2011

Backward Error Analysis as a Model of Computation for Numerical Methods

Nicolas Fillion

The UNIVERSITY *of*
WESTERN ONTARIO

Backward Error Analysis as a Model of Computation for Numerical Methods

(Spine title: Backward Error Analysis as a Model of Computation)
(Thesis Format: Integrated-Article)

by

Nicolas Fillion

Graduate Program in Applied Mathematics

A thesis submitted
in partial fulfillment of the requirements for
the degree of Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

SUPERVISOR	EXAMINERS
Robert M. Corless	Robert W. Batterman
	David J. Jeffrey
	Stephen M. Watt

The thesis by

Nicolas Fillion

entitled:

Backward Error Analysis as a Model of Computation for Numerical Methods

is accepted in partial fulfillment of the requirements for the degree of

Master of Science

Date _____ _____

Chair of the Thesis Examination Board

Abstract

This thesis delineates a generally applicable perspective on numerical methods for scientific computation called residual-based *a posteriori* backward error analysis, based on the concepts of condition, backward error, and residual, pioneered by Turing and Wilkinson. The basic underpinning of this perspective, that a numerical method's errors should be analyzable in the same terms as physical and modelling errors, is readily understandable across scientific fields, and it thereby provides a view of mathematical tractability readily interpretable in the broader context of mathematical modelling. It is applied in this thesis mainly to numerical solution of differential equations. We examine the condition of initial-value problems for ODEs and present a residual-based error control strategy for methods such as Euler's method, Taylor series methods, and Runge-Kutta methods. We also briefly discuss solutions of continuous chaotic problems and stiff problems.

Keywords: BACKWARD ERROR ANALYSIS, CONDITION NUMBER, RESIDUAL, NUMERICAL STABILITY, FLOATING-POINT ARITHMETIC, NUMERICAL SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS, STIFFNESS, LOCAL ERROR, TAYLOR SERIES METHOD, CONTINUOUS RUNGE-KUTTA METHOD

Co-Authorship Statement

This thesis includes three chapters from the forthcoming book co-authored by Robert M. Corless and Nicolas Fillion (hereafter referred to as Corless and Fillion, 201x). The relation of this thesis to the book is roughly the following:

1. Chapter 1 of this thesis will be the first chapter of the book.
2. Chapter 2 of this thesis will be chapter 12 of the book.
3. Chapter 3 of this thesis will be chapter 13 of the book.

The chapters are expected to bear the same titles in the published version of the book. Moreover, some excerpts from the introductory and concluding chapters of this thesis have been borrowed from the preface, introduction, and afterwords of Corless and Fillion (201x).

To a large extent, the material contained in Corless and Fillion (201x) bears the mark of both authors. The first chapter included in this thesis has been written by N. Fillion, but with constant interaction with R.M. Corless. This first chapter is definitely the chapter of Corless and Fillion (201x) that is most distinctively influenced by N. Fillion. The second and third chapters are based on notes used by R.M. Corless when teaching AM9561 or specifically drafted for the book. Nonetheless, the organization of the material and the actual writing has been done by N. Fillion. In particular, N. Fillion has been the main contributor to sections 3.1, 3.2, 3.3.1, 3.3.2, 3.5, 3.6, 4.1, 4.2, 4.3, and 4.6.1-4.6.3. The other sections of chapter 2 and 3 are either of equal contribution (4.4), or mainly contributed by R.M. Corless (3.3.3, 3.4, 4.5, 4.7).

Bibliography

Corless, R. M. and Fillion, N. (201x). A graduate survey of numerical methods. Forthcoming.

Although this may seem a paradox, all exact science is dominated by the idea of approximation.

The Scientific Outlook, Bertrand Russell, 1931

Acknowledgements

This thesis is the result of a long process and it would not have been actualized in any possible world were it not for the constant help, support, and opportunities brought to me by remarkable people who deserve my wholehearted gratitude. My first tip of the hat goes to my supervisor Robert M. Corless. When I started to take courses for the MSc program in 2009, my first course was the graduate numerical methods course taught by Rob. This course turned out to be my most brutal academic experience to this day. Nonetheless, in retrospect, words written by an undergraduate student in his teaching evaluation—"brutal but effective"—come to mind (writing this, I can hear people shout: "Don't encourage him!"). In any case, despite the pain, but perhaps including it, thank you very much Rob for the constant help, support, and opportunities you brought me through those two years. In particular I am extremely grateful for your confidence concerning the book project into which we're now involved.

In academia, it is hard to find someone on whom we can rely for many years. Since I came at Western in 2006, no matter what sort of turbulent flow I got myself into, Bob Batterman has always been there as a fixed point helping to stabilize my academic trajectory. As my co-supervisor for my PhD in the philosophy department, Bob encouraged me to undertake more serious studies in applied mathematics. For that, and for the countless hours of discussions, for the great advices, for moral and material support, and for accepting to be examiner for this thesis, thank you.

I would also like to thank my good friend Robert Moir. Since we met, I believe it was in 2007, I have always appreciated our enjoyable philosophico-mathematical discussions. The heedless optimism that brought us to begin this applied mathematics degree certainly lead us on unexpected paths! Who knew we would ever end up saying publicly that numerical methods are fun and philosophically interesting! Thank you for being a good friend and collaborator.

I consider it both a duty and a great pleasure to thank all my friends, especially the remarkable people in the philosophy department at Western. In particular, I am inordinately indebted to the philosophers of science and mathematics and to the members of the Rotman institute. You make Western an incredibly stimulating environment. Also, I would like to thank my family, and particularly my parents, for their never-ending support and encouragement.

The staff of the philosophy department and of the applied mathematics department at Western deserves much gratitude. In particular, I would like to thank Elisa Kilbourne and Audrey Kager, who have use their smarts to help me go around the red tape in the most agreeable way. Your presence made the administrative tasks a pleasant breeze instead of an abrasive tornado.

For the selfless and honourable sacrifice that being an examiner for this thesis demands, I also thank David Jeffrey and Stephen Watt.

Finally, I thank the Rotman institute for the incomparable office space and material support. Moreover, I thank the Schmeelk Canada Foundation for funding my last two years of study at Western.

Contents

Introduction

Corless and Fillion (201x), a forthcoming book of which this thesis will be a part, is a survey of numerical methods that gathers a wide range of material from floating-point arithmetic, numerical linear algebra, polynomials, function evaluation and root finding, interpolation, numerical differentiation and integration, and numerical solutions of differential equations. Numerical analysis can be succinctly and accurately described as "the theory of constructive methods in mathematical analysis" (Henrici, 1964). A slightly more long-winded definition would also specify that this disciplines develops, studies, and compares efficient numerical *methods* designed to find numerical approximations to the solution of mathematical *problems* stemming from practical difficulties in applications, while quantifying the magnitude of the computation error and qualifying the possible resulting misrepresentation of the system.

But why would a discipline devote energy to *approximate* solutions, instead of developing new methods to find *exact* solutions? The first reason is a pragmatic one, *i.e.*, the urgencies of scientific practice:

> The applications of mathematics are everywhere, not just in the traditional sciences of physics and chemistry, but in biology, medicine, agriculture and many more areas. Traditionally, mathematicians tried to give an exact solution to scientific problems or, where this was impossible, to give exact solutions to modified or simplified problems. With the birth of the computer age, the emphasis started to shift towards trying to build exact models but resorting to numerical approximations. (Butcher, 2008)

A second reason is that, even if we could exactly solve the problems from

applications, it would be practically necessary to resort to modification, uniformization, compression, and simplification of the data and the information specific to the problem. Finally, a third reason is brought about by theoretical necessity. More specifically, mathematicians have produced many impossibility theorems, *i.e.*, they have shown that some types of problems are not solvable, so that there is no computational route that leads to the exact solution. For instance, Abel and Galois showed that it is not possible to solve general polynomial equations of degree five or more in radicals (although there is a less-well-known algorithm using elliptic functions for the quintic itself), Liouville showed that many important integrals could not be expressed in terms of elementary functions (and provided a basic theory to decide just when this could in fact be done), Turing has shown that some number-theoretic problems cannot be finitarily decided, *etc.* With this in mind, Trefethen (1992) claims that the numerical analysts'

> [...] central mission is to compute quantities that are typically uncomputable, from an analytic point of view, and to do it with lightning speed.

Accordingly, both the nature of mathematics in itself and the role of mathematics in science requires a perspective and a theory on numerical approximation to answer the following central *epistemological* question: *when one cannot know the true solution of a mathematical problem, how should one determine how close to the true solution the approximate one is?*[1]

Our guiding principle for the choice of perspective is that numerical methods should be discussed as a part of a more general practice of mathematical modeling as is found in applied mathematics and engineering. Once mostly absent from texts on numerical methods, this *desideratum* has become an integral part of much of the active research in various fields of numerical analysis (see, *e.g.*, Higham, 2002; Enright, 2006). However, while this thesis focuses on

[1] As a philosopher by training, I cannot help but noticing the similarity of this question with the great questions that moved philosophers and scientists through the ages.

applicable computational mathematics it will not present many actual applications.

