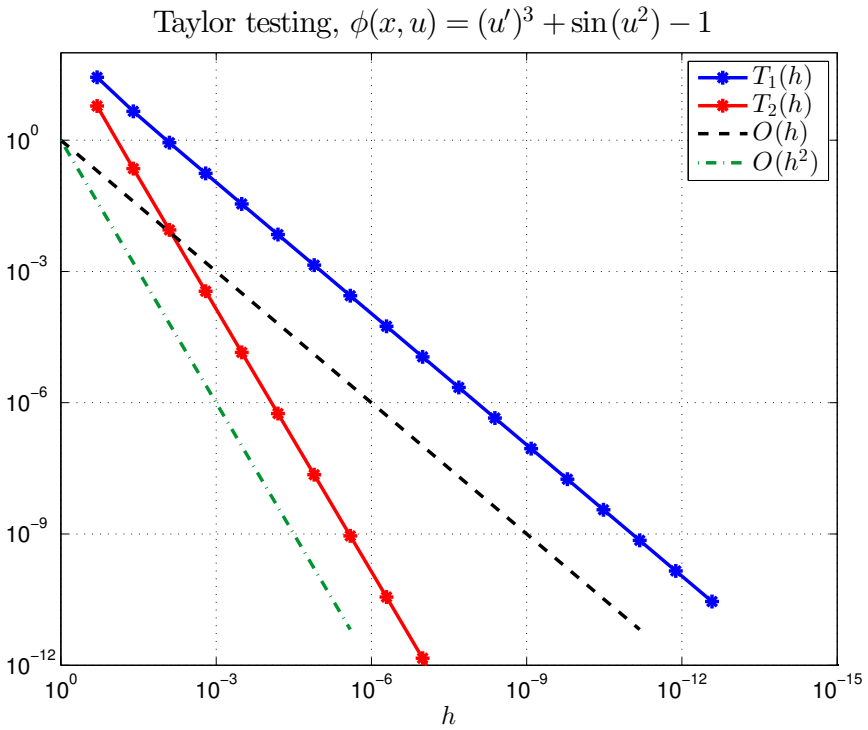

Figure 3.3: A log-log plot showing the results of Taylor testing for correctness of derivatives of the operator (3.6), supplied by AD in Chebfun. The $O(h^2)$ convergence for the AD supplied derivative (red line) is evident.

and again differentiating at $u(x) = e^x + x^2$, we get the results shown in Figure 3.4. Here, the quadratic behaviour of the second-order Taylor remainder does not become evident until the perturbations are small enough. With even less smoother operators, such as the absolute value operator

$$\phi(x,u) : u \mapsto |u|,$$

our Taylor testing algorithm fails, because the operator is not Fréchet differentiable.

3.6 Automatic numerical solution of ODE BVPs and the `chebop` class

Now that we have established that it is possible to compute Fréchet derivatives using automatic differentiation, we present Algorithm 7 for automatic numerical solution of the nonlinear BVP $\mathcal{F}(u) = 0$. The algorithm combines damped Newton iteration in function space and automatic Fréchet differentiation to generate a Newton sequence of guesses of solutions, $\{u_k\}$. At the kth Newton iteration, it evaluates the residual of the operator, $\mathcal{F}(u_k)$, and uses AD to compute the Fréchet derivative of $\mathcal{F}$ at u_k.

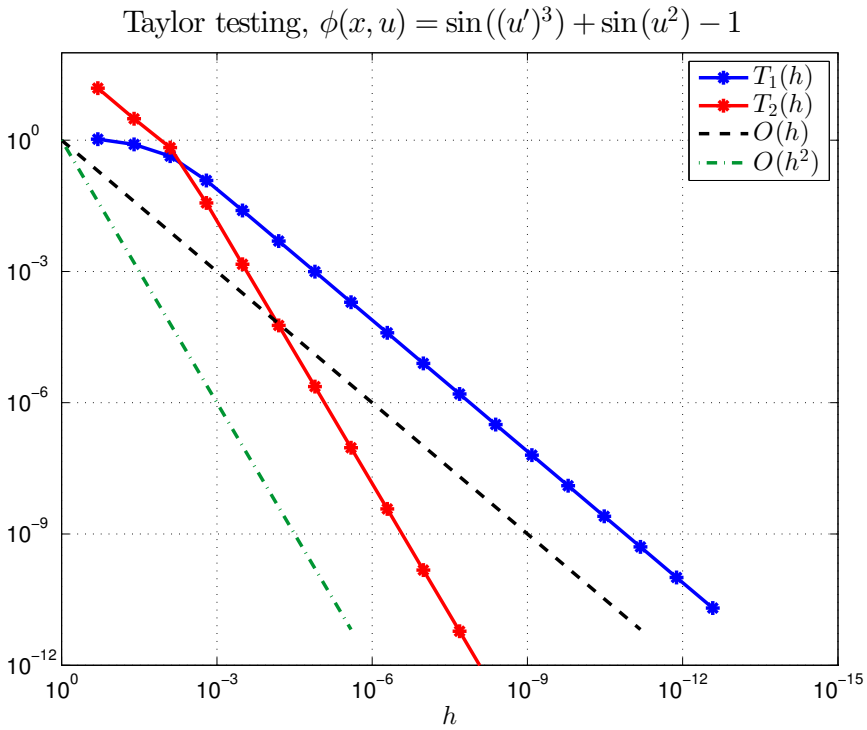

Figure 3.4: A log-log plot showing the results of Taylor testing for correctness of derivatives of the operator (3.7), supplied by AD in Chebfun. The $O(h^2)$ convergence for the AD supplied derivative (red line) only becomes evident once the perturbations are small enough.

It employs the algorithm `GENIB` to compute damping factors λ_k, and the algorithm `LENIB` to supply an estimate $[e_k]$ of the error $\|u_k - u^*\|_V$ (where u^* is the exact solution for which $\mathcal{F}(u^*) = 0$, and V is the function space in which we seek a solution).

The `chebop` class implements Algorithm 7 in Chebfun. It is designed to be a black-box solver for nonlinear ODEs, and has become an important part of Chebfun. Here, we give examples of how BVPs can be solved with the `chebop` class, in Section 3.6.1, we compare it with the built-in BVP solvers of MATLAB.

The `chebop` class uses anonymous functions to enable evaluation of the differential equation and boundary conditions of problems. For the differential equation, the first argument of the anonymous function is the independent function of the interval the problem is specified on; subsequent arguments denote the unknown functions. Assume we want to create a `chebop` representing the BVP operator of the problem we gave as an example in Section 2.3.3,

$$\phi(x,u) = 0.01u'' + uu' - xu = 0, \quad u(0) = -\frac{7}{6}, \quad u'(1) = \frac{3}{2}. \tag{3.8}$$

Algorithm 7 An error-oriented damped Newton iteration for nonlinear BVPs.

Input: A BVP operator, $\mathcal{F}$. An initial guess of a solution, u_0. Error tolerance η_{err}. Maximum number of iterations, $k_{\max}$.

Output: A solution u, for which $\|u - u^*\| \leq \eta_{\text{err}}$ (where $\mathcal{F}(u^*) = 0$), or a `flag` which denotes no convergence.

$u_k \leftarrow u_0$, $k \leftarrow 0$, `damped` $\leftarrow$ `TRUE` ▷ Initialisation

repeat

  Evaluate and store $\mathcal{F}_k = \mathcal{F}(u_k)$.

  Use AD to compute the Fréchet derivative of $\mathcal{F}$ at u_k, $\mathcal{F}'(u_k)$.

  Compute the Newton update δu_k via $\mathcal{F}'_k \delta u_k = -\mathcal{F}_k$.

  if `damped = TRUE` **then**

    Compute steplength λ_k using algorithm `GENIB`.

    $u_{k+1} \leftarrow u_k + \lambda_k \delta u_k$

  else

    $u_{k+1} \leftarrow u_k + \delta u_k$

    Compute error estimate, $[e_k]$ of $\|u - u^*\| \leq \eta_{\text{err}}$ using algorithm `LENIB`

  end if

  Monitor contraction factor to decide whether we need damping or not,
  and assign `damped` $\leftarrow$ `TRUE` or `damped` $\leftarrow$ `FALSE` accordingly.

  $k \leftarrow k + 1$

  if $k > k_{\max}$ **then**

    return `flag`: Newton iteration not convergent within $k_{\max}$ iterations.

  end if

until $[e_k] < \eta_{\text{err}}$

return $u \leftarrow u_k$

Here, we seek a solution $u \in H^2(0,1)$, and regard ϕ as an operator

$$\phi : H^2(0,1) \to L_2(0,1).$$

Before continuing, we confirm that ϕ is indeed a mapping into $L_2(0,1)$, and that its Fréchet derivative is bounded at any u we differentiate ϕ at. We have that

$$\|\phi(x,u)\|_{L_2} = \|0.01u'' + uu' - xu\|_{L_2} \leq \|0.01u''\|_{L_2} + \|uu'\|_{L_2} + \|xu\|_{L_2}$$

by the Minkowski inequality [AF03, Thm. 2.8]. Since $u \in H^2(0,1)$ by assumption, the first term is clearly finite. To check that the remaining terms are finite, we recall the Sobolev embedding theorem discussed in Section 2.2.1, in particular (2.12), which states that since $u \in H^2(0,1)$, both u and u' are continuous on the closed domain $[0,1]$. Hence, the functions uu' and xu are continuous on $[0,1]$, so $\|uu'\|_{L_2}$ and $\|xu'\|_{L_2}$ are finite. Thus, $\phi : H^2(0,1) \to L_2(0,1)$.

To confirm that the Fréchet derivative of ϕ is a bounded operator, we note that its Fréchet derivative at a point $u \in H^2(0,1)$ is the operator

$$\phi'_u : v \mapsto 0.01v'' + uv' + u'v - xv, \qquad v \in H^2(0,1).$$

To show that ϕ'_u is bounded, we need to show that for all $v \in H^2(0,1)$, there exists some C_u such that

$$\|\phi'_u v\|_{L_2} \leq C_u \|v\|_{H^2}.$$

Using the Minkowski inequality again, the Cauchy-Schwarz inequality for L_2 spaces (a special case of the Hölder inequality [AF03, Thm. 2.4]), which states that

$$\|fg\|_{L_2} \leq \|f\|_{L_2} \|g\|_{L_2}, \qquad \forall f,\, g \in L_2$$

and that

$$\|v\|_{H^p} = \left(\sum_{k=0}^{p} \int_0^1 |v^{(k)}(x)|^2 \mathrm{d}x \right)^{1/2} \geq \left(\int_0^1 |v^{(j)}(x)|^2 \right)^{1/2} = \left\|v^{(j)}\right\|_{L_2}, \quad j = 0, \ldots, p,$$

we find that

$$\begin{aligned}
\|\phi'_u v\|_{L_2} &\leq \|0.01v''\|_{L_2} + \|uv'\|_{L_2} + \|u'v\|_{L_2} + \|xu\|_{L_2} \\
&\leq 0.01 \|v''\|_{L_2} + \|u\|_{L_2} \|v'\|_{L_2} + \|u'\|_{L_2} \|v\|_{L_2} + \|x\|_{L_2} \|v\|_{L_2} \\
&\leq 0.01 \|v''\|_{L_2} + \|u\|_{H^1} \|v'\|_{L_2} + \left(\|u\|_{H^1} + \tfrac{1}{\sqrt{3}} \right) \|v\|_{L_2} \\
&\leq \tfrac{1}{3} C_u \left(\|v''\|_{L_2} + \|v'\|_{L_2} + \|v\|_{L_2} \right) \\
&\leq C_u \|v\|_{H^2},
\end{aligned}$$

where $C_u = 3\left(\frac{1}{\sqrt{3}} + \|u\|_{H^1}\right) < \infty$ since $u \in H^2 \hookrightarrow H^1$. Thus, ϕ'_u is a bounded operator.

To solve the BVP (3.8), we start by creating an anonymous function `phi` as follows:

```
>> phi = @(x,u) 0.01*diff(u,2) + u.*diff(u) - x.*u;
```

Next, we call the `chebop` constructor with `phi` and the interval that the problem is defined on

```
>> N = chebop(phi, [0 1]);
```

One imposes boundary conditions on the problem via the boundary conditions fields of the `chebop` object as follows:

```
>> N.lbc = @(u) u + 7/6;
>> N.rbc = @(u) diff(u) - 3/2;
```

Here, we have imposed the non-homogeneous Dirichlet and Neumann conditions $u(0) = -7/6$, $u'(1) = 3/2$, as left and right boundary conditions respectively.

As is evident from Chapter 2, algorithms implementing Newton's method require an initial guess of the solution, u_0, to start the iteration. One can either pass an initial guess for the `chebop` solver by assigning to the `init` field of the `chebop`:

```
>> N.init = chebfun(@(x) 3*x/2 - 7/6, [0 1]);
```

or if none is passed, the solver automatically constructs an initial guess. If the boundary conditions are linear, the initial guess is the polynomial of lowest order that satisfies the boundary conditions. If they are nonlinear, the initial guess is the polynomial of lowest order satisfying the boundary conditions linearised about $u = 0$. In the former case, all subsequent linearised problems solved have homogeneous boundary conditions. In practice, we have found that having initial guesses satisfying the boundary conditions can speed up, or even be necessary for the convergence of the Newton iteration.

To solve a `chebop` problem, one calls the overloaded "nonlinear backslash operator" with the right-hand side of the differential equation. The right-hand side can either be a `chebfun`, or a constant numerical value. The backslash operator calls the code `solvebvp_GEN`, which implements the global error oriented Newton algorithm `GENIB` described in Section 2.5.1. We have also implemented methods for the global residual based Newton algorithm `GRNIB` and the local Newton algorithms `LENIB` and `LRNIB` from Chapter 2. All methods use the default tolerance $\eta = 10^{-10}$ as a termination criterion (as described in the respective algorithms). The output of all solvers is a `chebfun` representing the solution u, as well as a MATLAB `struct` that contains useful information about solution of problem.

Using the lines of code above, we now solve (3.8) via

```
>> [u, info] = N\0;
```

In Figure 3.5, we plot the solution obtained, as well as the norm of the Newton updates and the step-size λ_k at each Newton iteration. Here, we loosened the tolerance of the solver to $\eta_{\text{err}} = 10^{-8}$, as the method runs into issues with ill-conditioning of the spectral differentiation matrices with the default tolerance. The iteration starts out in the damped phase, and although the convergence is not quadratic, we observe that as predicted by the results of Section 2.5, the H^2 norms of the Newton updates decrease monotonically. Once we reach the undamped phase, the quadratic convergence of Newton's method is evident. The solution returned, `u`, is a `chebfun` of length 233,

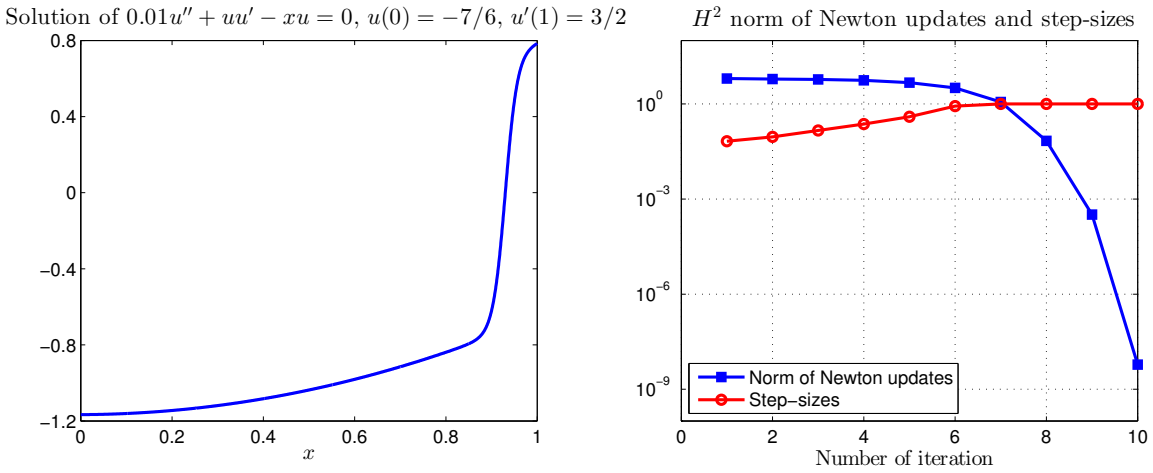

Figure 3.5: Solution of the BVP (3.8), obtained via `chebop` nonlinear backslash.

with a residual 2-norm 3.5×10^{-11}. Furthermore, the local Newton algorithm `LENIB` from Chapter 2 provides us with the error estimate $\|u^* - u\|_{H^2} \approx 6.0 \cdot 10^{-9}$. Note that even though the error estimate is derived to be an upper bound of the error, it relies on sampling of global Lipschitz constants, so it can not be guaranteed to be an upper bound. However, we have found in practice that when we solve nonlinear BVPs with a known exact solution to compare with, the error estimates are close to the actual error.

A useful feature of the `chebop` class is how easily problems can be modified. Suppose we want to solve a slight modification to the problem above:

$$0.01u'' + uu' - xu = \tfrac{1}{2}\sin(5\pi x), \quad u(0) = 0, \quad u'(1) = 1, \tag{3.9}$$

or modify the problem and impose periodic boundary conditions:

$$0.01u'' + uu' - xu = \tfrac{1}{2}\sin(5\pi x), \quad u(0) = u(1), \quad u'(0) = u'(1). \tag{3.10}$$

We then only need to execute

```
>> rhs = chebfun(@(x) .5*sin(5*pi*x), [0 1]);
>> N.lbc = 'dirichlet';
>> N.rbc = @(u) diff(u) - 1;
>> [u info] = N\rhs;
```

and

```
>> N.bc = 'periodic';
>> [u info] = N\rhs;
```

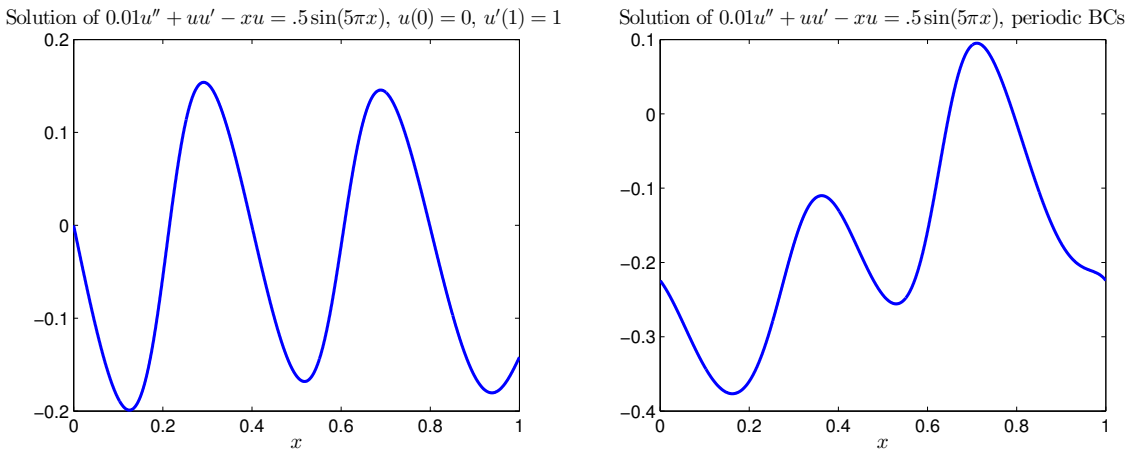

Figure 3.6: Solutions of the BVPs (3.9) and (3.10).

The solutions to (3.9) and (3.10) are shown in Figure 3.6.

We note that above, we assigned to the `N.bc` field an argument indicating that we were seeking a periodic solution. The `bc` field allows the imposition of quite general conditions on the solution, for example (still assuming that the problem to be solved is defined on $[0, 1]$),

- Non-separated conditions, such as $u(0) + 3u'(1) = 2$.
- Interior point conditions, such as $u(\frac{1}{2}) = 3$, $u'(\frac{3}{4}) = -1$.
- Conditions on the solution over the whole interval, such as $\int_0^1 u(x)dx = 1$.

The capability of being able to impose such conditions is based on the observation that they are all functionals. As described in Section 3.4, we are able to obtain derivatives of functionals, which gives us a constraint to impose on the solution of the linearised BVP solved at each step of Newton's method.

As an example of imposing such conditions, consider the differential equation governing the motion of a pendulum:

$$\ddot{\theta} + \sin(\theta) = 0.$$

Here, θ denotes the angular displacement (in radians) from the stable equilibrium at which the pendulum lies at rest, and $\ddot{\theta}$ is the second derivative of θ with respect to time. $\ddot{\theta}$ has the physical interpretation of the angular acceleration of the pendulum. Now suppose we are interested in computing two solutions in a timespan from the initial time $t_0 = 0$ to the final time $T = 5$:

1. Where the pendulum starts at $\theta = \frac{\pi}{2}$, and has the same angular displacement at time $t = 1$ and $t = 5$, that is

$$\theta(0) = \tfrac{\pi}{2}, \qquad \theta(1) = \theta(5). \tag{3.11}$$

2. The pendulum starts at $\theta = 0$, and traces out a curve with a arc-length 2π, that is

$$\theta(0) = 0, \qquad \int_0^5 \sqrt{1 + \dot{\theta}^2}\mathrm{d}t = 2\pi. \tag{3.12}$$

The first problem is an example of a mixture of boundary and interior point conditions, while the second problem imposes a nonlinear condition applying over the whole interval. We set up and solve the problems with the `chebop` class as follows:

```
>> N = chebop(@(t, theta) diff(theta,2) + sin(theta), [0 5]);
>> N.bc = @(t,theta) [theta(0) - pi/2, theta(1) - theta(5)];
>> [theta1, info1] = N\0;
>> % Impose different conditions
>> arcLength = @(theta) sum(sqrt(1 + diff(theta).^2));
>> N.bc = @(t,theta) [theta(0), arcLength(theta) - 2*pi];
>> [theta2, info2] = N\0;
```

Note that here, we used anonymous functions to define the conditions imposed on the solution, and assigned them to the `bc` field of the `chebop`. In Figure 3.7, we plot the solutions found, which have residual 2-norm of 2.5×10^{-12} and 3.8×10^{-12} respectively, and H^2 error estimates 1.2×10^{-13} and 5.6×10^{-12}. We also check that the non-standard conditions of (3.11) and (3.12) are satisfied:

```
>> norm(theta1(1) - theta1(5))
ans =
     2.6645e-15
>> norm(arcLength(theta2) - 2*pi)
ans =
     1.7764e-15
```

3.6.1 Comparison with existing software

Here, we list the results of a comparison of solving nonlinear BVPs using the `chebop` class, the built-in MATLAB solvers `bvp4c` [KS01] and `bvp5c` [KS08], and the solver `bvp4cAD` [SKF05]. The solver `bvp4cAD` is built on `bvp4c`, but uses the MATLAB AD toolbox [For06] to obtain derivatives via automatic differentiation, rather than via finite difference schemes. Such a comparison has previously been reported in [BD12].

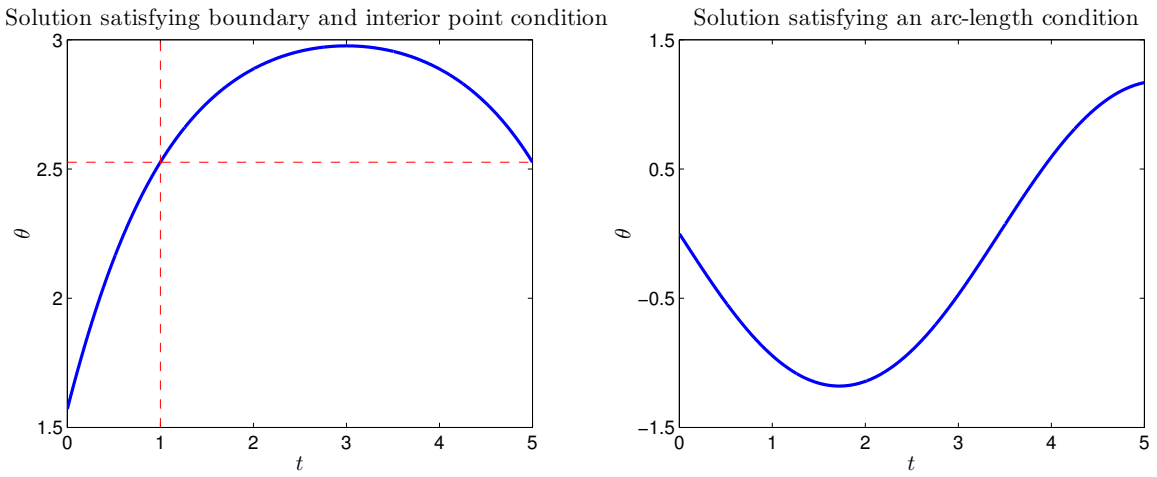

Figure 3.7: Solutions of the BVP describing the motion of a pendulum, satisfying the conditions (3.11) and (3.12).

From the algorithmic and the user-friendliness point of view, we believe that the `chebop` class has several important advantages over the other solvers listed. The key algorithmic advantages are:

- The solver is a pseudospectral method, so the `chebop` class is capable of highly accurate solutions.
- When using the error-oriented algorithm `solvebvp_GEN`, the convergence criterion is based on an error estimate of the solution in a Sobolev norm determined by the function space in which a problem is posed. On the other hand, in `bvp4c`, the convergence criteria is based on an estimate of the residual of the differential equation. In `bvp5c`, the criteria is based on an estimate of the pointwise error of the solution.
- Similarly, the damping strategy of the `chebop` solver is based on the natural Sobolev norms of the Newton updates. In `bvp4c` and `bvp5c`, the damping strategy is based on estimates of the L_2 norms of the updates.

The key implementation advantages are:

- We offer users a very convenient syntax, describing problems in a form close to what a scientist or an engineer would write down on a piece of paper.
- There is no need for the error-prone process of rewriting higher order differential equations as a system of first order equations. Thus, it greatly reduces the human time taken to set up problems.

- The solution returned is a function, that is, a `chebfun`. Thus, the user is able to do all the same things with the solution as with any other `chebfun`, for example, find its maximum, mean value, standard deviation etc.

The main drawback of the `chebop` solver as will become evident below is that when it is used to solve problems to low and intermediate tolerances, execution time can be longer than with `bvp4c` and `bvp5c`.

For comparing the efficiency of `chebop` with the other BVP solvers, we consider the six following nonlinear BVPs:

$$\begin{aligned}
u'' + 3.5e^u = 0, &\quad u(0) = u(1) = 0, \\
xu'' - \sqrt{\frac{u'}{x}} = u', &\quad u(1) = 1,\ u(2) = 2, \\
u'' - \cos(x)u' + u\log(u) = 0, &\quad u(0) = 1,\ u(1) = e, \\
u'' - u' + e^{2x}u + u^2 = \sin^2\left(e^x\right), &\quad u(0) = \sin(1),\ u\left(\tfrac{5}{2}\right) = \sin\left(e^{\frac{5}{2}}\right), \\
u'' + 18(u - u^3) = 0, &\quad -u(-1) = u(1) = \tanh(3), \\
u''u' - 48xu = 0, &\quad u(0) = 10^{-4},\ u(1) = 1.1^4.
\end{aligned}$$

They can be shown to have the exact solutions

$$u(x) = -2\ln\left[\frac{\cosh\left(\left(x - \frac{1}{2}\right)\frac{\theta}{2}\right)}{\cosh\left(\frac{\theta}{4}\right)}\right], \text{ where } \theta \text{ is a sol. of } \theta = \sqrt{7}\cosh\left(\frac{\theta}{4}\right),$$

$$u(x) = \frac{A^2}{8}(x^2 - 1) - \frac{A}{2}(x - 1) + \frac{1}{4}\log(x) + 1, \text{ where } A = \frac{2\left(1 + \sqrt{7 - \frac{3}{2}\log(2)}\right)}{3},$$

$$u(x) = e^{\sin(x)},$$

$$u(x) = \sin(e^x),$$

$$u(x) = \tanh(3x),$$

$$u(x) = (x + 0.1)^4,$$

respectively, which allows us to compare the true error of the solutions returned.

The first problem is the Bratu problem we encountered in Section 2.6. For our choice of $\lambda = 3.5$, we know it has two solutions, that is, there are two values of θ which satisfy $\theta = \sqrt{7}\cosh\left(\frac{\theta}{4}\right)$. We find one such value of θ numerically using Maple with 25 digits of accuracy; it can then be used to construct the exact solution. The solution corresponding to the value of θ we found turns out to be the same one as the solution found by all the solvers tested.

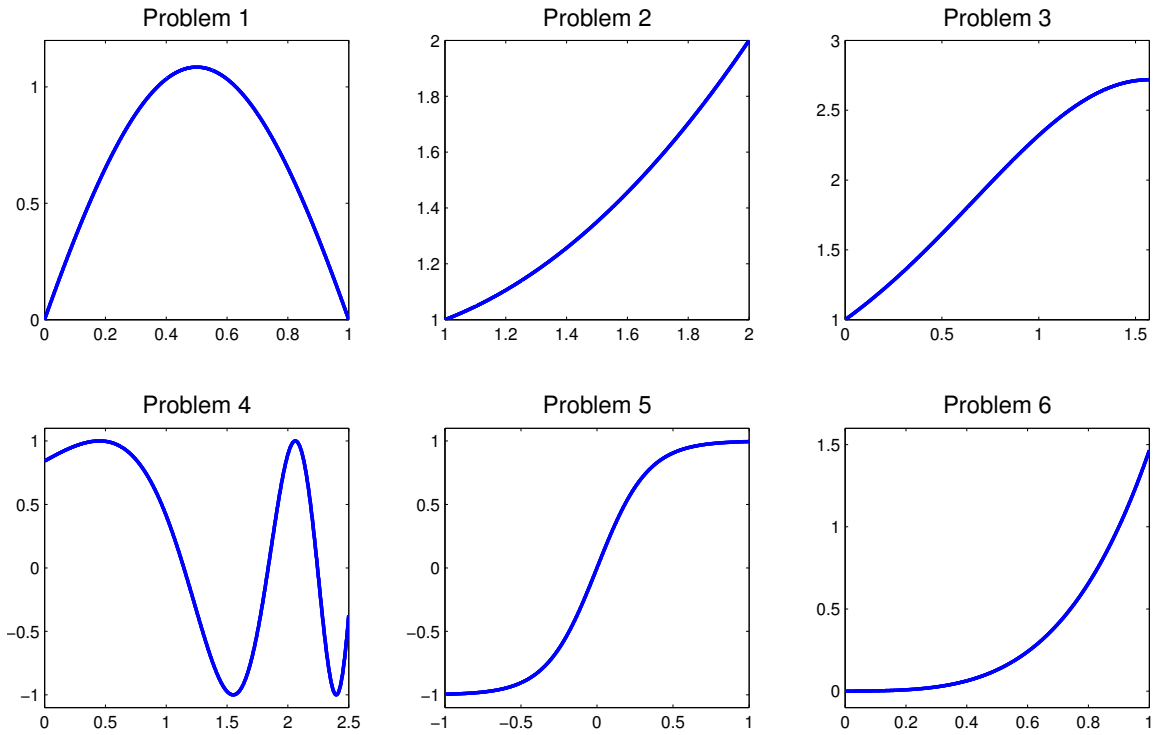

Figure 3.8: Plot of the solutions of the six nonlinear BVPs we consider for comparing solvers.

In Figure 3.8, we plot the solutions of these six problems. To highlight the different syntax of the solvers used, we solve the first problem using Chebfun with the following three lines of code

```
% Create a chebop describing the problem on the interval [0, 1]
N = chebop(@(u) diff(u,2) + 3.5*exp(u), [0 1]);
N.lbc = 0; N.rbc = 0;        % Assign homogeneous Dirichlet BCs
u = N\0;                     % Solve using nonlinear \
```

With the other solvers, we need to rewrite the problem as a first order system. Using `bvp4c`, the problem is then solved with the following lines of code:

```
dydx = @(x,y) [y(2), -3.5*exp(y(1))];          % First order system
bcres = @(ya,yb) [ya(1), yb(1)];               % BCs residual
solinit = bvpinit(linspace(0,1,5)', [0 1]);    % Initial guess
sol = bvp4c(dydx, bcres, solinit);             % Obtain solution
```

By changing the last line, it is possible to obtain solutions from different solvers, that is, `bvp5c` and `bvp4cAD`.

By changing the tolerances of the solvers, as well as limiting the maximum number

of Newton iterations allowed, it is possible to create work-precision diagrams.[6] These show how much time it takes to get a specific accuracy, by which we mean the maximum absolute value of the difference between the numerical solution and the analytical one. Such plots are shown in Figure 3.9. The running times are found by solving each problem 11 times at a given level of accuracy, subtracting the first running time (to avoid distortion from loading the programs into the cache) and taking the average of the others. For all the problems, we used simple initial guesses, either constants or the guesses constructed automatically by Chebfun.

From Figure 3.9, it is clear that for low accuracy, Chebfun is outperformed by the other methods. However, if we seek solutions with better accuracy, Chebfun usually has clear benefits, taking equivalent or shorter times to return solutions closer to the exact ones than the other solvers. The main reason why we observe this pattern is that the computational overheads for setting up problems with `chebop` are high compared to the other solvers, which rely less (`bvp4cAD`) or not at all (`bvp4c/5c`) on object-oriented programming in MATLAB. As the requested accuracy is increased, the number of gridpoints `bvp4c/5c` and `bvp4cAD` use grows very quickly, in most cases up to many thousands. On the other hand, thanks to using spectral methods, the `chebfun` solutions require much more modest numbers of gridpoints. Thus, the cost of the linear algebra operations for computing each Newton update grows for `bvp4c/5c/4cAD` as the requested accuracy is increased, while for Chebfun, the cost of computing each Newton update remains similar.

Importantly, as the problems only take a couple of seconds to solve, the human time used to set up problems, for example, the process of writing them as first-order systems, will greatly outweigh the computer time used to solve the problems. In that aspect, the `chebop` solver offers many benefits compared with existing software.

To comment further on the results of solving the test problems, we note that generally, the regular `bvp4cAD` method is found to be slower than all the other methods for intermediate and high accuracies. However, if we rewrite the problems in vectorised form, and turn on the vectorised option for `bvp4cAD`, the performance is drastically improved. This is in agreement with [SKF05].

To give a rough idea how time is spent in the `chebop` solver, consider the fourth problem above, which requires four Newton steps to achieve full accuracy, and has

[6]The limiting of Newton steps is necessary as it forces the solvers to return their current guess of the solution even though the other convergence criteria of the solvers have not been met. In this way, we can obtain solutions after various running times which we then use to compare with the true solution. Informally speaking, we give the solvers a fixed time to solve a problem, and measure the accuracy of the solutions obtained in that fixed time.