This thesis, as well as the book Corless and Fillion (201x), is based on a perspective on the quality of numerical solution known as *backward error analysis*, which is seen as very accurately arriving at the aforementioned desideratum. The first use of backward error analysis is credited by Wilkinson (1971) to Givens, but it is broadly agreed that it was Wilkinson himself who began the systematic exploitation of the idea in a broad collection of contexts.[2] As construed by Wilkinson and his followers, this approach gives a crucial role to the theory of *conditioning or sensitivity of a problem*, which originated in the works of Turing (Blum, 2004).[3] The basic underpinning of the backward error perspective, that a numerical method's errors should be analyzable in the same terms as whatever physical (or chemical or biological or social or what-have-you) modelling errors, is readily understandable across all fields of application. Similarly, the concept of sensitivity of a problem to changes in its data is also one that goes across disciplines.[4] These ideas will be introduced with full generality in the first chapter, resulting in an approach that Corless and Fillion (201x) call a residual-based *a posteriori* backward error analysis, that provides mathematically tractable numerical solutions readily interpretable in the broader context of mathematical modelling.

The pedagogical problem that justifies the existence of this work is that, even though many excellent numerical analysis books exist, no single one of them provides a unifying perspective based on the concept of backward error analysis. The objective is to provide the reader with a perspective on scientific computing that provides a systematic method for thinking about numerical solutions and about their interpretation and assessment, across the subfields of numerical analysis. Accordingly, this thesis is mostly about making *a new*

[2]The first synthesis of this point of view is probably Wilkinson (1963). However, until the last two decades, it was mostly confined to numerical linear algebra.

[3]In this respect, an interesting historical fact is that Wilkinson began his career as a research assistant under Turing (Hodges, 1992).

[4]A nice succinct presentation of these ideas in an elementary context can be found in Corless (1993).

synthesis of already known results. The full-fledged use of the conceptual apparatus of backward error analysis is nowadays standard in numerical linear algebra. In the last two decades, it has also been progressively implemented for the numerical solution of differential equations.[5] However, it is almost never encountered at a level of generality that shows what elements are common to all particular applications and how to apply them to various other subfields of numerical analysis, such as floating-point arithmetic, function evaluation, series manipulation, interpolation, *etc.*[6] In this thesis, the most important thing is the development and the presentation of residual-based backward error analysis as a general perspective on computation, in order to show the essential unity of the subject. Naturally, given the limited length of this thesis, it will *not* be possible to show how the perspective developed applies generally. However, this is what Corless and Fillion (201x) aim to achieve, and this thesis, particularly chapter 1, serves as an canvas for that.

In chapter 1, we motivate the need for an error analysis of numerical computation by looking at floating-point arithmetic. Appendix A provides a brief presentation of what a floating-point number system is. We begin with floating-point arithmetic since we consider it logically primary; indeed, it is the underlying ground on which lies almost all computer-assisted numerical computation. Moreover, beginning in this way puts at our disposal a technically simple theory that we can use to introduce the central concepts of error analysis without them being obfuscated by the internal difficulties of other fields of numerical analysis. Chapter 1 then goes on to introduce the central concepts of the residual-based *a posteriori* backward error analysis we promote. It successively introduce the concepts of forward and backward error, residual, and condition number. The very general definition of residual given here appears to be new. We think that it sheds light on many aspects of the

[5] See Moir (2010) for a review of the recent literature on the use of backward error analysis for the numerical solution of differential equations.

[6] A book not so far from doing this is Higham (2002), and the pair of books by Deuflhard and Bornemann (2002) and Deuflhard and Hohmann (2003) does exactly that. However, Deuflhard and his co-workers refrain from the use of the concept of residual, which is here taken to be central. Higham does not refrain from using it, but does not define it generally and apply it across the board.

literature. Finally, chapter 1 introduces properties of numerical algorithms such as stability and complexity. In this chapter as in the rest of the thesis, we make sure to present the material in a way that respects the distinction between properties of *problems* and properties of *methods*—a distinction often hard to trace in the literature.

Because of the limitation of space, Robert Corless and I had to decide on one type of problem for which we would show thoroughly how the backward error perspective developed can be applied. We decided upon the numerical solution of differential equations, specifically initial-value problems for ordinary differential equations (IVP). Accordingly, chapter 2 embarks on a presentation of the *problem-specific* aspect of it, while we reserve the *method-specific* aspect of it for chapter 3.[7] In chapter 2, we want to be able to use and obtain numerical solutions of IVPs without going into the details of particular methods. Accordingly, we open the chapter with an introduction to the use of MATLAB's codes and to the notation that we will require in the rest of the thesis. On that basis, we introduce the concept of residual and point at a simple connection with ideas from dynamical systems. Following this is the examination of the conditioning of IVPs from three perspectives: in terms of Lipschitz constants, in terms of the variational equations and Lyapunov exponents, and finally in terms of the Alexeev-Gröbner theorem. We also show how to track the condition number of problems numerically. After this abstract presentation, we exemplify how the concepts apply in practice in an extended example, the three-body problem. We close chapter 2 by discussing in what sense numerical methods can and can't be considered satisfactory for chaotic problems and, what is in some sense dual, what problem-specific properties contribute to the numerical phenomenon known as "stiffness."

Chapter 3 examines particular numerical methods from our general perspective. Our use of the concept of residual to characterize the properties of problems is extended to the method-specific concepts of error analysis such as convergence, consistency, order of a method, local and global error. We introduce these concepts by discussing Euler's method in a way that emphasizes

[7]These are the chapters 12 and 13 of Corless and Fillion (201x).

residual control as an error control strategy. We also briefly return to stiffness and sketch an argument that the success of implicit methods for stiff problems can be enlightened by examining the size of the residual of Taylor polynomials for the approximation of exponential growth. The way in which we introduce problem-specific and method-specific concepts of error analysis for the numerical solution of IVPs, however, demands that one treats numerical methods for the solution of differential equations as ones that produce differentiable solutions, and not merely a discrete set of solution values. We then introduce Taylor series method as a natural implementation of the idea that numerical methods provide piecewise continuous solutions. Thereafter, we introduce discrete Runge-Kutta methods and indicate how to move to continuous explicit Runge-Kutta methods from there.

Bibliography

Blum, L. (2004). Computing over the reals: Where Turing meets Newton. *Notices of the American Mathematical Society*, 51(9):1024–1037.

Butcher, J. (2008). Numerical analysis. *Journal of Quality Measurement and Analysis*, 4(1):1–9.

Corless, R. M. (1993). Six, lies, and calculators. *The American Mathematical Monthly*, 100(4):344–350.

Corless, R. M. and Fillion, N. (201x). A graduate survey of numerical methods. Forthcoming.

Deuflhard, P. and Bornemann, F. (2002). *Scientific computing with ordinary differential equations*. Springer Verlag.

Deuflhard, P. and Hohmann, A. (2003). *Numerical analysis in modern scientific computing: an introduction*, volume 43. Springer Verlag.

Enright, W. (2006). Software for ordinary and delay differential equations: Accurate discrete approximate solutions are not enough. *Applied Numerical Mathematics*, 56(3-4):459–471.

Henrici, P. (1964). *Elements of Numerical Analysis*. John Wiley & Sons.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 2nd edition.

Hodges, A. (1992). *Alan Turing: the Enigma*. Vintage Books.

Moir, R. H. C. (2010). Reconsidering backward error analysis for ordinary differential equations. MSc Thesis, The University of Western Ontario.

Trefethen, L. N. (1992). The definition of numerical analysis. *SIAM News*, 25(November 1992):6 and 22.

Wilkinson, J. H. (1963). *Rounding Errors in Algebraic Processes*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs.

Wilkinson, J. H. (1971). Modern error analysis. *SIAM Review*, 13(4):548–568.

Chapter 1

Computer Arithmetic & Fundamental Concepts of Computation

1.1 Mathematical problems and computability of solutions

We begin by introducing a few elementary concepts that we will use to discuss computation in the context of numerical methods, adding a few parenthetical remarks meant to contrast our perspective from that of others. We represent a mathematical problem by an operator φ, that has an *input* (data) space $\mathscr{I}$ as its domain and an *output* (result, solution) space $\mathscr{O}$ as its codomain:

$$\varphi : \mathscr{I} \to \mathscr{O},$$

and we write $y = \varphi(x)$. In many cases, the input and output spaces will be $\mathbb{R}^n$ or $\mathbb{C}^n$, in which case we will use the function symbols $f, g, \ldots$ and accordingly write

$$y = f(z_1, z_2, \ldots, z_n) = f(\mathbf{z}).$$

Here, y is the (exact) solution to the problem f for the input data $\mathbf{z}$.[1] But φ need not be a function; for instance, we will study problems involving differential and integral operators. That is, in other cases, both x and y will themselves be functions.

We can delineate two general classes of computational problems related to the mathematical objects x, y, and φ:

C1. *verifying* whether a certain output y is actually the value of φ for a given input x, *i.e.*, verifying whether $y = \varphi(x)$;

C2. *finding* the output y determined by applying the map φ to a given input x, *i.e.*, finding the y such that $y = \varphi(x)$.[2]

The computation required by each type of problem is normally determined by an *algorithm*, *i.e.*, by a procedure performing a sequence of primitive operations leading to the solution in a finite number of steps. Numerical analysis is a mathematical reflection on complexity and numerical properties of algorithms in contexts that involve *data error* and *computational error*.

In the study of numerical methods as in many other branches of mathematical sciences, the reflection involves a subtle concept of *computation*. With a precise model of computation at hand, we can refine our views on what's computationally achievable, and if it turns out to be, how much resources are required.

The classical model of computation used in most textbooks on logic, computability, and algorithm analysis stems from metamathematical problems addressed in the 1930s; specifically, while trying to solve Hilbert's *Entscheidungsproblem*, Turing developed a model of primitive mathematical operations that could be performed by some type of machine affording finite but unlimited time and memory. This model, that turned out to be equivalent to other models developed independently by Gödel, Church, and others, resulted in a notion of computation based on *effective computability*. From there, we can

[1] We use boldface font for vectors and matrices.

[2] It is normally computationally simpler to verify whether a certain value satisfies an equation than finding a value that satisfies it.

form an idea of what is "truly feasible" by further adding constraints on time and memory.[3]

Nonetheless, scientific computation requires an alternative, *complementary* notion of computation, because the methods and the objectives are quite different from those of metamathematics. A first important difference is the following:

> [...] the Turing model (we call it "classical") with its dependence on 0s and 1s is fundamentally inadequate for giving such a foundation to the modern scientific computation, where most of the algorithms—which origins in Newton, Euler, Gauss, et al.—are *real number algorithms.* (Blum et al., 1998, 3)

Blum et al. (1998) generalize the ideas found in the classical model to include operations on elements of arbitrary rings and fields. But the difference goes even deeper:

> Rounding errors and instability are important, and numerical analysts will always be experts in the subjects and at pains to ensure that the unwary are not tripped up by them. But our central mission is to compute quantities that are typically uncomputable, from an analytic point of view, and to do it with lightning speed. (Trefethen, 1992)

Even with an improved picture of effective computability, it remains that the concept that matters for a large part of applied mathematics (including engineering) is the different idea of *mathematical tractability*, understood in a context where there is error in the data, error in computation, and where approximate answers can be entirely satisfactory. Trefethen's seemingly contradictory phrase "compute quantities that are typically uncomputable" underlines the complementarity of the two notions of computation.

[3]For a presentation of the classical model of computation, see, *e.g.*, Davis (1982); Brassard and Bratley (1996); Pour-El and Richards (1989), and for a specific discussion of what is "truly feasible," see Immerman (1999).

This second notion of computability based on the engineering view of mathematical tractability addresses the proper computational difficulties posed by the application of mathematics to the solution of practical problems from the outset. Certainly, both pure and applied mathematics heavily use the concepts of real and complex analysis. From real analysis, we know that every real number can be represented by a nonterminating fraction:

$$x = \lfloor x \rfloor . d_1 d_2 d_3 d_4 d_5 d_6 d_7 \cdots$$

However, in contexts involving applications, only a finite number of digits is ever dealt with. For instance, in order to compute $\sqrt{2}$, one could use an iterative method (*e.g.*, Newton's method, which we cover in Corless and Fillion, 201x, chap. 2) in which the number of accurate digits in the expansion will depend upon the number of iterations. A similar situation would hold if we used the first few terms of a series expansion for the evaluation of a function.

However, one must also consider another source of error due to the fact that, within each iteration (or each term), only finite-precision numbers and arithmetic operations are being used. We find the same situation in numerical linear algebra, interpolation, numerical integration, numerical differentiation, *etc.*

Understanding the effect of limited-precision arithmetic is crucial to computation for problems of continuous mathematics. Since computers only store and operate on finite expressions, the arithmetic operations they process necessarily incur an error that may, in some cases, propagate and/or accumulate in alarming ways.[4] In this first chapter, we focus on the kind of error that arises in the context of computer arithmetic, namely representation and arithmetic error. In fact, we will limit ourselves to the case of floating-point arithmetic, which is by far the most widely used. Thus, the two errors we will concern ourselves with are the error that results from representing a real number by a

[4]But let us not panic: "These risks are very real, but the message was communicated all too successfully, leading to the current widespread impression that the main business of numerical analysis is coping with rounding errors. In fact, [...]" and we have already continued the quote on page 11 (Trefethen, 2008).

floating-point number and the error that results from computing using floating-point operations instead of real operations. For a brief review of floating-point number systems, the reader is invited to consult Appendix A.

The objective of this chapter is not so much an in-depth study of error in floating-point arithmetic as an occasion to introduce some of the most important concepts of error analysis in a context that should not pose important technical difficulty to the reader. In particular, we will introduce the concepts of residual, backward and forward error, condition number, which will be the central concepts around which this thesis revolves. Together, these concepts will give solid conceptual grounds to the main theme of this thesis: *A good numerical method gives you nearly the right solution to nearly the right problem.*

1.2 Representation and computation error

Floating-point arithmetic does not operate on real numbers, but rather on floating-point numbers. This generates two types of *roundoff* errors: representation error and arithmetic error. The first type of error we encounter, *representation error*, comes from the replacement of real numbers by floating-point numbers. If we let $x \in \mathbb{R}$ and $\bigcirc : \mathbb{R} \to \mathbb{F}$ be an operator for the standard rounding procedure to the nearest floating-point number[5] (see appendix A), then the *absolute representation error* Δx is

$$\Delta x = \bigcirc x - x = \hat{x} - x. \tag{1.1}$$

[5]For the sake of simplicity, we will always assume that x and the other real numbers are within the range of $\mathbb{F}$.

(We will usually write $\hat{x}$ for $x+\Delta x$.) If $x \neq 0$, the *relative representation error* δx is given by

$$\delta x = \frac{\Delta x}{x} = \frac{\hat{x} - x}{x}. \tag{1.2}$$

From those two definitions, we obtain the following useful equality:

$$\hat{x} = x + \Delta x = x(1 + \delta x). \tag{1.3}$$

The IEEE standard described in appendix A guarantees that $|\delta x| < \mu_M$, where μ_M is half the machine epsilon ε_M.

In a numerical computing environment such as MATLAB, $\varepsilon_M = 2^{-52} \approx 2.2 \cdot 10^{-16}$, so that $\mu_M \approx 10^{-16}$.

The IEEE standard also guarantees that the floating-point sum of two floating-point numbers, written $\hat{z} = \hat{x} \oplus \hat{y}$, is the floating-point number nearest to the real sum $z = \hat{x} + \hat{y}$ of the floating-point numbers, *i.e.*, it is guaranteed that

$$\hat{x} \oplus \hat{y} = \bigcirc(\hat{x} + \hat{y}). \tag{1.4}$$

In other words, the floating-point sum of two floating-point numbers is the correctly rounded real sum. As explained in appendix A, similar guarantees are given for $\ominus, \otimes$, and $\oslash$.

Parallelling the definitions of equations (1.1) and (1.2), we define the absolute and relative *computation errors* (in addition) by

$$\Delta z = \hat{z} - z = (\hat{x} \oplus \hat{y}) - (\hat{x} + \hat{y}) \tag{1.5}$$

$$\delta z = \frac{\Delta z}{z} = \frac{(\hat{x} \oplus \hat{y}) - (\hat{x} + \hat{y})}{\hat{x} + \hat{y}}. \tag{1.6}$$

As in equation (1.3), we obtain

$$\hat{x} \oplus \hat{y} = \hat{z} = z + \Delta z = z(1 + \delta z) \tag{1.7}$$

with $|\delta z| < \mu_M$. It follows directly from equation (1.3) that the real sum $z = \hat{x} + \hat{y}$ of the floating-point representations of x and y is given by

$$\begin{aligned} z = \hat{x} + \hat{y} &= x(1 + \delta x) + y(1 + \delta y) \\ &= (x + y)\left(1 + \frac{x\delta x + y\delta y}{x + y}\right). \end{aligned} \tag{1.8}$$

From this and equation (1.7), the total error generated by the floating-point addition of two floating-point representations of real numbers is given by

$$\hat{z} = \hat{x} \oplus \hat{y} = (1 + \delta z)(\hat{x} + \hat{y}) = (1 + \delta z)\left(1 + \frac{x\delta x + y\delta y}{x + y}\right)(x + y), \tag{1.9}$$

where δz is the computation error and δx and δy are the representation errors. This equation gives us an automatic way to transform expressions containing '$\hat{}$' and '$\oplus$' into expressions containing only real quantities and operations.