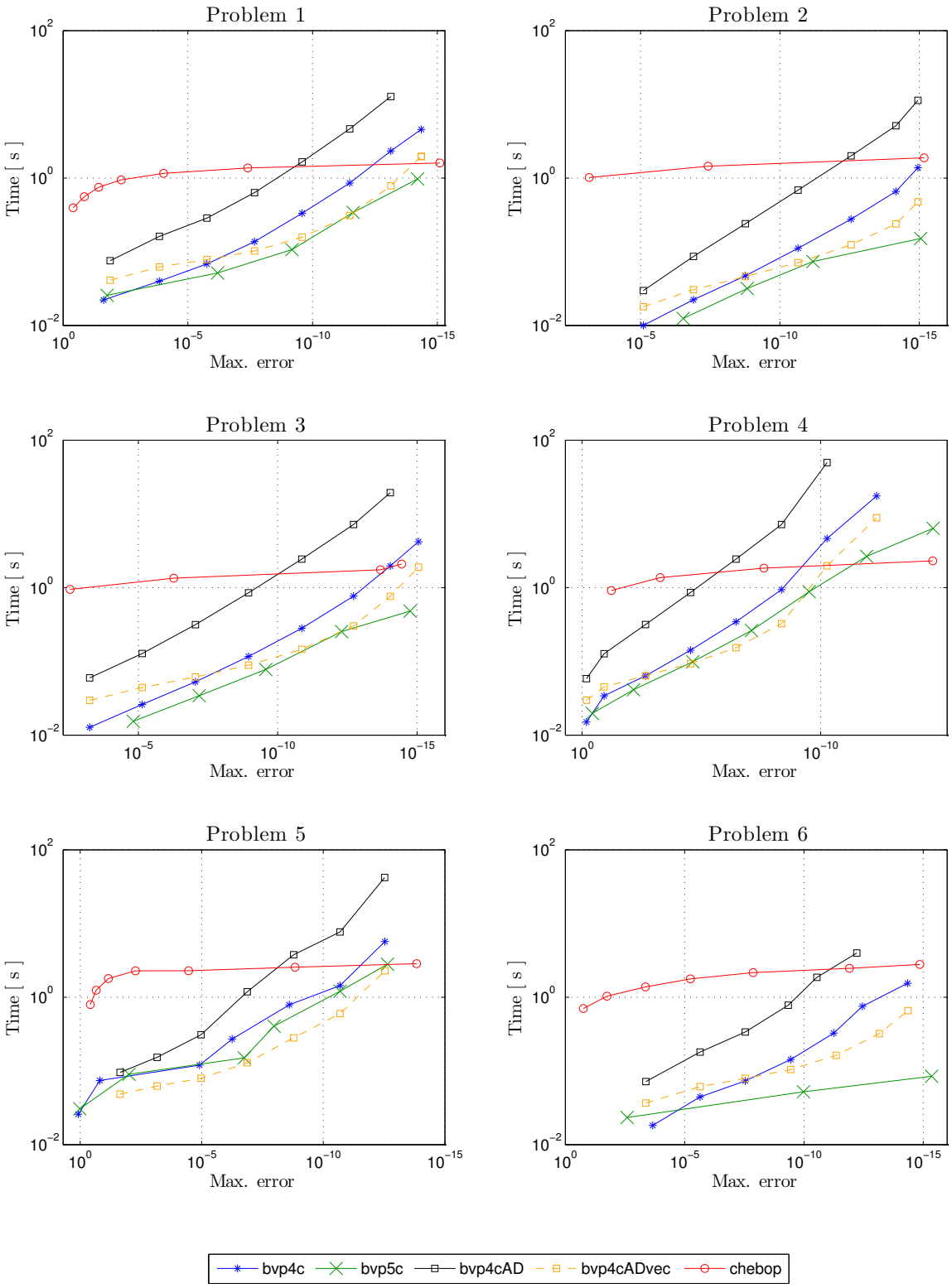

Figure 3.9: Running time vs. accuracy obtained with different solvers for six nonlinear BVPs.

as a solution a `chebfun` of length 55. Obtaining the solution takes 2.4 seconds. The set-up, before the first Newton update is computed (including the first linearisation of the operator), takes 0.44 seconds. At each Newton step, 0.23 seconds are spent on computing derivatives with AD, while 0.17 seconds are spent on discretising the Fréchet derivative and the linear algebra for computing the Newton updates. The rest of the computational time is spent on other operations, and overheads related to object oriented programming in MATLAB.

3.6.2 A graphical user interface to Chebfun – chebgui

To further increase the user-friendliness of using the `chebop` class to solve nonlinear BVPs, we introduced *chebgui* — a graphical user interface to the differential equations capabilities of Chebfun.

Although the syntax used for the `chebop` solver introduced above is close to what a scientist or an engineer would write down on a piece of paper to describe a problem, it still requires some knowledge about Chebfun syntax. On the other hand, `chebgui` is able to take strings describing problems in a "natural language", such as

```
u'' + x*sin(u) - 1,
```

and convert them to a MATLAB anonymous function that the `chebop` class is able to work with, in this case

```
@(x, u) diff(u,2) + x.*sin(u) - 1
```

This is achieved by performing lexical and grammatical analysis on the original string, concepts widely used in compiler theory, see for example [ALSU07]. Such techniques have a long history of applications in symbolic differentiation in computer algebra systems, such as Maple and Mathematica.

In Figure 3.10 we show a screenshot of `chebgui` solving the Carrier problem taken from [BO78, Sec. 9.7]

$$0.01u'' + 2(1 - x^2)u + u^2 = 1, \quad u(-1) = u(1) = 0. \tag{3.13}$$

Important features of `chebgui`, in addition to the capability of converting strings into expressions the `chebop` solver is capable of working with, include the option of exporting problems to executable `m`-files for further explorations, the option of exporting operators and functions to the MATLAB workspace, and a collection of demo problems.

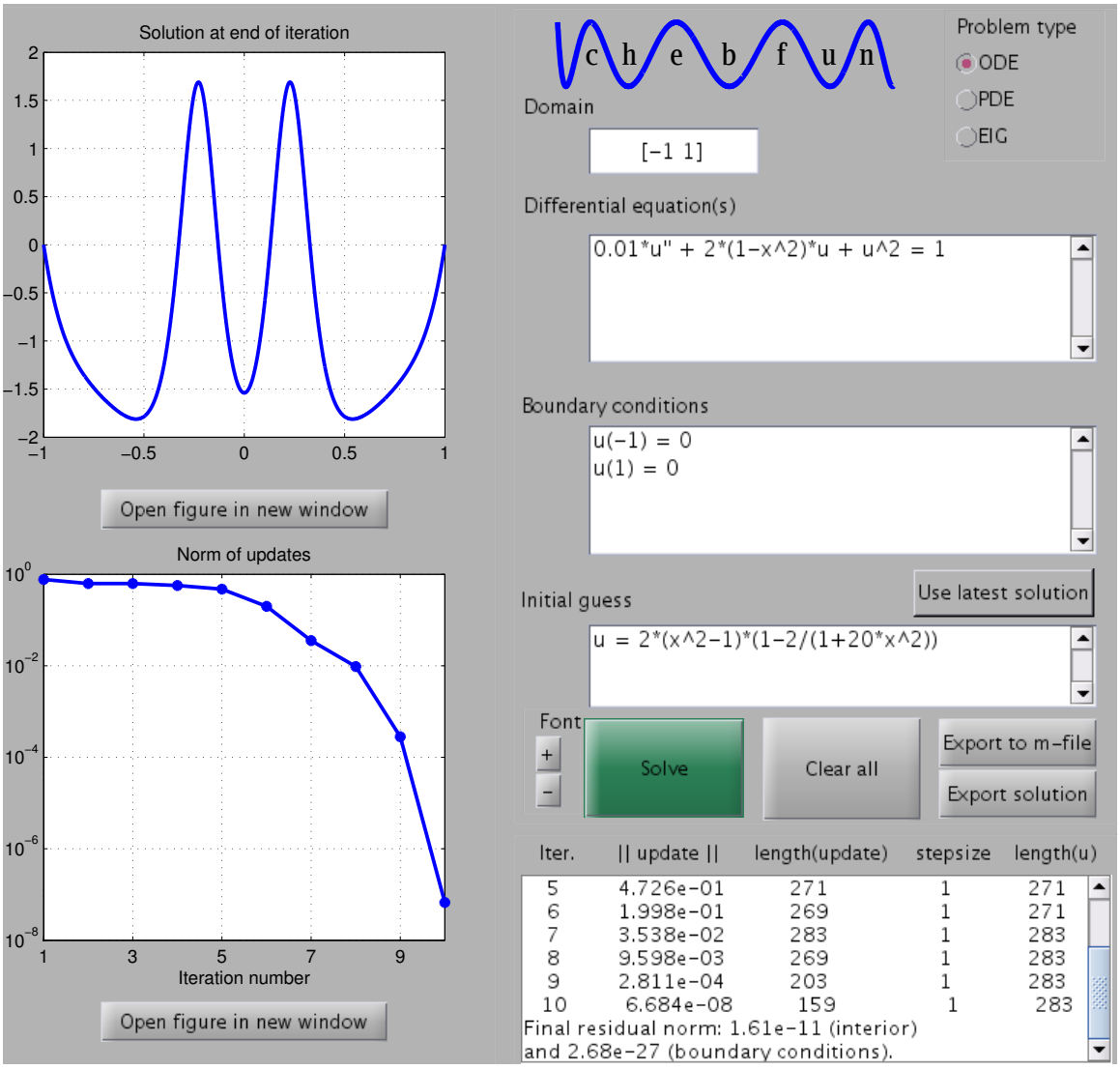

Figure 3.10: Screenshot of `chebgui`, the graphical user interface to Chebfun's differential equations capabilities.

Chapter 4

Automatic Linearity Detection

As discussed in previous chapters, most methods for solving nonlinear problems in scientific computing involve iteration over successive linearisations of the full problem. When the underlying problem is linear, such iterations should in principle converge exactly in one step. However, the additional overhead and setup required for the general nonlinear problem can be wasteful, making it useful for a black-box solver to detect linearity before proceeding. For example, in [GT04] the authors show that determining whether constraints in optimisation problems are linear can reduce problem sizes and speed up solution times. In the ODE context, it is useful for the solver to know whether it has to start the machinery of damped Newton iteration to find a solution.

As previously described, one method for obtaining the derivatives required by iteration schemes for nonlinear problems is via automatic differentiation (AD). AD can be used to detect linearity automatically, and this is discussed in a few places in the literature. In [SH06], the authors implement linearity detection via data-flow analysis. The work [Wal08] describes similar ideas to those presented here for the purpose of obtaining sparsity patterns of Hessians. However, there are some differences in both the implementation and the application between that work and the work presented here, as we will highlight. In particular, our approach is focused on applying linearity detection in the ODE context.

Another topic closely related to automatic linearity detection is also in the context of optimisation. It is highly useful for algorithm selection to be able to detect whether an optimisation problem is convex, so substantial effort is put into automatic techniques for this. For example, the CVX modelling language [GB12] applies a set of rules to determine whether the problems being solved are convex, maintaining a list of mathematical expressions that can break the convexity of a problem. DrAmpl [FO10] analyses expression trees of mathematical models constructed in the AMPL

modelling language and attempts to collect information about the problem such as linearity and convexity. Another example worth mentioning is the Microsoft Excel tool for optimisation problems, known as *Solver*, which combines automatic differentiation and techniques for gathering structural information about problems [NF03].

Our approach for detecting linearity automatically relies on observing that this is closely related to determining what partial derivatives of operators are equal to the zero operator. We will associate with each elementary operator in the evaluation trace *zero derivative information* and *Hessian sparsity information* operators. We denote these operators by ζ and η, respectively. This is similar to standard forward mode AD, where we associate with each elementary operator in the evaluation trace an operator for computing the derivatives of intermediate variables via the chain rule. We append two Boolean vectors z and h, which encode the sparsity patterns of the derivatives of operators, to all intermediate variables in the evaluation trace, and operate on them via ζ and η. Due to its close connections to AD, we call our approach *automatic linearity detection*.

This chapter describes the mathematical observations and algorithms enabling automatic linearity detection. In Section 4.1, we start by discussing properties of linear and affine operators. We give proofs that the Fréchet derivatives of affine operators are invariant over the space they are defined on, and conversely, if an operator is Fréchet differentiable on its domain (assumed to be open and connected) and has a constant derivative, it is affine. These results form the basis for automatic linearity detection of computed quantities. We give formal definitions of the operators ζ and η and the vectors z and h mentioned above. We describe the Boolean logic for propagating information about the sparsity patterns of derivatives of operators. We then prove that our definitions of ζ, η, z and h provide conditions which, when satisfied, guarantee that an operator is linear.

In Section 4.2, we describe how automatic linearity detection can be incorporated into software. We give examples of linearity detection in action, both in the `advar` class introduced in Section 1.3 and in Chebfun.

The origin of the work presented in this chapter is as follows. The original idea that a vector could be used to propagate linearity information is due to Tobin Driscoll. The observation that another vector was required to fully capture nonlinearities is due to the author. The implementation of linearity detection in Chebfun was done jointly by Driscoll and the author, while the linearity detection in the `advar` class was done by the author. Theorem 4.1.6 can for example be found in [Cia13]. Subsequent theorems proved in this chapter are due to the author. The material in this chapter

was first discussed in [BD13], but the presentation has been significantly extended, restructured and clarified for this thesis.

4.1 Linearity of computed quantities

This section derives an algorithm for detecting linearity of computed quantities. We proceed as follows:

1. Definitions 4.1.1–4.1.3 define what it means for an operator to be linear, affine, or affine in a particular argument.

2. Theorem 4.1.6 states that having a constant Fréchet derivative is a sufficient condition for an operator to be affine. Theorem 4.1.7 states that having a constant partial derivative with respect to the jth argument is a sufficient condition for an operator to be affine in the jth argument.

3. Definitions 4.1.9 and 4.1.10 define the zero derivative and Hessian sparsity vectors z and h.

4. Table 4.2 defines the zero derivative and Hessian sparsity information propagation operators ζ and η via their actions on z and h.

5. Theorem 4.1.11 states that the definitions of z and η provide conditions which, when satisfied, are sufficient to guarantee that an operator has a zero partial derivative with respect to a particular argument.

6. Theorem 4.1.12 states that the definitions of z, h, ζ and η provide conditions which, when satisfied, are sufficient to guarantee that a particular column of the Hessian of an operator consists only of zero operators.

7. Corollaries 4.1.13–4.1.15 state that our that our definitions of z, h, ζ and η provide conditions which, when satisfied, are sufficient to guarantee that an operator is affine.

4.1.1 Definitions of linear and affine operators

Let V and W be normed vector spaces, and let U be an open subset of V. In our typical application involving a second order ODE on the interval $[a, b]$, V might be the Sobolev space $H^2(a, b)$, U might be the domain of the differential operator, and W might be the Lebesgue space $L_2(a, b)$.

More generally, we are interested in systems of coupled ODEs. Let $U^{(n)} = U_1 \times \cdots \times U_n$, $V^{(n)} = V_1 \times \cdots \times V_n$ and $W^{(m)} = W_1 \times \cdots \times W_m$, where each V_i is a normed vector space, each U_i is an open subset of V_i, and each W_i is a normed vector space as well. Again, each V_i might be a Sobolev space (whose order is determined by the order of the ODE), U_i the domain of the differential operator corresponding to its ith argument, and W_i might be a Lebesgue space. We denote elements in $U^{(n)}$ by $\mathbf{u}$, that is, $\mathbf{u} = (u_1, \ldots, u_n)$, where each $u_i \in U_i$. We call such an $\mathbf{u}$ an *n-tuple*.

Suppose we have a linear or nonlinear operator $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$. The number of variables that $\mathcal{F}$ operates on, here denoted by n, is known as the *arity* of the operator. Using the notation introduced above, we write $\mathcal{F}(\mathbf{u})$ for $\mathcal{F}(u_1, \ldots, u_n)$.

As discussed in Section 2.1, if $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ is a differential operator representing a system of coupled ODEs, it consists of m components $\mathfrak{f}_1, \ldots, \mathfrak{f}_m$ where each $\mathfrak{f}_i \ : \ U^{(n)} \subset V^{(n)} \to W_i$, so that

$$\mathcal{F}(\mathbf{u}) = \mathcal{F}(u_1, \ldots, u_n) = \begin{bmatrix} \mathfrak{f}_1(u_1, \ldots, u_n) \\ \vdots \\ \mathfrak{f}_m(u_1, \ldots, u_n) \end{bmatrix} = \begin{bmatrix} \mathfrak{f}_1(\mathbf{u}) \\ \vdots \\ \mathfrak{f}_m(\mathbf{u}) \end{bmatrix}.$$

We now make the natural extension of Definition 2.2.3 of linear operators to operators acting on n-tuples of functions.

Definition 4.1.1. *Let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be an operator. $\mathcal{F}$ is said to be* linear *if*

$$\mathcal{F}(\mu\mathbf{u} + \psi\mathbf{v}) = \mu\mathcal{F}(\mathbf{u}) + \psi\mathcal{F}(\mathbf{v})$$

for all scalars μ and ψ and n-tuples of functions $\mathbf{u}, \mathbf{v} \in U^{(n)}$ such that $\mu\mathbf{u}+\psi\mathbf{v} \in U^{(n)}$.

Note that nonlinear dependence on the independent variable in a system of ODEs does not affect the linearity of the operator. Thus, for example, the mapping

$$\mathcal{F} : u_1, u_2 \mapsto x^2 u_1(x) + \tanh(x) u_2''(x)$$

is linear.

Detection of linear operators is not sufficient for our purposes. The reason is that a nonhomogeneous ODE might be expressed as either $\mathcal{L}\mathbf{u} = \mathbf{f}$ or $\mathcal{L}\mathbf{u} - \mathbf{f} = 0$; the operator $\mathcal{F}(\mathbf{u}) = \mathcal{L}\mathbf{u}$ is linear, but $\widetilde{\mathcal{F}}(\mathbf{u}) = \mathcal{L}\mathbf{u} - \mathbf{f}$ is not. Instead, both are instances of the larger class of affine operators. Even so, we refer to the process of determining whether a given operator is affine as "linearity detection", which seems to follow common use of the terminology.

Affine operators are defined as follows [Cia13, p. 480]. Recall from Chapter 2 that $L(V^{(n)}, W^{(m)})$ denotes the vector space of all continuous linear operators from $V^{(n)}$ into $W^{(m)}$.

Definition 4.1.2. *Let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be an operator. $\mathcal{F}$ is said to be* affine *if there exist an operator $\mathcal{A} \in L(V^{(n)}, W^{(m)})$ and a vector $\mathbf{b} \in W^{(m)}$ such that for all $\mathbf{u} \in U^{(n)}$,*

$$\mathcal{F}(\mathbf{u}) = \mathcal{A}\mathbf{u} + \mathbf{b}. \tag{4.1}$$

We now define the notion of an operator being affine in a particular argument.

Definition 4.1.3. *Let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be an operator. $\mathcal{F}$ is said to be* affine in its ith argument *if there exist an operator $\mathcal{A}_i \in L(V_i, W^{(m)})$, an operator*

$$\mathcal{G}_i : (U_1 \times \cdots \times U_{i-1} \times U_{i+1} \times \cdots \times U_n) \subset (V_1 \times \cdots \times V_{i-1} \times V_{i+1} \times \cdots \times V_n) \to W^{(m)}$$

and a vector $\mathbf{b}_i \in W^{(m)}$ such that for all $\mathbf{u} \in U^{(n)}$,

$$\mathcal{F}(\mathbf{u}) = \mathcal{G}_i(u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n) + \mathcal{A}_i u_i + \mathbf{b}_i.$$

If now there exist an operator $\mathcal{A}_j \in L(V_j, W^{(m)})$, an operator

$$\begin{aligned}\mathcal{G}_{i,j} :&(U_1 \times \cdots \times U_{i-1} \times U_{i+1} \times \cdots \times U_{j-1} \times U_{j+1} \times \cdots \times U_n) \\ &\subset (V_1 \times \cdots \times V_{i-1} \times V_{i+1} \times \cdots \times V_{j-1} \times V_{j+1} \times \cdots \times V_n) \to W^{(m)}\end{aligned}$$

and a vector $\mathbf{b}_{i,j} \in W^{(m)}$ such that for all $\mathbf{u} \in U^{(n)}$,

$$\mathcal{F}(\mathbf{u}) = \mathcal{G}_{i,j}(u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_{j-1}, u_{j+1}, \ldots, u_n) + \mathcal{A}_i u_i + \mathcal{A}_j u_j + \mathbf{b}_{i,j}, \tag{4.2}$$

we say that $\mathcal{F}$ is affine in its ith and jth *arguments.*

The notion of $\mathcal{F}$ being affine in more than two of its arguments is defined analogously. In particular, we say that $\mathcal{F}$ is *affine in every argument* if there exist operators $\mathcal{A}_i \in L(V_i, W^{(m)})$ for $1 \leq i \leq n$ and a vector $\mathbf{b} \in W^{(m)}$ such that for all $\mathbf{u} \in U^{(n)}$,

$$\mathcal{F}(\mathbf{u}) = \sum_{i=1}^{n} \mathcal{A}_i u_i + \mathbf{b}.$$

As the following lemma states, if an operator is affine in every argument, it is affine.

Lemma 4.1.4. *Let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be an operator. If $\mathcal{F}$ is affine in every argument, it is affine.*

Proof. If $\mathcal{F}$ is affine in every argument, there exist operators $\mathcal{A}_j \in L(V_j, W^{(m)})$ for $1 \leq j \leq n$ and a $\mathbf{b} \in W^{(m)}$ such that for all $\mathbf{u} \in U^{(n)}$,

$$\mathcal{F}(\mathbf{u}) = \sum_{j=1}^{n} \mathcal{A}_j u_j + \mathbf{b}. \tag{4.3}$$

Define $\mathcal{A} \in L(V^{(n)}, W^{(m)})$ by

$$\mathcal{A}\mathbf{u} = \sum_{j=1}^{n} \mathcal{A}_j u_j. \tag{4.4}$$

Combining (4.3) and (4.4), we see that $\mathcal{F}$ satisfies (4.1), so $\mathcal{F}$ is affine. □

4.1.2 Properties of affine operators

We now derive theorems necessary for computational linearity detection. We recall Definition 2.2.4 of Fréchet derivatives, stated here again in a form suited for our purpose. Let $V^{(n)}$ and $W^{(m)}$ be normed vector spaces, and let $U^{(n)}$ be an open subset of $V^{(n)}$. An operator $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ is said to be *Fréchet differentiable* (or simply *differentiable*) at $\mathbf{u} \in U^{(n)}$ if and only if there exists an operator $\mathcal{L} \in L(V^{(n)}, W^{(m)})$ such that

$$\lim_{\|\mathbf{h}\|_{V^{(n)}} \to 0} \frac{\|\mathcal{F}(\mathbf{u}+\mathbf{h}) - \mathcal{F}(\mathbf{u}) - \mathcal{L}\mathbf{h}\|_{W^{(m)}}}{\|\mathbf{h}\|_{V^{(n)}}} = 0. \tag{4.5}$$

The operator $\mathcal{L}$ is then called the *Fréchet derivative* (or simply the *derivative*) of $\mathcal{F}$ at $\mathbf{u}$, denoted by $\mathcal{F}'(\mathbf{u})$. If $\mathcal{F}$ is differentiable at all points in $U^{(n)}$, we say that $\mathcal{F}$ is differentiable in $U^{(n)}$. Note that in (4.5) (and elsewhere), it is tacitly understood that only points $\mathbf{u} + \mathbf{h}$ that belong to $U^{(n)}$ should be considered [Cia13, p. 452]; this condition will always hold if $\|\mathbf{h}\|$ is small enough since $U^{(n)}$ is open.

Proposition 4.1.5 states that any affine operator is differentiable and has a constant derivative. We clarify that by a constant derivative, we mean a derivative that does not depend on the point $\mathbf{u}$ at which we differentiate the operator. The derivative is still allowed to depend on the independent variable x.

Proposition 4.1.5. *Let $V^{(n)}$ and $W^{(m)}$ be normed vector spaces, let $U^{(n)}$ be an open subset of $V^{(n)}$, and let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be an affine operator. Then $\mathcal{F}$ is differentiable in $U^{(n)}$, and there exists an $\mathcal{A} \in L(V^{(n)}, W^{(m)})$ such that*

$$\mathcal{F}'(\mathbf{u}) = \mathcal{A} \quad \textit{for all } \mathbf{u} \in U^{(n)}.$$

Proof. Since $\mathcal{F}$ is affine, we know from Definition 4.1.2 that there exist an operator $\mathcal{A} \in L(V^{(n)}, W^{(m)})$ and a $\mathbf{b} \in W^{(m)}$ such that

$$\mathcal{F}(u) = \mathcal{A}\mathbf{u} + \mathbf{b}$$

for all $\mathbf{u} \in U^{(n)}$. We then have for any $\mathbf{v} \in V^{(n)}$ that

$$\begin{aligned}\mathcal{F}(\mathbf{u}+\mathbf{h}) - \mathcal{F}(\mathbf{u}) - \mathcal{A}\mathbf{v} &= [\mathcal{A}(\mathbf{u}+\mathbf{v}) + \mathbf{b}] - [\mathcal{A}\mathbf{u} + \mathbf{b}] - \mathcal{A}\mathbf{v} \\ &= \mathcal{A}\mathbf{u} + \mathcal{A}\mathbf{h} - \mathcal{A}\mathbf{u} - \mathcal{A}\mathbf{h} = \mathbf{0},\end{aligned}$$

where $\mathbf{0}$ is the zero element of $W^{(m)}$. Comparing with (4.5), we see that $\mathcal{F}$ is differentiable at $\mathbf{u}$, with $\mathcal{F}'(\mathbf{u}) = \mathcal{A}$. But $\mathbf{u}$ is an arbitrary point in $U^{(n)}$, so $\mathcal{F}$ is differentiable in $U^{(n)}$, with a derivative equal to $\mathcal{A}$ everywhere. □

The converse of Proposition 4.1.5 also holds. This is stated in Theorems 4.1.6 and 4.1.7. The first theorem states that if an operator has a constant derivative, it is affine. The second states that if an operator has a constant partial derivative with respect to an argument, it is affine in that argument. These are the properties that we will detect computationally.

The proofs of Theorems 4.1.6 and 4.1.7 rely on a few standard definitions and results from topology, which can be found for example in [Cho66, Sut09, RF10, Cia13].

- In a topological space X, a *neighbourhood* of a point $x \in X$ is a subset of X that contains an open set containing x.
- Let E be a subset of a topological space X. A point $x \in X$ is a *point of closure* of E if every neighbourhood of x contains a point in E.
- Let E be a subset of a topological space X. Then E is *closed* if it contains all its points of closure.
- A subset E of a metric space X is closed if and only if

$$x_n \in E,\, n \geq 0, \text{ and } x_n \to x \in X \text{ as } n \to \infty \text{ implies } x \in E.$$

- A topological space X is said to be *connected* if there does not exist any partition of X into two disjoint nonempty closed sets. Equivalently, X is connected if the only nonempty subset of X which is both open and closed relative to X is X itself.

We remark that Theorem 4.1.6 is given by [Cia13, Thm. 7.2-4]. The proof in [Cia13] and our proof of Theorem 4.1.7 essentially follow the same steps, relying on the properties of connected topological spaces. The main difference is that [Cia13] makes use of the mean value theorem in normed vector spaces to show openness, while we make use of the dual space $(W^{(m)})^*$ (which requires fewer preliminary results).

Theorem 4.1.6. *[Cia13, Thm. 7.2-4] Let $V^{(n)}$ and $W^{(m)}$ be normed vector spaces, let $U^{(n)}$ be a connected open subset of $V^{(n)}$, and let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be a mapping differentiable in $U^{(n)}$. Assume that there exists $\mathcal{A} \in L(V^{(n)}, W^{(m)})$ such that*

$$\mathcal{F}'(\mathbf{u}) = \mathcal{A} \quad \textit{for all } \mathbf{u} \in U^{(n)}.$$

Then there exists $\mathbf{b} \in W^{(m)}$ such that

$$\mathcal{F}(\mathbf{u}) = \mathcal{A}\mathbf{u} + \mathbf{b} \quad \textit{for all } \mathbf{u} \in U^{(n)}.$$

Theorem 4.1.6 states that an operator whose derivative is constant, and whose domain is open and connected, is affine. What happens if instead, only a particular partial derivative of the operator is constant? The next theorem asserts that if the jth partial derivative of an operator $\mathcal{F}$ is constant, $\mathcal{F}$ will be affine in the jth argument. Before presenting the theorem, we make two further remarks on the mathematical background.

- If an operator $\mathcal{F}$ is differentiable, all partial derivatives of $\mathcal{F}$ exist [AH05, Prop. 5.3.14].
- Let $U^{(n)} = U_1 \times \cdots \times U_n$ be a product space. Then $U^{(n)}$ is connected iff every U_i, $1 \leq i \leq n$, is connected [Sut09, Thm. 12.18].

For the following theorem, we use the notation

$$U^- := U_1 \times \cdots \times U_{j-1}, \qquad U^+ := U_{j+1} \times \cdots \times U_n,$$
$$V^- := V_1 \times \cdots \times V_{j-1}, \qquad V^+ := V_{j+1} \times \cdots \times V_n.$$

Theorem 4.1.7. *Let $V^{(n)}$ and $W^{(m)}$ be normed vector spaces, let $U^{(n)}$ be a connected open subset of $V^{(n)}$, and let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be a mapping differentiable in $U^{(n)}$. Assume that there exists $\mathcal{A}_j \in L(V_j, W^{(m)})$ such that*

$$\frac{\partial \mathcal{F}}{\partial u_j}(\mathbf{u}) = \mathcal{A}_j \quad \textit{for all } \mathbf{u} \in U^{(n)}.$$

Then there exists a vector $\mathbf{b}_j \in W^{(m)}$ *and a differentiable mapping* $\mathcal{G}_j : (U^- \times U^+) \subset (V^- \times V^+) \to W^{(m)}$ *such that for all* $\mathbf{u} \in U^{(n)}$,

$$\mathcal{F}(\mathbf{u}) = \mathcal{G}_j(u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_n) + \mathcal{A}_j u_j + \mathbf{b}_j.$$

Proof. Let points

$$\mathbf{u}^- = (u_1, \ldots, u_{j-1}) \in U^-, \qquad \mathbf{u}^+ = (u_{j+1}, \ldots, u_n) \in U^+$$

be given. With $u_j \in U_j$, we write

$$\mathcal{F}(\mathbf{u}^-; u_j; \mathbf{u}^+) = \mathcal{F}(u_1, \ldots, u_{j-1}, u_j, u_{j+1}, \ldots, u_n). \tag{4.6}$$

Fix some $u_j^0 \in U_j$, and let $\mathcal{G}_j : (U^- \times U_j \times U^+) \subset (V^- \times V_j \times V^+) \to W^{(m)}$ be the operator defined by

$$\mathcal{G}_j(\mathbf{u}^-; u_j; \mathbf{u}^+) = \mathcal{F}(\mathbf{u}^-; u_j; \mathbf{u}^+) - \mathcal{A}_j u_j - \mathbf{f}_0, \tag{4.7}$$

where $\mathbf{f}_0 := \mathcal{F}(\mathbf{u}^-; u_j^0; \mathbf{u}^+) \in W^{(m)}$. Then

$$\mathcal{G}_j(\mathbf{u}^-; u_j^0; \mathbf{u}^+) = -\mathcal{A}u_j^0 =: \mathbf{g}_j^0,$$

with $\mathbf{g}_j^0 \in W^{(m)}$.

By assumption, $\mathcal{F}$ is differentiable. Therefore, $\mathcal{G}_j$ is also differentiable, and its jth partial derivative exists. We observe that $\mathcal{G}_j$ has as its jth partial derivative, $\dfrac{\partial \mathcal{G}_j}{\partial u_j}$, the zero operator $\mathcal{Z}$ everywhere in $U^{(n)}$. Since $\mathcal{G}_j$ is differentiable in $U^{(n)}$, it is continuous on $U^{(n)}$, cf. Section 2.2.4.

Now define the set $\Upsilon_j \subset U_j$ by $\Upsilon_j = \left\{u_j \in U_j \,|\, \mathcal{G}_j(\mathbf{u}^-; u_j^0; \mathbf{u}^+) = \mathbf{g}_j^0\right\}$. Clearly, Υ_j is non-empty, since $u_j^0 \in \Upsilon_j$. We want to show that Υ_j is also both open and closed with respect to U_j.

To show that Υ_j is closed with respect to U_j, let a sequence $\{v_m\} \in \Upsilon_j$ be given, such that $v_m \to v^* \in U_j$ as $m \to \infty$. Since $\mathcal{G}_j$ is continuous, we have that

$$\mathcal{G}_j(\mathbf{u}^-; v^*; \mathbf{u}^+) = \lim_{m\to\infty} \mathcal{G}_j(\mathbf{u}^-; v_m; \mathbf{u}^+) = \mathbf{g}_j^0.$$

Hence, $v^* \in \Upsilon_j$, so Υ_j is closed with respect to U_j.

To show that Υ_j is open relative to U_j, we need to show that for any $\hat{v} \in \Upsilon_j$, there exists a $\delta > 0$ such that the open ball $B(\hat{v}, \delta) \cap U_j \subset \Upsilon_j$. Since U_j is open, we know that there must exist a δ such that $B(\hat{v}, \delta) \subset U_j$. Let $\overline{v} \in B(\hat{v}, \delta)$, let ω^* be a linear

functional in the dual space of $W^{(m)}$, $(W^{(m)})^*$, and define the function $\phi : [0,1] \to \mathbb{R}$ by

$$\phi(t) = \langle \omega^*,\, \mathcal{G}_j(\mathbf{u}^-;\, \hat{v} + t(\overline{v} - \hat{v});\, \mathbf{u}^+) - \mathcal{G}_j(\mathbf{u}^-;\, \hat{v};\, \mathbf{u}^+) \rangle.$$

Here, the bracket notation denotes the action of ω^* on $W^{(m)}$,

$$\langle \omega^*,\, \mathbf{w} \rangle := \omega^*(\mathbf{w}), \qquad \omega^* \in (W^{(m)})^*, \quad \mathbf{w} \in W^{(m)}.$$

We see that $\phi(0) = 0$. Furthermore, since $\dfrac{\partial \mathcal{G}_j}{\partial u_j}$ is equal to $\mathcal{Z}$ everywhere on $U^{(n)}$, we have that

$$\phi'(t) = 0 \qquad \forall t \in [0,1].$$

Thus, we have $\phi(t) = 0$ for all $t \in [0,1]$, so in particular

$$0 = \phi(1) = \langle \omega^*,\, \mathcal{G}_j(\mathbf{u}^-;\, \overline{v};\, \mathbf{u}^+) - \mathcal{G}_j(\mathbf{u}^-;\, \hat{v};\, \mathbf{u}^+) \rangle. \tag{4.8}$$

But ω^* is an arbitrary member of $(W^{(m)})^*$, so (4.8) holds for every $\omega^* \in (W^{(m)})^*$. As asserted by [Cia13, Thm. 5.9.5] (a corollary of the Hahn-Banach theorem in normed spaces), this implies that

$$\mathcal{G}_j(\mathbf{u}^-;\, \overline{v};\, \mathbf{u}^+) - \mathcal{G}(\mathbf{u}^-;\, \hat{v};\, \mathbf{u}^+) = \mathbf{0}.$$

Therefore, $\overline{v} \in \Upsilon_j$, so $B(\hat{v}, \delta) \subset \Upsilon_j$, and Υ_j is open.

We have now shown that Υ_j is non-empty and both open and closed with respect to U_j. But the only subset of a connected set which has these three properties is the set itself, so $\Upsilon_j = U_j$. Therefore, we must have $\mathcal{G}(\mathbf{u}^-;\, u_j;\, \mathbf{u}^+) = \mathbf{g}_j^0$ for all $u_j \in U_j$, so $\mathcal{G}_j$ is constant over U_j, that is $\mathcal{G}_j(\mathbf{u}^-;\, u_j;\, \mathbf{u}^+) \equiv \mathcal{G}_j(\mathbf{u}^-;\, \mathbf{u}^+)$.

From 4.6 and 4.7, we see that

$$\mathcal{F}(u_1,\, \ldots,\, u_{j-1},\, v,\, u_{j+1},\, \ldots,\, u_n) = \mathcal{G}_j(\mathbf{u}^-;\, \mathbf{u}^+) + \mathcal{A}_j v + \mathbf{f}_0.$$

□

Observe that in order to be able to conclude that an operator is affine, we need to consider the derivatives over all variables as a single entity. An example of an operator with arity $n = 2$ demonstrates this. The operator $\mathcal{F}(u_1, u_2) = u_1 u_2$ has a derivative with respect to u_1 that is independent of u_1, and similarly for u_2, but the derivative with respect to the pair of variables is not independent of the pair. In other words, the derivative of $\mathcal{F}$ with respect to u_1 changes as u_2 is perturbed. Theorems 4.1.6 and 4.1.7 are consistent with the fact that $\mathcal{F}$ is not a linear operator.

4.1.3 Limitations of automatic linearity detection

At this point we need to clarify realistic expectations for automatic linearity detection. Proposition 4.1.5 and Theorem 4.1.6 assert that an operator is affine if and only if it has a constant derivative. However, once we implement linearity detection on a computer, a "false negative" result, in which a linear operator is mistakenly not identified as linear, is all but inevitable. While the operator

$$\mathfrak{f}(u) = [u + \sin(u)] - [\sin(u)] \tag{4.9}$$

is clearly linear, when it is evaluated numerically on a computer, we cannot expect robust results for linearity detection. The impact of such false negatives, besides them being a rather trivial subset of problems, is negligible. This is because in the larger context, a nonlinear algorithm will be applied to the problem and arrive at correct final results nevertheless. On the other hand, a "false positive" result, in which a nonlinear problem is identified as linear, will likely have a disastrous effect on subsequent calculations. One should thus test for the conditions of Theorems 4.1.6 and 4.1.7, which are sufficient conditions for an operator to be affine or affine in a particular argument, respectively.

The potential for false positive results argues against linearity detection through the simple expedient of testing (4.1) for a few random inputs. The possibility of unfortunate choices for the inputs makes false positives impossible to rule out in theory, and difficult to avoid robustly in practice. We mention, however, that such sampling techniques have successfully been applied for convexity detection in MProbe [Chi02], a tool for analysing mathematical programs written in modelling languages such as AMPL.

4.1.4 Boolean algebra notation

The core of our algorithms for linearity detection is to operate on Boolean vectors to propagate the necessary information through a computation. We now describe the notation used to work with this information. For further details on Boolean algebra, we refer to [Whi95].

Boolean vectors are elements of $\mathbb{B}^n$, the space of vectors with n elements which are either 0 or 1. We represent members of $\mathbb{B}^n$ as row vectors. By r_i, we mean the ith element of the vector $r \in \mathbb{B}^n$. We use the following standard Boolean algebra notation for vectors r, s, $t \in \mathbb{B}^n$:

$\wedge$ Logical AND. If $t = r \wedge s$, then $t_i = 1$ iff $r_i = 1$ and $s_i = 1$.

Identity laws	$1 \wedge r = r$	$0 \vee r = r$
Null laws	$0 \wedge r = \mathbf{0}$	$1 \vee r = \mathbf{1}$
Idempotent laws	$r \wedge r = r$	$r \vee r = r$
Commutative laws	$r \wedge s = s \wedge r$	$r \vee s = s \vee r$
Associative laws	$(r \wedge s) \wedge t = r \wedge (s \wedge t)$	$(r \vee s) \vee t = r \vee (s \vee t)$
Distributive laws	$r \vee (s \wedge t) = (r \vee s) \wedge (r \vee t)$	$r \wedge (s \vee t) = (r \wedge s) \vee (r \wedge t)$
Absorption laws	$r \wedge (r \vee s) = r$	$r \vee (r \wedge s) = r$

Table 4.1: List of Boolean identities [Whi95]. Here, r, s and t denote Boolean vectors, and the $\mathbf{0}$ and $\mathbf{1}$ symbols on the right-hand side of the null laws equations denote Boolean vectors with all elements equal to 0 or 1, respectively.

$\vee$ Logical OR. If $t = r \vee s$, then $t_i = 1$ iff $r_i = 1$ or $s_i = 1$ (or both).

$\neg$ Logical NOT. If $t = \neg r$, then $t_i = 1$ iff $r_i = 0$.

We also allow operations involving a scalar Boolean $\rho \in \mathbb{B}$ and a vector $s \in \mathbb{B}^n$. These represent operators that map $\mathbb{B} \times \mathbb{B}^n$ to $\mathbb{B}^n$, with results given as follows:

$\wedge$ If $t = \rho \wedge s$, then $t_i = s_i$ if $\rho = 1$, and $t_i = 0$ if $\rho = 0$.

$\vee$ If $t = \rho \vee s$, then $t_i = 1$ if $\rho = 1$, and $t_i = s_i$ if $\rho = 0$.

In Table 4.1, we list some Boolean algebra identities that will be useful in Section 4.1.5.

The logical function $\forall(\cdot) : \mathbb{B}^n \to \mathbb{B}$ will also be useful in Section 4.1.5. It is defined as follows.

Definition 4.1.8. *Let $r \in \mathbb{B}^n$. The function $\forall(\cdot)$ is defined by $\forall(r) = 1$ iff $r_i = 1$ for all $1 \leq i \leq n$.*

4.1.5 Propagation of information for linearity detection

Let $U^{(n)} = U_1 \times \cdots \times U_n$, $V^{(n)} = V_1 \times \cdots \times V_n$ and $W^{(m)} = W_1 \times \cdots \times W_m$, where each V_i is a normed vector space, each U_i is an open and connected subset of V_i, and each W_i is a normed vector space as well. The normed vector spaces we consider consist of functions of one variable, denoted by x. The functions are assumed to be sufficiently smooth so that operations such as differentiation with respect to x are well defined. To simplify our notation, without loss of generality, we let $U_1 = \ldots = U_n =: U$, $V_1 = \ldots = V_n =: V$, and $W_1 = \ldots = W_m =: W$.

Let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be a twice continuously differentiable mapping of arity n, consisting of m components $\mathfrak{f}_1, \ldots, \mathfrak{f}_m$, so that each $\mathfrak{f}_i : U^{(n)} \subset V^{(n)} \to W$. As (4.9) makes clear, the computational detection of linearity (as opposed to linearity itself) depends not just on the operator $\mathcal{F}(u_1, \ldots, u_n)$, but also the path taken to evaluate it. It is thus closely related to the *evaluation trace* discussed in Section 1.3.

In the evaluation trace, we associate with each elementary operator an intermediate variable. We quickly review our notation. We assume that when evaluating $\mathcal{F}$, we evaluate its components $\mathfrak{f}_1, \ldots, \mathfrak{f}_m$ one by one. We denote the kth elementary operator in the sequence by φ_k, and the result of evaluating it by the intermediate variable v_k. We assume that every elementary operator is twice continuously differentiable and that its arguments are sufficiently smooth, so that every partial derivative of every intermediate variable exists, and operator composition and addition of partial derivatives of intermediate variables are well defined.

At the start of a computation, for $k \leq 0$, we initialise the intermediate variables v_k to be the independent variable and the input variables, so that

$$v_{-n} = x \tag{4.10}$$

and

$$v_k = u_{n+k}, \qquad k = -n+1, -n+2, \ldots, 0. \tag{4.11}$$

For instance, if $\mathcal{F}$ is a ternary operator, we have $v_{-3} = x$, $v_{-2} = u_1$, $v_{-1} = u_2$ and $v_0 = u_3$.

For $k > 0$, we have

$$v_k = \varphi_k(v_i), \quad \text{or} \quad v_k = \varphi_k(v_i, v_j),$$

depending on whether φ_k is unary or binary. Here, v_i and v_j are intermediate variables, with $i, j < k$.

In the evaluation trace, each elementary operator φ_k maps from its domain $\mathfrak{U}_k$, assumed to be an open and connected subset of a normed vector space $\mathfrak{V}_k$, to a normed vector space $\mathfrak{W}_k$. As described below, the precise nature of these spaces is not important for computational linearity detection. Rather, the crucial observation is that we need to determine whether an intermediate variable v_k has zero partial derivatives with respect to the inputs. To simplify our notation, we let $\mathcal{Z}$ denote zero operators throughout this chapter, tacitly understood to map from the space of the inputs to the appropriate function spaces.

For computing derivatives in standard forward mode AD, we associate with each intermediate variable v_k the tangent variable $\dot{v}_k$. Assuming that we are computing

derivatives with respect to all inputs at the same time in "vector-mode" AD, $\dot{v}_k$ will be a "block-operator" of the form

$$\dot{v}_k = \left(\frac{\partial v_k}{\partial u_1}, \ldots, \frac{\partial v_k}{\partial u_n} \right).$$

We observe that the intermediate variable $v_{-n} = x$ has as its derivative the block operator

$$\begin{bmatrix} \mathcal{Z} & \cdots & \mathcal{Z} \end{bmatrix}. \tag{4.12}$$

For $-n < k \leq 0$, we have from (4.11) that v_k has as its Fréchet derivative the block operator

$$\begin{bmatrix} \mathcal{Z} & \cdots & \mathcal{Z} & \mathcal{I} & \mathcal{Z} & \cdots & \mathcal{Z} \end{bmatrix}, \tag{4.13}$$

where the identity operator $\mathcal{I}$ is in the $(n+k)$th position of the block operator. The variables $\dot{v}_k$ are seeded accordingly, as described in Section 1.3.

For $k > 0$, we apply the chain rule to compute the entries of $\dot{v}_k$ at the same time as v_k is computed. The values of the entries of $\dot{v}_k$ are obtained by using only the values of previously computed intermediate and tangent variables.

For linearity detection, we introduce similar associated variables that allow us to carry the necessary information through the evaluation trace. As motivation, we note the following. Suppose that a component $\mathfrak{f}_i$ of $\mathcal{F}$ has a constant pth partial derivative, that is, for any $\mathbf{u} \in U^{(n)}$,

$$\frac{\partial \mathfrak{f}_i}{\partial u_p}(\mathbf{u}) = \mathcal{A}_{i,p}, \tag{4.14}$$

with $\mathcal{A}_{i,p} \in L(V, W)$. Then, according to Theorem 4.1.7, $\mathfrak{f}_i$ is affine in the pth argument.

Let us now consider higher order derivatives of $\mathfrak{f}_i$. Following [Cia13, Ch. 7.8], we note that since $\mathcal{F}$ and hence $\mathfrak{f}_i$ are assumed to be sufficiently differentiable, the mapping

$$\frac{\partial \mathfrak{f}_i}{\partial u_p} : \mathbf{u} \in U^{(n)} \subset V^{(n)} \mapsto \frac{\partial \mathfrak{f}_i}{\partial u_p}(\mathbf{u}) \in L(V, W)$$

is well defined and again differentiable at $\mathbf{u}$. The block operator

$$\mathcal{H}^{\mathfrak{f}_i}(\mathbf{u}) = \begin{pmatrix} \frac{\partial^2 \mathfrak{f}_i}{\partial u_1^2}(\mathbf{u}) & \cdots & \frac{\partial^2 \mathfrak{f}_i}{\partial u_1 u_n}(\mathbf{u}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathfrak{f}_i}{\partial u_n u_1}(\mathbf{u}) & \cdots & \frac{\partial^2 \mathfrak{f}_i}{\partial u_n^2}(\mathbf{u}) \end{pmatrix} \tag{4.15}$$

will have as its entries elements from the space of bilinear operators $L(V \times V, W)$, see (2.16). We call $\mathcal{H}^{\mathfrak{f}_i}$ the *Hessian* of $\mathfrak{f}_i$, and let $\mathcal{H}^{\mathfrak{f}_i}_{q,p}$ denote the (q,p)th element of $\mathcal{H}^{\mathfrak{f}_i}$.

Let $\bar{\mathcal{Z}}$ denote the bilinear zero operator that acts on elements of $V \times V$. Analogously to above, $\bar{\mathcal{Z}}$ will be assumed to map to the appropriate function space wherever used. From (4.14), we observe that if $\mathfrak{f}_i$ has a constant pth partial derivative, then for all $\mathbf{u} \in U^{(n)}$,

$$\frac{\partial^2 \mathfrak{f}_i}{\partial u_q \partial u_p}(\mathbf{u}) = \bar{\mathcal{Z}}, \qquad q = 1, \ldots, n.$$

Conversely, if $\mathcal{H}^{\mathfrak{f}_i}_{q,p}(\mathbf{u}) = \bar{\mathcal{Z}}$ for all $\mathbf{u} \in U^{(n)}$ and $q = 1, \ldots, n$, $\mathfrak{f}_i$ has a has a constant pth partial derivative (this follows from Theorem 4.1.7). Since it is a sufficient condition for an operator to be affine in the pth argument that its pth partial derivative is constant, *it is a sufficient condition for $\mathfrak{f}_i$ to be affine in the pth argument that the pth column of its Hessian is all zero.*

Another important case we consider is when $\mathfrak{f}_i$ does not depend on the input u_p. In other words, if we have a point $\mathbf{u} \in U^{(n)}$ and perturb the pth component of $\mathbf{u}$ but keep the other inputs fixed, $\mathfrak{f}_i(\mathbf{u})$ is constant. In this case, we have for the pth partial derivative of $\mathfrak{f}_i$ that

$$\frac{\partial \mathfrak{f}_i}{\partial u_p}(\mathbf{u}) = \mathcal{Z}. \tag{4.16}$$

We say that $\mathfrak{f}_i$ is *independent of u_p*.

We now give definitions for vectors that encode sufficient conditions for partial derivatives of intermediate variables to be of the special forms (4.14) or (4.16). These vectors encode the structural sparsity patterns of the Fréchet derivatives and Hessians of intermediate variables. They are associated with every intermediate variable v_k in the evaluation trace — the result of evaluating the elementary operator φ_k. Each elementary operator is assumed to be sufficiently elementary that its Fréchet derivatives with respect to each argument are known (or known not to exist), which is a requirement for computing AD information in any case.

Definition 4.1.9. Zero derivative vector. *Let φ_k be the kth elementary operator in the evaluation trace of an operator $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$, and let v_k be the intermediate variable that is the result of evaluating φ_k. The* zero derivative vector *$z^{v_k} \in \mathbb{B}^n$ is associated with v_k. It is defined by the following conditions.*

For $k = -n$, z^{v_k} is initialised as

$$z^{v_k} = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}. \tag{4.17}$$

For $-n < k \leq 0$, z^{v_k} is initialised as

$$z^{v_k} = \begin{bmatrix} 1 & \cdots & 1 & 0 & 1 & \cdots & 1 \end{bmatrix}, \tag{4.18}$$

where the 0 entry is in the $(n+k)$th position of the vector.

For $k > 0$, z^{v_k} is given by

$$z^{v_k} = \zeta_k(z^{v_i}) \quad or \quad z^{v_k} = \zeta_k(z^{v_i}, z^{v_j}),$$

depending on whether φ_k is unary or binary. Here, ζ_k is the zero derivative information propagation operator *associated with φ_k, defined via its action listed in Table* 4.2*, and v_i and possibly v_j are the intermediate variables that v_k is computed from.*

Definition 4.1.10. Hessian sparsity vector. *Let φ_k be the kth elementary operator in the evaluation trace of an operator $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$, and let v_k be the intermediate variable that is the result of evaluating φ_k. The* Hessian sparsity vector *$h^{v_k} \in \mathbb{B}^n$ is associated with v_k. It is defined by the following conditions.*

For $k \leq 0$, h^{v_k} is initialised as

$$h^{v_k} = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}. \tag{4.19}$$

For $k > 0$, h^{v_k} is given by

$$h^{v_k} = \eta_k(z^{v_i}, h^{v_i}) \quad or \quad h^{v_k} = \eta_k(z^{v_i}, z^{v_j}, h^{v_i}, h^{v_j}),$$

depending on whether φ_k is unary or binary. Here, η_k is the Hessian sparsity information propagation operator *associated with φ_k, defined via its action listed in Table* 4.2*, v_i and possibly v_j are the intermediate variables that v_k is computed from, and z^{v_i} and possibly z^{v_j} are the associated zero derivative vectors.*

We note that if we are only interested in finding out whether a given operator is affine or not, it is possible to simplify the approach for linearity detection described in this chapter. Rather than working with the n-vectors z and h, one could simply use scalar Boolean variables encoding overall linearity information. However, we believe that being able to determine what parts of coupled ODEs are affine with respect to what inputs is valuable — an idea that has been exploited at least as long ago as the VARPRO code by Golub and Pereyra [GP73]. In particular, it has the potential to be useful in the analysis of block operators — an upcoming research topic for Chebfun. Thus, the approach to linearity detection described here is slightly more complicated than might be required for some purposes.

We now prove that our definitions of z, h, ζ and η provide conditions which, when satisfied:

Operator	Values of z^{v_k} and h^{v_k}	Comment
$v_k = \pm v_i$ $v_k = \frac{\mathrm{d}}{\mathrm{d}x} v_i$ $v_k = \int v_i \mathrm{d}x$	$z^{v_k} = z^{v_i}$ $h^{v_k} = h^{v_i}$	The unary + and − operators are linear. Differentiation and integration are also linear operations. They do not have any effect on the zero patterns of the derivative or the Hessian.
$v_k = \sin(v_i)$	$z^{v_k} = z^{v_i}$ $h^{v_k} = z^{v_i}$	If the pth partial derivative of v_i is equal to $\mathcal{Z}$, the pth column of $\mathcal{H}^{v_k}$ will be all zero. The same formulae apply for other univariate functions, such as cos, exp, etc.
$v_k = v_i \pm v_j$	$z^{v_k} = z^{v_i} \wedge z^{v_j}$ $h^{v_k} = h^{v_i} \wedge h^{v_j}$	If the pth columns of both $\mathcal{H}^{v_i}$ and $\mathcal{H}^{v_j}$ are all zero, the pth column of $\mathcal{H}^{v_k}$ will also be all zero.
$v_k = v_i * v_j$	$z^{v_k} = z^{v_i} \wedge z^{v_j}$ $h^{v_k} = h^{v_i} \wedge h^{v_j} \wedge$ $(z^{v_i} \vee \forall(z^{v_j})) \wedge$ $(\forall(z^{v_i}) \vee z^{v_j})$	If both $\mathcal{H}^{v_i}$ and $\mathcal{H}^{v_j}$ have the same all zero column, and furthermore, they both have a zero derivative, or one argument has derivative only consisting of zero blocks, $\mathcal{H}^{v_k}$ will have the same all zero column. This ensures that xu_1 is detected as linear, u_1u_1 as linear in u_2, while u_1u_2 is nonlinear in both variables.
$v_k = v_i / v_j$	$z^{v_k} = z^{v_i} \wedge z^{v_j}$ $h^{v_k} = h^{v_i} \wedge z^{v_j} \wedge$ $(z^{v_i} \vee \forall(z^{v_j}))$	The expression of h^{v_k} for the division operator is derived from the expressions for the unary inverse operator and the multiplication operator.
$v_k = v_i \,\hat{}\, v_j$	$z^{v_k} = z^{v_i} \wedge z^{v_j}$ $h^{v_k} = z^{v_i} \wedge z^{v_j}$	Here, observe that $h^{v_k} = z^{v_k}$. If v_k does not depend on a variable, the corresponding column in $\mathcal{H}^{v_k}$ will be all zero.

Table 4.2: The actions of the zero derivative and Hessian sparsity information propagation operators. This table defines the operators ζ_k and η_k via

$$z^{v_k} = \zeta_k(z^{v_i}), \qquad h^{v_k} = \eta_k(z^{v_i}, h^{v_i})$$

if φ_k is unary, and

$$z^{v_k} = \zeta_k(z^{v_i}, z^{v_j}), \qquad h^{v_k} = \eta_k(z^{v_i}, z^{v_j}, h^{v_i}, h^{v_j})$$

if φ_k is binary.

- Are sufficient to guarantee that an intermediate variable has a zero partial derivative with respect to a particular argument (Theorem 4.1.11);
- Are sufficient to guarantee that a particular column of the Hessian of an intermediate variable consists only of zero operators (Theorem 4.1.12).

The proofs proceed by induction, starting from the initialisations of the vectors z and h. We recall that if r is a Boolean vector, then r_i refers to the ith element of r, and that $\mathcal{M}^{g(x)}$ denotes the multiplication operator

$$\mathcal{M}^{g(x)} : f(x) \mapsto g(x)f(x).$$

Theorem 4.1.11. Justification of the zero derivative vector. *Let v_k be an intermediate variable in the evaluation trace of an operator $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$. Let z^{v_k} and ζ_k be as defined in Definition* 4.1.9. *Then,*

$$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

Proof. If $k \leq 0$, we are in the initialisation phase of the evaluation trace. If $k = -n$, the initalisation expression will be of the form

$$v_k = x.$$

From (4.12), v_k has as its Fréchet derivative the block operator

$$\begin{bmatrix} \mathcal{Z} & \cdots & \mathcal{Z} \end{bmatrix}.$$

From Definition 4.1.9, in particular (4.17), we have that

$$z^{v_k} = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}.$$

We thus have for $k = -n$ that

$$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}. \tag{4.20}$$

If $-n < k \leq 0$, all expressions will be of the form

$$v_k = u_{n+k}.$$

From (4.13), v_k has as its Fréchet derivative the block operator

$$\begin{bmatrix} \mathcal{Z} & \cdots & \mathcal{Z} & \mathcal{I} & \mathcal{Z} & \cdots & \mathcal{Z} \end{bmatrix},$$

where the identity operator $\mathcal{I}$ is in the $(n+k)$th position of the block operator. From Definition 4.1.9, in particular (4.18), we have that

$$z^{v_k} = \begin{bmatrix} 1 & \cdots & 1 & 0 & 1 & \cdots & 1 \end{bmatrix},$$

where the 0 entry is in the $(n+k)$th position of the vector. We thus have for $-n < k \leq 0$ that

$$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}. \tag{4.21}$$

Combining (4.20) and (4.21), we have the base case for our induction.

Now consider $k > 0$. Depending on whether φ_k is unary or binary, we have

$$v_k = \varphi_k(v_i), \quad \text{or} \quad v_k = \varphi_k(v_i, v_j),$$

where v_i and v_j are intermediate variables, with $i, j < k$.

Our induction hypothesis is that for every previously computed intermediate variable v_i, $i < k$, the condition $z_p^{v_i} = 1$ is a sufficient condition for v_i to have a zero pth partial derivative, that is

$$z_p^{v_i} = 1 \Rightarrow \frac{\partial v_i}{\partial u_p} = \mathcal{Z}. \tag{4.22}$$

We observe that from Definition 4.1.9 and Table 4.2, we have for every elementary operator φ_k that

$$z^{v_k} = \zeta_k(z^{v_i}) = z^{v_i}, \quad \text{or} \quad z^{v_k} = \zeta_k(z^{v_i}, z^{v_j}) = z^{v_i} \wedge z^{v_j},$$

depending on whether φ_k is unary or binary. Therefore, we have that

$$z_p^{v_k} = z_p^{v_i} \quad \text{or} \quad z_p^{v_k} = z_p^{v_i} \wedge z_p^{v_i} \tag{4.23}$$

for each $1 \leq p \leq n$. Combining (4.23) with the induction hypothesis (4.22), we see that

$$z_p^{v_k} = 1 \Rightarrow z_p^{v_i} = 1 \Rightarrow \frac{\partial v_i}{\partial u_p} = \mathcal{Z} \tag{4.24}$$

or

$$z_p^{v_k} = 1 \Rightarrow [z_p^{v_i} = 1] \wedge [z_p^{v_j} = 1] \Rightarrow \frac{\partial v_i}{\partial u_p} = \mathcal{Z} \text{ and } \frac{\partial v_j}{\partial u_p} = \mathcal{Z}, \tag{4.25}$$

depending on whether φ_k is unary or binary. We now consider each category of elementary operators φ_k separately.

1. **Unary + and −.**

   Here we have that $v_k = \pm v_i$ for some $i < k$. Therefore,

   $$\frac{\partial v_k}{\partial u_p} = \pm \frac{\partial v_i}{\partial u_p}. \tag{4.26}$$

   Combining (4.24) and (4.26), we have that

   $$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

2. **Differentiation and integration operators.**

   As an example, consider first order differentiation (higher order differentiation and integration work analogously). Here we have that $v_k = \frac{\mathrm{d}}{\mathrm{d}x} v_i$ for some $i < k$. Therefore,

   $$\frac{\partial v_k}{\partial u_p} = \mathcal{D} \frac{\partial v_i}{\partial u_p}, \tag{4.27}$$

   where $\mathcal{D}$ is the differentiation operator on the interval that v_k is defined on. Combining (4.24) and (4.27), we have that

   $$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

3. **Unary elementary and special mathematical functions, such as sine.**

   Here we have that $v_k = \varphi_k(v_i)$ for the specific operator φ_k and some $i < k$. From the chain rule, we have that

   $$\frac{\partial v_k}{\partial u_p} = \frac{\partial v_k}{\partial v_i} \frac{\partial v_i}{\partial u_p}. \tag{4.28}$$

   Combining (4.24) and (4.28), we have that

   $$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

4. **Binary + and −.**

   Here we have that $v_k = v_i \pm v_j$ for some $i, j < k$. Therefore,

   $$\frac{\partial v_k}{\partial u_p} = \frac{\partial v_i}{\partial u_p} \pm \frac{\partial v_j}{\partial u_p}. \tag{4.29}$$

   Combining (4.25) and (4.29), we have that

   $$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

5. **The multiplication operator $*$.**

   Here we have that $v_k = v_i * v_j$ for some $i, j < k$. Therefore,

$$\frac{\partial v_k}{\partial u_p} = \mathcal{M}^{v_j}\frac{\partial v_i}{\partial u_p} + \mathcal{M}^{v_i}\frac{\partial v_j}{\partial u_p}. \tag{4.30}$$

   Combining (4.25) and (4.30), we have that

$$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

6. **The division operator /.**

   Here we have that $v_k = v_i/v_j$ for some $i, j < k$. Therefore,

$$\frac{\partial v_k}{\partial u_p} = \mathcal{M}^{1/v_j}\frac{\partial v_i}{\partial u_p} - \mathcal{M}^{v_i/v_j^2}\frac{\partial v_j}{\partial u_p}. \tag{4.31}$$

   Combining (4.25) and (4.31), we have that

$$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

7. **The power operator $\wedge$.**

   Here we have that $v_k = v_i \wedge v_j$ for some $i, j < k$. Therefore,

$$\frac{\partial v_k}{\partial u_p} = \mathcal{M}^{v_j v_i \wedge (v_j - 1)}\frac{\partial v_i}{\partial u_p} + \mathcal{M}^{v_i \wedge (v_j)\log(v_i)}\frac{\partial v_j}{\partial u_p}. \tag{4.32}$$

   Combining (4.25) and (4.32), we have that

$$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

We have thus shown that for all k, and every elementary operator φ_k we consider,

$$z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z}.$$

□

For the following theorem, we recall that $\mathcal{H}^{v_k}_{q,p}$ denotes the (q,p)th element of the Hessian $\mathcal{H}^{v_k}$ (see (4.15)), and that $\bar{\mathcal{Z}}$ denotes the the bilinear zero operator that acts on elements of $V \times V$, tacitly understood to map to the appropriate function space wherever used.

Theorem 4.1.12. Justification of the Hessian sparsity vector. *Let v_k be an intermediate variable in the evaluation trace of an operator $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$. Let z^{v_k}, h^{v_k}, ζ_k and η_k be as defined in Definitions* 4.1.9 *and* 4.1.10*. Then,*

$$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}},$$

for all $1 \leq q \leq n$.

Proof. If $k \leq 0$, then we are in the initialisation phase of the evaluation trace. If $k = -n$, we have that $v_k = x$. If $-n < k \leq 0$, all expressions will be of the form

$$v_k = u_{n+k}.$$

If $k = -n$, v_k will have as its Fréchet derivative the block operator

$$\begin{bmatrix} \mathcal{Z} & \cdots & \mathcal{Z} \end{bmatrix}.$$

If $-n < k \leq 0$, v_k will have as its Fréchet derivative the block operator

$$\begin{bmatrix} \mathcal{Z} & \mathcal{Z} & \cdots & \mathcal{I} & \cdots & \mathcal{Z} \end{bmatrix} \qquad ((n+k)\text{th position}).$$

We thus have that for all $k \leq 0$, every column of the Hessian $\mathcal{H}^{v_k}$ will only consist of zero operators.

From Definition 4.1.9, in particular (4.19), we have that

$$h^{v_k} = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}.$$

Thus, for $k < 0$, we have that

$$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

This is the base case for our induction.

Now consider $k > 0$. Our induction hypothesis is that for every previously computed intermediate variable v_i, $i < k$, $h_p^{v_i} = 1$ is a sufficient condition for the pth column of the Hessian $\mathcal{H}^{v_i}$ to be all zero, that is

$$h_p^{v_i} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_i} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n. \tag{4.33}$$

We now consider each category of elementary operators φ_k separately.

1. **Unary + and −.**

   Here we have that $v_k = \pm v_i$ for some $i < k$. Therefore,

   $$\frac{\partial^2 v_k}{\partial u_q \partial u_p} = \pm \frac{\partial^2 v_i}{\partial u_q \partial u_p} = \pm \mathcal{H}_{q,p}^{v_i}. \tag{4.34}$$

   From Definition 4.1.10 and Table 4.2, we have for this φ_k that

   $$h^{v_k} = \eta_k(z^{v_i}, h^{v_i}) = h^{v_i}.$$

   Therefore, $h_p^{v_k} = h_p^{v_i}$, so from our induction hypothesis (4.33),

   $$h_p^{v_k} = 1 \Rightarrow h_p^{v_i} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_i} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n. \tag{4.35}$$

   Combining (4.34) and (4.35), we have that

   $$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

2. **Differentiation and integration operators.**

   As an example, consider first order differentiation (higher order differentiation and integration work analogously). Here we have that $v_k = \frac{\mathrm{d}}{\mathrm{d}x} v_i$ for some $i < k$. This is a linear operation, so

   $$\frac{\partial^2 v_k}{\partial u_q \partial u_p} = \mathcal{D} \frac{\partial^2 v_i}{\partial u_q \partial u_p} = \mathcal{D} \mathcal{H}_{q,p}^{v_i}, \tag{4.36}$$

   where $\mathcal{D}$ is the differentiation operator on the interval of v_k.

   From Definition 4.1.10 and Table 4.2, we have for this φ_k that

   $$h^{v_k} = \eta_k(z^{v_i}, h^{v_i}) = h^{v_i}.$$

   Therefore, $h_p^{v_k} = h_p^{v_i}$, so

   $$h_p^{v_k} = 1 \Rightarrow h_p^{v_i} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_i} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n. \tag{4.37}$$

   from (4.33). Combining (4.36) and (4.37), we have that

   $$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

3. **Unary elementary and special mathematical functions, such as sine.**

   Here, we have that $v_k = \varphi_k(v_i)$ for some $i < k$. We observe that if $\dfrac{\partial v_k}{\partial u_p} = \mathcal{Z}$, then $\dfrac{\partial^2 v_k}{\partial u_q \partial u_p} = \bar{\mathcal{Z}}$ for any q. In other words,

$$\frac{\partial v_k}{\partial u_p} = \mathcal{Z} \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n. \tag{4.38}$$

   From Definition 4.1.9 and Table 4.2, we have for this φ_k that

$$h^{v_k} = \eta_k(z^{v_i}, h^{v_i}) = z^{v_i}.$$

   But from Table 4.2, we furthermore have that $z^{v_k} = z^{v_i}$, so $h^{v_k} = z^{v_k}$. Therefore, $h_p^{v_k} = z_p^{v_k}$, so

$$h_p^{v_k} = 1 \Rightarrow z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z} \tag{4.39}$$

   from Theorem 4.1.11. Combining (4.38) and (4.39), we have that

$$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

4. **Binary $+$ and $-$.**

   Here we have that $v_k = v_i \pm v_j$ for some i, $j < k$. Therefore,

$$\frac{\partial^2 v_k}{\partial u_q \partial u_p} = \frac{\partial^2 v_i}{\partial u_q \partial u_p} \pm \frac{\partial^2 v_j}{\partial u_q \partial u_p} = \mathcal{H}_{q,p}^{v_i} \pm \mathcal{H}_{q,p}^{v_j}. \tag{4.40}$$

   From Definition 4.1.10 and Table 4.2, we have for this φ_k that

$$h^{v_k} = \eta_k(z^{v_i}, z^{v_j}, h^{v_i}, h^{v_j}) = h^{v_i} \wedge h^{v_j}.$$

   Therefore, $h_p^{v_k} = h_p^{v_i} \wedge h_p^{v_j}$, so from our induction hypothesis (4.33),

$$h_p^{v_k} = 1 \Rightarrow [h_p^{v_i} = 1] \wedge [h_p^{v_j} = 1] \Rightarrow \mathcal{H}_{q,p}^{v_i} = \bar{\mathcal{Z}} \text{ and } \mathcal{H}_{q,p}^{v_j} = \bar{\mathcal{Z}} \tag{4.41}$$

   for all $1 \leq q \leq n$. Combining (4.40) and (4.41), we have that

$$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

5. **The multiplication operator $*$.**

   Here we have that $v_k = \varphi_k(v_i, v_j) = v_i * v_j$ for some i, $j < k$. We have that

$$\frac{\partial v_k}{\partial u_p} = \mathcal{M}^{v_j} \frac{\partial v_i}{\partial u_p} + \mathcal{M}^{v_i} \frac{\partial v_j}{\partial u_p},$$

so the (q,p)th element of the Hessian $\mathcal{H}^{v_k}$ is the bilinear operator given by

$$\begin{aligned}\frac{\partial^2 v_k}{\partial u_q \partial u_p} &= \mathcal{M}^{v_j}\frac{\partial^2 v_i}{\partial u_q \partial u_p} + \frac{\partial v_j}{\partial u_q}\frac{\partial v_i}{\partial u_p} + \frac{\partial v_i}{\partial u_q}\frac{\partial v_j}{\partial u_p} + \mathcal{M}^{v_i}\frac{\partial^2 v_j}{\partial u_q \partial u_p} \\ &= \mathcal{M}^{v_j}\mathcal{H}^{v_i}_{q,p} + \frac{\partial v_j}{\partial u_q}\frac{\partial v_i}{\partial u_p} + \frac{\partial v_i}{\partial u_q}\frac{\partial v_j}{\partial u_p} + \mathcal{M}^{v_i}\mathcal{H}^{v_j}_{q,p}.\end{aligned}$$

From Definition 4.1.10 and Table 4.2, we have for this φ_k that

$$h^{v_k} = h^{v_i} \wedge h^{v_j} \wedge (z^{v_i} \vee \forall(z^{v_j})) \wedge (\forall(z^{v_i}) \vee z^{v_j}).$$

Therefore,

$$h_p^{v_k} = 1 \Rightarrow [h_p^{v_i} = 1] \wedge [h_p^{v_j} = 1] \wedge [(z_p^{v_i} \vee \forall(z^{v_j})) = 1] \wedge [(\forall(z^{v_i}) \vee z_p^{v_j}) = 1]. \quad (4.42)$$

In other words, if $h_p^{v_k} = 1$, all the Boolean factors on the right hand side of the $\Rightarrow$ sign are equal to 1. We now inspect each factor separately.

(a) First, consider $h_p^{v_i}$ From our induction hypothesis (4.33), we have that

$$h_p^{v_i} = 1 \Rightarrow \mathcal{H}^{v_i}_{q,p} = \bar{\mathcal{Z}} \quad (4.43)$$

for all $1 \leq q \leq n$.

(b) Similarly, for $h_p^{v_j}$, we have that

$$h_p^{v_j} = 1 \Rightarrow \mathcal{H}^{v_j}_{q,p} = \bar{\mathcal{Z}} \quad (4.44)$$

for all $1 \leq q \leq n$.

(c) Now consider the term $z_p^{v_i} \vee \forall(z^{v_j})$. We have that

$$(z_p^{v_i} \vee \forall(z^{v_j})) = 1 \Rightarrow [z_p^{v_i} = 1] \vee [\forall(z^{v_j}) = 1].$$

From the results of Theorem 4.1.11, we have that

$$z_p^{v_i} = 1 \Rightarrow \frac{\partial v_i}{\partial u_p} = \mathcal{Z},$$

and that

$$\forall(z^{v_j}) = 1 \Rightarrow \frac{\partial v_j}{\partial u_q} = \mathcal{Z}$$

for all $1 \leq q \leq n$. Thus, we have that

$$(z_p^{v_i} \vee \forall(z^{v_j})) = 1 \Rightarrow \frac{\partial v_j}{\partial u_q}\frac{\partial v_i}{\partial u_p} = \bar{\mathcal{Z}} \quad (4.45)$$

for all $1 \leq q \leq n$.

(d) Similarly, for $\forall(z^{v_i}) \vee z_p^{v_j}$, we obtain that

$$(\forall(z^{v_i}) \vee z_p^{v_j}) = 1 \Rightarrow \frac{\partial v_i}{\partial u_q}\frac{\partial v_j}{\partial u_p} = \bar{\mathcal{Z}} \tag{4.46}$$

for all $1 \leq q \leq n$.

Combining (4.42) and (4.43)–(4.46), we have that

$$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

6. **The division operator /.**

We observe that the operation $v_k = v_i/v_j$ can equally be regarded as the sequence of operations

$$w_j = (v_j)^{-1},$$
$$v_k = v_j * w_j,$$

that is, a combination of the unary inverse operator and the binary $*$ operator. From Table (4.2), we have that

$$z^{w_j} = z^{v_j} \qquad \text{and} \qquad h^{w_j} = z^{v_j}. \tag{4.47}$$

We have already shown that these assignments provide sufficient condition for w_j to have a zero partial derivative, and an all zero column in its Hessian.

Using the Boolean identities from Table 4.1, we now have for $v_k = v_j * w_j$ that

$$\begin{aligned}
h^{v_k} &= h^{v_i} \wedge h^{w_j} \wedge (z^{v_i} \vee \forall(z^{w_j})) \wedge (\forall(z^{v_i}) \vee z^{w_j}) \\
&= h^{v_i} \wedge z^{v_j} \wedge (z^{v_i} \vee \forall(z^{v_j})) \wedge (\forall(z^{v_i}) \vee z^{v_j}) && \text{(using (4.47))} \\
&= [h^{v_i} \wedge z^{v_j} \wedge (z^{v_i} \vee \forall(z^{v_j})) \wedge \forall(z^{v_i})] \vee \\
&\quad [h^{v_i} \wedge z^{v_j} \wedge (z^{v_i} \vee \forall(z^{v_j})) \wedge z^{v_j}] && \text{(using distr. law)} \\
&= [h^{v_i} \wedge z^{v_j} \wedge (z^{v_i} \vee \forall(z^{v_j})) \wedge \forall(z^{v_i})] \vee && \text{(using comm. and} \\
&\quad [h^{v_i} \wedge z^{v_j} \wedge (z^{v_i} \vee \forall(z^{v_j}))] && \text{idemp. law)} \\
&= h^{v_i} \wedge z^{v_j} \wedge (z^{v_i} \vee \forall(z^{v_j})) && \text{(using absorp. law) .}
\end{aligned}$$

This is indeed the entry for the / operator in Table 4.2.