Similar results hold for subtraction, multiplication, and division. The real subtraction and multiplication of two floating-point numbers give us the following equations:

$$\hat{z} = \hat{x} - \hat{y} = \left(1 + \frac{x\delta x + y\delta y}{x - y}\right)(x - y) \tag{1.10}$$

$$\begin{aligned} \hat{z} = \hat{x} \times \hat{y} &= (1 + \delta x + \delta y + \delta x\delta y)xy \\ &\doteq (1 + \delta x + \delta y)xy \end{aligned} \tag{1.11}$$

Note that we use the "dot equal" symbol $\doteq$ to signify that we neglect the higher-order terms (in this case, $O(\delta^2)$). For division, since $|\delta| < 1$, we use the

fact that

$$\frac{1}{1-\delta^2} = \frac{1}{1-\delta^2}\frac{1+\delta^2}{1+\delta^2} = \frac{1+\delta^2}{1-\delta^4}$$
$$= \frac{(1+\delta^2)(1+\delta^4)}{1-\delta^8} = \frac{(1+\delta^2)(1+\delta^4)(1+\delta^8)}{1-\delta^{16}} = \ldots, \tag{1.12}$$

which is true since all the δs cancel out (we will often use this trick to obtain only the first-order error terms). So, for the real division of two floating-point numbers, we obtain

$$\hat{z} = \hat{x}/\hat{y} = \frac{x(1+\delta x)}{y(1+\delta y)} = \frac{x}{y}(1+\delta x)(1-\delta y)\prod_{i=1}^{\infty}(1+\delta y^{2^i})$$
$$\doteq (1+\delta x-\delta y)\frac{x}{y} \tag{1.13}$$

Finally, we obtain the following equations for the floating-point subtraction, multiplication, and division of floating-point numbers:

$$\hat{x} \ominus \hat{y} = (1+\delta z)\left(1+\frac{x\delta x - y\delta y}{x-y}\right)(x-y) \tag{1.14}$$

$$\hat{x} \otimes \hat{y} \doteq (1+\delta x+\delta y+\delta z)xy \tag{1.15}$$

$$\hat{x} \oslash \hat{y} \doteq (1+\delta x-\delta y-\delta z)\frac{x}{y} \tag{1.16}$$

We can usually assume that $\sqrt{x}$ also provides the correctly rounded result, but it is not generally the case for other operations, such as e^x, $\ln x$, and the trigonometric functions (see Muller et al., 2009).

The important point to observe about equations 1.8, 1.10, 1.11, 1.12 and 1.9, 1.14, 1.15, 1.16 is that *they contain no floating-point numbers and no floating-point operations* on the right-hand side. Together, they provide a set of equations that allow us to examine floating-point arithmetic using real arithmetic only. That makes it easier to analyze the error of computational methods, since we can rely on familiar tools.

1.3 Problematic cases for floating-point arithmetic

In this section, we motivate the introduction of the key concepts of error analysis by presenting some typical problematic cases that arise from representation and computation errors. Note that, from now on, we will use the term *rounding error* to refer (ambiguously) to representation or computation error.

Failure of standard axioms of arithmetic on fields To understand floating-point arithmetic better, it is important to verify whether the standard axioms of fields are satisfied, or at least nearly satisfied. As it turns out, many standard axioms do not hold, not even nearly. Consider the following sentences (for $\hat{x}, \hat{y}, \hat{z} \in \mathbb{F}$):

1. Associative law of $\oplus$:

$$\hat{x} \oplus (\hat{y} \oplus \hat{z}) = (\hat{x} \oplus \hat{y}) \oplus \hat{z} \tag{1.17}$$

2. Associative law of $\otimes$:

$$\hat{x} \otimes (\hat{y} \otimes \hat{z}) = (\hat{x} \otimes \hat{y}) \otimes \hat{z} \tag{1.18}$$

3. Cancellation law (for $\hat{x} \neq 0$):

$$\hat{x} \otimes \hat{y} = \hat{x} \otimes \hat{z} \Rightarrow \hat{y} = \hat{z} \tag{1.19}$$

4. Distributive law:

$$\hat{x} \otimes (\hat{y} \oplus \hat{z}) = (\hat{x} \otimes \hat{y}) \oplus (\hat{x} \otimes \hat{z}) \tag{1.20}$$

5. Multiplication canceling division:

$$\hat{x} \otimes (\hat{y} \oslash \hat{x}) = \hat{y} \tag{1.21}$$

In general, the associative and distributive laws fail, but commutativity still holds. As a result of these failures, mathematicians find it very difficult to work directly in floating-point arithmetic—its algebraic structure is weak and unfamiliar. Nonetheless, floating-point arithmetic *has* some algebraic structure in which one *can* provide an error analysis—Von Neumann and Goldstine (1947) famously did go through this excruciatingly painful process.

It will typically be more practical to use a different, less harrowing approach. Using the results of section 1.2, we know how to translate a problem involving floating-point operations into a problem involving only real arithmetic on real quantities $(x, \Delta x, \delta x, \ldots)$. This approach allows us to use the mathematical structures that we are familiar with in algebra and analysis. So, instead of making our error analysis directly in floating-point arithmetic, we try to work on a problem which is *exactly* (or nearly exactly) equivalent to the original floating-point problem, by means of the study of perturbations of real (and eventually complex) quantities. This insight was first exploited systematically by J.H. Wilkinson.

Error Accumulation and Catastrophic Cancellation In applications, it is usually the case that a large number of operations have to be done sequentially before results are obtained. In sequences of floating-point operations, arithmetic error may accumulate. The magnitude of the accumulating error will usually be negligible for well-tested algorithms.[6] Nonetheless, it is important to be aware of the possibility of massive *accumulating rounding error* in some cases. For instance, even if the IEEE standard guarantees that, for $\hat{x}, \hat{y} \in \mathbb{F}$, $\hat{x} \oplus \hat{y} = \bigcirc(\hat{x} + \hat{y})$, it does not guarantee that equations of the form

$$\bigoplus_{i=1}^{k} \hat{x}_i = \bigcirc \sum_{i=1}^{k} \hat{x}_i, \qquad k > 2 \tag{1.22}$$

hold true. This can potentially cause problems for the computation of sums, *e.g.*, for the computation of an inner product $\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{k} x_i y_i$. In this case,

[6] In fact, as explained by Higham (2002, chap. 1), errors can often cancel each other out to give surprisingly accurate results.

the direct floating-point computation would be

$$\bigoplus_{i=1}^{k}(x_i \otimes y_i). \tag{1.23}$$

If we consider $k = 1000$, each multiplication introduces a maximum error μ_M, and so does each addition, for a total of $1,999\mu_M$. So, in 16-bit precision floating-point arithmetic, the result is only guaranteed to be accurate to about 12 places. But this is a worst-case analysis, that returns the maximum error that can result from the satisfaction of the IEEE standard. In practice, it will often be much better. In fact, if you use a built-in routine for inner products, the accumulating error will be well-below that (Brent and Zimmermann, 2011).

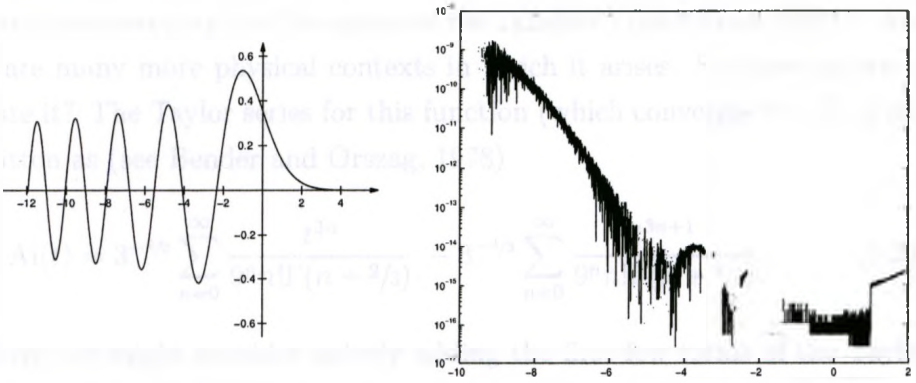
Another typical case in which the potential difficulty with sums poses a problem is in the computation of the value of a function using a convergent series expansion and floating-point arithmetic. Consider the simple case of the exponential function (from Forsythe, 1970), $f(x) = e^x$, which can be represented by the uniformly convergent series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \tag{1.24}$$

If we work in a floating-point system with a 5-digit precision, we obtain the sum

$$\begin{aligned} e^{-5.5} &\approx 1.0000 - 5.5000 + 15.125 - 27.730 + 38.129 - 41.942 + 38.446 \\ &\quad - 30.208 + 20.768 - 12.692 + 6.9803 - 3.4902 + 1.5997 + \cdots \\ &= 0.0026363. \end{aligned}$$

This is the sum of the first 25 terms, following which the first few digits do not change, perhaps leading us to believe (incorrectly) that we have reached an accurate result. But in fact, $e^{-5.5} \approx 0.00408677$, so that $\Delta y = \hat{y} - y \approx 0.0015$. This might not seem very much, when posed in absolute terms, but it corresponds to $\delta y = 35\%$, an enormous relative error! Note, however, that it would be within what would be guaranteed by the IEEE standard for this



(a) Plot of the Airy Function.