7. **The power operator $^\wedge$.**

   Here we have that $v_k = \varphi_k(v_i, v_j) = v_i \,^\wedge v_j$ for some i, $j < k$. We observe that if $\dfrac{\partial v_k}{\partial u_p} = \mathcal{Z}$, then $\mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}$ for all $1 \leq q \leq n$. In other words,

$$\frac{\partial v_k}{\partial u_p} = \mathcal{Z} \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n. \tag{4.48}$$

   From Definitions 4.1.9, 4.1.10 and Table 4.2, we have for this φ_k that

$$h^{v_k} = \eta_k(z^{v_i}, z^{v_j}, h^{v_i}, h^{v_j}) = z^{v_i} \wedge z^{v_j}.$$

   But from Table 4.2, we furthermore have that $z^{v_k} = z^{v_i} \wedge z^{v_j}$, so $h^{v_k} = z^{v_k}$. Therefore, $h_p^{v_k} = z_p^{v_k}$, so

$$h_p^{v_k} = 1 \Rightarrow z_p^{v_k} = 1 \Rightarrow \frac{\partial v_k}{\partial u_p} = \mathcal{Z} \tag{4.49}$$

   from Theorem 4.1.11. Combining (4.48) and (4.49), we have that

$$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

We have thus shown that for all k, and every elementary operator φ_k we consider,

$$h_p^{v_k} = 1 \Rightarrow \mathcal{H}_{q,p}^{v_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

□

As above, let $\mathcal{F} : U^{(n)} \subset V^{(n)} \to W^{(m)}$ be an operator, consisting of m components $\mathfrak{f}_i : U^{(n)} \subset V^{(n)} \to W$. Let N_i denote the total number of steps in the evaluation trace taken when evaluating $\mathfrak{f}_i(u_1, \ldots, u_n)$ finishes. We then let $y_i = v_{N_i}$ denote the output variable associated with each $\mathfrak{f}_i$, that is

$$\begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \mathcal{F}(u_1, \ldots, u_n) = \begin{bmatrix} \mathfrak{f}_1(u_1, \ldots, u_n) \\ \vdots \\ \mathfrak{f}_m(u_1, \ldots, u_n) \end{bmatrix}.$$

Combining Theorems 4.1.7 and 4.1.12 gives rise to the following corollaries on which automatic linearity detection is based.

Corollary 4.1.13. *If $h_j^{y_i} = 1$, then $\mathfrak{f}_i$ is affine in the jth argument.*

Corollary 4.1.14. *If $\forall(h^{y_i}) = 1$, then $\mathfrak{f}_i$ is affine.*

Corollary 4.1.15. *If* $\forall(h^{y_i}) = 1$ *for all* $1 \leq i \leq n$*, then* $\mathcal{F}$ *is affine.*

Corollary 4.1.14 asserts that for $y_i = \mathfrak{f}_i(u_1, \ldots, u_n)$,

$$\forall(h^{y_i}) = 1 \quad \Rightarrow \quad \frac{\partial^2 \mathfrak{f}_i}{\partial u_j \partial u_k} = \bar{\mathcal{Z}}, \qquad 1 \leq q \leq n.$$

To see that "$\Leftarrow$" does not necessarily hold, consider the example

$$y = \mathfrak{f}(u_1, u_2) = u_1 + u_2^2 - u_2^2.$$

Here, the Hessian of $\mathfrak{f}$ is

$$\begin{pmatrix} \dfrac{\partial^2 \mathfrak{f}}{\partial u_1^2} & \dfrac{\partial^2 \mathfrak{f}}{\partial u_1 \partial u_2} \\ \dfrac{\partial^2 \mathfrak{f}}{\partial u_2 \partial u_1} & \dfrac{\partial^2 \mathfrak{f}}{\partial u_2^2} \end{pmatrix} = \begin{pmatrix} \bar{\mathcal{Z}} & \bar{\mathcal{Z}} \\ \bar{\mathcal{Z}} & \bar{\mathcal{Z}} \end{pmatrix},$$

but $\forall(h^y) = 0$.

As an example of how derivative pattern information is propagated in the evaluation trace, let y be a function of two variables that has the formula

$$y = [\, 2(u_1 + u_2) - \sqrt{u_1}\,] \cos(u_1 + u_2) + x^2,$$

and suppose that we want to obtain the value of y with $x = 0.5$, $u_1 = 4$ and $u_2 = 0.75$. We begin by initalising the z^{v_k} and h^{v_k} vectors according to (4.18) and (4.19) respectively. We then compute the values of the vectors h^{v_k} and z^{v_k} according to the rules listed in Table 4.2 to arrive at the linearity information for the final result y. In Table 4.3, we show the augmented evaluation trace.

4.2 Linearity detection in AD software

Here, we demonstrate the algorithmic ideas presented above by incorporating linearity detection into the standard forward-mode, operator-overloaded AD implementation through the `advar` class, introduced in Section 1.3.

To incorporate linearity detection into the `advar` class, we modify it and introduce the fields `zeroder` and `linear`. Each field stores a Boolean MATLAB vector of length n, where n is the number of input variables in a given problem. Those vectors are used to propagate linearity information through z and h as described in Section 4.1.5. We chose the name `linear` to represent the h vector, as we believe it is a more intuitive name than `zeroColHessian`. This is because as we saw in the previous section, it is

Assignment	Value of v^k	Value of z^{v_k}	Value of h^{v_k}
$v_{-2} = x$	0.5000	[1 1]	[1 1]
$v_{-1} = u_1$	4.0000	[0 1]	[1 1]
$v_0 = u_2$	0.7500	[1 0]	[1 1]
$v_1 = v_{-1} + v_0$	$4.0000 + 0.7500 = 4.7500$	[0 0]	[1 1]
$v_2 = 2v_1$	$2 \times 4.7500 = 9.5000$	[0 0]	[1 1]
$v_3 = \sqrt{v_{-1}}$	$\sqrt{4.0000} = 2.0000$	[0 1]	[0 1]
$v_4 = v_2 - v_3$	$9.5000 - 2.0000 = 7.5000$	[0 0]	[0 1]
$v_5 = \cos(v_1)$	$\cos(4.7500) = 0.0376$	[0 0]	[0 0]
$v_6 = v_4 * v_5$	$7.5000 \times 0.0376 = 0.2820$	[0 0]	[0 0]
$v_7 = (v_{-2})^2$	$0.5000^2 = 0.2500$	[1 1]	[1 1]
$v_8 = v_7 + v_8$	$0.2820 + 0.2500 = 0.5320$	[0 0]	[0 0]
$y = v_8$	0.5320	[0 0]	[0 0]

Table 4.3: An evaluation trace for $y = \left[2(u_1 + u_2) - \sqrt{u_1} \right] \cos(u_1 + u_2) + x^2$, showing the propagation of linearity information.

equivalent that a variable is affine in an input and that it has an all zero column in the Hessian with respect to the input. We mention that this way of storing linearity information in the `advar` objects is the most elementary one. It is obvious that in order to avoid running into difficulties with running out of memory if the arity of the original operator is large, the vectors in the `zeroder` and `linear` fields should be represented as sparse vectors. However, logical vectors in MATLAB already require only one eighth of the memory of a vector of doubles.

The previously overloaded methods of the `advar` class also have to be modified to account for linearity detection. All such methods carry out the expected commands on the values of the variables, compute derivatives using AD techniques, and compute information about linearity as described in Section 4.1.5. For example, the `plus` method of the `advar` class looks as follows with linearity detection incorporated:

```
function advOut = plus(adv1, adv2)
    % Construct a new advar, adding values and derivatives of inputs
    advOut = advar(adv1.val + adv2.val, adv1.der + adv2.der);
    % Assign linearity information to output
    advOut.zeroder = adv1.zeroder & adv2.zeroder;
    advOut.linear  =  adv1.linear & adv2.linear;
end
```

Another example is the `cos` method:

```
function advout = cos(advin)
    % Take cos of values and construct deriv. according to chain rule
    advout = advar(cos(advin.val), -sin(advin.val).*advin.der);
    % Assign linearity information to output
    advout.zeroder = advin.zeroder;
    advout.linear  = advin.zeroder;
end
```

We use similar ideas as for the `advar` class above to implement linearity detection in Chebfun. However, like the lazy evaluation of derivatives described in Chapter 3, the linearity detection is done in a delayed manner; that is, when the information is required, not when functions are evaluated. This requires the implementation of linearity detection to be slightly different in Chebfun than for the `advar` class. Rather than attaching the vectors z and h to `chebfun` variables, we append a vector corresponding to the z vector to `linop` objects as derivatives are computed, and the h vector is returned as the second output argument of the overloaded `diff` method. We then do the same Boolean arithmetic on the vectors as described in Table 4.2 for yielding linearity information about operators. As discussed in Chapter 3, we expect to replace the delayed evaluation of derivatives in Chebfun with a run-time evaluation. Therefore, we expect that linearity detection in future versions of Chebfun will be similar to what is described here for the `advar` class.

We are interested in the extra computational cost required in the evaluation trace for detecting linearity. We point out that the operations required for detecting linearity should only be carried out the first time we compute the derivatives of an operator; as in the case of nonlinear problems, linearity information does not change between iterations. We look at unary and binary operators separately:

1. For a unary operator, $v_k = \varphi_k(v_i)$, we need two memory write operations, and one or two memory read operations, depending on whether φ_k is linear or not. We do not need any arithmetic operations, since z^{v_k} and h^{v_k} are simply assigned the values of previous zero derivative or Hessian sparsity vectors.

2. For a binary operator, $v_k = \varphi_k(v_i, v_j)$, as is evident from Table 4.2, we need four memory access and two memory write operations. Furthermore, we need $O(n)$ logical arithmetic operations for computing z^{v_k} and h^{v_k}.

The differences between the work presented here and that described in [Wal08] include:

- We are able to detect that a function such as $x^2 + u_1'$ is linear. This is important when linearity detection is applied in solving differential equations, as we want

to allow nonlinear dependence on the independent space/time variable without flagging the problem as nonlinear.

- Rather than working with sets of index domains for every intermediate variable and maintaining nonlinear interaction domains for every input variable as described in [Wal08], we work entirely with Boolean vectors (vectors for z and h) that contain the information.

- In particular, by not working with nonlinear interaction domains, we avoid the "dead end" problem discussed in [Wal08]. The dead end problem is caused by performing nonlinear operations on variables that are not required for computing the final output variable, such as the sequence (assuming `u1` and `u2` have been defined to be variables of the correct type in the code):

  ```
  v = sin(u1)*cos(u2);
  w = u1 + u2;
  ```

  This sequence would lead to u_1 and u_2 being included in the nonlinear interaction domains (which determine sparsity patterns of Hessians), but if we are only interested in the linearity of w, this is unnecessary. Note that while these two implementations of linearity detection give different results for the linearity of expressions such as $x^2 + u_1'$, our avoidance of the dead end problem is *not* due to that. Rather, it is because we determine whether each intermediate variable is linear in the input variables, instead of keeping track of what input variables appear in nonlinear expressions.

4.2.1 Examples

As an example of `advars` in use, consider the expression

$$y = \sin(x) + 3u_1 + u_2^2,$$

evaluated at $x = 2$, $u_1 = 4$, $u_2 = -1$. We are interested in obtaining derivatives with respect to u_1 and u_2, so we begin by creating those variables as described in Section 1.3:

```
>> u1 = advar( 4, [1 0]);
>> u2 = advar(-1, [0 1])
u2 =
       val: -1
       der: [0 1]
    linear: [1 1]
  zeroder: [1 0]
```

We remind the reader about the syntax of the `advar` constructor used here. The first argument to the constructor above is the value of the variable. The second argument to the constructor for `u2`, `[0 1]`, indicates that u_2 has the derivatives 0 with respect to u_1, and 1 with respect to u_2. The information printed for `u2` tells us that it is linear in both `u1` and `u2`, but as expected, we cannot guarantee that its derivative with respect to `u2` is zero.

To create an `advar` to represent an independent variable x we use:

```
>> x = advar(2, [0 0])
x =
       val: 2
       der: [0 0]
    linear: [1 1]
  zeroder: [1 1]
```

The information above tells us that the variable x is linear in both u_1 and u_2, and that the sufficient conditions for its derivative with respect to both u_1 and u_2 to be zero are satisfied.

We now create new variables from `x`, `u1` and `u2`, which will contain the correct derivative and linearity information. Using the `getval` and `getder` methods described in Section 1.3, we can obtain the values of variables and derivatives respectively. For our expression above, we have

```
>> y = sin(x) + 3*u1 + u2.^2;
>> getval(y)
ans =
  13.909297426825681
>> getder(y)
ans =
     3    -2
```

The `getlin` method returns information about linearity, that is, the vector h for that variable. An entry equal to 1 indicates that the input variable is linear in the corresponding variable we are differentiating with respect to. By executing

```
>> getlin(y)
ans =
     1     0
```

we see that y is linear in u_1 but not u_2. For overall linearity information, that is, whether y can be detected to be affine as according to Definition 4.1.2, we call the `islinear` method. It returns either a logical 0 or 1:

```
>> islinear(y)
ans =
```

```
     0
```

We can now check for linearity of various mathematical expressions:

```
>> islinear(x.*u1 + u2)
ans =
     1
>> islinear(x.*exp(u1))
ans =
     0
>> islinear(x + u1.*u2)
ans =
     0
```

In the Chebfun context, the most important use of linearity detection is to distinguish linear `chebop` objects from nonlinear ones. A part of Chebfun where linearity detection is crucial is for eigenvalue problems. Linear eigenvalue problems, such as the harmonic oscillator,

$$-u'' + x^2 u = \lambda u, \quad u(-8) = u(8) = 0,$$

can be easily solved with using the `chebop` class as follows:

```
>> N = chebop(@(x,u) -diff(u,2) + x.^2.*u, [-8 8]); % Create a chebop
>> N.lbc = 0; N.rbc = 0;        % Assign BCs
>> [U, Lambda] = eigs(N, 6);    % Find six eigenmodes
```

However, nonlinear eigenvalue problems are currently not supported, so an error is thrown:

```
>> N = chebop(@(x,u) -diff(u,2) + x.^2.*u.^2, [-8 8]); % Nonlin. chebop
>> N.lbc = 0; N.rbc = 0;        % Assign BCs
>> [U, Lambda] = eigs(N, 6);    % Expect an error to be thrown
Error using chebop/eigs (line 50)
Chebop appears to be nonlinear. Currently, eigs only
supports linear chebops.
```

Without linearity detection, Chebfun would not be able to offer the nice `chebop` syntax for linear eigenvalue problems, and users would have to resort to the more inconvenient `linop` syntax. Linearity detection holds the same importance for operator exponentiation via the `expm` method, which is only supported for linear operators.

Another application area of linearity detection is for solving BVPs. If a problem is solved with a nonlinear `chebop`, damped Newton iteration in function space is performed, while for a linear `chebop`, the solver knows it should only take one full Newton step. For example, the output of solving the linear BVP

$$u'' + x^2 u = 0, \qquad u(-1) = -1, \quad u'(5) = 0,$$

is as follows:

```
>> % Create a linear chebop on [−1,5]
>> N = chebop(@(x,u) diff(u,2) + x.^2.*u, [−1 5]);
>> N.lbc = −1; N.rbc = @(u) diff(u);              % Assign BCs
>> u = N\0;                                       % Solve
Converged in one step. (Chebop is linear).
```

Thanks to a simpler algorithm we can use when solving linear BVPs, the solution of the problem above takes 0.29 seconds. On the other hand, it takes 0.65 seconds to solve the problem by calling the full damped Newton method.

Chapter 5

Multiple Solutions of Nonlinear BVPs

In the preceding chapters, we have set up algorithms for the solution of nonlinear BVPs by Newton's method in function space and automatic differentiation. The crucial inputs to the algorithms are a nonlinear differential operator $\mathcal{N}$ and an initial guess u_0. The output from the algorithm, if it converges, is a function u, representing a good approximation to a true solution u^*, for which $\mathcal{N}(u^*) = 0$.

Like nonlinear problems in finite dimensions, nonlinear BVPs can have multiple solutions.[1] However, for each initial guess u_0, Newton's method can only be expected to converge to one of those solutions. This chapter describes algorithms for obtaining multiple solutions of nonlinear BVPs, via deflation in Section 5.1, and path-following in infinite dimensions in Section 5.2. In Section 5.3, we then present an algorithm combining deflation and path-following.

To motivate our discussion, consider the differential equation governing the motion of a mathematical pendulum:

$$\ddot{\theta} + \sin(\theta) = 0.$$

Here, θ denotes the angular deviation in radians from the stable equilibrium of the pendulum at $\theta = 0$. Assume that we are interested in finding trajectories of the pendulum that start at position $\theta = 2$ at time $t = 0$, and finish in the same position at a final time $T = 10$. We can formulate this problem as the two-point BVP

$$\ddot{\theta} + \sin(\theta) = 0, \qquad \theta(0) = \theta(10) = 2. \tag{5.1}$$

[1] To clarify, we use the term *multiple solutions* in the sense that we say that the equation $x^2 - 1 = 0$ has multiple solutions, that is, $x = -1$ and $x = 1$. We do not use it for describing solutions with multiplicity greater than 1, for example in the equation $(x-1)^2 = 0$, where $x = 1$ is a double root.

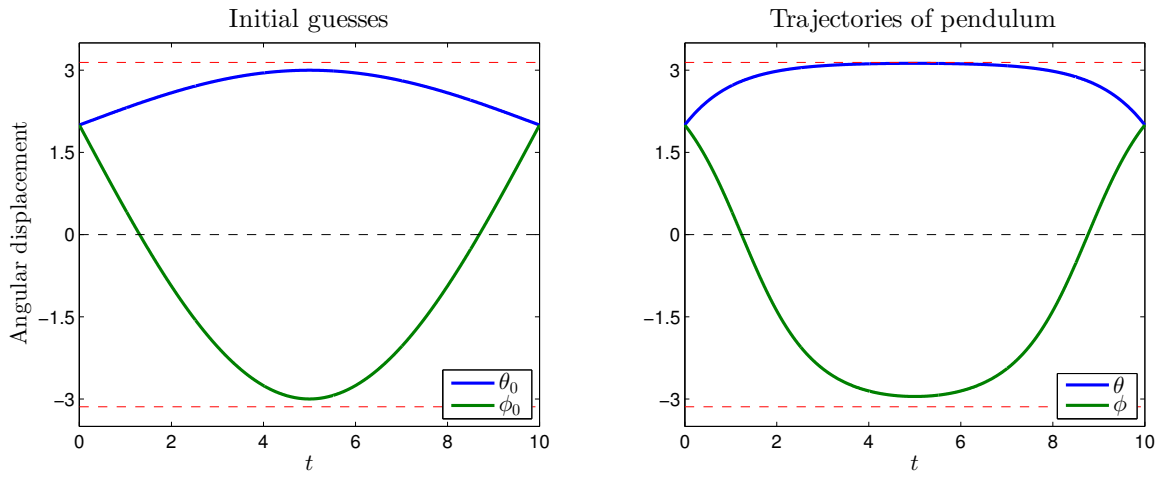

Figure 5.1: Two solutions of the BVP (5.1), describing the motion of a nonlinear pendulum. Each solution was obtained by starting the Newton iteration with the corresponding initial guess on the left plot. The black dashed line indicates the stable equilibrium of the pendulum, and the two red dashed lines indicate the unstable one.

Intuitively, there might be more than one trajectory satisfying these boundary conditions. For example, we might have the pendulum going up towards the unstable equilibrium at $\theta = \pi$, and staying there before returning to its initial position. Another solution might be that the pendulum swings over the stable equilibrium at $\theta = 0$ fast enough to be able to make it back to $\theta = 2$ at time $T = 10$. Indeed, numerical experiments confirm that both solutions exist. In Figure 5.1, we plot two solutions $\theta(t)$ and $\phi(t)$ obtained from the initial guesses

$$\theta_0(t) = 2 + \sin\left(\pi\frac{t}{10}\right), \qquad \phi_0(t) = 2 - 5\sin\left(\pi\frac{t}{10}\right).$$

The origin of the work presented in this chapter is as follows: All material presented in Sections 5.1 and 5.3 is due to the author (the topic of deflation was suggested by Nick Trefethen). The author was introduced by Tobin Driscoll to the idea of using path-following for obtaining multiple solutions of nonlinear ODEs, discussed in Section 5.2. The original implementation of the `chebmatrix` class, necessary for implementing path-following in Chebfun, was done jointly by the author and Driscoll. The presentation in Section 5.2 is due to the author.

5.1 Deflation for multiple solutions of nonlinear BVPs

The aim of this section is to develop algorithms which, starting from a single initial guess u_0, are capable of returning multiple solutions to nonlinear BVPs. This is desirable in order to construct a black-box solver capable of returning multiple solutions to nonlinear BVPs. We achieve this via the technique of *deflation*. We start by discussing deflation in the context of polynomial rootfinding, before describing deflation for systems of nonlinear equations. Finally, we present deflation for nonlinear BVPs.

5.1.1 Deflation for polynomial rootfinding

Perhaps the best known use of deflation is in the context of polynomial rootfinding. Deflation, in combination with Newton iteration or another rootfinding method, can be used to find multiple roots of a polynomial $p_n(x)$. We note that there exist superior methods for finding all roots of polynomials, such as the global approach of computing eigenvalues of colleague or companion matrices. However, looking at deflation for polynomial rootfinding helps us set the stage for discussions below of deflation for operator equations.

Overviews of the use of deflation for polynomial rootfinding can be found in [PW71] and [PTVF07]. In their treatment, Peters and Wilkinson make a distinction between two related techniques:

- *Deflation*, in which, if a root r is found, the original polynomial $p_n(x)$ of degree n is divided by $x - r$ explicitly to give a polynomial of degree $n - 1$ with the remaining roots.

- *Suppression*, where subsequent rootfinding iterations are performed using the function $\hat{p}_n(x) = p_n(x)/(x-r)$, without performing the polynomial division explicitly. The values of $\hat{p}_n(x)$ are then obtained by computing $p_n(x)$ and dividing it by $x - r$.

As described in [PW71], both methods have advantages and disadvantages.

- With deflation, the polynomial degree gradually decreases, making the polynomials cheaper to evaluate. However, errors can accumulate as we find the roots, and further roots might be contaminated. Hence, it is advisable to *polish* the roots found by deflation; that is, compute a root of the deflated polynomial, but then apply Newton's method on the undeflated polynomial, using the unpolished root as an initial guess.

- Using suppression, the function becomes increasingly more expensive to evaluate as more roots are found. However, one keeps working with the original polynomial $p_n(x)$ which still contains the true roots, so errors in roots previously found will not affect further roots found.

Importantly, deflation is limited to the polynomial context, while suppression works for more general functions. Although there is a distinction between deflation and suppression, the literature often uses the term deflation also for the non-polynomial case, and we follow that tradition here. To be precise, in this section on polynomial rootfinding, we are truly using deflation as defined by Peters and Wilkinson, while in subsequent sections, our method should technically be called suppression.

As discussed in [BFST03], deflation works well in combination with Horner's scheme for polynomial evaluation. To see why, let a polynomial p_n of degree n be given, and suppose for simplicity that p_n has n distinct and real roots. Also, assume we are working in the monomial basis. Then, we can write

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n. \tag{5.2}$$

We can also write the polynomial in a first order remainder form as

$$p_n(x) = (x - \tilde{x}) q_{n-1}(x) + R = (x - \tilde{x})(b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}) + R, \tag{5.3}$$

where q_{n-1} is the polynomial of degree $n-1$ obtained by polynomial division of p_n by $x - \tilde{x}$, and $R = p_n(\tilde{x})$.

Suppose that starting from an initial guess x_0, Newton's method converges to a root $r^{(1)}$. Then, $R = 0$, and (5.3) simplifies to

$$p_n(x) = (x - r^{(1)}) q_{n-1}(x) = (x - r^{(1)})(b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}). \tag{5.4}$$

Because of the assumption that p_n has distinct roots, $r^{(1)}$ is not a root of q_{n-1}. Thus, if we are able to find a root of q_{n-1}, we have found another root of the original polynomial p_n. This process is repeated until all the roots are found.

Recall that in Horner's scheme for polynomial evaluation, we rewrite the polynomial p_n from (5.2) in *nested form*:

$$p_n(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)). \tag{5.5}$$

To evaluate $p_n(x)$ at some value $x = \tilde{x}$, we then successively evaluate

$$\gamma_0 = a_n, \quad \gamma_{k+1} = \tilde{x}\gamma_k + a_{n-1-k}, \tag{5.6}$$

for $k = 0, 1, \ldots, n-1$, yielding the value $p_n(\tilde{x}) = \gamma_n$.

Comparing the coefficients of q_{n-1} from (5.3), and the nested form of p_n from (5.5), we see that for the coefficients b_i of the quotient polynomial at a point $\tilde{x}$, we must have

$$b_{n-1} = a_n, \quad b_{n-k} = a_{n-k+1} + \tilde{x} b_{n-k+1}, \tag{5.7}$$

for $k = 2, \ldots, n$. This leads to a recurrence algorithm for computing the coefficients of q_{n-1}. Hence, if we find a root $r^{(1)}$, the coefficients of the deflated polynomial are given by

$$b_{n-1} = a_n, \quad b_{n-k} = a_{n-k+1} + r^{(1)} b_{n-k+1}, \quad k = 2, \ldots, n.$$

Finally, we note that differentiating (5.3) on both sides with respect to x gives

$$p_n'(x) = q_n(x) + q_n'(x)(x - \tilde{x}). \tag{5.8}$$

Evaluating (5.8) at $x = \tilde{x}$ (which is not necessarily a root) yields

$$p_n'(\tilde{x}) = q_n(\tilde{x}). \tag{5.9}$$

Hence, if we are able to evaluate the quotient polynomial q_n at $\tilde{x}$, we can obtain the value of the derivative of p_n at the same point. But this is simple, as comparing (5.6) and (5.7), we notice that the coefficients of q_n for each $\tilde{x}$ are given by the intermediate values of γ when evaluating p_n at $\tilde{x}$, that is,

$$b_{n-k-1} = \gamma_k, \quad k = 0, 1, \ldots, n-1. \tag{5.10}$$

Thus, when we perform nested evaluation of p_n at a point $\tilde{x}$, using (5.10) we can also evaluate the quotient polynomial q_{n-1} at $\tilde{x}$ at the same time. By (5.9), this gives the value of p_n' at $\tilde{x}$.

We summarise deflation for polynomial rootfinding using Horner's method in Algorithm 8. Figure 5.2 shows MATLAB code implementing the algorithm. Note that in the code, we have to compute the coefficients of the deflated polynomial after we find each root. Although these coefficients are equal to the intermediate values in the Horner evaluation scheme, we have not taken the last Newton step when the last intermediate values are computed. Hence, we can not use the last intermediate values as the next coefficients straightaway.

We call the method for the polynomial $p_4(x) = x^4 - 10x^3 + 35x^2 - 50x + 24$, which has the roots $r_1 = 1$, $r_2 = 2$, $r_3 = 3$ and $r_4 = 4$. In Figure 5.3, we show polynomials of successively lower degrees as the roots of p_4 gradually get deflated.

Algorithm 8 Deflation in combination with Horner's method and Newton iteration.

Input: A vector a of length $n+1$, containing the $n+1$ coefficients of a polynomial p_n of degree n in decreasing order, assumed to have n distinct and real roots. Initial guess of the roots, x_0.

Output: A vector r, containing the n roots of p_n (assuming Newton's method will not diverge).

$m \leftarrow n$ ▷ Store number of coeffs. of polyn. currently being worked with
for $j = 1, \ldots, n$ **do** ▷ Seek n roots.
  Newton iteration for finding a root of p_m, using Horner scheme for evaluation.
  $x \leftarrow x_0$ ▷ Assign x the initial guess.
  repeat
    $\gamma \leftarrow a_{m+1}$ ▷ Initialise for computing value of polynomial at x.
    $\delta \leftarrow 0$ ▷ Initialise for computing value of derivative at x.
    for $k = m, \ldots, 1$ **do** ▷ Nested evaluation of polynomial and derivative
      $\delta \leftarrow x\delta + \gamma$ ▷ Intermediate derivative value
      $\gamma \leftarrow x\gamma + a_k$ ▷ Intermediate polynomial value
    end for
    $\Delta = -\gamma/\delta$ ▷ Compute Newton correction
    $x \leftarrow x + \Delta$ ▷ Take Newton step
  until $|\Delta| < 10^{-11}$ ▷ Newton iteration until converged
  $r_j \leftarrow x$ ▷ Store the root found
  Deflation step. Compute coefficients of the quotient polynomial q_{m-1}.
  $b \leftarrow (0, \ldots, 0)$ (m times) ▷ Initialise vector for coefficients of q_{m-1}.
  $b_m = a_{m+1}$ ▷ Highest coefficient of quotient polynomial
  for $l = m-1, \ldots, 1$ **do** ▷ Nested computation of coefficients of q
    $b_l = a_{l+1} + x\, b_{l+1}$
  end for
  $a \leftarrow b$ ▷ Update coefficients for next iteration, when we seek roots of q_{m-1}.
end for
return r.

```
function r = HornerDeflation(a)
% HornerDeflation: Compute roots of polynomial with coefficients A, via
% deflation in combination with Horner's method and Newton iteration.
r = zeros(1, length(a) - 1);         % For storing roots found
for rootCounter = 1:length(a) - 1    % Seek n roots
    x = 0;                           % Initial guess for root
    normDel = Inf;                   % Initialise control variable
    while normDel > 1e-11            % Newton iter. until ||del|| < 1e-11
        % Horner's method for evaluating polynomial and its derivative
        gamma = a(end);
        der = 0;

        for k = (length(a) - 1):-1:1  % Recurrent evaluation
            der = x*der + gamma;
            gamma = x*gamma + a(k);
        end

        delta = -gamma/der;           % Newton correction
        normDelta = norm(delta);      % Check norm for convergence check
        x = x + delta;                % Take Newton step
    end
    r(rootCounter) = x;          % Store root found

    % Deflate the current root and obtain coeffs. of quotient polynomial
    b = zeros(1, length(a) - 1);
    b(end) = a(end);
    for k = (length(b) - 1):-1:1
        b(k) = a(k + 1) + x*b(k + 1);
    end
    a = b;                        % Update coeffs. used for next iter.
end
end
```

Figure 5.2: MATLAB code implementing Algorithm 8.

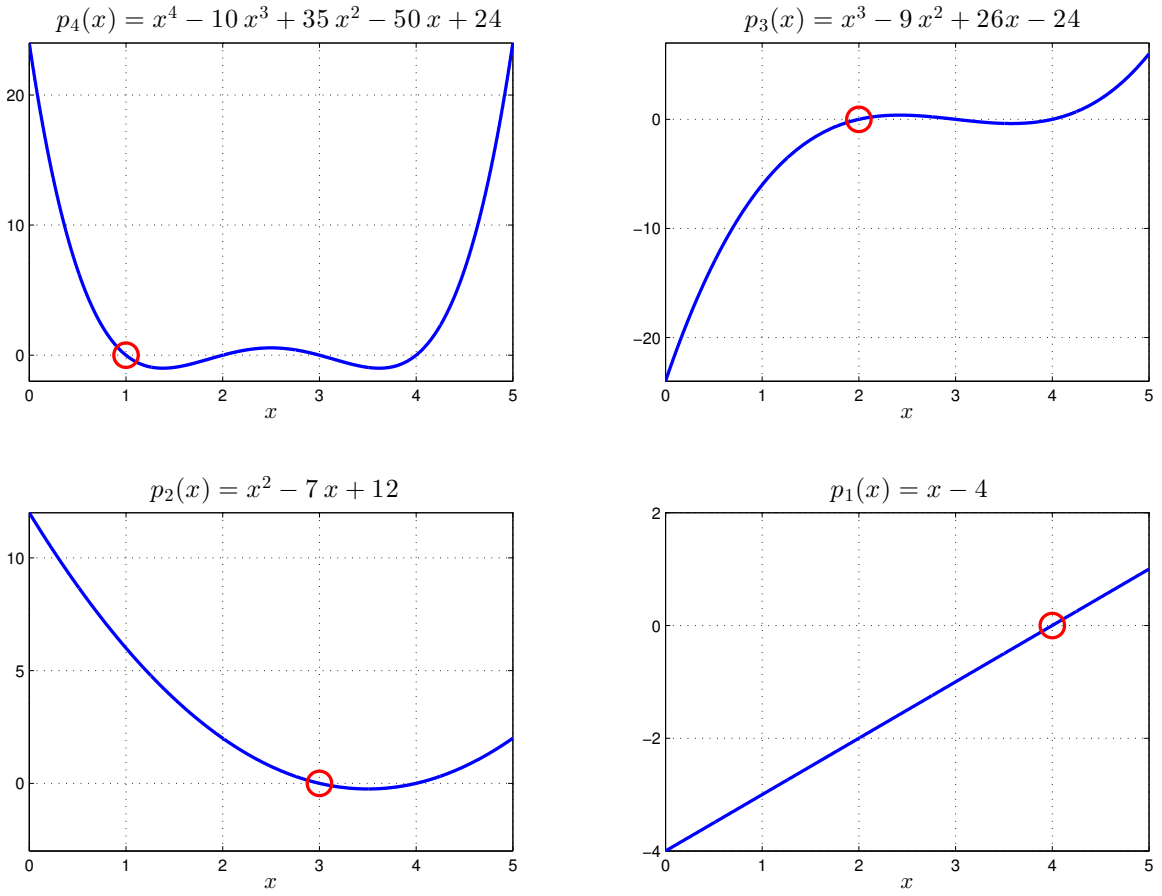

Figure 5.3: Obtaining multiple roots of the polynomial $p_4(x) = x^4 - 10x^3 + 35x^2 - 50x + 24$, via deflation in combination with Horner's method and Newton iteration. The plots show the polynomials we seek roots of as the root previously found gets deflated. We mark with a red circle the root found by Newton's method for each polynomial. In all cases, Newton's method starts from the same initial guess $x_0 = 0$.

5.1.2 Deflation for nonlinear systems of equations

After these preliminary discussions of deflation for polynomial rootfinding, we now discuss deflation for nonlinear systems of equations. Here, we want to find multiple roots of the equation

$$F(x) = 0,$$

where $x \in \mathbb{R}^n$ and $F : \mathbb{R}^n \to \mathbb{R}^n$. We equip $\mathbb{R}^n$ with the standard Euclidean inner product $\langle x, y \rangle = x^T y$, and the associated norm $\|x\| = \sqrt{\langle x, x \rangle^2}$. We assume that F is defined on all of $\mathbb{R}^n$, and that the roots of F are simple. Also, we assume that the components of F, denoted by $f_1, \ldots, f_n$, are differentiable with respect to each argument. The approach of applying deflation to this problem has been discussed in the literature for example in [BG71], although in general, numerical continuation

methods as discussed in Section 5.2 are a more popular technique. We remind the reader that technically, the method should be categorised as suppression as defined in [PW71], but we will use the term deflation preferred in the literature.

Assume that we have have found a root $r^{(1)}$ of the system $F(x) = 0$, using Newton iteration starting from an initial guess x_0. Then it is clear that to find another root $r^{(2)}$ starting from the same initial guess, we have to modify the problem. Following [BG71], we define *deflation matrices* as follows:

Definition 5.1.1. *For each $r \in \mathbb{R}^n$, let $M(x; r)$ be a $n \times n$ matrix, defined for all $x \in X_r$, where X_r is an open subset of $\mathbb{R}^n$ such that r belongs to the closure of X_r. We say that M is a* deflation matrix *if for any differentiable $F : \mathbb{R}^n \to \mathbb{R}^n$ such that $F(r) = 0$ and $F'(r)$ is nonsingular, we have*

$$\liminf_{i\to\infty} \left\| M(x^{(i)}; r)F(x^{(i)}) \right\| > 0 \tag{5.11}$$

for any sequence $x^{(i)} \to r$, $x^{(i)} \in X_r$. Here, $F'(r)$ denotes the Jacobian of F, evaluated at r.

Having found a root $r^{(1)}$, we form the function

$$G^{(1)}(x) := M(x; r^{(1)})F(x),$$

and apply our iterative method for finding a zero of $G^{(1)}$. By (5.11), if a sequence of points converges to $r^{(1)}$, it does not give a root of $G^{(1)}$, so the sequence generated by a rootfinding algorithm does not converge to $r^{(1)}$. We say that $G^{(1)}$ is the function obtained by *deflating out* the root $r^{(1)}$. If our iterative method finds a root of $G^{(1)}$, we then polish it by applying Newton's method to the undeflated problem.

If we have found m roots of $F(x) = 0$, denoted by $r^{(1)}$, $r^{(2)}$, $\ldots$, $r^{(m)}$, then to look for the next root, we construct the deflated function

$$G^{(m)}(x) := M(x; r^{(m)}) \cdots M(x; r^{(2)})M(x; r^{(1)})F(x), \tag{5.12}$$

and apply the rootfinding algorithm to $G^{(m)}$.

The question is, what kind of matrices can we use as deflation matrices? The following lemma [BG71, lemma 2.1] is useful for determining whether a given matrix is a deflation matrix.

Lemma 5.1.2. *Suppose that the matrix $M(x; r)$ has the property that for each $r \in \mathbb{R}^n$, and any sequence $x^{(i)} \to r$, $x^{(i)} \in X_r$, where X_r is an open subset of $\mathbb{R}^n$ such that r belongs to the closure of X_r, if*

$$\left\| x^{(i)} - r \right\| M(x^{(i)}; r)z^{(i)} \to 0, \tag{5.13}$$

for some sequence $\left\{z^{(i)}\right\}$, $z^{(i)} \in \mathbb{R}^n$, *then* $z^{(i)} \to 0$. *Then* M *is a deflation matrix.*

Proof. We proceed via a proof by contradiction. If M is not a deflation matrix as defined by Definition 5.1.1, then there exists a differentiable function $F : \mathbb{R}^n \to \mathbb{R}^n$ and an $r \in \mathbb{R}^n$, such that $F(r) = 0$, $F'(r)$ nonsingular, and

$$\liminf_{i\to\infty} \left\| M(x^{(i)}; r)F(x^{(i)}) \right\| = 0$$

for some sequence $x^{(i)} \to r$, $x^{(i)} \in X_r$. Then there exists a subsequence $\left\{y^{(i)}\right\}$ of $\left\{x^{(i)}\right\}$, such that $M(y^{(i)}; r)F(y^{(i)}) \to 0$. Setting $z^{(i)} = F(y^{(i)}) / \left\|y^{(i)} - r\right\|$, we have that

$$\left\|y^{(i)} - r\right\| M(y_i; r)z^{(i)} \to 0.$$

From the property of M stated in the assumptions, we conclude that $z^{(i)} \to 0$, that is,

$$\frac{F(y^{(i)})}{\|y^{(i)} - r\|} \to 0. \tag{5.14}$$

Since F is differentiable, we can expand it in Taylor series around r to give

$$F(y^{(i)}) = F(r) + F'(r)(y^{(i)} - r) + o(\left\|y^{(i)} - r\right\|) = F'(r)(y^{(i)} - r) + o(\left\|y^{(i)} - r\right\|),$$

as $F(r) = 0$. We then have that

$$\frac{F(y^{(i)})}{\|y^{(i)} - r\|} = \frac{1}{\|y^{(i)} - r\|} \left[F'(r)(y^{(i)} - r) + o(\left\|y^{(i)} - r\right\|)\right] \approx F'(r)\overline{y}^{(i)},$$

where $\overline{y}^{(i)} = \dfrac{y^{(i)} - r}{\|y^{(i)} - r\|}$ is a unit vector for all $y^{(i)} \in X_r$. But then, (5.14) leads to a contradiction of the nonsingularity of $F'(r)$. □

In other words, if we define our matrix $M(x^{(i)}; r)$, and inspect whether the sequence (5.13) can converge to 0 for any other sequence $\left\{z^{(i)}\right\}$ other than $z^{(i)} \to 0$, we can determine whether M is a deflation matrix.

This immediately gives rise to a selection of deflation matrices, which we call *norm deflation* matrices, all satisfying Lemma 5.1.2:

- We can take $M(x; r^{(i)}) = \dfrac{1}{\|x - r^{(i)}\|} I$, where I is the identity matrix on $\mathbb{R}^n$.
- More generally, we can take $M(x; r^{(i)}) = \dfrac{1}{\mu(\|x - r^{(i)}\|)} A$, where $A \in \mathbb{R}^{n\times n}$ is a nonsingular matrix, and $\mu : \mathbb{R} \to \mathbb{R}$ is a function such that $\mu(h)$ is $O(h)$.

When using norm deflation matrices, the domain of definition for (5.12) will be $\mathbb{R}^n \setminus \bigcup_{i=1}^{k} \{r^{(i)}\}$. It is clear that the selection of A and μ can affect the success of our deflation scheme. A particular choice with nice differentiability properties is the "half norm-squared" deflation matrix

$$M(x; r^{(i)}) := \frac{1}{\frac{1}{2}\|x - r^{(i)}\|_2^2} I.$$

We seek to derive expressions for the corrections at each step of the Newton iteration of the deflated function. This is similar to *Maehly's procedure* in the context of deflation for one-dimensional problems. In [Mae54], Maehly pointed out for a polynomial $p(x)$, the derivative of the reduced polynomial

$$\hat{p}_m(x) := \frac{p(x)}{(x - x_1)\cdots(x - x_m)}$$

can be written as

$$\hat{p}_m'(x) = \frac{p'(x)}{(x - x_1)\cdots(x - x_m)} - \frac{p(x)}{(x - x_1)\cdots(x - x_m)} \sum_{j=1}^{m} \frac{1}{x - x_j}. \tag{5.15}$$

Formula (5.15) also holds for the derivative of a more general function $\hat{f}_m(x)$ after deflating m roots of the original function $f(x)$. Assume we apply Newton's method in the deflation scheme for the function $\hat{f}_m$. Then, at the kth Newton iteration when finding the $(m+1)$st root, we can take the Newton step via

$$x_{k+1} = x_k - \frac{\hat{f}_m(x_k)}{\hat{f}_m'(x_k)} = x_k - \frac{f(x)}{f'(x_k) - f(x_k)\sum_{j=1}^{m}(x_k - x_j)^{-1}},$$

so there is no need to compute the derivatives of the deflated function $\hat{f}_m$ — we have an expression for the derivative of the deflated function in terms of the original function.

To determine the Newton step when using the half norm-squared deflation matrix, we note that after finding one root, the deflated function G is given by

$$G(x) = M(x; r)F(x) = \begin{pmatrix} \dfrac{f_1(x)}{\frac{1}{2}\|x - r\|_2^2} \\ \vdots \\ \dfrac{f_n(x)}{\frac{1}{2}\|x - r\|_2^2} \end{pmatrix},$$

where we have dropped the superscript of r. If we define the functional η as

$$\eta(x) := \tfrac{1}{2}\|x - r\|^2 = \tfrac{1}{2}\langle x - r, x - r\rangle,$$

we have that

$$\begin{aligned}\eta(x+y)-\eta(x) &= \frac{1}{2}\left(\langle x+y-r, x+y-r\rangle - \langle x-r, x-r\rangle\right)\\ &= \frac{1}{2}\left(\langle x,y\rangle + \langle y,x\rangle - \langle y,r\rangle - \langle r,y\rangle + \langle y,y\rangle\right)\\ &= \langle x-r, y\rangle + O(\|y\|^2).\end{aligned}$$

This indicates that the first derivative of the functional η at a point x is the functional represented by an inner product with the row vector $(x-r)^T$. Proceeding to look at the gradients of each component of $G(x)$, that is, each row of $G'(x)$, we have that

$$\nabla g_i(x) = \frac{1}{\eta(x)}\left(\nabla f_i(x) - \frac{f_i(x)}{\eta(x)}(x-r)^T\right).$$

Thus, we we see that

$$G'(x) = \frac{1}{\eta(x)}\left[F'(x) - \frac{1}{\eta(x)}F(x)(x-r)^T\right],$$

where we have taken the outer product between $F(x)$ and $x-r$. Recalling that the Newton correction for $G:\mathbb{R}^n\to\mathbb{R}^n$ at the kth step is given by

$$x_{k+1} = x_k - (G'(x_k))^{-1}G(x_k),$$

we obtain

$$\begin{aligned}(G'(x))^{-1}G(x) &= \left[F'(x) - \frac{1}{\eta(x)}F(x)(x-r)^T\right]^{-1}\eta(x)G(x)\\ &= \left[F'(x) - \frac{1}{\eta(x)}F(x)(x-r)^T\right]^{-1}F(x),\end{aligned} \tag{5.16}$$

yielding a convenient way to compute the Newton correction for the deflated function, in terms of the original function F, its Jacobian, and values of the functional η.

Another option for deflation matrices suggested by [BG71] is the use of the so called *inner product deflation* matrices. Here we take $M(x;r^{(i)})$ in (5.24) to be a diagonal matrix, whose jth diagonal element is given by

$$m_{jj} = \frac{1}{\langle a^{(i,j)}, x-r^{(i)}\rangle}. \tag{5.17}$$

Here, $a^{(i,1)},\ldots,a^{(i,n)}\in\mathbb{R}^n$ are nonzero vectors for each $i=1,\ldots,k$ — in other words, for every root $r^{(i)}$, we have a set of n vectors. Letting $C(a)$ denote the subspace of $\mathbb{R}^n$ orthogonal to a where $a\in\mathbb{R}^n$, that is

$$C(a) := \left\{u\in\mathbb{R}^n : \langle a,\, u\rangle = 0\right\},$$

we have that the domain of definition for (5.12) is

$$\mathbb{R}^n \setminus \bigcup_{i=1}^{k} \left\{ \{r^{(i)}\} \bigcup_{j=1}^{n} C\left(a_j^{(i)}\right) \right\}. \tag{5.18}$$

To confirm that the choice of M given by (5.17) does indeed define a deflation matrix for a given root $r^{(i)}$, we check that it satisfies Lemma 5.1.2. Let $\left\{x^{(j)}\right\}$ be a sequence in X_r such that $x^{(j)} \to r^{(i)}$. We define the sequence $\left\{s^{(j)}\right\}$ via

$$s^{(j)} := \left\|x^{(j)} - r^{(i)}\right\| M(x_j; r) z^{(j)}, \tag{5.19}$$

where $\left\{z^{(j)}\right\}$ is some sequence in $\mathbb{R}^n$. Defining $y^{(j)} := x^{(j)} - r^{(i)}$, we have that the kth element of the vector $s^{(j)}$ from (5.19) is given by

$$s_k^{(j)} = \left\|y^{(j)}\right\| \frac{1}{\left\langle a^{(i,j)}, y^{(j)} \right\rangle} z_k^{(j)}, \tag{5.20}$$

where $z_k^{(j)}$ is the kth element of the vector $z^{(j)}$.

Let $a_k^{(i,j)}$ denote the kth element of the vector $a^{(i,j)}$, and y_k denote the kth element of y. Taking the norm and expanding the inner product in the denominator, (5.20) becomes

$$\begin{aligned} \left|s_k^{(j)}\right| &= \left\|y^{(j)}\right\| \frac{1}{\left|a_1^{(i,j)} y_1^{(j)} + \cdots + a_n^{(i,j)} y_n^{(j)}\right|} \left|z_k^{(j)}\right| \\ &\geq \frac{\left\|y^{(j)}\right\|}{|a_1^{(i,j)}| \left|y_1^{(j)}\right| + \cdots + |a_n^{(i,j)}| \left|y_n^{(j)}\right|} \left|z_k^{(j)}\right|, \end{aligned} \tag{5.21}$$

using the triangle inequality. With $\hat{a} := \max\limits_{1 \leq k \leq n} |a_k^{(i,j)}|$, (5.21) becomes

$$\left|s_k^{(j)}\right| \geq \frac{1}{\hat{a}} \frac{\left\|y^{(j)}\right\|}{\left|y_1^{(j)}\right| + \cdots + \left|y_n^{(j)}\right|} \left|z_k^{(j)}\right|. \tag{5.22}$$

Let p, $1 \leq p \leq k$, denote the position of the largest absolute entry of $y^{(j)}$, that is,

$$\left|y_p^{(j)}\right| \geq \left|y_k^{(j)}\right|, \quad k = 1, \ldots, n.$$

Then (5.22) becomes

$$\begin{aligned}
|s_k^{(j)}| &\geq \frac{1}{\hat{a}} \frac{\left\|y^{(j)}\right\|}{n\left|y_p^{(j)}\right|} \left|z_k^{(j)}\right| \\
&= \frac{1}{\hat{a}n\left|y_p^{(j)}\right|} \sqrt{\left|y_1^{(j)}\right|^2 + \cdots + \left|y_n^{(j)}\right|^2} \left|z_k^{(j)}\right| \\
&= \frac{1}{\hat{a}n\left|y_p^{(j)}\right|} \left|y_p^{(j)}\right| \sqrt{1 + \sum_{\substack{k=1 \\ k\neq p}}^{n} \frac{\left|y_k^{(j)}\right|^2}{\left|y_p^{(j)}\right|^2}} \left|z_k^{(j)}\right| \\
&\geq \frac{1}{\hat{a}n} \left|z_k^{(j)}\right|. \qquad (5.23)
\end{aligned}$$

We see that for $s_k^{(j)} \to 0$ for any sequence $\left\{y^{(j)}\right\}$, we need $z_k^{(j)} \to 0$. Therefore, we can conclude that for $s^{(j)} \to 0$ for any sequence $x^{(j)} \to r$, we need $z^{(j)} \to 0$, so $M(x, r^{(i)})$ is a deflation matrix.

We now have to select vectors $a^{(i,j)}$ to apply in our inner product deflation scheme. Motivated by Maehly's procedure again, we seek choices of $a^{(i,j)}$ for which we can obtain derivatives of the deflated function in terms of the original function and its Jacobian. Following [BGH69], we choose

$$a^{(i,j)} = \nabla f_i(r^{(j)}),$$

that is, $a^{(i,j)}$ is the gradient of the ith component of F, evaluated at the jth root $r^{(j)}$. We call such deflation *gradient deflation*. We note that $\nabla f_i(r^{(j)})$ is equal to the ith row of $F'(r^{(j)})$ — the Jacobian of F, evaluated at $r^{(j)}$.

Once we have found m roots, we have that the ith component of the deflated function $G^{(m)}$ is given by $g_i^{(m)}$, where $g_i^{(m)}$ is constructed successively by

$$\begin{aligned}
g_i^{(0)}(x) &= f_i(x), \\
g_i^{(k+1)}(x) &= \frac{g_i^{(k)}(x)}{\langle \nabla g_i^{(k)}(r^{(k)}), x - r^{(k)} \rangle} = \frac{f_i(x)}{\prod_{j=1}^{k} \langle \nabla f_i(r^{(j)}), x - r^{(j)} \rangle}, \qquad (5.24)
\end{aligned}$$

for $k = 0, \ldots, m-1$.

We now work out the Jacobian of the deflated function when using gradient deflation. We drop the superscripts for notational simplicity, and let r be the first root of F we find. We have that the ith component of G in (5.24) can be written as

$$g_i(x) = \frac{f_i(x)}{\langle \nabla f_i(r), x - r \rangle},$$

which gives the gradient of the ith component (that is, the ith row of the Jacobian G'):

$$\nabla g_i(x) = \frac{1}{\langle \nabla f_i(r), x - r\rangle}\left(\nabla f_i(x) - \frac{f_i(x)}{\langle \nabla f_i(r), x - r\rangle}\nabla f_i(r)\right).$$

Letting $D(x;r)$ and $\grave{F}(x)$ be the diagonal matrices with entries

$$d_{ii} = \langle \nabla f_i(r), x - r\rangle, \qquad \grave{f}_{ii} = f_i(x),$$

respectively, we see that

$$G'(x) = D(x;r)^{-1}\left[F'(x) - \grave{F}(x)D(x;r)^{-1}F'(r)\right], \tag{5.25}$$

(by (5.18), we know that $d_{ii} \neq 0$, so $D(x;r)$ is invertible). Again, recalling that Newton iteration for $G : \mathbb{R}^n \to \mathbb{R}^n$ at the kth step is given by

$$x_{k+1} = x_k - (G'(x_k))^{-1}G(x_k),$$

and that

$$G(x) = D(x;r)^{-1}F(x),$$

we arrive at

$$\begin{aligned}(G'(x))^{-1}G(x) &= \left[F'(x) - \grave{F}(x)D(x;r)^{-1}F'(r)\right]^{-1} D(x;r)G(x)\\ &= \left[F'(x) - \grave{F}(x)D(x;r)^{-1}F'(r)\right]^{-1} F(x),\end{aligned}$$

which gives a convenient expression for the Newton correction.

More generally, assume we have found m roots. Then, defining

$$\sigma_i^{(m)}(x) = \prod_{j=1}^{m}\langle \nabla f_i(r^{(j)}), x - r^{(j)}\rangle,$$

we obtain for the gradient of the ith component of $G^{(k)}$ from (5.24)

$$\begin{aligned}\nabla g_i^{(m)}(x) &= \frac{1}{\sigma_i^{(m)}(x)}\left(\nabla f_i(x) - \frac{f_i(x)}{\sigma_i^{(m)}(x)}\left[\sum_{j=1}^{m}\left(\prod_{\substack{k=1\\k\neq j}}^{m}\langle \nabla f_i(r^{(k)}), x - r^{(k)}\rangle\right)\nabla f_i(r^{(j)})\right]\right)\\ &= \frac{1}{\sigma_i^{(m)}(x)}\left(\nabla f_i(x) - \frac{f_i(x)}{\sigma_i^{(m)}(x)}\left[\sum_{j=1}^{m}\nabla f_i(r^{(j)})\frac{\sigma_i^{(m)}(x)}{\langle f_i(r^{(j)}), x - r^{(j)}\rangle}\right]\right)\\ &= \frac{1}{\sigma_i^{(m)}(x)}\left(\nabla f_i(x) - f_i(x)\sum_{j=1}^{m}\frac{\nabla f_i(r^{(j)})}{\langle f_i(r^{(j)}), x - r^{(j)}\rangle}\right).\end{aligned}$$

Now, defining $D^{(j)}(x; r^{(j)})$ for each $j \leq m$, and $S^{(m)}(x)$ to be the diagonal matrices with entries

$$d_{ii}^{(j)} = \langle f_i(r^{(j)}), x - r^{(j)} \rangle \qquad \text{(elements of the matrix } D^{(j)}(x; r^{(j)})\text{)},$$
$$s_{ii}^{(m)} = \sigma_i^{(m)}(x),$$

and $\grave{F}(x)$ as above, we see that

$$(G^{(m)})'(x) = S^{(m)}(x)^{-1} \left[F'(x) - \grave{F}(x) \sum_{j=1}^{m} D^{(j)}(x; r^{(j)})^{-1} F'(r^{(j)}) \right].$$

Hence, the Newton correction is given by

$$-[(G^{(m)})']^{-1} G^{(m)}(x) = - \left[F'(x) - \grave{F}(x) \sum_{j=1}^{m} D^{(j)}(x; r^{(j)})^{-1} F'(r^{(j)}) \right]^{-1} F(x),$$

which again is a convenient expression for computation.

We present our finite dimensional deflation algorithm as Algorithm 9. For simplicity, we assume that the Newton iteration always converges, and that we are always able to find the number of roots requested by a user — in an implementation, we detect when those assumptions fail, and exit the iteration gracefully. We apply an error-oriented damping strategy as described in the algorithm `GENIB` in Section 2.5 for the Newton iteration. We note that while the `GENIB` algorithm for finding the steplength in Newton iteration for each separate root is affine invariant for each deflated problem, the complete deflation algorithm is not affine invariant with respect to the original problem. We consider constructing an completely affine invariant deflation algorithm to be an interesting future research topic.

Also note that the damping strategy is now based on the simplified Newton updates for the deflated function G, rather than of the original function F. As a consequence, it can happen that the Newton sequence generated does not converge to a new solution of the original problem $F(x) = 0$. As the sequence moves away from previously found roots, the algorithm is led to believe that it is making improvements of its current guess, while in fact, it is moving away from a new solution. Thus, the deflation strategy can subdue convergence for the original problem.

When this happens, the Newton direction ceases to give significant improvements, causing the contraction factors Θ_k to be larger than 1 for acceptable values of λ_k. This means that we will not pass the acceptance test $\Theta_k < 1$. If we deflate the first root found (seeking solutions of the system $G^{(1)}(x) = 0$), and observe that the steplength

prescribed by the damping strategy is less than $\lambda_{\min}$, we switch back to the original problem $F(x) = 0$, and continue the iteration for F. In other words, we stop using deflation, and solve a system where the previously found root is again a solution. We say that we have *inflated* the previously found root.

More generally, assume that we have deflated m roots, so that we are performing Newton iteration on the function

$$G^{(m)}(x) = M(x; r^{(m)}) \cdots M(x; r^{(1)}) F(x),$$

and detect that the steplength prescribed by the damping strategy is less than $\lambda_{\min}$. That is a sign that the Jacobian might have started to become ill-conditioned. Our strategy for selecting which root to inflate is as follows: We compute the Jacobians of the modified deflated functions

$$\begin{aligned} \overline{G}^{(1)}(x; r^{(1)}) &= M(x; r^{(1)}) F(x), \\ &\vdots \\ \overline{G}^{(m)}(x; r^{(m)}) &= M(x; r^{(m)}) F(x), \end{aligned}$$

and compare their condition numbers. The root which has the largest condition number then gets inflated.

We now consider the example given in [BG71], that of finding the roots of the nonlinear cubic-parabolic system

$$f_1(x) = 4x_1^3 - 3x_1 - x_2, \qquad f_2(x) = x_1^2 - x_2, \tag{5.26}$$

where the subscript denotes elements of the vector x. This system has three roots:

$$r^{(1)} = \left(-\frac{3}{4}, \frac{9}{16} \right)^T, \qquad r^{(2)} = (0, 0)^T, \qquad r^{(3)} = (1, 1)^T.$$

Before deflating, we set up a grid of equally distributed initial guesses on the square $\left[-\frac{5}{4}, \frac{5}{4}\right] \times \left[-\frac{5}{4}, \frac{5}{4}\right]$, and inspect what root the guesses converge to, using an error-oriented damped Newton's method. Out of the 160,801 grid-points, the iteration converges successfully for all of them. The domain of attraction for each root is shown in Figure 5.5. The sharp bounds between the green, red, and blue areas correspond to lines where the Jacobian of the system is singular.

Let x_0 denote some grid-point used as an initial guess. Assume $\overline{r}$ is the root that the Newton iteration originally converged to from x_0. To find another root, we then construct the deflated function

$$G(x, \overline{r}) = M(x; \overline{r}) F(x),$$

Algorithm 9 Deflation for finite dimensional systems.

Input: A function $F : \mathbb{R}^n \to \mathbb{R}^n$. Initial guess of a roots, x_0. Number of roots sought, m. Deflation strategy to be applied, $M(x, r^{(k)})$. Minimum allowed steplength $\lambda_{\min}$.

Output: A matrix R, containing the roots of F found.

$x \leftarrow x_0$ ▷ Assign x the initial guess
repeat ▷ Find the first root
  $\Delta \leftarrow -(F'(x))^{-1}F(x)$ ▷ Newton correction
  Find steplength λ using Algorithm ***GENIB****,*
  based on simplified Newton steps of F.
  $x \leftarrow x + \lambda\Delta$ ▷ Take damped Newton step
until `GENIB` has converged ▷ Newton iteration until converged
$r^{(1)} \leftarrow x$ ▷ Store first root found
for $j = 1, \ldots, m$ **do** ▷ Seek $m - 1$ more roots.
  $x \leftarrow x_0$ ▷ Reassign x the initial guess.
  $G(x) := M(x, r^{(j)}) \cdots M(x, r^{(1)})F(x)$ ▷ Form deflated function.
  repeat ▷ Find the first root
    $\Delta \leftarrow -(G'(x))^{-1}G(x)$ ▷ Newton correction
    Find steplength λ using Algorithm ***GENIB****,*
    based on simplified Newton steps of G.
    if $\lambda < \lambda_{\min}$ **then**
      for $l = 1, \ldots, j$ **do** ▷ Find out what root we want to inflate
        $\overline{G}^{(l)}(x) := M(x, r^{(l)})F(x)$ ▷ Form modified deflated function
        $\kappa_l \leftarrow \mathrm{cond}((\overline{G}^{(l)})'(x))$
      end for
      Find p such that $\kappa_p \geq \kappa_l, \forall l$, and inflate the pth root.
      $G(x) := M(x, r^{(j)}) \cdots M(x, r^{(p+1)})M(x, r^{(p-1)}) \cdots M(x, r^{(1)})F(x)$
    else
      $x \leftarrow x + \lambda\delta$ ▷ Take damped Newton step
    end if
  until `GENIB` has converged ▷ Newton iteration until converged
  $r^{(j)} \leftarrow x$ ▷ Store jth root found
end for
Return roots found as a matrix, where each column of R is a root.
return $R \leftarrow \left(r^{(0)} \mid \cdots \mid r^{(m)}\right)$

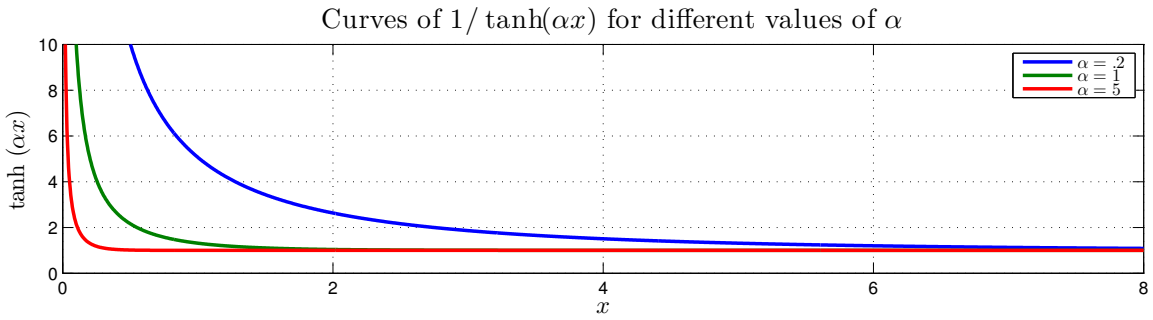

Figure 5.4: Curves of $1/\tanh(\alpha(x))$ for different values of α, showing the range of influence for tanh-deflation strategies.

and apply Newton iteration to G to try to converge to a solution different from $\overline{r}$. We do this for all the points on our original grid. We apply three different choices of deflation matrices:

1. Half norm-squared deflation, with

$$M(x;\overline{r}) = \frac{1}{\frac{1}{2}\left\|x-\overline{r}\right\|^2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}.$$

2. "tanh-deflation", with

$$M(x;\overline{r}) = \frac{1}{\tanh\left(\alpha\left\|x-\overline{r}\right\|^2\right)}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}.$$

   This strategy has the property that the influence of a previously found root does not "tail off" as we move further away from it. The parameter α controls how quickly the deflation strategy goes off to infinity as we approach a previously found solution, as depicted in Figure 5.4.

3. Gradient deflation, with

$$M(x;\overline{r}) = \begin{pmatrix}\dfrac{1}{\langle\nabla f_1(\overline{r}), x-\overline{r}\rangle} & 0\\ 0 & \dfrac{1}{\langle\nabla f_2(\overline{r}), x-\overline{r}\rangle}\end{pmatrix}. \tag{5.27}$$

   When we apply gradient deflation to find three roots, after first finding the root $\overline{r}$ and then $\hat{r}$, we construct the deflated function

$$\hat{G}(x,\overline{r},\hat{r}) = M(x;\hat{r})G(x,\overline{r}),$$

   with $M(x;\hat{r})$ defined similar to (5.27).

Scheme	# Roots sought	# Grid-points	# Converged	Success rate
No deflation	1	160,801	160,801	1.00
Norm-squared deflation, no inflation	2	160,801	50,896	0.32
Norm-squared deflation	2	160,801	100,331	0.62
tanh-deflation, $\alpha = 2$	2	160,801	111,104	0.69
Gradient deflation	2	160,801	158,701	0.99
Gradient deflation	3	158,701	94,536	0.62

Table 5.1: Performance of deflation schemes for the cubic-parabola system (5.26). All schemes, except for the second line, employ the inflation technique as listed in Algorithm 9.

In Table 5.1, we report the success rate of the deflation schemes. We make a few remarks about the results:

- Norm deflation is significantly more successful if we employ the inflation technique described above.
- The tanh-deflation gives better results than standard norm deflation.
- The success rate for gradient deflation is approximately one.

In Figures 5.5–5.10, we show plots of the results, including the domains of attractions for the first root found, and what roots the deflation schemes converged to if they were successful. We make a remark about the black streaks on the figures of the results using gradient deflation. These lines consist of points which share an x coordinate with a root. At such points, the value of the first component of the deflated function is given by

$$g_1(x) = \frac{f_1(x)}{\langle \nabla f_1(r), x - r \rangle} = \frac{4x_1^3 - 3x_1 - x_2}{\begin{bmatrix} 12r_1^2 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ x_2 - r_2 \end{bmatrix}} = \frac{4r_1^3 - 3r_1 - x_2}{r_2 - x_2},$$

where subscripts denote the elements of the vectors r and x. But since r is a root, we have that $4r_1^3 - 3r_1 = r_2$, so $g_1(x)$ must be equal to one. Similarly, $g_2(x) = 0$. Hence, the matrix $\grave{F}(x)D(x;r)^{-1}$ in (5.25) is the identity matrix. Since $F'(x) = F'(r)$ at the grid-points in question, the Jacobian matrix $G'(x)$ is the zero matrix, and the Newton iteration fails.

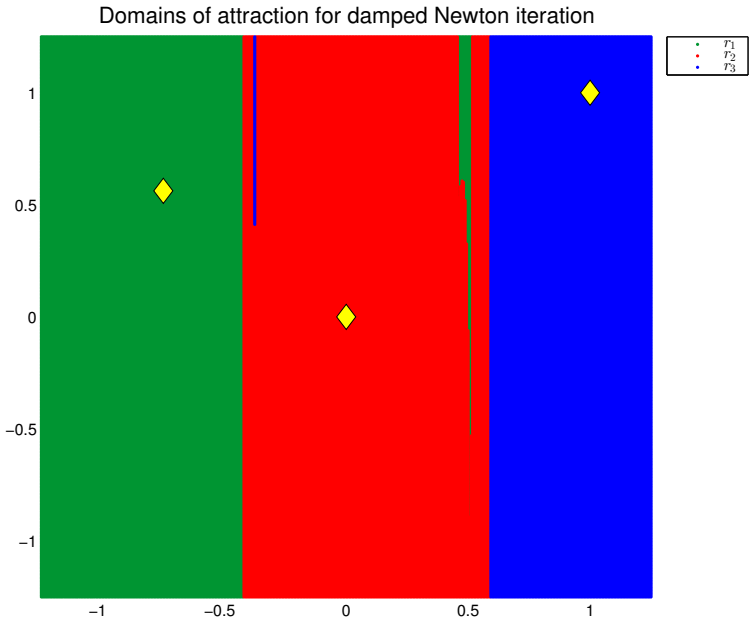

Figure 5.5: Domains of attraction for each root of the nonlinear cubic-parabola system (5.26), showing yellow diamonds at the location of the roots. Green points converged to $r^{(1)} = (-\frac{3}{4}, \frac{9}{16})$, red points to $r^{(2)} = (0, 0)$ and blue points to $r^{(3)} = (1, 1)$.

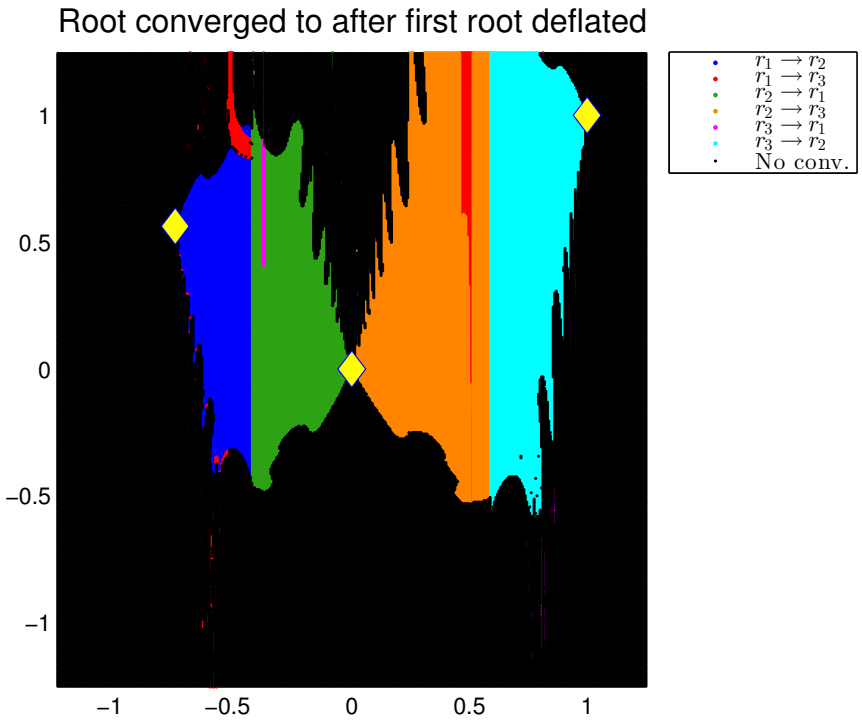

Figure 5.6: Grid-points for which norm deflation, without using inflation, successfully converged to a second root.

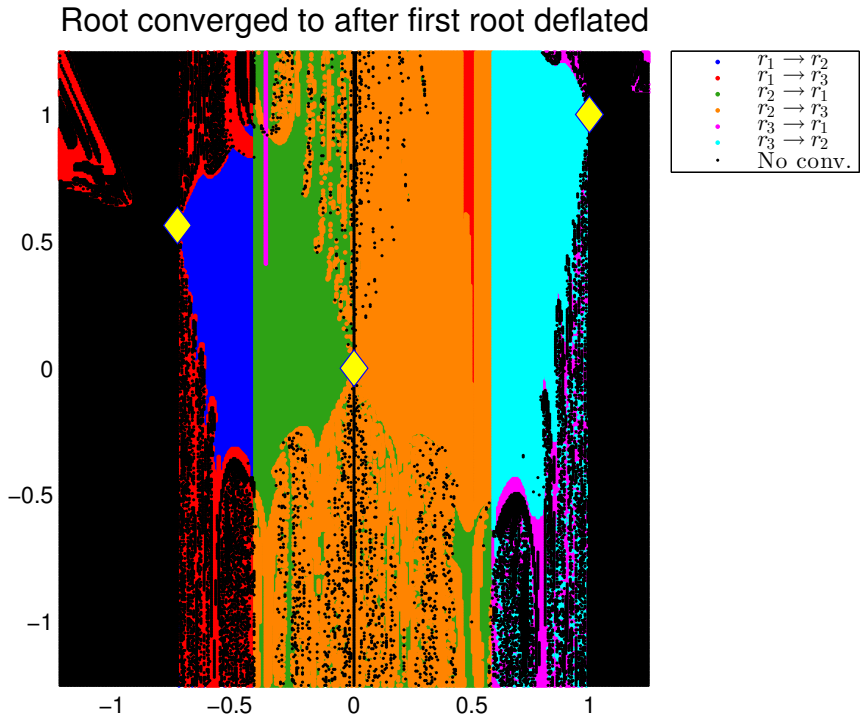

Figure 5.7: Grid-points for which norm deflation, using inflation, successfully converged to a second root.