(b) Forward error in the MATLAB naive Taylor series computation.

Figure 1.1: The Airy function

number system. To decrease the magnitude of the maximum rounding error, we would need to add precision to the number system, thereby decreasing the magnitude of the machine epsilon. But as we will see below, this would not save us either. A more efficient solution would be to use a more accurate formula for e^{-x}.

There usually are excellent built-in algorithms for the exponential function. But a similar situation could occur with the computation of values of some transcendental function for which no built-in algorithm is provided, such as the Airy function. The Airy function (see figure 1.1(a)) is a solution of the differential equation $\ddot{x} - tx = 0$ The first Airy function can be defined by the integral

$$\mathrm{Ai}(t) = \frac{1}{\pi} \int_0^\infty \cos\left(\frac{1}{3}\zeta^3 + t\zeta\right) d\zeta. \tag{1.25}$$

Similarly, the function $\mathrm{Ai}(-t)$ is a solution of $\ddot{x} + tx = 0$. This function occurs often in physics. For instance, if we study the undamped motion of a weight attached to a Hookean spring that becomes linearly stiffer with time, we get the equation of motion $\ddot{x} + tx = 0$, and so the motion is described by $\mathrm{Ai}(-t)$ (Nagle et al., 2000). Another case would be the important role the zeros of

the Airy function play for the optics of the rainbow (Batterman, 2002). And there are many more physical contexts in which it arises. So, how are we to evaluate it? The Taylor series for this function (which converges for all x) can be written as (see Bender and Orszag, 1978)

$$\mathrm{Ai}(t) = 3^{-2/3} \sum_{n=0}^{\infty} \frac{t^{3n}}{9^n n! \Gamma(n + 2/3)} - 3^{-4/3} \sum_{n=0}^{\infty} \frac{t^{3n+1}}{9^n n! \Gamma(n + 4/3)} . \tag{1.26}$$

As above, we might consider naively adding the first few terms of the Taylor series using floating-point operations, until convergence (*i.e.*, until adding new terms does not change the solution anymore because they are small). This can be done simply with a MATLAB code of this ilk (using 16-digit floating-point arithmetic):

```
function [ Ai ] = AiTaylor( z )
%AiTaylor Taylor series about 0 evaluation of Ai(z)
%   Try to use (naively) the explicitly-known Taylor series at z=0
%   to evaluate Ai(z).  Ignore rounding errors, overflow/underflow, NaN.
%   The input argument z may be a vector of complex numbers.
%   y = AiTaylor( z );
%
    THREETWOTH  = 3.0^(-2/3);
    THREEFOURTH = 3.0^(-4/3);

    Ai = zeros(size(z));
    zsq = z.*z;
    n = 0;
    zpow = ones(size(z));  % zpow = z^(3n)

    term = THREETWOTH*ones(size(z))/gamma(2/3); % recall n! = gamma(n+1)
    nxtAi = Ai + term;

    % Convergence is deemed to occur when adding new terms
    % makes no difference numerically.
    while any( nxtAi ~= Ai ),
```

```
22          Ai = nxtAi;
23          zpow = zpow.*z;  % zpow = z^(3n+1)
24          term = THREEFOURTH*zpow/9^n/factorial(n)/gamma(n+4/3);
25          nxtAi = Ai - term;
26          if all( nxtAi == Ai ), break, end;
27          Ai = nxtAi;
28          n = n + 1;
29          zpow = zpow.*zsq;  % zpow = z^(3n)
30          term = THREETWOTH*zpow/9^n/factorial(n)/gamma(n+2/3);
31          nxtAi = Ai + term;
32      end
33
34      % We are done.  If the loop exits, Ai = AiTaylor(z).
35
36  end
```

Using this algorithm, can one expect to have a high accuracy, with error close to ε_M? Figure 1.1(b) displays the difference between the correct result (as computed with MATLAB's function `airy`) and the naive Taylor series approach. So, suppose we want to use this algorithm to compute $f(-12.82)$, a value near the tenth zero (counting from the origin towards $-\infty$), the absolute error is

$$\Delta y = |\mathrm{Ai}(x) - \mathrm{AiTaylor}(x)| = 0.002593213070374, \tag{1.27}$$

resulting in a relative error $\delta y \approx 0.277$. The solution is only accurate to two digits! Even if, theoretically, the series converge for all x, it is of no practical use. We examine this example in more detail in Corless and Fillion (201x, chap. 2, "Polynomials and Series") when discussing the evaluation of polynomial functions.

The underlying phenomenon in the former examples, sometimes known as "the hump phenomenon," could also occur in a floating-point number systems with higher precision. What happened exactly? If we consider the magnitude of some of the terms in the sum, we find out that they are much larger than the returned value (and the real value). We observe that this series is an alternating series in which the terms of large magnitude mostly cancel each other

out. When such a phenomenon occurs—a phenomenon that Lehmer coined *catastrophic cancellation*—we are more likely to encounter erratic solutions. After all, how can we expect that numbers such as 38.129, a number with only three significant decimal points, could be used to accurately obtain the sixth or seventh decimal point in the answer? This explains why one must be careful in cases involving catastrophic cancellation.

Another famous example of catastrophic cancellation involves finding the roots of a degree-2 polynomial $ax^2 + bx + c$ using the quadratic equation (Forsythe, 1966):

$$x_{\pm}^{*} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If we take an example for which $b^2 \gg 4ac$, catastrophic cancellation can occur. Consider this example:

$$a = 1 \cdot 10^{-2} \qquad b = 1 \cdot 10^{7} \qquad c = 1 \cdot 10^{-2}$$

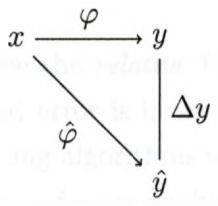
Such numbers could easily arise in practice. Now, a MATLAB computation returns $x_{+}^{*} = 0$, which is obviously not a root of the polynomial. In this case, the answer returned is 100% wrong, in relative terms.

1.4 Perspectives on error analysis: forward, backward, and residual-based

The problematic cases can provoke a feeling of insecurity. When are the results provided by actual computation satisfactory? Sometimes, it is quite difficult to know intuitively whether it is the case. And how exactly should satisfactoriness be understood and measured? Here, we discuss the concepts that will warrant confidence or non-confidence in some results based on an error analysis of the computational processes involved.

Our starting point is that problems arising in scientific computation are such that we typically do not compute the exact value $y = \varphi(x)$, for the

reference problem φ, but instead some other more convenient value $\hat{y}$. The value $\hat{y}$ is not an exact solution of the reference problem, so that many authors regard it as an approximate solution, *i.e.*, $\hat{y} \approx \varphi(x)$. We find it much more fruitful to regard the quantity $\hat{y}$ as the *exact* solution of a modified problem, *i.e.*, $\hat{y} = \hat{\varphi}(x)$, where $\hat{\varphi}$ denotes the modified problem. For reasons that will become clearer later, we also call the modified problem an *engineered problem*, because it consists in modifying φ in a way that makes computation easier, or even possible at all. We thus get this general picture:



(1.28)

For example, if we have a simple problem of addition to do, instead of computing $y = f(x_1, x_2) = x_1 + x_2$, we might compute $\hat{y} = \hat{f}(\hat{x}_1, \hat{x}_2) = \hat{x}_1 \oplus \hat{x}_2$. Here, we regard the computation of the floating-point sum as a modified problem, and we regard $\hat{y}$ as the exact solution of this engineered problem. Similarly, if the problem is to find the zeros of a polynomial, we can use various methods that will give us pseudozeros. Instead of regarding the pseudozeros as approximate solutions of the reference problem "find the zeros," we regard those pseudozeros as the *exact* solution to the modified problem "find some pseudozeros" (see Corless and Fillion, 201x, chap. 2, "Polynomials and Series"). If the problem is to find a vector $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$, given a matrix $\mathbf{A}$ and a vector $\mathbf{b}$, we can use various methods that will give us a vector that almost satisfies the equation, but not quite. Then we can regard this vector as the solution for a matrix with slightly modified entries (see Corless and Fillion, 201x, chap. 4, "Solving $\mathbf{Ax} = \mathbf{b}$"). The whole book is about cases of this sort arising from all branches of mathematics.