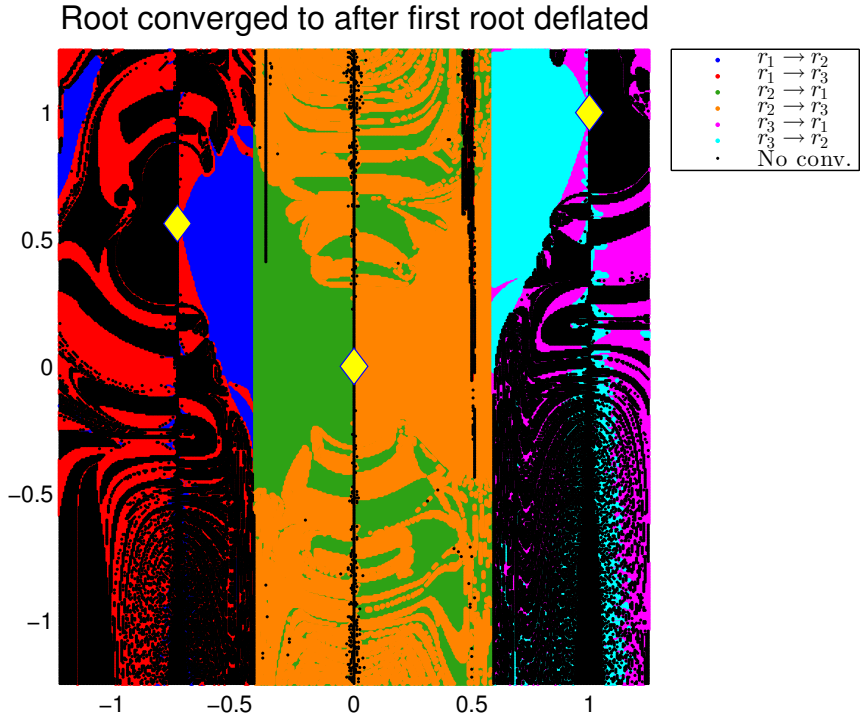

Figure 5.8: Grid-points for which tanh-deflation, with $\alpha = 2$, successfully converged to a second root.

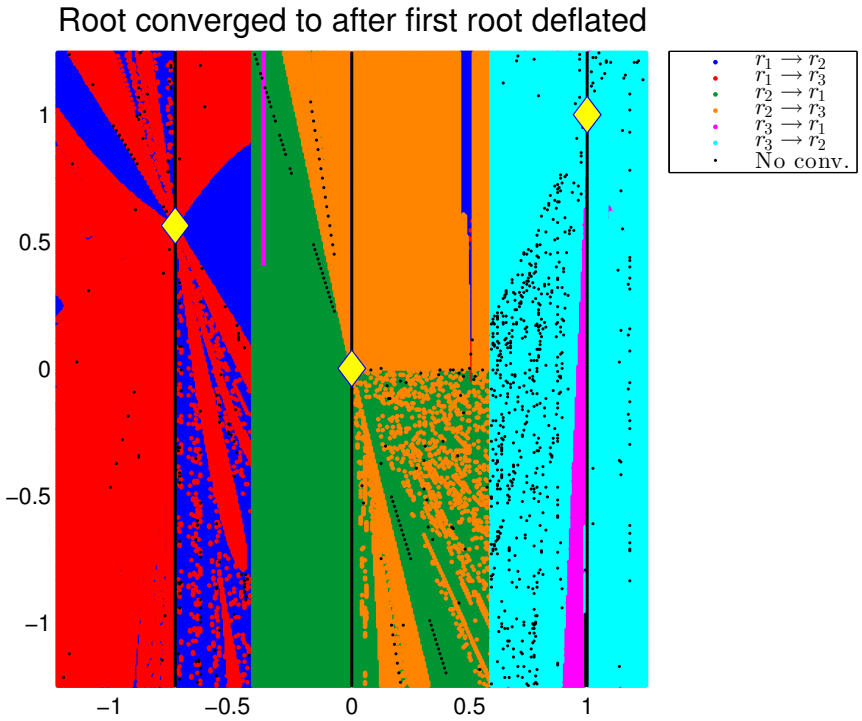

Figure 5.9: Grid-points for which gradient deflation successfully converged to a second root.

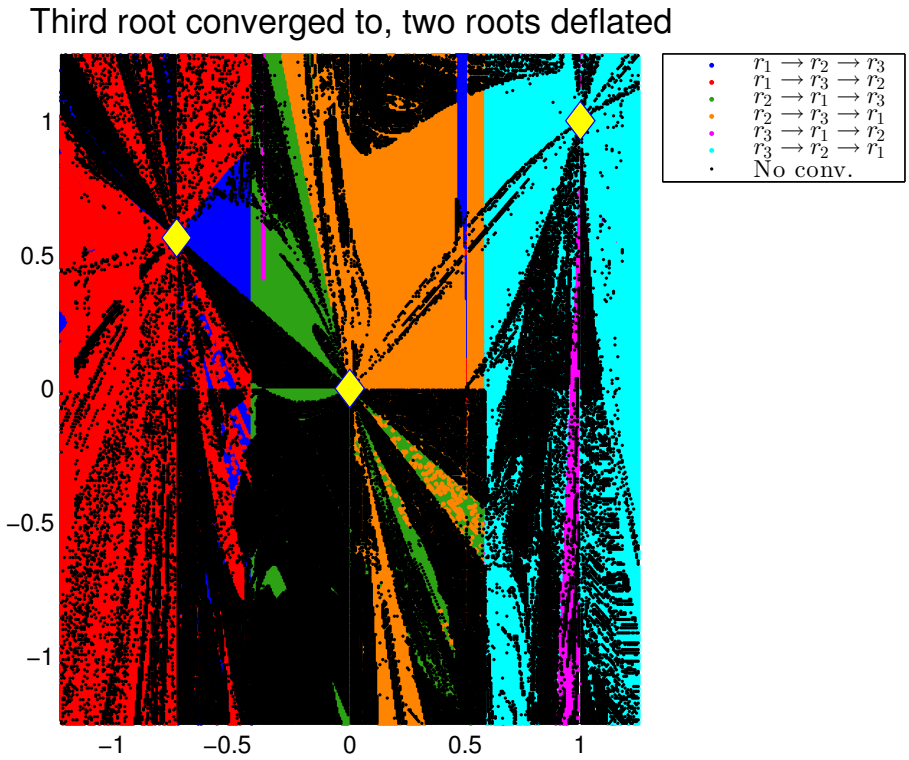

Figure 5.10: Grid-points for which gradient deflation successfully converged to a third root.

5.1.3 Deflation for operator equations

We now shift our focus to deflation in the continuous framework, where vectors become functions, and matrices become linear operators. As in the preceding section, the aim is to construct an algorithm capable of converging to multiple solutions of a nonlinear problem for the single initial guess u_0. Here, we focus on nonlinear scalar BVPs as described in Section 2.1, consisting of a differential equation and conditions to be imposed on the solution (which often are boundary conditions). The initial guess will be the function $u_0(x)$. For concise notation, we write the problem as $\mathcal{F}(u) = 0$, where as before, $\mathcal{F} : U \subset V \to W$ denotes a BVP operator, V and W are Banach spaces, and U is an open subset of V.

Similar to Definition 5.1.1 for deflation matrices, we now define *deflation operators*. We use the notation that when we write $u^{(i)} \xrightarrow{U} v$ for a sequence $\left\{u^{(i)}\right\}$, where $u^{(i)}$ and v are members of $U \subset V$, we mean that $\left\{u^{(i)}\right\}$ converges to v in the norm V is equipped with, that is, $\lim_{i\to\infty} \|u - v\|_V = 0$.

Definition 5.1.3. *Let a Fréchet differentiable operator $\mathcal{F} : U \subset V \to W$, be given. For each $r \in U$, let $\mathcal{M}(u; r) : W \to Z$ be an invertible linear operator, defined for all $u \in U_r$, where U_r is an open subset of U such that r belongs to the closure of U_r. We say that $\mathcal{M}$ is a* deflation operator *if for any $\mathcal{F}$, such that $\mathcal{F}(r) = 0$ and $\mathcal{F}'(r)$ is nonsingular, we have*

$$\liminf_{i\to\infty} \|\mathcal{M}(u_i; r)\mathcal{F}(u_i)\|_Z > 0 \tag{5.28}$$

for any sequence $u^{(i)} \xrightarrow{U} r$, $u^{(i)} \in U_r$. Here, $\mathcal{F}'(r)$ denotes the Fréchet derivative of $\mathcal{F}$, evaluated at r.

Assume that we have found a solution $r^{(1)}$ to the nonlinear BVP $\mathcal{F}(u) = 0$. We then form the nonlinear BVP

$$\mathcal{G}^{(1)}(u) := \mathcal{M}(u; r^{(1)})\mathcal{F}(u),$$

and apply our iterative method for finding a zero of $\mathcal{G}^{(1)}$. By (5.28), any sequence of point converging to $r^{(1)}$ does not give a solution of $\mathcal{G}^{(1)}(u)$, so the sequence generated by Newton's method will stay away from $r^{(1)}$. We say that $\mathcal{G}^{(1)}$ is the nonlinear BVP operator obtained by *deflating out* the solution $r^{(1)}$. If our iterative method is successful in finding a solution of $\mathcal{G}^{(1)}(u) = 0$, we then polish the root by using Newton's method on the undeflated problem, using the solution found for $\mathcal{G}^{(1)}$ as the initial guess.

If we have found m solutions to the original nonlinear BVP $\mathcal{F}(u) = 0$, we define the nonlinear BVP

$$\mathcal{G}^{(m)}(u) := \mathcal{M}(u; r^{(m)}) \cdots \mathcal{M}(u; r^{(1)}) \mathcal{F}(u),$$

and apply our iterative procedure to $\mathcal{G}^{(m)}(u)$ to seek further roots.

As above in the case of deflation for finite dimensional problems, we now have to make a choice of a deflation operator to apply in our problem. We extend Lemma 5.1.2 to the continuous framework.

Lemma 5.1.4. *Let $\mathcal{F} : U \subset V \to W$, be a Fréchet differentiable operator. Suppose that the linear operator $\mathcal{M}(u; r) : W \to Z$ has the property that for each $r \in U$, and any sequence $u^{(i)} \overset{U}{\to} r$, $u^{(i)} \in U_r$, if*

$$\left\| u^{(i)} - r \right\| \mathcal{M}(u^{(i)}; r) w_i \overset{Z}{\to} 0, \tag{5.29}$$

for some sequence $\{w_i\}$, $w_i \in W$, then $w_i \overset{W}{\to} 0$. Then $\mathcal{M}$ is a deflation operator.

Proof. We proceed similar to the proof of Lemma 5.1.2 above. If $\mathcal{M}$ is not a deflation operator as defined by Definition 5.1.3, then there exists a Fréchet differentiable operator $\mathcal{F} : U \subset V \to W$ and an $r \in U$, such that $\mathcal{F}(r) = 0$, $\mathcal{F}'(r)$ nonsingular, and

$$\liminf_{i \to \infty} \left\| \mathcal{M}(u^{(i)}; r) \mathcal{F}(u^{(i)}) \right\| = 0$$

for some sequence $u^{(i)} \overset{U}{\to} r$, $u^{(i)} \in U_r$. Then there exists a subsequence $\left\{v^{(i)}\right\}$ of $\left\{u^{(i)}\right\}$, such that $\mathcal{M}(v^{(i)}; r) F(v^{(i)}) \overset{Z}{\to} 0$. Defining $w^{(i)} \in W$ such that $w^{(i)} = \mathcal{F}(y^{(i)}) / \left\| v^{(i)} - r \right\|$, we have that

$$\left\| v^{(i)} - r \right\| M(v_i; r) w^{(i)} \overset{Z}{\to} 0.$$

From the property of $\mathcal{M}$ stated in the assumptions, we conclude that $w^{(i)} \overset{W}{\to} 0$, that is,

$$\frac{\mathcal{F}(v^{(i)})}{\|v^{(i)} - r\|} \overset{W}{\to} 0. \tag{5.30}$$

Since $\mathcal{F}$ is Fréchet differentiable, we can expand it in Taylor series around r to give

$$\mathcal{F}(v^{(i)}) = \mathcal{F}(r) + \mathcal{F}'(r)(v^{(i)} - r) + o(\left\| v^{(i)} - r \right\|^2) = \mathcal{F}'(r)(v^{(i)} - r) + o(\left\| v^{(i)} - r \right\|),$$

as $\mathcal{F}(r) = 0$. We then have that

$$\frac{F(v^{(i)})}{\|v^{(i)} - r\|} = \frac{1}{\|v^{(i)} - r\|} \left[\mathcal{F}'(r)(v^{(i)} - r) + o(\left\| v^{(i)} - r \right\|) \right] \approx F'(r) \overline{v}^{(i)},$$

where $\overline{v}^{(i)} = \dfrac{v^{(i)} - r}{\|v^{(i)} - r\|} \in U_r$ is a function with a unit norm for all $v^{(i)} \in U_r$. But then, (5.30) leads to a contradiction of the nonsingularity of $\mathcal{F}'(r)$. □

As in the finite dimensional case, if we define the operator $\mathcal{M}(x_i; r)$, and check whether the sequence (5.29) can converge to 0 for any other sequence $\{w_i\}$ other than $w_i \overset{W}{\to} 0$, we can determine whether $\mathcal{M}$ is a deflation operator. Again, this gives rise to a selection of deflation operators, which we call *norm deflation* operators. They all satisfy Lemma 5.1.4, and will be on the form

$$\mathcal{M}(x; r^{(i)}) = \frac{1}{\mu(\|x - r^{(i)}\|)}\mathcal{A},$$

where $\mathcal{A} : V \to V$ is a nonsingular operator, and $\mu : \mathbb{R} \to \mathbb{R}$ is a function such that $\mu(h)$ is $O(h)$. When using the norm deflation operators, the domain of definition for (5.12) will be $U \setminus \bigcup_{i=1}^{k} \left\{r^{(i)}\right\}$. The selection of A and μ can affect the success of our deflation scheme.

One choice of deflation operator we consider is the half norm-squared deflation operator

$$\mathcal{M}(x; r^{(i)}) = \frac{1}{\frac{1}{2}\left\|u - r^{(i)}\right\|^2}\mathcal{I}, \tag{5.31}$$

where $\mathcal{I}$ is the identity operator on W. This operator is the continuous analogue of the choice we made in Section 5.1.2, and related to the half norm-squared functional from Section 3.4.

We seek a convenient expression for the Newton step of the deflated BVP operator when using the deflation operator (5.31), similarly to what we arrived at in (5.16) for the finite dimensional case. We need to be a bit careful when constructing the derivative. Defining the functional η_i as

$$\eta_i(u) := \tfrac{1}{2}\left\|u - r^{(i)}\right\|^2 = \tfrac{1}{2}\langle u - r^{(i)}, u - r^{(i)}\rangle,$$

we have that

$$\eta_i(u + v) - \eta_i(u) \approx \langle u - r^{(i)}, v\rangle$$

to first order. Thus, the Fréchet derivative of η_i is the functional that corresponds to taking an inner product with the function $u - r^{(i)}$, that is,

$$\eta_i'(u)v = \langle u - r^{(i)}, v\rangle = (u - r^{(i)})^T v,$$

where we denote by $(u - r^{(i)})^T$ the transpose of the function $u - r^{(i)}$. Analogously to the finite dimensional case, if we interpret the function $u - r^{(i)}$ as an $\infty \times 1$ dimensional mathematical object, $(u - r^{(i)})^T$ will be of dimensions $1 \times \infty$.

Hence, with

$$\mathcal{G}^{(1)}(u) = \frac{1}{\eta_1(u)}\mathcal{F}(u),$$

we have that

$$
\begin{aligned}
(\mathcal{G}^{(1)})'(u) &= \frac{1}{\eta_1(u)}\mathcal{F}'(u) - \mathcal{F}(u)\frac{1}{(\eta_1(u)^2)}\eta_1'(u) \\
&= \frac{1}{\eta_1(u)}\left[\mathcal{F}'(u) - \frac{1}{\eta_1(u)}\mathcal{F}(u)\eta_1'(u)\right] \\
&= \frac{1}{\eta_1(u)}\left[\mathcal{F}'(u) - \mathcal{F}(u)\frac{(u-r^{(1)})^T}{\eta_1(u)}\right]. \qquad (5.32)
\end{aligned}
$$

In the last line, we have taken the outer product of the functions $\mathcal{F}(u)$ and $u - r^{(1)}$, which results in a rank-1 linear operator. By Taylor testing, as described in Section 3.5, we confirm that the derivatives we compute are indeed the Fréchet derivatives of $\mathcal{G}^{(1)}$.

We now arrive at the Newton correction for the deflated BVP operator, given by

$$
\begin{aligned}
-[(\mathcal{G}^{(1)})'(u)]^{-1}\mathcal{G}^{(1)}(u) &= -\left[\mathcal{F}'(u) - \mathcal{F}(u)\frac{(u-r^{(1)})^T}{\eta_1(u)}\right]^{-1}\eta_1(u)\mathcal{G}^{(1)}(u) \\
&= -\left[\mathcal{F}'(u) - \mathcal{F}(u)\frac{(u-r^{(1)})^T}{\eta_1(u)}\right]^{-1}\mathcal{F}(u).
\end{aligned}
$$

More generally, if we have found m roots of the nonlinear BVP $\mathcal{F}(u)$, $r^{(1)}$, $\ldots$, $r^{(m)}$, we form the deflated BVP operator

$$
\mathcal{G}^{(m)}(u) = \frac{1}{\eta_m(u)\cdots\eta_1(u)}\mathcal{F}(u) = \frac{1}{\eta^{(m)}(u)}\mathcal{F}(u),
$$

with $\eta^{(m)}(u) = \prod_{j=1}^{m}\eta_j(u)$. The Fréchet derivative of $\mathcal{G}^{(m)}$ is given by

$$
\begin{aligned}
(\mathcal{G}^{(m)})'(u) &= \frac{1}{\eta^{(m)}(u)}\mathcal{F}'(u) - \mathcal{F}(u)\frac{1}{(\eta^{(m)}(u))^2}\sum_{j=1}^{m}\left(\prod_{\substack{k=1\\k\neq j}}^{m}\eta_k(u)\right)\eta_j'(u) \\
&= \frac{1}{\eta^{(m)}(u)}\left[\mathcal{F}'(u) - \mathcal{F}(u)\frac{1}{\eta^{(m)}(u)}\sum_{j=1}^{l}\eta^{(m)}(u)\frac{\eta_j'(u)}{\eta_j(u)}\right] \\
&= \frac{1}{\eta^{(m)}(u)}\left[\mathcal{F}'(u) - \mathcal{F}(u)\sum_{j=1}^{l}\frac{\left(u-r^{(j)}\right)^T}{\eta_j(u)}\right].
\end{aligned}
$$

Thus, the Newton step for the deflated operator when m roots have been found is

given by

$$-[(\mathcal{G}^{(m)})'(u)]^{-1}\mathcal{G}^{(m)}(u) = -\left[\mathcal{F}'(u) - \mathcal{F}(u)\sum_{j=1}^{l}\frac{\left(u-r^{(j)}\right)^T}{\eta_j(u)}\right]^{-1}\eta^{(m)}(u)\mathcal{G}^{(m)}(u)$$
$$= -\left[\mathcal{F}'(u) - \mathcal{F}^{(m)}(u)\sum_{j=1}^{l}\frac{\left(u-r^{(j)}\right)^T}{\eta_j(u)}\right]^{-1}\mathcal{F}(u).$$

Another choice of deflation operator we have found to be useful in practice is the tanh-deflation operator, where after finding m roots, we take

$$\mathcal{G}^{(m)}(u) = \frac{1}{\tau_m(u)\cdots\tau_1(u)}\mathcal{F}(u) = \frac{1}{\tau^{(m)}}\mathcal{F}(u),$$

with $\tau_i(u) = \tanh(\alpha\left\|u-r^{(i)}\right\|)$, and $\tau^{(m)}(u) = \prod_{j=1}^{m}\tau_j(u)$. The choice of α allows us to control how close we allow the Newton iteration to approach previously found solutions.

Other options of deflation operators we have considered include

- Norm deflation operators, rather than norm-squared deflation operators. For example, after finding one root r, we form the deflated BVP

$$\mathcal{G}(u) = \frac{1}{\|u-r\|}\mathcal{F}(u).$$

- tanh-deflation operators, similar to discussed in Section 5.1.2. In this scheme, after finding one root r, we form the deflated BVP

$$\mathcal{G}(u) = \frac{1}{\tanh(\alpha\|u-r\|^2)}\mathcal{F}(u),$$

  for some value of α (and the obvious extension once more roots have been found).

Given the success of gradient deflation in Section 5.1.2, it is worth considering whether gradient deflation can also be applied to operators. However, this turns out not to be straightforward. To see why, in gradient deflation, after finding one root r, each component of the deflated function G was given by

$$g_i(x) = \frac{f_i(x)}{\langle\nabla f_i(r), x-r\rangle}.$$

Interpreting $\nabla f_i(r)$ as the jth row of the Jacobian of F, evaluated at r, we have that

$$G(x) = K(x,r)^{-1}F(x),$$

where K is a diagonal matrix with entries of the vector $F'(r)(x-r)$ on the diagonal.

We often consider the continuous setting as the limit when the number of components of a function becomes infinite. Correspondingly, we might try to compute the function $\mathcal{F}'(r)(u-r)$ and the multiplication operator

$$\mathcal{K} : v \mapsto (\mathcal{F}'(r)(u-r))v,$$

and consider the gradient deflated BVP operator

$$\mathcal{G}(u) = \mathcal{K}^{-1}F(x).$$

However, while $w(x) := \mathcal{F}'(r)(u(x) - r(x)) \in L_2[a,b]$, it might happen that $w(x)$ has zeros on the interval $[a,b]$. In that case, $\mathcal{K}^{-1}$ will not be a mapping from L_2 to L_2. Furthermore, any computational representation of $\mathcal{K}$ is basis dependent, and thus its discretisation as well. For example, if we were working with spectral methods, and one of the coefficients of $w(x)$ in the expansion we choose was zero, a discretisation of $\mathcal{K}$ would have a zero on the diagonal, and the matrix would not be invertible. Hence, we abandon gradient deflation in the continuous setting.

The algorithm for deflation for operator equations will be similar to Algorithm 9. Again, the steplength strategy for the damped Newton iteration is based on the algorithm `GENIB`, a globalised error oriented Newton method. If the steplength strategy predicts a value of λ smaller then $\lambda_{\min}$ at the first iteration, we take a moderately sized Newton step (for example, $\lambda = .1$) in the hope that will move the iteration into a domain from which it converges to another solution. When using the tanh-deflation for some α, if the algorithm is unable to find more solutions, we change the value of α and restart the algorithm. If the algorithm appears to be converging to a previously found solution before giving up, we decrease the value of α again, making the Newton iteration more repelled by the previously found roots. If the algorithm gives up because the Newton iteration diverged, we increase the value of α, allowing the Newton iteration to get closer to the previously found solutions.

Like in the finite dimensional case, the Newton method `GENIB` will only be affine invariant with respect to the problem currently being solved as more and more roots are found. The algorithm is thus not affine invariant with respect to the original problem. Again, we consider that to be an interesting future research topic.

We mention that the success of the deflation algorithm for nonlinear BVPs depends on the initial guess. In particular, consider BVPs where the differential equation is autonomous, so that it does not depend explicitly on the independent variable, and has symmetric boundary conditions. Then, if we start with an initial guess that is

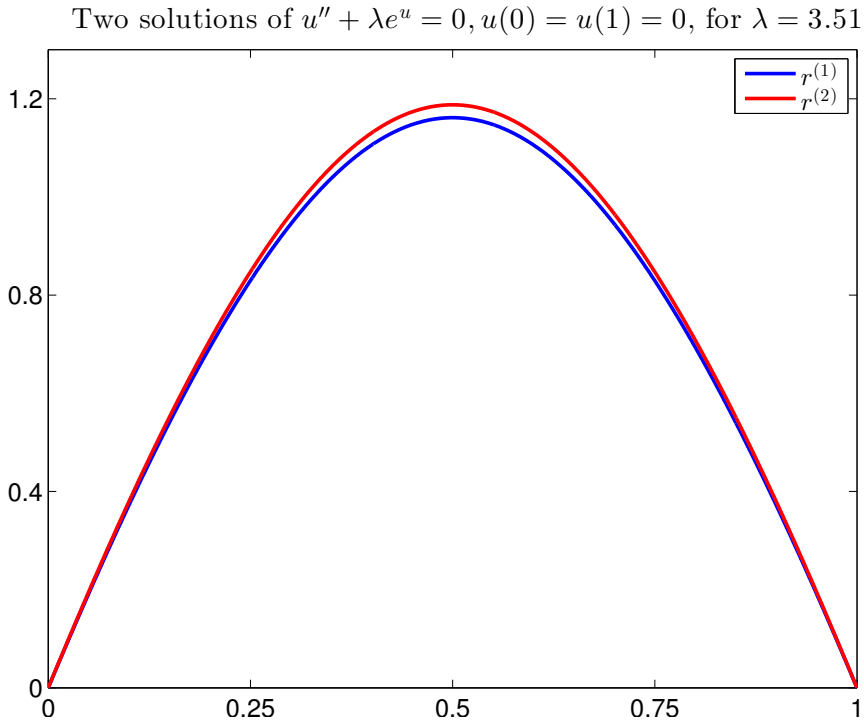

Figure 5.11: Two solutions of the Bratu problem (5.33), obtained using norm-deflation. The blue curve is the first solution obtained, and the red curve is the second one. Both solutions were obtained from the same initial guess, $u_0(x) = 0$.

symmetric about the midpoint of the domain and do not take take measures to break the symmetry, we only expect to find symmetric solutions. If the deflation algorithm is unable to find more solutions for some initial guess, perturbing it slightly to make it asymmetric can lead to further solutions being found. Should the algorithm find an asymmetric solution, we also expect that reflecting the solution around midpoint of the interval gives a solution as well.

We are now in position to apply the deflation algorithm for obtaining multiple solutions of nonlinear BVPs. As we will be working with functions and operators, Chebfun is a natural framework in which to implement the algorithms.

The first problem we consider is the *Bratu problem* [Dav62]:

$$u'' + \lambda e^u = 0, \quad u(0) = u(1) = 0. \tag{5.33}$$

As described in Section 2.6, we know that for values of λ less than the critical value $\lambda_c \approx 3.51383$, (5.33) has two solutions. For $\lambda = \lambda_c$ it has one solution, and no solutions exist for $\lambda > \lambda_c$. Using norm-deflation, we are successful in obtaining both solutions for values of λ up to $\lambda = 3.5138$, using the initial guess $u_0(x) = 0$. We

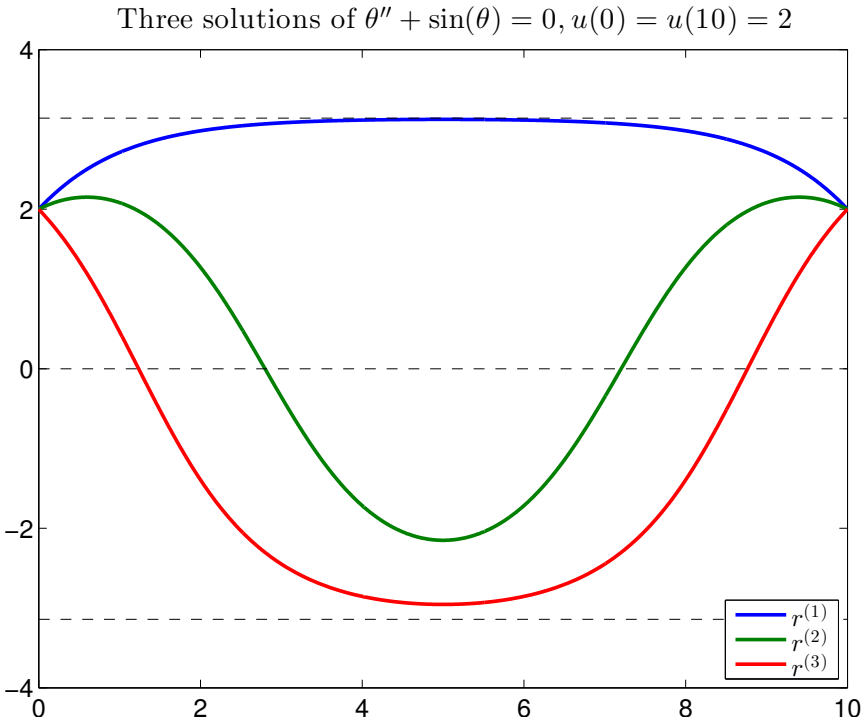

Figure 5.12: Three solutions of the nonlinear pendulum BVP, (5.34), obtained using tanh-deflation. The black dashed line indicated the stable and unstable equilibria of the pendulum at $\theta = 0$ and $\theta = \pm\pi$ respectively.

report that neither norm-squared deflation nor tanh-deflation successfully converges to a second root. In Figure 5.11, we plot the two solutions obtained for $\lambda = 3.51$.

The second problem we consider is that of the nonlinear pendulum we discussed at the start of this chapter,

$$\ddot{\theta} + \sin(\theta) = 0, \qquad \theta(0) = \theta(10) = 2. \tag{5.34}$$

Here, tanh-deflation reveals the existence of one more solution than plotted in Figure 5.1. All three solutions are obtained from the same initial guess $\theta_0(x) = 2$. This new solution corresponds to the pendulum swinging upward for a bit, before going past the stable equilibrium, then returning to the initial position at the final time $T = 10$. In Figure 5.12, we plot the three solutions found.

The third example we consider is the Painlevé I problem [Dav62]

$$u'' - 6u^2 = x, \qquad x(-L) = \sqrt{\frac{L}{6}}, \quad u(0) = 0. \tag{5.35}$$

This problem is known to have two solutions — the "Heaven" solution, which is easy to compute numerically, and most initial guesses converge to, and the "Hell" solution,

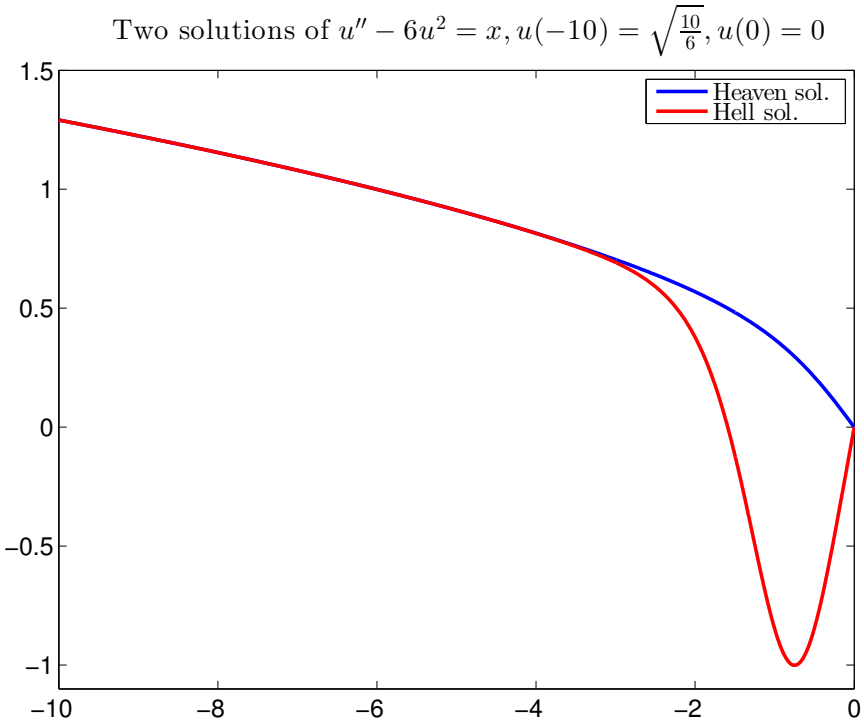

Figure 5.13: The "Heaven" and "Hell" solutions of the nonlinear Painlevé I problem (5.35), obtained using norm-squared deflation.

which is difficult to obtain numerically.[2] Here, we observe that norm deflation is unable to obtain the Hell solution. On the other hand, both norm-squared and tanh-deflation obtain both solutions, starting from the same linear initial guess satisfying the boundary conditions; $u_0(x) = -\sqrt{L/6}\,x$. In Figure 5.13, we plot the Heaven and Hell solutions.

The final problem we consider is a simplified version of the Carrier problem we originally encountered in (3.13), taken from taken from [BO78, Sec. 9.7]. Here, we seek multiple solutions of

$$0.1u'' + u^2 = 1, \qquad u(0) = u(\pi) = 0, \tag{5.36}$$

using tanh-deflation. Starting from the initial guess $u_0(x) = 0$, we find four solutions symmetric around $x = \frac{\pi}{2}$. By then adding a low-degree polynomial perturbation to the initial guess, we find two further solutions. In Figure 5.14, we plot the solutions found.

[2]The author of this thesis learnt about the nicknames of the solutions from André Weideman.

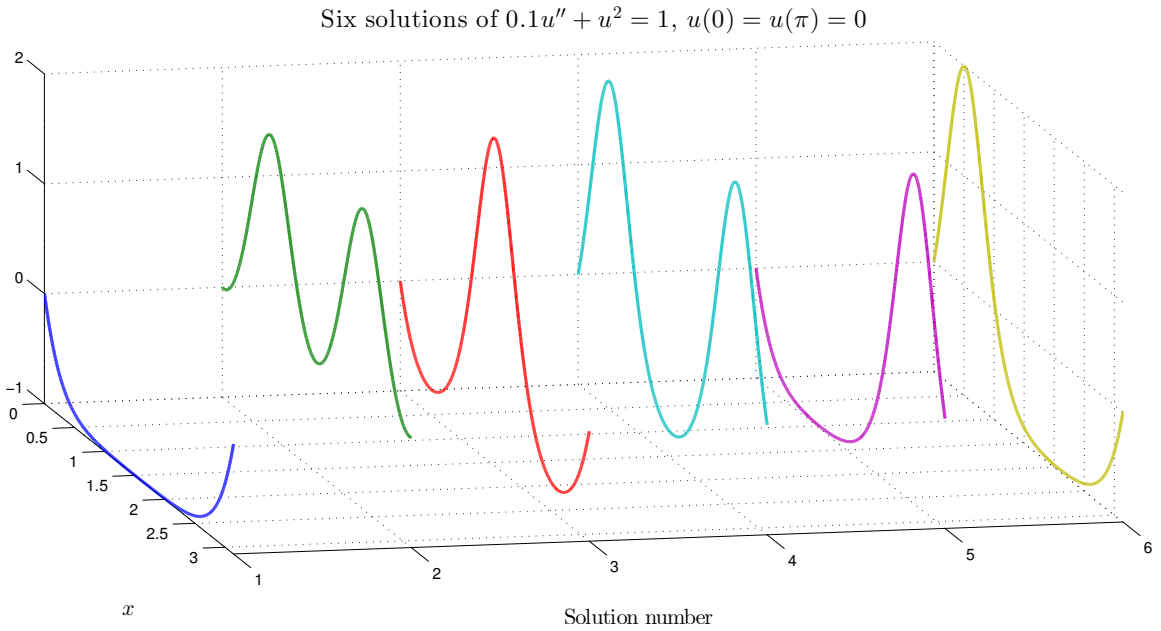

Figure 5.14: Six solutions of the simplified Carrier problem (5.36). The first four solutions were obtained from the same initial guess $u_0(x) = 0$. The last two were obtained by adding a low degree polynomial perturbation to u_0 to start the Newton iteration with an asymmetric initial guess.

5.2 Path following

A popular way in scientific computing to obtain multiple solutions of nonlinear problems are the so-called *path-following* methods. Such numerical continuation methods are a large subject in computational mathematics, and we will only discuss them in the context of how to use such techniques for obtaining multiple solutions of nonlinear ODEs. For an overview of the topic of numerical continuation methods, see [AG03] and references therein. Many standard techniques for path-following are not affine invariant, for an exception, see [Deu06, Ch. 5]. Our treatment of path-following is not in the affine invariant framework, but we regard it as a prospective future research topic. For a discussion of numerical continuation methods in the particular context of ODEs, see for example [AMR95] and [Sey10]. On the software side, we mention that AUTO, developed by Doedel and collaborators [DPC+07], is an established tool for continuation and bifurcation analysis of ordinary differential equations. The MATLAB software project MatCont focuses on the numerical continuation and bifurcation study of continuous and discrete parameterised dynamical systems [DGK03].

5.2.1 Path following for boundary value problems

So far, we have focused on finding solutions for a single differential equation $\mathcal{F}(u) = 0$, where $\mathcal{F}$ is a differential operator that maps from an open subset U of one function space V, to another, W. However, many problems naturally include a parameter λ, in which case, we can consider the differential operator as a mapping from the augmented function space $(U \subset V) \times \mathbb{R}$ to W. In this case, we write the problem of finding a solution of the form

$$\mathcal{H}(u, \lambda) = 0. \tag{5.37}$$

Here, $\mathcal{H}$ stands for a differential system or a BVP operator as described in Section 2.1, encapsulating a differential equation $\phi(u, \lambda) = 0$ (omitting explicit x dependence), and boundary conditions to be imposed on the solution, $\mathbf{B}(u, \lambda) = \boldsymbol{\gamma}$. Studying how a solution u of (5.37) varies with λ is a part of the subject of *bifurcation theory*.

For example, the Bratu problem (5.33), restated here for convenience,

$$u'' + \lambda e^u = 0, \quad u(0) = u(1) = 0, \tag{5.38}$$

is clearly of the form (5.37). As discussed before, there exist two solutions for λ smaller than the critical value $\lambda_c \approx 3.51383$. The solutions u depend on the value of λ, as depicted in Figure 5.15, where we plot both solutions for different values of λ.

Standard treatment of path-following methods, such as that found in [AG03], assumes a discretisation of the problem. However, it is possible to extend the ideas of numerical continuation methods to the continuous setting of Banach spaces. For this, we need the implicit function theorem for maps between Banach spaces:

Theorem 5.2.1 (Implicit function theorem)**.** *[HN01, Thm. 13.22] Suppose that U, V, W are Banach spaces, and $\mathcal{H} : T \subset (U \times V) \to W$ is a continuously differentiable map defined on an open subset T of $U \times V$. If $(u_0, v_0) \in T$ is a point such that $\mathcal{H}(u_0, v_0) = 0$, and $D_u\mathcal{H}(u_0, v_0) : U \to W$ is a one-to-one, onto, bounded linear map, then there is an open neighborhood $\breve{V} \subset V$ of v_0, an open neighborhood $\breve{U} \subset U$ of u_0, and a unique function $f : \breve{V} \to \breve{U}$ such that*

$$\mathcal{H}(f(v), v) = 0 \quad \textit{for all } v \in \breve{V}.$$

The function f is continuously differentiable, and

$$f'(v) = -[D_u\mathcal{H}(f(v), v)]^{-1} D_v\mathcal{H}(f(v), v).$$

Here, $D_u\mathcal{H}(u, v) : U \to W$ and $D_v\mathcal{H}(u, v) : V \to W$ denote the partial Fréchet derivatives of the operator $\mathcal{H}$, evaluated at (u, v), with respect to the first and second arguments respectively.

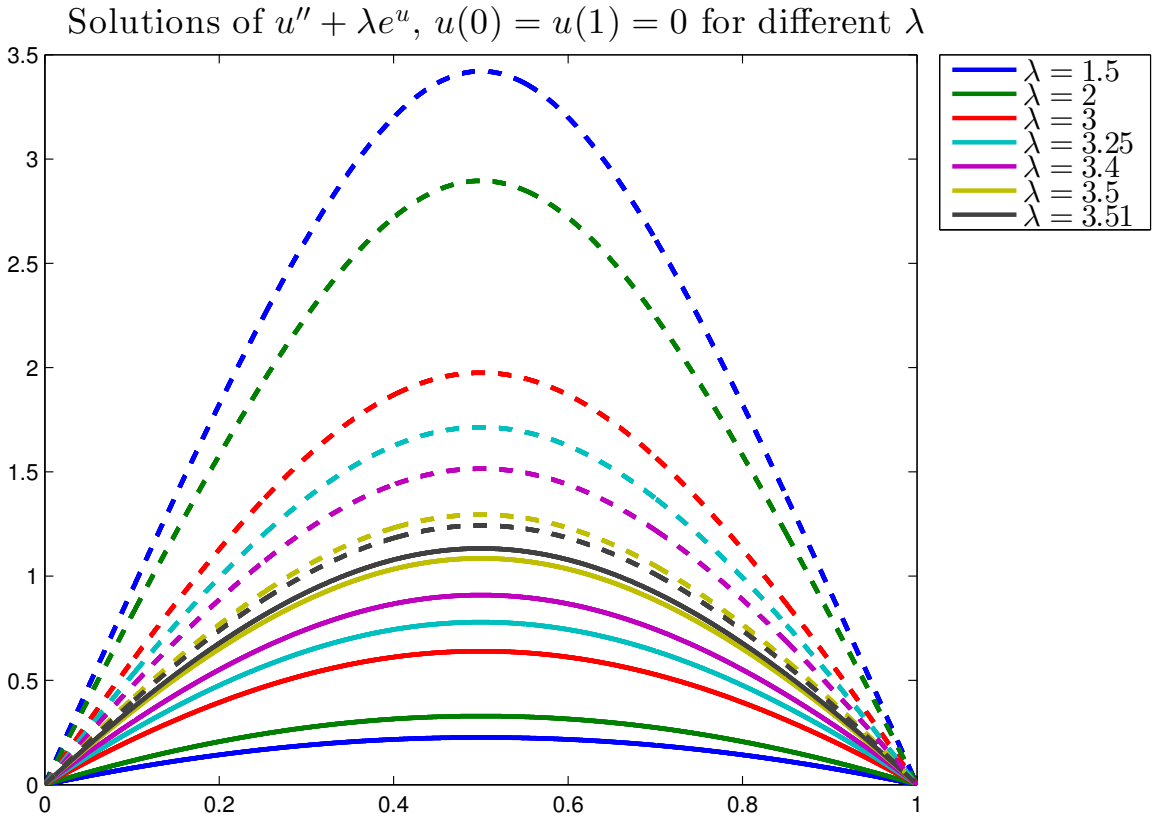

Figure 5.15: Solutions of the Bratu problem (5.38), depending on the value of λ. The solutions plotted with a solid line were obtained using Newton's method starting from the initial guess $u_0(x) = 0$. The dashed lines are solutions for the same values of λ, obtained via Newton's method starting from the initial guess $v_0(x) = 10x(1-x)$.

Following [HN01], we define a *solution branch* or a *solution curve* of (5.37) as a smooth map $\overline{u} : I \subset \mathbb{R} \to V$ from an open interval I in $\mathbb{R}$ into V such that

$$\mathcal{H}(\overline{u}(\lambda), \lambda) = 0, \qquad \text{for all } \lambda \in I.$$

Then, λ_* is said to be a *bifurcation point* of (5.37) from the solution branch $\overline{u}$, if there exists a sequence of solutions $(u_n, \lambda_n) \in V \times I$ such that

$$\mathcal{H}(u_n, \lambda_n) = 0, \qquad u_n \neq \overline{u}(\lambda_n),$$
$$u_n \to \overline{u}(\lambda_*),\ \lambda_n \to \lambda_* \qquad \text{as } n \to \infty.$$

In other words, at a bifurcation point, multiple solutions come together, as we see in Figure 5.16. We will only consider the simplest type of such points, namely *turning points*.

The implicit function theorem implies that if the derivative

$$D_u \mathcal{H}(\overline{u}(\lambda), \lambda) \tag{5.39}$$

is a nonsingular, bounded linear map from U to W, then there is a unique local solution branch. Hence, it is a necessary (but not sufficient) condition for λ_* to be a bifurcation point that $D_u\mathcal{H}(\overline{u}(\lambda), \lambda)$ is singular at $\lambda = \lambda_*$. At such points, the parameterisation of the solution branch with respect to λ breaks down.

The possibility of continuation failing by encountering a turning point indicates that λ is not the ideal choice for parameterising the curve. A common approach to overcome such issues is to parameterise the solution branch with respect to arclength (or pseudo-arclength, see below) s, so that u and λ are considered to be functions of the arclength parameter s, that is, $u = u(s)$ and $\lambda = \lambda(s)$. Then, (5.37) is written as

$$\mathcal{H}(u(s), \lambda(s)) = 0. \tag{5.40}$$

Taking the derivative on both sides of (5.40), one obtains

$$\mathcal{H}_u(u, \lambda)\frac{\mathrm{d}u}{\mathrm{d}s} + \mathcal{H}_\lambda(u, \lambda)\frac{\mathrm{d}\lambda}{\mathrm{d}s} = 0, \tag{5.41}$$

where we have written $\mathcal{H}_u(u, \lambda) := D_u\mathcal{H}(u, \lambda)$ and $\mathcal{H}_\lambda(u, \lambda) := D_\lambda\mathcal{H}(u, \lambda)$. Defining

$$\dot{u} := \frac{\mathrm{d}u}{\mathrm{d}s}, \qquad \dot{\lambda} := \frac{\mathrm{d}\lambda}{\mathrm{d}s},$$

(5.41) becomes

$$\mathcal{H}_u(u, \lambda)\dot{u} + \mathcal{H}_\lambda(u, \lambda)\dot{\lambda} = 0, \tag{5.42}$$

where $\dot{u}$ and $\dot{\lambda}$ satisfy the arclength condition

$$\|\dot{u}\|^2 + \|\dot{\lambda}\|^2 = 1. \tag{5.43}$$

We see that if we have found one solution to (u_0, λ_0) to $\mathcal{H}(u, \lambda) = 0$, then (5.42) defines an initial value problem, often called the *Davidenko differential equation*.

One could imagine tracing out the solution branch by simply integrating the Davidenko differential equation numerically. For reasons why that is not recommended, see [Deu06, Ch. 5]. Instead, a common approach is to trace the curve using the so-called *predictor-corrector* methods, which, starting from one solution point on the curve (u_0, λ_0), generate a sequence of solutions

$$(u_j, \lambda_j) \longrightarrow (u_{j+1}, \lambda_{j+1}).$$

Predictor-corrector methods can be described as methods that approximately follows the exact solution curve. Another set of popular methods for path-following are the so-called *piecewise linear* methods, where instead, one follows exactly a piecewise linear approximation to the curve. For a discussion of such methods, see [AG03].

We use a predictor-corrector method to obtain the next solution point on the curve (u_{j+1}, λ_{j+1}) from (u_j, λ_j) as follows:

1. First, we take a *tangent prediction step* that potentially takes us off the solution curve:

$$(u_j, \lambda_j) \longrightarrow (\overline{u}_j, \overline{\lambda}_j).$$

The tangent step is in the direction (t_j, τ_j), where t_j is a function and τ_j is a scalar that satisfy the coupled linear differential equation from (5.42):

$$\begin{bmatrix} \phi_u(u_{j-1}, \lambda_{j-1}) & \phi_\lambda(u_{j-1}, \lambda_{j-1}) \\ \langle t_{j-1}\,, \cdot \rangle & \tau_{j-1} \end{bmatrix} \begin{bmatrix} t_j \\ \tau_j \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \tag{5.44a}$$

with boundary conditions

$$\begin{bmatrix} \mathbf{B}_u^{(i)}(u_{j-1}, \lambda_{j-1}) & \mathbf{B}_\lambda^{(i)}(u_{j-1}, \lambda_{j-1}) \end{bmatrix} \begin{bmatrix} t_j \\ \tau_j \end{bmatrix} = \mathbf{0}. \tag{5.44b}$$

By ϕ_u, ϕ_λ, $\mathbf{B}_u$ and $\mathbf{B}_\lambda$, we mean the Fréchet derivatives of the components of the BVP operator $\mathcal{H}$ with respect to u and λ. In (5.44), we have relaxed the arclength condition (5.43) to the *pseudo-arclength* condition

$$\langle t_{j-1}, t_j \rangle + \tau_{j-1}\tau_j = 1,$$

which is a linear constraint, based on information about the tangent of the previous iteration. The intermediate value $(\overline{u}_j, \overline{\lambda}_j)$ is then given by

$$(\overline{u}_j, \overline{\lambda}_j) = (u_j, \lambda_j) + \sigma(t_j, \tau_j), \tag{5.45}$$

where σ is the steplength.

2. We then use Newton iteration as a corrector to get us back onto the solution curve:

$$(\overline{u}_j, \overline{\lambda}_j) \longrightarrow (u_{j+1}, \lambda_{j+1}).$$

The Newton update formula at a point (u^c, λ^c) near the curve is given by

$$\mathcal{H}_u(u^c, \lambda^c)\delta u + \mathcal{H}_\lambda(u^c, \lambda^c)\delta\lambda = -\mathcal{H}(u^c, \lambda^c).$$

We wish to move away from the intermediate point $(\overline{u}_j, \overline{\lambda}_j)$ back to the curve in a direction perpendicular to the tangent (t_j, τ_j). As described in [AG03], this ensures that the point we converge to on the curve, (u_{j+1}, λ_{j+1}), minimizes $\big\|(u_{j+1}, \lambda_{j+1}) - (\overline{u}_j, \overline{\lambda}_j)\big\|$. To that end, we impose the orthogonality constraint

$$\langle t_j, \delta u \rangle + \tau_j \delta\lambda = 0.$$

Each Newton correction $(\delta u, \delta\lambda)$ will thus be a solution of the linear coupled ODE

$$\begin{bmatrix} \phi_u(u^c,\lambda^c) & \phi_\lambda(u^c,\lambda^c) \\ \langle t_j\,,\,\cdot\rangle & \tau_j \end{bmatrix} \begin{bmatrix} \delta u \\ \delta\lambda \end{bmatrix} = \begin{bmatrix} -\phi(u^c,\lambda^c) \\ 0 \end{bmatrix}, \tag{5.46a}$$

with boundary conditions

$$\begin{bmatrix} \mathbf{B}_u^{(i)}(u^c,\lambda^c) & \mathbf{B}_\lambda^{(i)}(u^c,\lambda^c) \end{bmatrix} \begin{bmatrix} \delta u \\ \\ \delta\lambda \end{bmatrix} = \boldsymbol{\gamma} - \mathbf{B}(u^c,\lambda^c). \tag{5.46b}$$

Solving (5.46), and adding the Newton correction to the current guess of the solution as we move back to the curve from $(\overline{u}_j, \overline{\lambda}_j)$, we iterate until we have converged back on the solution branch.

Close to the solution curve, we can expect the Newton iteration to enjoy strong attraction properties. Hence, we should anticipate that a local Newton method, such as the algorithm `LENIB` from Chapter 2, converges quadratically, and will only need a few Newton steps. If we detect that the Newton iteration has not converged quadratically in few iterations, we decrease the steplength σ in (5.45), to obtain a new intermediate point $(\overline{u}_j, \overline{\lambda}_j)$, and retry. If in subsequent iterations we observe good convergence of the Newton corrector, we increase the predictor steplength again.

In Algorithm 10, we give an algorithm for path-following for BVPs, which follows the solution curve using a predictor-corrector method.

We now wish to numerically follow the solution branch $\overline{u}(\lambda)$ of the nonlinear BVP $\mathcal{H}(u,\lambda) = 0$. We choose Chebfun as the framework for implementing Algorithm 10. Observe that (5.44) and (5.46) define linear coupled BVPs, where the solution is an "$\infty + 1$" dimensional object, consisting of a function and a scalar. Recent advances in Chebfun, an object class named `chebmatrix`, to be included in a future release, allow efficient solution of such systems.

When following the solution branches, we want to be able to visualise how the solution u evolves with the parameter λ. For that purpose, we make use of *bifurcation diagrams*, which are two dimensional diagrams with the value of λ on the x-axis, and some scalar-valued *monitor functional*:

$$[\cdot] : V \to \mathbb{R}, \quad u \mapsto [u] \in \mathbb{R}$$

of the function u on the y axis. The choice of the monitor functional depends on where in the problem the parameter λ appears; we list our choice of $[u]$ in all bifurcation diagrams shown.

Algorithm 10 Path-following for nonlinear BVPs.

Input: An operator $\mathcal{H}(u, \lambda)$. Initial point on the solution curve (u_0, λ_0) for which $\mathcal{H}(u_0, \lambda_0) = 0$. Maximum number of points to be computed on the curve, m. Minimum and maximum tangent steplengths allowed, $\sigma_{\min}$ and $\sigma_{\max}$. Maximum number of Newton correction steps allowed, $k_{\max}$. Initial tangent steplength, σ.

Output: A solution curve $\mathbf{c}$, or a `flag` denoting that we were unable to trace the curve.

$(t_{-1}, \tau_{-1}) = (0, 0)$ ▷ Want first tangent to be orth. to $(0, 0)$
for $j = 0, \ldots, m$ **do** ▷ Trace the solution curve
  Obtain a tangent (t, τ) *to* $\mathbf{c}$ *by solving*

$$\begin{bmatrix} \phi_u(u_j, \lambda_k) & \phi_\lambda(u_j, \lambda_k) \\ \langle t_{j-1}\,,\, \cdot \rangle & \tau_{j-1} \end{bmatrix} \begin{bmatrix} t_j \\ \tau_j \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \text{and BCs (5.44b).}$$

  $t \leftarrow t/\sqrt{\|t\|^2 + \tau^2}, \quad \tau \leftarrow \tau/\sqrt{\|t\|^2 + \tau^2}$ ▷ Normalise the tangent
  `accept` $\leftarrow$ `FALSE` ▷ Monitors whether point is accepted
  repeat
    $(u^c, \lambda^c) \leftarrow (u_j, \lambda_j) + \sigma(t_j, \tau_j)$ ▷ Take a tangent step of size σ
    $k \leftarrow 1$ ▷ Newton counter
    repeat ▷ Newton iteration to to get back on $\mathbf{c}$
      Obtain a Newton correction $(\delta u, \delta\lambda)$ *by solving*

$$\begin{bmatrix} \phi_u(u^c, \lambda^c) & \phi_\lambda(u^c, \lambda^c) \\ \langle t_j\,,\, \cdot \rangle & \tau_j \end{bmatrix} \begin{bmatrix} \delta u \\ \delta\lambda \end{bmatrix} = \begin{bmatrix} -\phi(u^c, \lambda^c) \\ 0 \end{bmatrix} \text{ and BCs (5.46b).}$$

      $(u^c, \lambda^c) \leftarrow (u^c, \lambda^c) + (\delta u, \delta\lambda)$ ▷ Take a Newton step
      $k \leftarrow k + 1.$ ▷ Update Newton counter
    until $\mathcal{H}(u^c, \lambda^c) \approx 0$ or $k > k_{\max}$
    if $k > k_{\max}$ **then** ▷ Had to use too many Newton steps
      $\sigma = \max(\sigma/2, \sigma_{\min})$ ▷ Try a smaller tangent step
      if $\sigma < \sigma_{\min}$ **then**
        return `flag`: Unable to trace $\mathbf{c}$, $\sigma < \sigma_{\min}$.
      end if
    else
      $\sigma = \min(2\sigma, \sigma_{\max})$ ▷ Good Newton convergence, try increasing σ
      `accept` $\leftarrow$ `TRUE` ▷ Accept (u^c, λ^c)
    end if
  until `accept` = `TRUE`
  $(u_{j+1}, \lambda_{j+1}) \leftarrow (u^c, \lambda^c)$ ▷ Accept next point on the curve
  $\mathbf{c}(j) \leftarrow (u_{j+1}, \lambda_{j+1})$ ▷ Store the new solution point
end for
return c

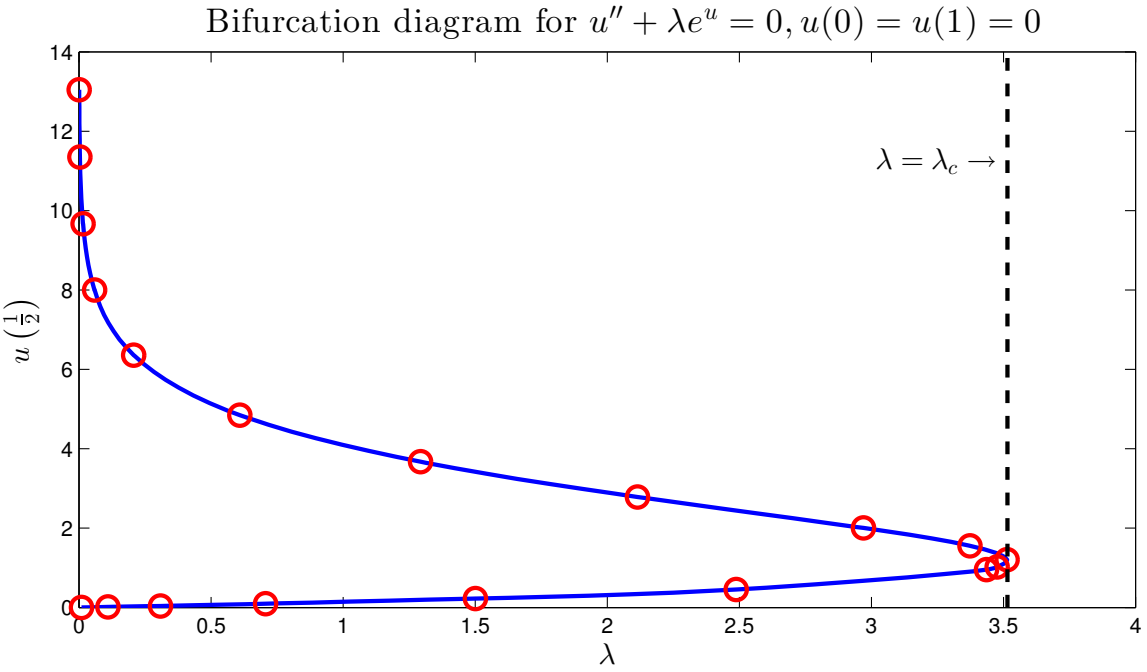

Figure 5.16: Bifurcation diagram for the Bratu problem (5.47). The black dashed line indicates the critical value λ_c. At $\lambda = \lambda_c$, there is a turning point. There exist two solutions for $\lambda < \lambda_c$. For $\lambda = \lambda_c$, there exists one solution, and for $\lambda > \lambda_c$, there are no solutions. The red circles mark points on the solution curve (u, λ), obtained using Algorithm (10). Here, we used $[u] = u\left(\frac{1}{2}\right)$.

The first example we consider in the path-following context is the Bratu problem,

$$u'' + \lambda e^u = 0, \quad u(0) = u(1) = 0. \tag{5.47}$$

Here, the coefficient λ is the obvious path-following parameter. In Figure 5.16, we shown the bifurcation diagram obtained, where we used $[u] = u\left(\frac{1}{2}\right)$. From the figure, it is clear that for values of λ less than the critical value, two solutions exist to the problem $\mathcal{H}(u, \lambda) = 0$.

We note that on the bifurcation diagram shown in Figure 5.16, we have solved a slightly different problem than stated in Section 5.1 on deflation. There, we had a fixed BVP operator $\mathcal{F}$, and sought multiple solutions for the problem $\mathcal{F}(u) = 0$. Here however, we have obtained a sequence of solutions of the problem $\mathcal{H}(u, \lambda) = 0$, for varying values of λ. If we were only interested in solutions for some fixed value of λ, say $\lambda = 2$, only two of the solutions to the problem $\mathcal{H}(u, \lambda) = 0$ are of interest to us.

To address this issue, we reconsider where the parameter λ appears in the problem. This approach is also useful for obtaining multiple solutions to problems where no obvious continuation parameter exists, such as the nonlinear pendulum (5.49).

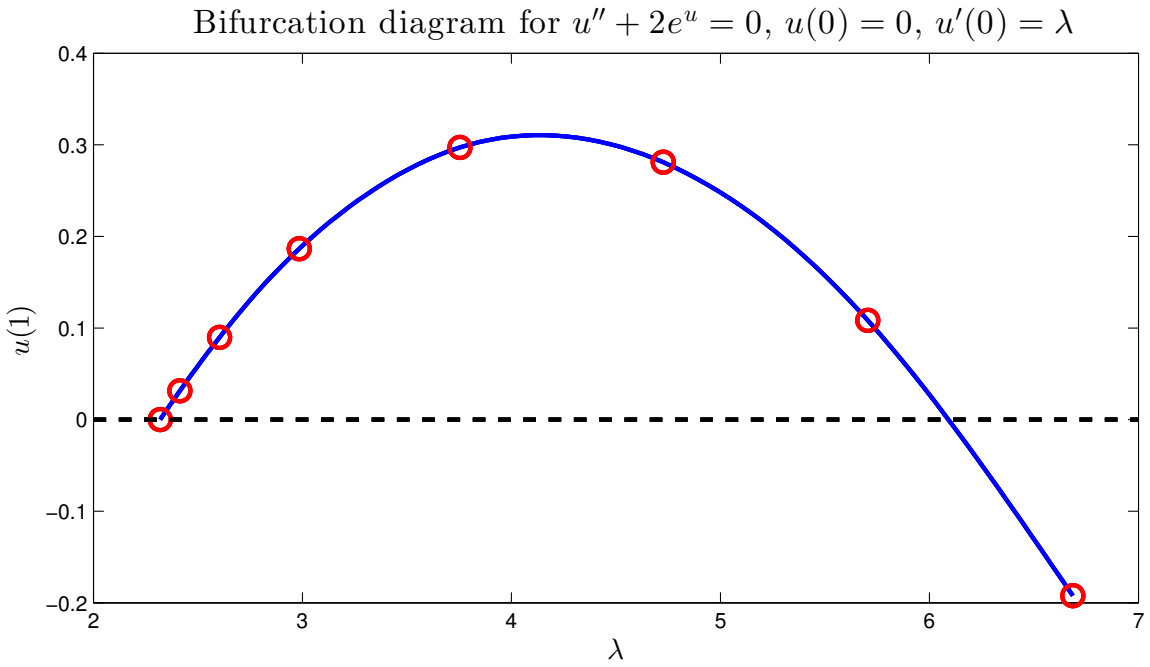

Figure 5.17: Bifurcation diagram for the Bratu problem (5.48). Here, we used $[u] = u(1)$ for the y-axis. Points where the blue solution curve crosses the black dashed line indicate solutions which satisfy the boundary conditions of the original problem, $u(0) = u(1) = 0$.

To demonstrate the alternative approach, let us keep λ fixed in the differential equation of (5.47), for example, $\lambda = 2$. Assume we have found a solution to (5.47) for that value of λ, which we denote by u_0. Then, we apply the path-following algorithm to the problem

$$u'' + 2e^u = 0, \quad u(0) = 0,\ u'(0) = \lambda. \tag{5.48}$$

For the initial value of λ in our path-following algorithm, we take the slope of the first solution u_0 at the left endpoint, that is,

$$\lambda_0 = u_0'(0).$$

In other words, we convert one of the Dirichlet conditions of the original problem to a Neumann condition at the opposite endpoint. Now, the natural monitor functional to plot on the bifurcation diagram is the original Dirichlet condition we converted, in this case, $[u] = u(1)$. To obtain a new solution to the original problem, we pick the solution on the resulting path closest to satisfying the original boundary conditions, and apply a Newton iteration to the original problem using that solution as an initial guess. In Figure 5.17, we show the bifurcation diagram obtained for the problem (5.48).

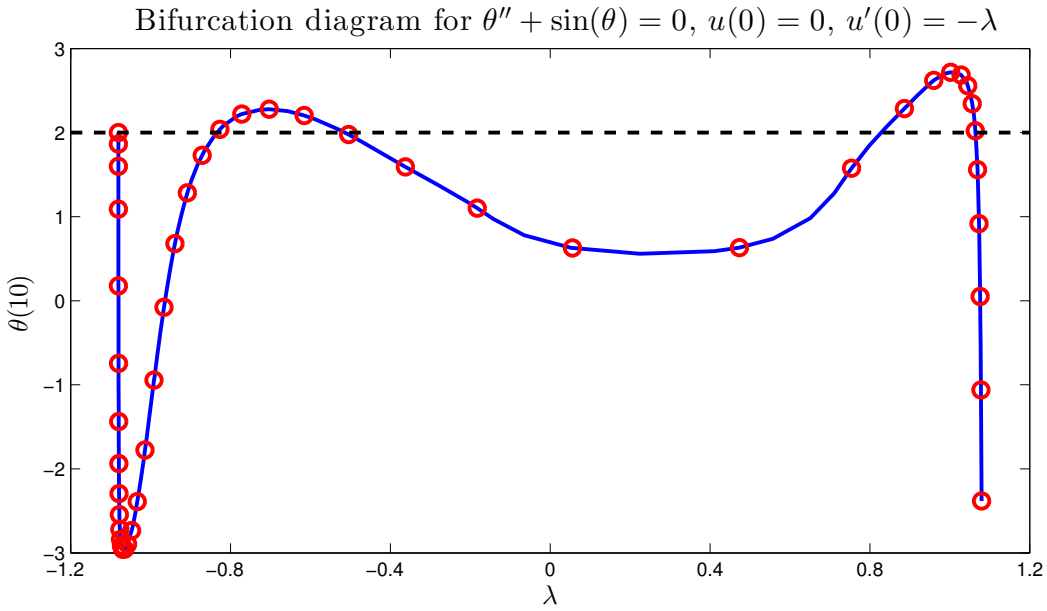

Figure 5.18: Bifurcation diagram for the nonlinear pendulum (5.49). Here, we used $[\theta] = \theta(10)$ for the y-axis. Points where the blue solution curve crosses the black dashed line indicate solutions which satisfy the boundary conditions of the original problem, $\theta(0) = \theta(10) = 2$.

The next problem we consider is that of the nonlinear pendulum,

$$\ddot{\theta} + \sin(\theta) = 0, \qquad \theta(0) = \theta(10) = 2. \tag{5.49}$$

To obtain multiple solutions via path-following, we change the Dirichlet condition on the right endpoint to a Neumann condition at the left endpoint, as described above for the Bratu problem. The resulting bifurcation diagram is shown in Figure 5.18. Here, we observe that the solution curve intersects the line $u(10) = 2$ five times, so there exist two more solutions than we found by deflation. It is not a surprise that we missed these solutions via deflation; they are asymmetric around $t = 5$, so deflating starting from a symmetric initial guess was ineffective. In Figure 5.19, we plot the five solutions to (5.49). They were obtained by starting the Newton iteration with each of the five solution points on the solution branch closest to where it intersects the line $u(10) = 2$.

The next problem we look at is the simplified Carrier equation from Section 5.1:

$$0.1u'' + u^2 = 1, \qquad u(0) = u(\pi) = 0. \tag{5.50}$$

Keeping the value of the solution fixed at the left endpoint, and letting the slope vary, gives the bifurcation diagram shown in Figure 5.20. It reveals that there are

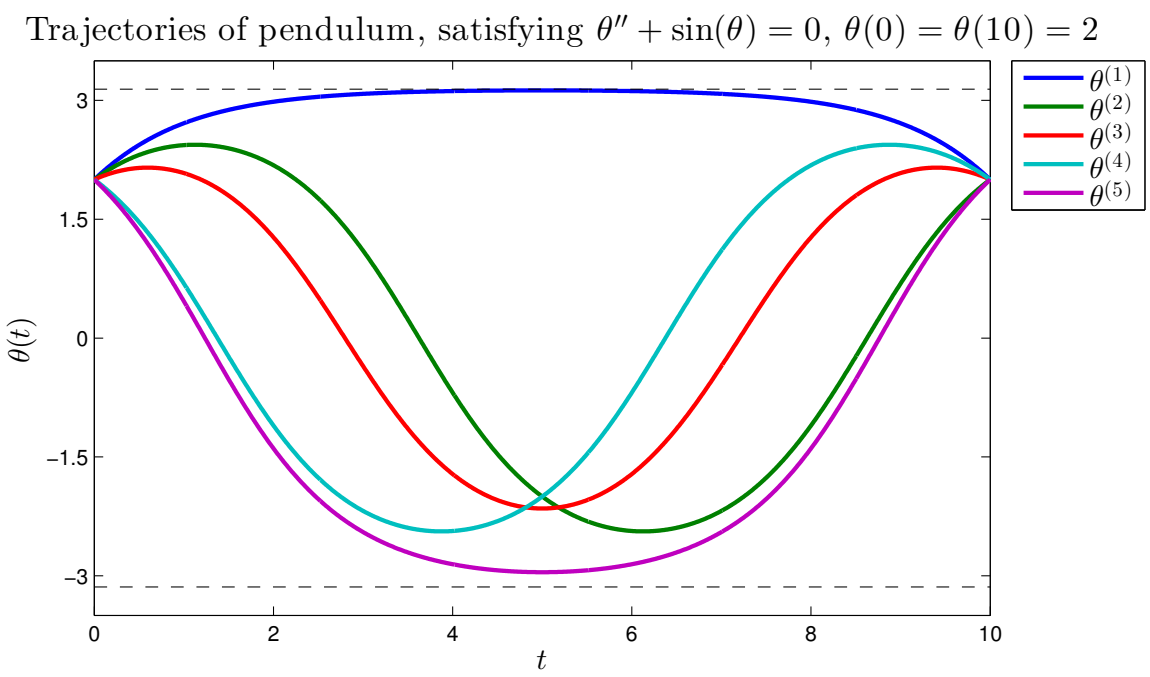

Figure 5.19: Five solutions for the nonlinear pendulum (5.49). We note that we did not obtain the solutions $\theta^{(2)}$ and $\theta^{(4)}$ with deflation, as they are asymmetric around $t = 5$. The black dashed lines indicate the unstable equilibria of the pendulum.

two further asymmetric solutions we did not include in Figure 5.14 when we sought multiple solutions of the problem using deflation. In Figure 5.21, we plot the two new solutions. As expected, they are reflections of one another about $x = \frac{\pi}{2}$.

For the final example, we consider the Painlevé I problem

$$u'' - 6u^2 = x, \qquad u(-L) = \sqrt{\frac{L}{6}}, \quad u(0) = 0, \tag{5.51}$$

for $L = 10$. Here, we keep the value of u at $x = -L$ fixed, and vary the value at $u(0)$, that is, we perform path-following on the problem

$$u'' - 6u^2 = x, \qquad u(-L) = \sqrt{\frac{L}{6}}, \quad u(0) = -\lambda, \tag{5.52}$$

where we take $\lambda_0 = u_{\text{Heaven}}(0) = 0$ as the initial value. We monitor the value of u at the left endpoint, that is, $[u] = u(0)$. Here, we find that the path taken after the turning point lies extremely close to the path towards the turning point. Thus, the solution is very sensitive to small variations in λ, and could easily jump between solutions that bear a resemblance to the Heaven or Hell solutions. In Figure 5.22, we show the resulting bifurcation diagram. In Figure 5.23, we present a few "snapshots" from the path-following, showing the solutions u associated with every point (u, λ) on the solution path shown in the bifurcation diagram, obtained using Algorithm 10.

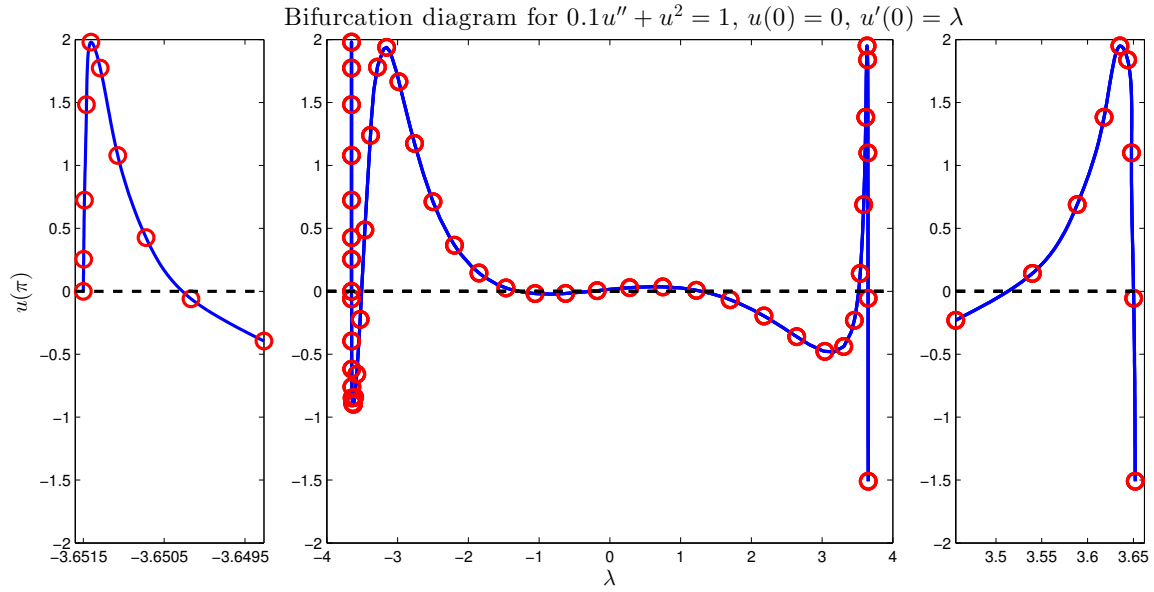

Figure 5.20: Bifurcation diagram for the simplified Carrier problem (5.50). Here, we used $[u] = u(\pi)$. Points where the blue solution curve crosses the black dashed line indicate solutions which satisfy the boundary conditions of the original problem, $\theta(0) = \theta(10) = 2$. Note that $[u]$ is very sensitive to the value of λ for the smallest and largest values of λ shown, so we have zoomed in on those regions in the left and right subplots.