What is so fruitful about this seemingly trivial change in the way the problems and solutions are discussed? Once this change of perspective is adopted, we do not focus so much on the question "how far is the computed

solution from the exact one?" (*i.e.*, in diagram 1.28, how big is Δy) than on the question "how closely related are the original problem and the engineered problem?" (*i.e.*, in diagram 1.28, how closely related are φ and $\hat{\varphi}$). If the modified problem behaves closely like the reference problem, we will say it is a *nearby problem.*

The quantity labeled Δy in diagram 1.28 is called the *forward error*, which is defined by

$$\Delta y = y - \hat{y} = \varphi(x) - \hat{\varphi}(x). \tag{1.29}$$

We can of course also introduce the *relative* forward error by dividing by y. In certain contexts, the forward error is in some sense the key quantity that we want to control when designing algorithms to solve a problem.Then, a very important task is to carry a forward error analysis; the task of such an analysis is to put an upper bound on $\|\Delta y\| = \|\varphi(x) - \hat{\varphi}(x)\|$. However, as we will see, there are also many contexts in which the control of the forward is not so crucial.

Even in contexts requiring a control of the forward error, direct forward error analysis will play a very limited role in our analyses, for a very simple reason. We engineer problems and algorithms because we don't know or don't have efficient means of computing the solution of the reference problem. But directly computing the forward error involves solving a computational problem of type C2 (as defined on p. 10), which is often unrealistic. As a result, scientific computation presents us situations in which we usually don't know or don't have efficient ways of computing the forward error. Somehow, we need a more manageable concept that will also reveal if our computed solutions are good. Fortunately, there's another type of *a priori* error analysis—*i.e.*, antecedent to actual computation—one can carry, namely, *backward error analysis.* We explain the perspective it provides in the next subsection. Then, in subsection 1.4.2 and 1.4.3, we show how to supplement a backward error analysis with the notions of condition and residual in order to obtain an informative assessment of the forward error. Finally, in the next section, we will provide definitions

for the stability of algorithms in these terms.

1.4.1 Backward Error Analysis

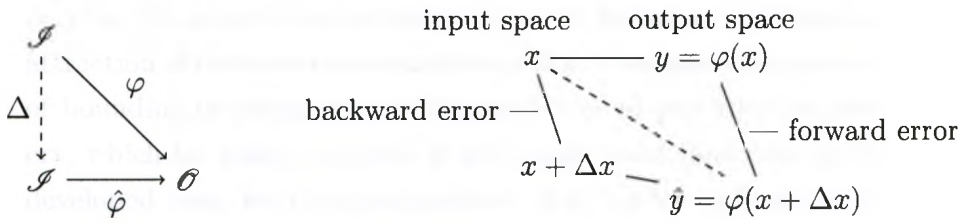
Let us generalize our concept of error to include any type of error, whether it comes from data error, measurement error, rounding error, truncation error, discretization error, *etc.* In effect, the success of backward error analysis comes from the fact that it treats all types of errors (physical, experimental, representational, and computational) on an equal footing. Thus, $\hat{x}$ will be some approximation of x, and Δx will be some absolute error that may be or may not be the rounding error. Similarly, in what follows, δx will be the relative error, that may or may not be the relative rounding error. The error terms will accordingly be understood as *perturbations of the initially specified data.* So, in a backward error analysis, if we consider the problem $y = \varphi(x)$, we will in general consider all the values of the data $\hat{x} = x(1 + \delta x)$ satisfying a condition $|\delta x| < \varepsilon$, for some ε prescribed by the modeling context,[7] and not only the rounding errors determined by the real number x and the floating-point system. In effect, this change of perspective shifts our interest from particular values of the input data to sets of input data satisfying certain inequalities.[8]

Now, if we consider diagram 1.28 again, we could ask: can we find a perturbation of x that would have effects on φ comparable to the effect of changing the reference problem φ by the engineered problem $\hat{\varphi}$? Formally, we are asking: can we find a Δx such that $\varphi(x + \Delta x) = \hat{\varphi}(x)$? The smallest such Δx is what is called the *backward error.* For input spaces whose elements are numbers, vectors, matrices, functions, and the like, we use norms as usual to determine how big the backward error is.[9] For other types of mixed inputs, we might have to use a set of norms for each component of the input. The resulting general picture is illustrated in figure 1.2(b) (see, *e.g.*, Higham, 2002), and we see that this analysis amounts to *reflect* the forward error *back* into the backward

[7]Note that, since modeling contexts usually include the proper choice of scale, the value of ε will usually be given in relative rather than absolute terms.

[8]For an alternative, more rigorous presentation of the concepts presented here, see Deuflhard and Hohmann (2003, chap. 2).

[9]The choice of norm may be a delicate issue, but we will leave it aside for the moment.



(a) Reflecting back the backward error: finding maps Δ

(b) Input and output space in a backward error analysis

Figure 1.2: Backward error analysis: The general picture.

error.[10] In effect, the question that is central to backward error analysis is, *when we modified the reference problem φ to get the engineered problem $\hat{\varphi}$, for what set of data have we actually solved the problem φ?* If solving the problem $\hat{\varphi}(x)$ amounts to having solved the problem $\varphi(x + \Delta x)$ for a Δx smaller than the perturbations inherent in the modeling context, then our solution $\hat{y}$ must be considered completely satisfactory.[11]

Adopting this approach, we benefit from the possibility of using well-known perturbation methods to talk about different problems and functions:

> The effects of errors in the data are generally easier to understand than the effects of rounding errors committed during a computation, because data errors can be analysed using perturbation theory for the problem at hand, while intermediate rounding errors require an analysis specific to the given method. (Higham, 2002, 6)

> [t]he process of bounding the backward error of a computed solution is called *backward error analysis*, and its motivation is twofold. First, it interprets rounding errors as being equivalent to perturbations in the data. The data frequently contains uncertainties due to previous computations or errors committed in storing numbers on the computer. If the backward error is no larger than these uncertainties then the computed solution can hardly be criticized—it

[10] The 'reflecting back' terminology goes back to Wilkinson (1963).

[11] There are cases, however, where finding such a Δx will not be possible. See Higham (2002, p. 71).

> may be the solution we are seeking, for all we know. The second attraction of backward error analysis is that it reduces the question of bounding or estimating the forward error to perturbation theory, which for many problems is well understood (and only to be developed once, for the given problem, and not for each method). (Higham, 2002, 7-8)

One can examine the effect of perturbations of the data using basic methods we know from calculus, various orders of perturbation theory, and the general methods used for the study of dynamical systems. Consider this (almost trivial!) example using only first-year calculus. Take the polynomial $p(x) = 17x^3 + 11x^2 + 2$; if there is a measurement uncertainty or a perturbation of the argument x, then how big will be the effect? One finds that

$$\Delta y = p(x + \Delta x) - p(x) = 51x^2\Delta x + 51x(\Delta x)^2 + 17(\Delta x)^3 + 22x\Delta x + 11(\Delta x)^2.$$

Now, since typically $|\Delta x| \ll 1$, we can ignore the higher degrees of Δx, so that

$$\Delta y \doteq 51x^2\Delta x.$$

Consequently, if $x = 1 \pm .1$, we get $y \doteq 35 \pm 5.1$; the perturbation in the input data has been magnified by about 50, and that would get worse if x were bigger.

To end this subsection, let us consider an example showing concretely how to reflect back the forward error into the backward error, in the context of floating-point arithmetic. Suppose we want to compute $y = f(x_1, x_2) = x_1^3 - x_2^3$ for the input $\mathbf{x} = [12.5, 0.333]$. For the sake of the example, suppose we have to use a computer working with a floating-point arithmetic with 3-digit precision. So, we will really compute $\hat{y} = ((x_1 \otimes x_1) \otimes x_1) \ominus ((x_2 \otimes x_2) \otimes x_2)$. We assume that $\mathbf{x}$ is a pair of floating-point numbers, so there is no representation error. The result of the computation is $\hat{y} = 1950$, and the the exact answer is $y = 1953.014111$, leaving us with a forward error $\Delta y = 3.014111$ (or, in relative terms, $\delta y = {}^{3.014111}/_{1953.014111} \approx 1.5\%$). In a backward error analysis,

we want to reflect the arithmetic (forward) error back in the data, *i.e.*, we need to find some Δx_1 and Δx_2 such that

$$\Delta y = 3.014111 = \hat{y} - y = (12.5 + \Delta x_1)^3 - (0.333 + \Delta x_2)^3 - 1953.014111.$$

A solution is $\Delta \mathbf{x} \approx [0.0064, 0]$ (whereby $\delta x_1 = 0.05\%$). But as one sees, the condition determines an infinite set of real solutions S, with real and complex elements. In such cases, where the entire set of solutions can be characterized, it is possible to find particular solutions, *e.g.*, the solution that would minimize the 2-norm of the vector $\Delta \mathbf{x}$.