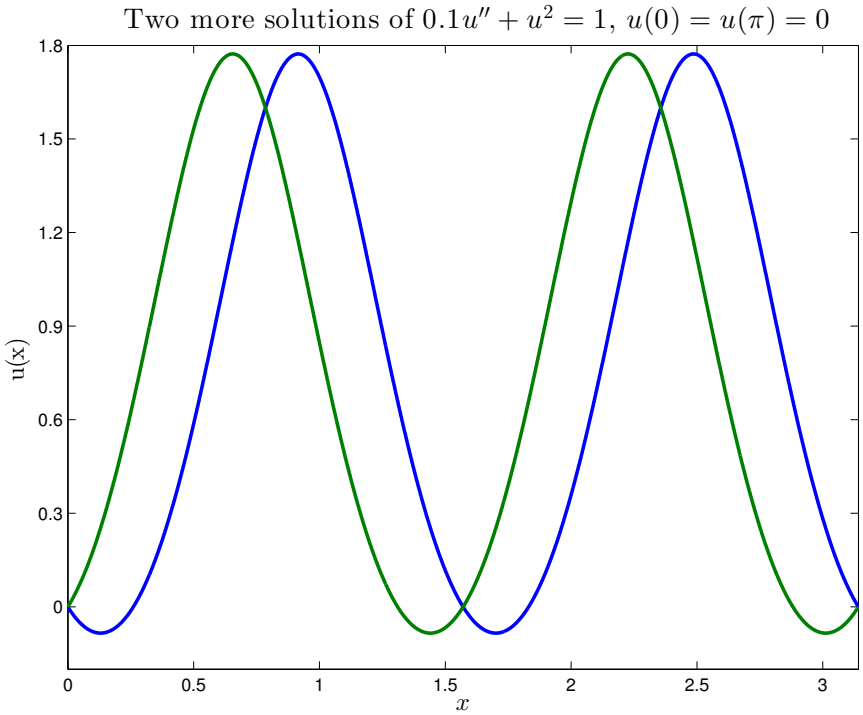

Figure 5.21: Two more asymmetric solutions to the simplified Carrier problem (5.50), which were not part of the solutions obtained through deflation.

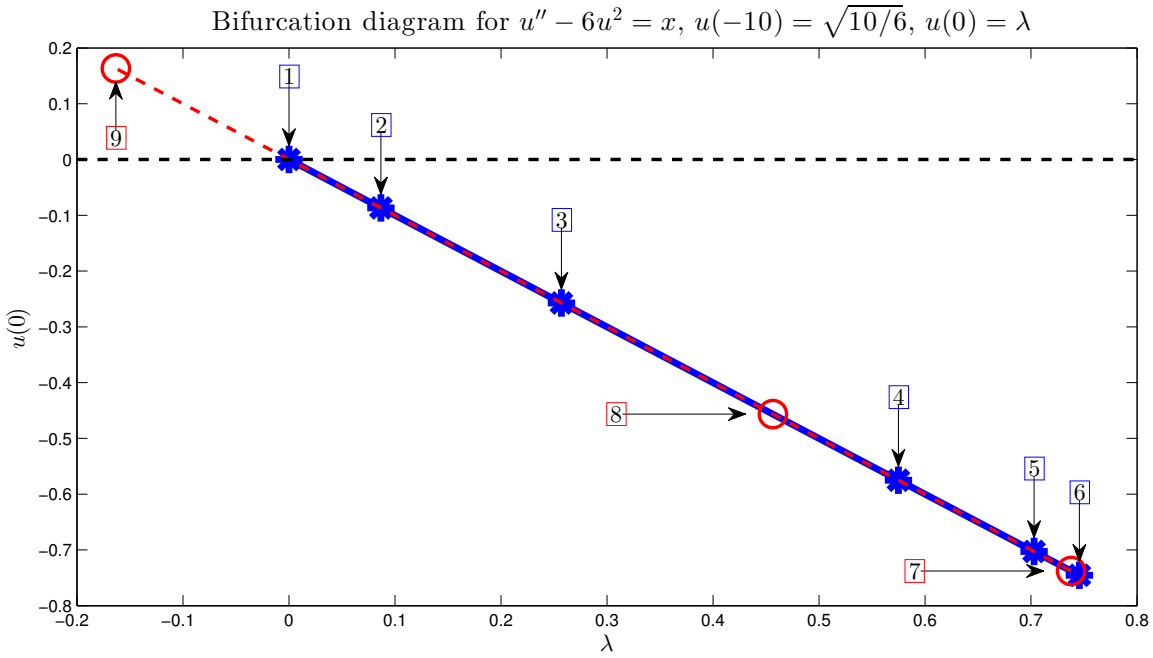

Figure 5.22: Bifurcation diagram for the Painlevé I problem (5.52). Here, we used $[u] = u(0)$. We observe that paths as we first travel from the Heaven solution to the turning point (blue solid line) and then from the turning point to the Hell solution (red dashed line) lie extremely close together. The black dashed line indicates solutions which satisfy the boundary conditions of the original problem, $u(0) = 0$. The numbers on the plot correspond to solutions shown in Figure 5.23.

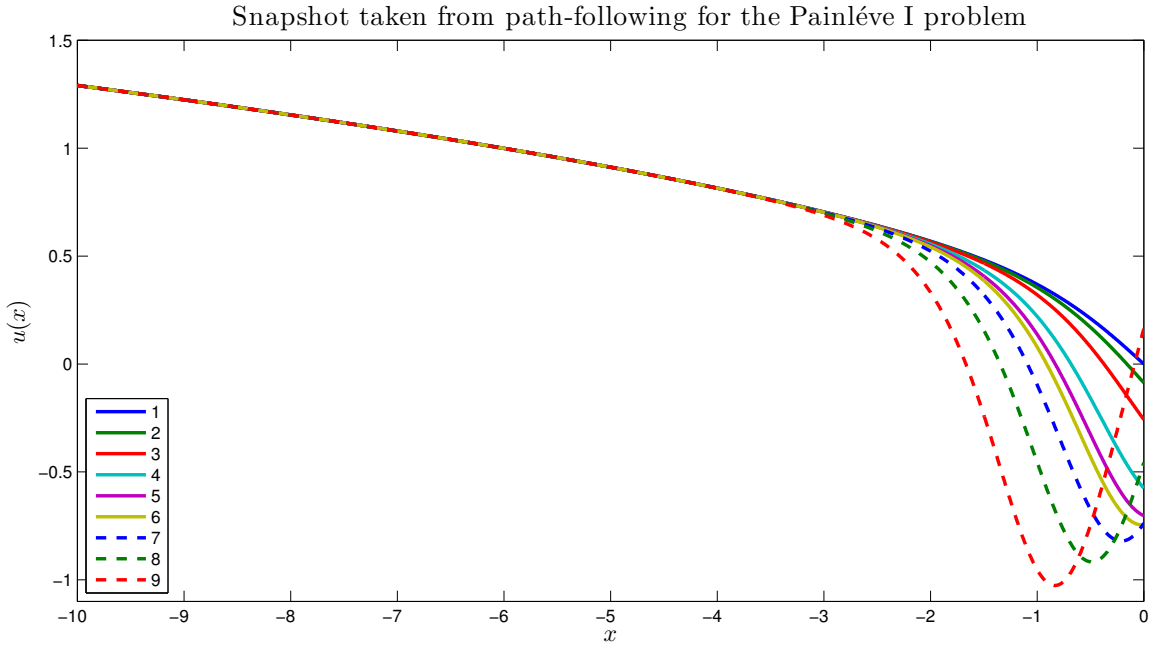

Figure 5.23: Nine snapshots taken from traversing the (u, λ) solution curve for the Painlevé I problem (5.52).

5.3 Comparing and combining deflation and path following

From Sections 5.1 and 5.2, it is clear that both deflation and path-following are useful tools for obtaining multiple solutions of nonlinear BVPs. Both approaches have their benefits and drawbacks:

- Deflation can easily be implemented as a black-box algorithm — there is no need to decide a useful continuation parameter, or to modify the original problem. For instance, had the examples we considered in Section 5.2 had conditions different from Dirichlet conditions, it might not have been feasible to simply convert one of the Dirichlet conditions to a Neumann condition. Also, even though it might be possible to perform continuation on the slope of the solution, it might not be the ideal continuation parameter.

- Deflation can rapidly obtain two or three new solutions to a problem, but should not be relied upon to find larger numbers of new solutions. Path following is more useful when we want to guarantee that we have obtained all solutions lying on a particular solution branch. As the nonlinear pendulum and simplified Carrier problems demonstrated, deflation missed out on solutions that were asymmetric around the midpoint of the domain. However, this can be addressed by breaking the symmetry of the deflated problem, for example by starting with an asymmetric initial guess.

- It is clear that by always starting the deflation algorithm with the same initial guess, no use is made of previously acquired information to aid convergence of the deflated algorithm. Path-following on the other hand traces out curves which depend on the previously obtained solutions.

- Starting from an initial solution on a particular solution branch, path-following can only be expected to find other solutions lying on the same branch. On the other hand, deflation has the potential to reach a different solution branch as it might land on it when seeking other solutions by Newton iteration.

- Path-following can spend a long time in areas where the solution is sensitive to variations in the parameter λ, as was evident in the simplified Carrier problem. Deflation on the other hand goes straight to finding a new solution of the original problem we are interested in.

We conclude that deflation and path-following complement each other as tools for finding multiple solutions. Thus, we propose an algorithm combining both tools, listed as the informal Algorithm 11. It starts by finding a solution to the problem $\mathcal{F}(u) = 0$, then tries deflation. If the number of solutions found by deflation is equal to the number of solutions requested by the user, it returns the solutions found. If the number of solutions found by deflation is smaller than requested, it modifies the problem (for example, by changing one of the boundary conditions) and performs path-following, starting from the first solution found.

If points on the solution curve are found to be close to satisfying the original boundary condition, those points are then used as initial guesses for Newton iteration of the original problem. The algorithm returns the new solutions acquired this way. It finally compares the solutions found from path-following and from deflation. If any solutions obtained by deflation are not on the traced curve (meaning that there might exist another solution curve), it then restarts the path-following from those solutions.

We plan to release our implementations of the algorithms for deflation, path-following, and the combined algorithm in a future version of Chebfun.

Algorithm 11 Algorithm for finding multiple roots of nonlinear BVPs, combining deflation and path-following.

Input: An operator $\mathcal{F}$. Initial solution u_0 for which $\mathcal{F}(u_0) = 0$. Number of roots sought, m.

Output: An "$\infty \times m$" dimensional object $\mathbf{R}$, containing the solutions of $\mathcal{F}(0) = 0$ found.

$k \leftarrow 1$ ▷ Initial number of solutions we know
repeat
  Apply deflation on the equation $\mathcal{F}(u) = 0$, deflating out previous solutions.
  $m_d \leftarrow$ number of new roots found by deflation.
  $k \leftarrow k + m_d$
  if $k < m$ **then**
    Apply path-following to trace curves the previously found roots lie on.
      Start the path-following from every solutions in R not lying on a curve previously traced.
    Apply Newton iteration to $\mathcal{F}$, using points on the curves closest to satisfying original problem as initial guesses.
    $m_p \leftarrow$ number of new roots found by deflation.
    $k \leftarrow k + m_p$
  end if
  Store in $\mathbf{R}$ the new roots found in deflation and path-following steps.
until $k \geq m$ or no new roots found
return R

Chapter 6

Conclusions and Further Work

6.1 Conclusions

In Chapter 1, we gave a review of the background material of this thesis. We started by discussing the problem of polynomial approximation, and how the Lebesgue constant of the nodes selected affects the approximation properties. We saw that for equispaced nodes, the Lebesgue constant grows exponentially, while for Chebyshev nodes,

$$x_j = -\cos\left(\frac{j\pi}{n}\right), \quad 0 \le j \le n,$$

it only grows logarithmically. This is the major reason why interpolation in equispaced points is a bad idea, while interpolation in Chebyshev nodes is a well-behaved process. We then summarised key results about approximation in Chebyshev nodes, and presented Chebfun, a collection of algorithms and an open-source software system in MATLAB, which is based on the useful approximation properties of Chebyshev nodes and polynomials.

In Section 1.2, we discussed spectral methods for numerical solution of differential equations, where the idea is to write the numerical solution u_N as a linear combination of global basis functions

$$u_N(x) = \sum_{k=0}^{N} a_k \varphi_k(x). \tag{6.1}$$

We described two categories of spectral methods; coefficient methods, which operate directly on the coefficients of the expansion (6.1), and collocation methods, which operate on the values of functions on the underlying grid. We saw how differential operators can be discretised on a Chebyshev grid, and how boundary bordering is used to impose conditions on the solution. We then discussed the recent work of Hale and Driscoll about rectangular discretisation of differential operators.

In Section 1.3, we gave an overview of automatic (or algorithmic) differentiation, AD. AD aims at introducing techniques for overcoming drawbacks of traditional finite difference schemes and symbolic differentiation. With AD, it is possible to compute derivatives in scientific computing in a fast, accurate, and robust manner. We gave the definition of the evaluation trace of a mathematical program, and demonstrated how programs are evaluated using simple operations on the so-called intermediate variables. We saw how the evaluation trace could be modified using both forward mode and reverse mode AD to yield derivatives of the outputs with respect to the inputs.

In Chapter 2, we discussed the theory and algorithms of Newton iteration in function space. We gave an overview of some of the theoretical background required, including the definition of Banach and Hilbert spaces, and discussed properties of operators between such spaces. We defined Fréchet derivatives, which are the continuous analogue of Jacobian matrices for maps between Banach spaces.

After giving a historical overview of the Newton's method in Section 2.3, we described Kantorovich' contribution of extending Newton's method for finite dimensional systems to the setting of Banach spaces. Kantorovich proved under the assumptions stated in the *Newton–Kantorovich theorem* that Newton's method in function space converges quadratically for a sufficiently good initial guess. We showed an example of how Newton's method in function space can be implemented in Chebfun to solve a nonlinear BVP.

In Section 2.4, we gave a treatment of the affine invariant framework for Newton's method, originally developed by P. Deuflhard. We investigated the effect of affine transformations on operator equations, and saw that Newton's method enjoys invariance properties. The Newton iterates were found to be invariant under the transformation of the image space, which is known as affine covariance. Furthermore, the Newton iterates are transformed in the same way as the original domain space, which is known as affine contravariance. We then discussed the importance of operator norms and Lipschitz constants for the theorems about convergence of Newton's method, and observed that it was possible to obtain affine invariant Lipschitz constants. The algorithmic paradigm implied that easily available computational estimates of such affine invariant Lipschitz constant are an important tool for constructing practical algorithms for Newton's method.

The two invariance properties of Newton's method lead two different algorithms. Via affine covariance, we derived an error-oriented algorithm, while with affine contravariance, we derived a residual based algorithm. In Section 2.5, we described

how damping could be used to globalise the algorithms, increasing their domains of convergence.

In Chapter 3, we described how AD techniques can be used to supply the Fréchet derivatives required for Newton iteration in function space. We described two different implementations of such an automatic Fréchet differentiation. The first implementation was delayed and recursive, and enabled us to differentiate any function with respect to any other function. The second implementation was similar to a standard forward mode, operator overloaded AD implementation. We then presented the `chebop` class for numerical solution of ODE BVPs. The `chebop` class encapsulates the algorithms of damped Newton iteration, automatic differentiation, and object-oriented programming in MATLAB. It offers a user-friendly syntax, and we gave a number of examples demonstrating its capabilities. We then compared the `chebop` solver with the built-in BVP solvers of MATLAB. Our results indicate that while for low accuracy, the `chebop` solver is slower, execution times are comparable at higher accuracy. Importantly, the `chebop` solver requires much less human time to setup problems than the other solvers, which in many cases will outweigh the computer time taken to solve them.

Most methods for solving nonlinear problems in scientific computing involve an iteration over successive linearisations of the full problem. When implementing a black-box solver for a class of problems, it can be valuable to be able to detect whether a given problem is linear or not, as this determines whether an iteration scheme has to be employed for solving the problem. In Chapter 4, we described how such an automatic linearity detection can be implemented. It is based on Theorem 4.1.6, which states that an operator whose Fréchet derivative is constant, and whose domain is open and connected, is affine.

We described how to modify the evaluation trace when evaluating operators to propagate the *zero derivative* and *Hessian sparsity vectors* z and h, which encode the sparsity patterns of the Fréchet derivatives and Hessians of intermediate variables. Using that technique, it is possible to detect linearity in an automatic way, based on the results of Theorems 4.1.11 and 4.1.12, and Corollaries 4.1.13–4.1.15. We discussed the limitations of such a detection scheme, and argued that the potential of a false negative result (linear operator detected as nonlinear), had much less potential disastrous effect than the occurrence of false positive results (nonlinear operator detected as linear). We described how linearity detection can be incorporated into AD software, both for a standard forward mode, operator overloaded AD implementation and in Chebfun, and gave examples of automatic linearity detection in use.

In Chapter 5, we extended algorithms for finite dimensional systems to the infinite dimensional setting, to derive algorithms for obtaining multiple solution of nonlinear BVPs. First, we discussed the technique of deflation in Section 5.1. Deflation works by modifying the original problem being solved. With deflation, previously found solutions cease to be solutions, so Newton sequences generated by a rootfinding algorithm do not converge to solutions found earlier. We introduced deflation in the context of polynomial rootfinding, before describing it for systems of nonlinear equations. We presented a criterion useful for determining whether a matrix works as a deflation matrix, and investigated different choices of deflation schemes for a nonlinear system of equation. We then extended the idea of deflation to the infinite dimensional setting, and discussed how deflation can be used to obtain multiple solutions of operator equations. Again, we derived a criterion for what operators we can use as deflation operators. We gave a number of examples demonstrating the capabilities of deflation for obtaining multiple solutions of BVPs, including for the Painlevé I problem, which is known to be numerically challenging.

In Section 5.2, we described how numerical continuation methods can be employed for obtaining multiple-solutions of nonlinear BVPs. Here, the idea was to consider the BVP solved as parameter dependent, so that the solution u depended on some parameter λ. This allowed us to consider the differential operator as a mapping from the augmented function space, where each point consists of a function and a scalar. While the standard treatment of such a path-following method assumes discretisation of the problem, the idea could be extended to the continuous framework via the implicit function theorem for maps between Banach spaces. The implicit function theorem implies the existence of solution curves lying in the augmented function space. Rather than starting from an initial point on the curve and following it directly, we generated a chain of solution points using a predictor-corrector method. For the predictor, we used a tangent step, while for the corrector, we employed Newton's method. Again, we gave a number of examples demonstrating how path-following can be used to obtain multiple solutions of nonlinear BVPs.

Finally, in Section 5.3, we compared the deflation and path-following algorithms, and proposed an algorithm combining the two techniques.

The following list summarises the main achievements and novel contributions of this thesis:

- Showing that the affine covariant and affine contravariant Newton methods derived by Deuflhard in [Deu06] could be elegantly implemented in the function space framework, where Fréchet derivatives of nonlinear operators are required.

- Efficiently implemented a delayed, recursive approach to forward mode AD, using the `anon` class for overcoming computational inefficiencies. This enabled us to implement the `chebop` class in Chebfun, which uses AD to allow users a convenient and easy way to apply the affine invariant Newton methods to solving nonlinear BVPs of ODEs.
- Derived the necessary results and implemented linearity detection for computed quantities, and showed examples of the benefits that automatic linearity detection brings to Chebfun.
- Introduced the concept of deflation for operator equations, and derived a criteria for an operator to be a deflation operator.
- Demonstrated through a number of examples that deflation can be useful for obtaining multiple solutions of nonlinear BVPs, including the computationally challenging Painléve I problem.
- We argued that Chebfun is a good framework for implementing path-following in infinite dimensions for obtaining multiple solutions of nonlinear BVPs, and gave a number of examples demonstrating the capabilities of our implementation.
- Derived Algorithm 11 for combining deflation for operator equations and path-following in infinite dimensions.

6.2 Further work

The work in this thesis readily suggests avenues for further investigation and research. Here, we list a few of them:

- It would be very interesting to apply our algorithms and software to a great number of real-life problems. The field of optimal control contains problems that give rise to ODEs, and is one potential source of real-life problems [BH75].
- Another source of interesting real-life problems is in biology and chemistry, where coupled BVPs, involving a significant number of unknown functions, arise [AMR95].
- At the moment, our Newton algorithms do not take into account the effect of discretisation errors for the error estimates of solutions returned. It would be useful to be able to quantify this effect, and return to the user a more accurate error estimate.

- The `chebmatrix` class introduced in this thesis will doubtless open up new possibilities. One area where we hope the `chebmatrix` class will prove to be useful is for problems in the calculus of variations. Another potential area where the `chebmatrix` class might prove to be useful is for problems of ODE constrained optimization — such problems should be possible to naturally express using the `chebop` and `chebmatrix` syntax, which are solvers could then be applied to.

- Our treatment of finding multiple solutions of nonlinear BVPs in Chapter 5 still lacks the desirable affine invariance property.

  - While we use affine covariant Newton method when solving each separate problem for finding a new root, the algorithm is not affine invariant with respect to the original problem. If we were able to derive a deflation algorithm that enjoyed that property, we should expect that the new algorithm would have a higher success rate converging to new roots than our current one.

  - An affine invariant path-following algorithm has already been presented by Deuflhard in [Deu06]. That algorithm uses a Gauss-Newton method for finding the tangent to the solution curve, rather then the idea of pseudo-arclength continutation. In order to be able to implement that algorithm, we need to have the capability of computing the Moore-Penrose pseudo-inverse of the Fréchet derivatives of nonlinear operators. Computing the Moore-Penrose pseudo-inverse of `linop` objects is certainly an exciting topic by itself, and it would be interesting to compare the resulting path-following algorithm to the one we currently have implemented.

Bibliography

[AF03] R. A. Adams and J. J. F. Fournier. *Sobolev Spaces*, volume 140 of *Pure and Applied Mathematics*. Elsevier Science, Amsterdam, 2003.

[AG03] E. L. Allgower and K. Georg. *Introduction to Numerical Continuation Methods*. SIAM, Philadelphia, 2003.

[AH05] K. E. A. Atkinson and W. Han. *Theoretical Numerical Analysis: Functional Analysis Framework*, volume 39 of *Texts in Applied Mathematics Series*. Springer, New York, 3rd edition, 2005.

[ALSU07] A. V. Aho, M. S. Lam, R. Seth, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Addison Wesley, Boston, 2nd edition, 2007.

[AMR95] U. M. Ascher, R. M. M. Mattheij, and R. D. Russell. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, volume 13 of *Classics in Applied Mathematics*. SIAM, Philadelphia, 1995.

[BB00] C. H. Bischof and H. M. Bücker. Computing derivatives of computer programs. In J. Grotendorst, editor, *Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition*, volume 3 of *NIC Series*, pages 315–327. NIC-Directors, Jülich, 2000.

[BBL+02] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and A. Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 65–72, Los Alamitos, 2002. IEEE Computer Society.

[BD97] W. E. Boyce and R. C. DiPrima. *Elementary Differential Equations and Boundary Value Problems*. Wiley, New York, 1997.

[BD12] A. Birkisson and T. A. Driscoll. Automatic Fréchet differentiation for the numerical solution of boundary-value problems. *ACM Trans. Math. Softw.*, 38(4):26:1–26:29, 2012.

[BD13] A. Birkisson and T. A. Driscoll. Automatic linearity detection, 2013. Oxford University Mathematical Institute, Numerical Analysis Group Technical Report no. [13/04].

[BD14] A. Birkisson and T. A. Driscoll. The `advar` class. `http://people.maths.ox.ac.uk/birkisson/advar`, 2014.

[Ber12] S. N. Bernstein. Sur l'ordre de la meilleure approximation des fonctions continues par des polynômes de degré donné. *Mem. Acad. Roy. Belg.*, 4:1–104, 1912.

[BFST03] C. S. Burrus, J. W. Fox, G. A. Sitton, and S. Treitel. Horner's Method for Evaluating and Deflating Polynomials. `http://www.ece.rice.edu/dsp/software/FVHDP/horner2.pdf`, November 2003.

[BG71] K. M. Brown and W. B. Gearhart. Deflation techniques for the calculation of further solutions of a nonlinear system. *Numerische Mathematik*, 16:334–342, 1971.

[BGH69] K. M. Brown, W. B. Gearhart, and H. A. Hall. On the automatic calculation of solutions — different from those previously obtained — of nonlinear systems of equations. Technical Report 69-42, Cornell University, Computer Science, 1969.

[BH75] A. E. Bryson and Y. C. Ho. *Applied Optimal Control: Optimization, Estimation and Control*. CRC Press, 1975.

[BKMC96] C. Bischof, P. Khademi, A. Mauer, and A. Carle. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Comput. Sci. Eng.*, 3(3):18–32, 1996.

[BO78] C. M. Bender and S. A. Orszag. *Advanced Mathematical Methods for Scientists and Engineers*. International series in pure and applied mathematics. McGraw–Hill, New York, 1978.

[Boy00] J. P. Boyd. *Chebyshev and Fourier Spectral Methods*. Dover Publications, New York, 2nd edition, 2000.

[Chi02] J. W. Chinneck. Discovering the characteristics of mathematical programs via sampling. *Optimization Methods and Software*, 17(2):319–352, 2002.

[Cho66] G. Choquet. *Topology*, volume 19 of *Pure and Applied Mathematics*. Elsevier Science, New York, 1966.

[Cia13] P. G. Ciarlet. *Linear and Nonlinear Functional Analysis with Applications:*, volume 130 of *Applied mathematics*. Society for Industrial and Applied Mathematics, 2013.

[CP91] G. F. Carrier and C. Pearson. *Ordinary Differential Equations*, volume 06 of *Classics in Applied Mathematics*. SIAM, Philadelphia, 1991.

[Dav62] H. T. Davis. *Introduction to Nonlinear Differential and Integral Equations*. Dover Publications, New York, 1962.

[DB02] P. Deuflhard and F. Bornemann. *Scientific Computing with Ordinary Differential Equations*, volume 42 of *Texts in Applied Mathematics*. Springer-Verlag, New York, 2002. Transl. by W.C. Rheinboldt.

[DBT08] T. A. Driscoll, F. Bornemann, and L. N. Trefethen. The chebop system for automatic solution of differential equations. *BIT Numerical Mathematics*, 48:701–723, 2008.

[DES82] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton methods. *SIAM J. Numer. Anal.*, 19(2):400–408, 1982.

[Deu06] P. Deuflhard. *Newton Methods for Nonlinear Problems*, volume 35 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, Heidelberg, 2006.

[DGK03] A. Dhooge, W. Govaerts, and Yu. A. Kuznetsov. MatCont: A Matlab package for numerical bifurcation analysis of ODEs. *ACM Trans. Math. Softw.*, 29(2):141–164, 2003.

[DH14] T. A. Driscoll and N. Hale. Rectangular spectral collocation. *SIAM J. Sci. Comp.*, 2014. Submitted.

[DM85] M. Deville and E. Mund. Chebyshev pseudospectral solution of second-order elliptic equations with finite element preconditioning. *Journal of Computational Physics*, 60(3):517 – 533, 1985.

[DNW02] P. Deuflhard, U. Nowak, and M. Weiser. Affine invariant adaptive Newton codes for discretized PDEs. Technical Report ZIB-Report 02-33, Zuse Institute Berlin, 2002.

[DP92] P. Deuflhard and F. Potra. Asymptotic mesh independence of Newton-Galerkin methods via a refined Mysovskii theorem. *SIAM J. Numer. Anal.*, 29(5):1395–1412, 1992.

[DPC+07] E. J. Doedel, R. C. Paffenroth, A. R. Champneys, T. F. Fairgrieve, Yu. A. Kuznetsov, B. E. Oldeman, B. Sandstede, and X. J. Wang. AUTO-07P: Continuation and bifurcation software for ordinary differential equations. `http://indy.cs.concordia.ca/auto/`, 2007.

[Dri10] T. A. Driscoll. Automatic spectral collocation for integral, integro-differential, and integrally reformulated differential equations. *J. Comput. Phys.*, 229(17):5980–5998, 2010.

[Elb06] E. Elbarbary. Integration preconditioning matrix for ultraspherical pseudospectral operators. *SIAM J. Sci. Comput.*, 28(3):1186–1201, 2006.

[FHFR13] P. Farrell, D. Ham, S. Funke, and M. E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM J. Sci. Comput.*, 35:C369–C393, 2013.

[FO10] R. Fourer and D. Orban. DrAmpl: a meta solver for optimization problem analysis. *Computational Management Science*, 7(4):437–463, 2010.

[For96] B. Fornberg. *A Practical Guide to Pseudospectral Methods*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 1996.

[For06] S. A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Trans. Math. Soft.*, 32:195–222, 2006.

[Fox57] L. Fox. *The Numerical Solution of Two-Point Boundary Problems in Ordinary Differential Equations*. Oxford University Press, London, 1957.

[GB12] M. Grant and S. Boyd. CVX: MATLAB software for disciplined convex programming, version 2.0 beta. `http://cvxr.com/cvx`, September 2012.

[GO77] D. Gottlieb and S. A. Orszag. *Numerical Analysis of Spectral Methods: Theory and Applications*. CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia, 1977.

[GP73] G. H. Golub and V. Pereyra. The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate. *SIAM J. Numer. Anal.*, 10(2):413–432, 1973.

[Gre91] L. Greengard. Spectral integration and two-point boundary value problems. *SIAM J. Numer. Anal.*, 28:1071–1080, 1991.

[GT04] N. Gould and P. L. Toint. Preprocessing for quadratic programming. *Math. Program.*, 100:95–132, 2004.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, 2nd edition, 2008.

[HN01] J. K. Hunter and B. Nachtergaele. *Applied Analysis*. World Scientific, Singapore, 2001.

[Hoh94] A. Hohmann. *Inexact Gauss Newton Methods for Parameter Dependent Nonlinear Problems*. PhD thesis, Freie Universität, Berlin, 1994.

[HP13] L. Hascoet and V. Pascual. The TAPENADE automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, May 2013.

[HPC05] V. Hutson, J. S. Pym, and M. J. Cloud. *Applications of Functional Analysis and Operator Theory*, volume 200 of *Mathematics in Science and Engineering*. Elsevier Science, Amsterdam, 2005.

[KA64] L. V. Kantorovich and G. P. Akilov. *Functional Analysis in Normed Spaces*, volume 46 of *International series of monographs in pure and applied mathematics*. Pergamon Press; [distributed in the Western Hemisphere by Macmillan, New York], 1964.

[Kan48] L. V. Kantorovich. On Newton's method for functional equations (in Russian). *Doklady Akademii Nauk SSSR*, 59:1237–1240, 1948.

[Kol92] N. Kollerstrom. Thomas Simpson and Newton's method of approximation: an enduring myth. *The British Journal for the History of Science*, 25:347–354, 1992.

[KS01] J. Kierzenka and L. F. Shampine. A BVP solver based on residual control and the MATLAB PSE. *ACM Trans. Math. Softw.*, 27:299–316, 2001.

[KS08] J. Kierzenka and L. F. Shampine. A BVP solver that controls residual and error. *Journal of Numerical Analysis, Industrial and Applied Mathematics*, 3(1–2):27–41, 2008.

[Lan38] C. Lanczos. Trigonometric interpolation of empirical and analytical functions. *J. Math. Phys.*, 17:123–199, 1938.

[Lax02] P. D. Lax. *Functional Analysis*, volume 55 of *Pure and applied mathematics*. Wiley, New York, 2002.

[LL72] G. E. Ladas and V. Lakshmikantham. *Differential Equations in Abstract Spaces*. Mathematics in Science and Engineering. Academic Press, Inc., New York, 1972.

[Mae54] H. J. Maehly. Zur iterativen Auflösung algebraischer Gleichungen. *Z. Angew. Math. Mech.*, 5(3):260–263, 1954.

[Mys49] I. Mysovskikh. On convergence of Newton's method (in Russian). *Trudy Mat. Inst. Steklov*, 28:145–147, 1949.

[Nau07] U. Naumann. Optimal Jacobian accumulation is NP-complete. *Math. Program.*, 112(2):427–441, 2007.

[Nau12] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*, volume 24 of *Software, Environments, and Tools*. SIAM, Philadelphia, 2012.

[Nei10] R. D. Neidinger. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Review*, 52(3):545–563, 2010.

[NF03] I. P. Nenov and D. H. Fylstra. Interval methods for accelerated global search in the Microsoft Excel Solver. *Reliable Computing*, 9(2):143–159, 2003.

[OR70] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, Orlando, Florida, 1970.

[Ors71] S. A. Orszag. Accurate solution of the Orr-Sommerfeld stability equation. *Journal of Fluid Mechanics*, 50:689–703, 1971.

[Ors80] S. A. Orszag. Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37(1):70–92, 1980.

[Ort69] E. L. Ortiz. The tau method. *SIAM J. Numer. Anal.*, 6:480–492, 1969.

[OT13] S. Olver and A. Townsend. A fast and well-conditioned spectral method. *SIAM Review*, 55:462–489, 2013.

[Pey02] R. Peyret. *Spectral Methods for Incompressible Viscous Flow*. Springer-Verlag, New York, 2002.

[Pol06] B. T. Polyak. Newton-Kantorovich method and its global convergence. *Journal of Mathematical Sciences*, 133(4):1513–1523, 2006.

[Pow81] M. J. D. Powell. *Approximation Theory and Methods*. Cambridge University Press, Cambridge, 1981.

[PTVF07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 3rd edition, 2007.

[PW71] G. Peters and J. H. Wilkinson. Practical problems arising in the solution of polynomial equations. *J. Inst. Maths Applics*, 8:16–35, 1971.

[RF10] H. L. Royden and P. M. Fitzpatrick. *Real Analysis*. Prentice Hall, 2010.

[Rum99] S. M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. `http://www.ti3.tuhh.de/rump/`.

[Run01] C. Runge. Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten. *Zeitschrift für Mathematik und Physik*, 46:465–519, 1901.

[Sey10] R. Seydel. *Practical Bifurcation and Stability Analysis*, volume 5 of *Interdisciplinary Applied Mathematics*. Springer, New York, 2010.

[SH06] M. M. Strout and P. Hovland. Linearity analysis for automatic differentiation. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 574–581, Heidelberg, 2006. Springer.

[SKF05] L. F. Shampine, R. Ketzscher, and S. A. Forth. Using AD to solve BVPs in MATLAB. *ACM Trans. Math. Soft.*, 31:79–94, 2005.

[SM04] L. F. Shampine and P. H. Muir. Estimating conditioning of BVPs for ODEs. *Mathematical and Computer Modelling*, 40(11–12):1309–1321, 2004.

[SS86] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7(3):856–869, 1986.

[ST98] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40(1):pp. 110–112, 1998.

[STW11] J. Shen, T. Tang, and L.-L. Wang. *Spectral Methods*, volume 41 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, Heidelberg, 2011.

[Sut09] W. A. Sutherland. *Introduction to Metric and Topological Spaces*. Oxford University Press, 2009.

[SV90] J. Szabados and P. Vértesi. *Interpolation of Functions*. World Scientific, Singapore, 1990.

[T+11] L. N. Trefethen et al. *Chebfun Version 4.2*. The Chebfun Development Team, 2011. `http://www.maths.ox.ac.uk/chebfun/`.

[Tre00] L. N. Trefethen. *Spectral Methods in* MATLAB, volume 14 of *Software, Environments, and Tools*. SIAM, Philadelphia, 2000.

[Tre13] L. N. Trefethen. *Approximation Theory and Approximation Practice*. SIAM, Philadelphia, 2013.

[Wal08] A. Walther. Computing sparse Hessians with automatic differentiation. *ACM Transaction on Mathematical Software*, 34(1):3:1–3:15, 2008.

[Wel97] B. D. Welfert. Generation of pseudospectral differentiation matrices I. *SIAM J. Numer. Anal.*, 34(4):pp. 1640–1657, 1997.

[WG12] A. Walther and A. Griewank. Getting started with ADOL-C. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, chapter 7, pages 181–202. Chapman-Hall CRC Computational Science, 2012.

[Whi95] J. E. Whitesitt. *Boolean Algebra and Its Applications*. Dover books on mathematics. Dover Publications, 1995.

[WR00] J. A. Weideman and S. C. Reddy. A MATLAB differentiation matrix suite. *ACM Trans. Math. Softw.*, 26(4):465–519, 2000.

[Yam85] T. Yamamoto. A unified derivation of several error bounds for Newton's process. *J. Comput. Appl. Math.*, 12–13:179–191, 1985.

[Ypm95] T. J. Ypma. Historical development of the Newton-Raphson method. *SIAM Review*, 37(4):531–551, 1995.

[Zei85] E. Zeidler. *Nonlinear Functional Analysis and its Applications: Part I: Fixed-point Theorems*. Springer-Verlag, New York, 1985.