1.4.2 Condition of problems

We have seen how we can reflect back the forward error in the backward error. Now, the question we ask is: *what is the relationship between the forward and the backward error*? In fact, in modeling contexts, we are not really after an expression or a value for the forward error *per se*. The only reason for which we want to estimate the forward error is to ascertain whether is it smaller than a certain user-defined "tolerance," prescribed by the modeling context. To do so, all one needs is to find how the perturbations of the input data (the so-called backward error we discussed) are magnified by the reference problem. Thus, the relationship we seek lies in a problem-specific coefficient of magnification, *i.e.*, the sensitivity of the solution to perturbations in the data, that we call the *conditioning of the problem*. The conditioning of a problem is measured by the *condition number*. As for the errors, the condition number can be defined in relative and absolute terms, and it can be measured normwise or componentwise.

The *normwise relative condition number* κ_{rel} is the maximum of the ratio of the relative change in the solution to the relative change in input,

which is expressed by

$$\kappa_{rel} = \sup_x \frac{\|\delta y\|}{\|\delta x\|} = \sup_x \frac{\|\Delta y/y\|}{\|\Delta x/x\|} = \sup_x \frac{\|(\varphi(\hat{x}) - \varphi(x))/\varphi(x)\|}{\|\hat{x} - x/x\|}$$

for some norm $\|\cdot\|$. As a result, we obtain the relation

$$\|\delta y\| \leq \kappa_{rel}\|\delta x\| \tag{1.30}$$

between the forward and the backward error. Knowing the backward error and the conditioning thus gives us an upper bound on the forward error.

In the same way, we can define the *normwise absolute condition number* κ_{abs} as $\sup_x \|\Delta y\|/\|\Delta x\|$, thus obtaining the relation

$$\|\Delta y\| \leq \kappa_{abs}\|\Delta x\|. \tag{1.31}$$

If κ has a moderate size, we say that the problem is *well-conditioned.* Otherwise, we say that the problem is *ill-conditioned.*[12] Consequently, even for a very good algorithm, the approximate solution to an ill-conditioned problem may have a large forward error.[13] It is important to observe that this fact is totally independent of any method used to compute φ. What matters is the existence of κ and what its size is.

Suppose that our problem is a scalar function. It is convenient to observe immediately that, for a sufficiently differentiable problem f, we can get an approximation of κ in terms of derivatives. Since

$$\lim_{\Delta x \to 0} \frac{\delta y}{\delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \cdot \frac{x}{y} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \frac{x}{f(x)} = \frac{x f'(x)}{f(x)},$$

[12] When κ is unbounded, the problem is sometimes said to be *ill-posed.*

[13] Note the "may", which means that backward error analysis often provides pessimistic upper bounds on the forward error.

the approximation of the condition number

$$\kappa_{rel} \approx \frac{|x||f'(x)|}{|f(x)|} \tag{1.32}$$

will provide a sufficiently good measure of the conditioning of a problem for small Δx. In the absolute case, we have $\kappa_{abs} \approx |f'(x)|$. This approximation will become useful in later chapters. If f is a multivariable function, the derivative $f'(x)$ will be the Jacobian matrix

$$\mathbf{J}_f(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \partial f/\partial x_1 & \partial f/\partial x_2 & \cdots & \partial f/\partial x_n \end{bmatrix},$$

and the norm used for the computation of the condition number will be the induced matrix norm $\|\mathbf{J}\| = \max_{\|\mathbf{x}\|=1} \|\mathbf{J}\mathbf{x}\|$. In effect, this approximation amounts to ignoring the terms $O(\Delta x^2)$ in the Taylor expansion of $f(x+\Delta x) - f(x)$; using this approximation will thus result in a *linear error analysis.*

Though normwise condition numbers are convenient in many cases, it is often important to look at the internal structure of the arguments of the problem, *e.g.*, the dependencies between the entries of a matrix or between the components of a function vector. In such cases, it is better to use a componentwise analysis of conditioning. The relative componentwise condition number of the problem φ is the smallest number $\kappa_{rel} \geq 0$ such that

$$\max_i \frac{|f_i(\hat{x}) - f_i(x)|}{|f_i(x)|} \dot{\leq} k_{rel} \max_i \frac{|\hat{x}_i - x_i|}{|x_i|}, \quad \hat{x} \to x,$$

where $\dot{\leq}$ indicates that the inequality holds in the limit $\Delta x \to 0$ (so, again, it holds for a linear error analysis). If the condition number is in this last form, we get a convenient theorem:

Theorem 1 (Deuflhard and Hohmann (2003)). *The condition number is submultiplicative,* i.e.,

$$\kappa_{rel}(g \circ h, x) \leq \kappa_{rel}(g, h(x)) \cdot \kappa_{rel}(h, x).$$

In other words, the condition number of a composed problem $g \circ h$ evaluated near x is smaller than or equal to the product of the condition number of the problem h evaluated at x by the condition number of the problem g evaluated at $h(x)$. □

Consider two simple examples. Firstly, let us take the identity function $f(x) = x$ near $x = a$ (this is of course a trivial example). As one would expect, we get the absolute condition number

$$\kappa_{abs} = \sup \frac{\|f(a + \Delta a) - f(a)\|}{\|\Delta a\|} = \frac{\|a + \Delta a - a\|}{\|\Delta a\|} = 1. \tag{1.33}$$

As a result, we get the relation $\|\Delta y\| \leq \|\Delta x\|$ between the forward and the backward error. This surely has moderate size in any context, since it does not amplify the input error. Secondly, consider addition, $f(a, b) = a + b$. Now, the derivative of f is

$$f'(a, b) = \begin{bmatrix} \frac{\partial f}{\partial a} & \frac{\partial f}{\partial b} \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix}.$$

Suppose we use the 1-norm on the Jacobian matrix. Then, the condition numbers are $\kappa_{abs} = \|f'(a, b)\|_1 = \| \begin{bmatrix} 1 & 1 \end{bmatrix} \|_1 = 2$ and

$$\kappa_{rel} = \frac{\left\| \begin{bmatrix} a \\ b \end{bmatrix} \right\|_1}{\|a + b\|_1} \left\| \begin{bmatrix} 1 & 1 \end{bmatrix} \right\|_1 = 2\frac{|a| + |b|}{|a + b|}. \tag{1.34}$$

(Since the function is linear, the approximation of the definitions is an equality.) Accordingly, if $|a + b| \ll |a| + |b|$, we consider the problem to be ill-conditioned. We examine many more cases in Corless and Fillion (201x). Moreover, many other examples are to be found in Deuflhard and Hohmann (2003).

1.4.3 Residual-based *a posteriori* error analysis

The key concept we exploit in what follows is the *residual*. For a given problem φ, the image y can have many forms. For example, if the reference problem φ consists in finding the roots of the equation $\xi^2+x\xi+2=0$, then for each value of x the object y will be a set containing two numbers satisfying $\xi^2+x\xi+2=0$, *i.e.*,

$$y = \left\{ \xi \mid \xi^2 + x\xi + 2 = 0 \right\}. \tag{1.35}$$

In general, we can then define a problem to be a map

$$x \xrightarrow{\varphi} \left\{ \xi \mid \phi(x,\xi) = 0 \right\}, \tag{1.36}$$

where $\phi(x,\xi)$ is some function of the input x and the output ξ. The function $\phi(x,\xi)$ is called the *defining function* and the equation $\phi(x,\xi)=0$ is called the *defining equation* of the problem. On that basis, we can introduce the very important concept of *residual*:

> Given the reference problem φ—whose value at x is a y such that the defining equation $\phi(x,y)=0$ is satisfied—and an engineered problem $\hat{\varphi}$, the residual r is defined by
>
> $$r = \phi(x,\hat{y}). \tag{1.37}$$
>
> As we see, we obtain the residual by substituting the computed value $\hat{y}$ (*i.e.*, the exact solution of the engineered problem) for y as the second argument of the defining function.

Let us consider some examples in which we apply our concept of residual to various kinds of problems.

1. The reference problem consists in finding the roots of $a_2x^2+a_1x+a_0=0$. The corresponding map is $\varphi(\mathbf{a}) = \{x \mid \phi(\mathbf{a},x)=0\}$ where the defining

equation is $\phi(\mathbf{a}, x) = a_2x^2 + a_1x + a_0 = 0$. Our engineered problem $\hat{\varphi}$ could consist in computing the roots to three correct places. With the resulting 'pseudozeros' $\hat{x}$, we can then easily compute the residual $r = a_2\hat{x}^2 + a_1\hat{x} + a_0$.

2. The reference problem consists in finding a vector $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$, for a non-singular matrix $\mathbf{A}$. The corresponding map is $\varphi(\mathbf{A}, \mathbf{b}) = \{\mathbf{x} | \phi(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \mathbf{0}\}$ where the defining equation is $\phi(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \mathbf{Ax} - \mathbf{b} = \mathbf{0}$. In this case, the set is a singleton since there's only one such $\mathbf{x}$. Our engineered problem could consist in using Gaussian elimination in 5-digit floating-point arithmetic. With the resulting solution $\hat{\mathbf{x}}$ we can compute the residual $\mathbf{r} = \mathbf{A}\hat{\mathbf{x}} - \mathbf{b}$.

3. The reference problem consists in finding a function $x(t)$ on the interval $0 < t \leq 1$ such that

$$\dot{x} = f(t, x) = t^2 + x(t) - \frac{1}{10}x^4(t) \tag{1.38}$$

and $x(0) = 0$. The corresponding map is

$$\varphi(x(0), f(t, x)) = \{x(t) \mid \phi(x(0), f(t, x), x(t)) = 0\} \tag{1.39}$$

where the defining equation is

$$\phi(x(0), f(t, x), x(t)) = \dot{x} - f(t, x) = 0 \tag{1.40}$$

together with $x(0) = 0$ (on the given interval). In this case, if the solution exists and is unique (as happens when f is Lipschitz) the set is a singleton since there's only one such $x(t)$. Our engineered problem could consist in using, say, a continuous Runge-Kutta method. With the resulting solution $\hat{x}(t)$ we can compute the residual $r = \dot{\hat{x}} - f(t, \hat{x})$.

Many more examples of different kinds could be included, but this should sufficiently illustrate the idea for now.

In cases similar to our third example above, we can rearrange the equation $r = \dot{\hat{x}} - f(t, \hat{x})$ to have $\dot{\hat{x}} = f(t, \hat{x}) + r$, so that the residual is itself a perturbation (or a backward error) of the function defining the integral operator for our initial value problem. The new "perturbed" problem is

$$\tilde{\varphi}(x(0), f(t,x) + r(t,x)) = \{x(t) \mid \tilde{\phi}(x(0), f(t,x) + r(t,x), x(t)) = 0\},$$

and we observe that our computed solution $\hat{x}(t)$ is an exact solution of this problem. When such a construction is possible, we say that $\tilde{\varphi}$ is a *reverse-engineered problem.*

The remarkable usefulness of the residual comes from the fact that in scientific computation we normally choose $\hat{\varphi}$ so that we can compute it efficiently. Consequently, even if finding the solution of $\hat{\varphi}$ is a problem of type C2 (as defined on p. 10), it is normally not too computationally difficult because we engineered the problem specifically to guarantee it is so. All that remains to do to compute the residual is the evaluation of $\phi(x, \hat{y})$, a simpler problem of type C1. Thus, the computational difficulty of computing the residual is much less than that of the forward error. Accordingly, we can usually compute the residual efficiently, thereby getting a measure of the quality of our solution. Consequently, it is simpler to reverse-engineer a problem by reflecting back the residual into the backward error than by reflecting back the forward error

Thus, the efficient computation of the residual allows us to gain important information concerning the reliability of a method on the grounds of what we have managed to compute with this method. In this context, we do not need to know as much about the intrinsic properties of a problem; we can use our computation method *a posteriori* to replace an *a priori* analysis of the reliability of the method. This allows us to use a feedback control method to develop an adaptive procedure that controls the quality of our solution "as we go." This shows why *a posteriori* error estimation is tremendously advantageous in practice.

The residual-based *a posteriori* error analysis that we emphasize in this thesis thus proceeds as follows:

1. For the problem φ, use an engineered version of the problem to compute the value $\hat{y} = \hat{\varphi}(x)$.

2. Compute the residual $r = \phi(x, \hat{y})$.

3. Use the defining equation and the computed value of the residual to obtain an estimate of the backward error. In effect, this amounts to (sometimes only approximately) reflecting back the residual as a perturbation of the input data.

4. Draw conclusions about the satisfactoriness of the solution in one of two ways:

   (a) If you do not require an assessment of the forward error, but only need to know that your have solved the problem for small enough perturbation Δx, conclude that your solution is satisfactory if the backward error (reflected back from the residual) is small enough.

   (b) If you require an assessment of the forward error, examine the condition of the problem. If the problem is well-conditioned and the computed solution amounts to a small backward error, then conclude that your solution is satisfactory.

We still have to add some more concepts regarding the stability of algorithms, and we will do so in the next section.

But before, it is important to not mislead the reader into think that this type of error analysis solves *all* the problems of computational applied mathematics! There are cases involving a complex interplay of quantitative and qualitative properties that prove to be challenging. We examine one such example from the computation of trajectories in chaotic systems in subsection 2.5. This reminds us of the following:

> A useful backward error-analysis is an explanation, not an excuse, for what may turn out to be an extremely incorrect result. The

> explanation seems at times merely a way to blame a bad result upon the data regardless of whether the data deserves a good result. (Kahan, 2009)

Thus, even if the perspective on backward error analysis presented here is extremely fruitful, it does not cure all evils.

1.5 Numerical properties of algorithms

An algorithm to solve a problem is a complete specification of how, exactly, to solve it: each step must be unambiguously defined in terms of known operations, and there must only be a finite number of steps. Algorithms to solve a problem φ correspond to the engineered problems $\hat{\varphi}$. There are many variants on the definition of an algorithm in the literature, and we will use the term loosely here. As opposed to the more restrictive definitions, we will count as algorithms methods that may fail to return the correct answer, or perhaps fail to return at all, and sometimes the method will use random numbers, thus failing to be deterministic. The key point for us is that the algorithms allow us to do computation with satisfactory results, this being understood from the point of view of mathematical tractability discussed before.

Whether $\hat{\varphi}(x)$ is satisfactory can be understood in different ways. In the literature, the algorithm-specific aspect of satisfactoriness is developed in terms of the numerical properties known as *numerical stability*, or just stability for short. Unfortunately "stability" is the most overused word in applied mathematics, and there is a particularly unfortunate clash with the use of the word in the theory of dynamical systems. In the terms introduced here, the concept of stability used in dynamical systems—which is a property of problems, not numerical algorithms—corresponds to "well-conditioning." Here, "stability" refers to the fact that an algorithm returns results that are about as accurate as the problem and the resources available allow. The takeaway message is that well-conditioning and ill-conditioning are properties of problems, while stability and instability are properties of algorithms.

The first sense of numerical stability corresponds to the forward analysis point of view: an algorithm $\hat{\varphi}$ is *forward stable* if it returns a solution $y = \hat{\varphi}(x)$ with a small forward error Δy. Note that, if a problem is ill-conditioned, there will typically not be any forward stable algorithm to solve it. Nonetheless, as we explained earlier, the solution can still be satisfactory from the backward error point of view. This leads us to define *backward stability*:

Definition 1. *An algorithm $\hat{\varphi}$ engineered to compute $y = \varphi(x)$ is* backward stable *if, for any x, there is a sufficiently small Δx such that*

$$\hat{y} = f(x + \Delta x), \qquad \|\Delta x\| \leq \varepsilon.$$

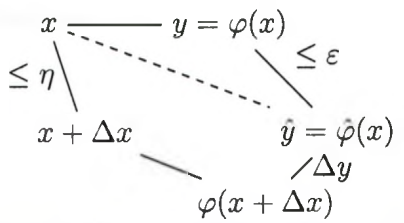
As mentioned before, what is considered "small", i.e., *how big ε is, is prescribed by the modeling context and, accordingly, is context dependent.* □

For example, the IEEE standard guarantees that $x \oplus y = x(1 + \delta x) + y(1 + \delta y)$, with $\delta \leq \varepsilon_M$. So, the IEEE standard in effect guarantees that the algorithms for basic floating-point operations are backward stable.

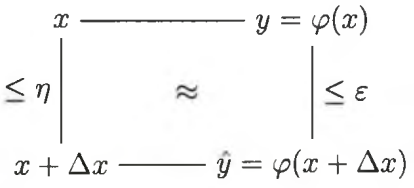
Note that an algorithm returning values with large forward errors can be backward stable. This happens particularly when we are dealing with ill-conditioned problems. As Higham (2002, p. 35) puts it:

> From our algorithm we cannot expect to accomplish more than from the problem itself. Therefore we are happy when its error $\hat{f}(x) - f(x)$ lies within reasonable bounds of the error $f(\hat{x}) - f(x)$ caused by the input error.

On that basis, we can introduce the concept of stability that we will use the most. It guarantees that we obtain theoretically informative solutions, while at the same time being very convenient in practice. Often, we only establish that $\hat{y} + \Delta y = f(x + \Delta x)$ for some small Δx and Δy. We do so either for convenience of proof, or because of theoretical limitations, or because we are implementing an adaptive algorithm as we described in subsection 1.4.3. Nonetheless, this is often sufficient from the point of view of error analysis. This leads us to the following definition (de Jong, 1977; Higham, 2002):



(a) Representation as a commutative diagram (Higham, 2002).



(b) Representation as an "approximately" commuting diagram (Robidoux, 2002). We can replace "$\approx$" by the order to which the approximation holds.

Figure 1.3: Stability in the mixed forward-backward sense

Definition 2. *An algorithm $\hat{\varphi}$ engineered to compute $y = \varphi(x)$ is* stable in the mixed forward-backward sense *if, for any x, there are sufficiently small Δx and Δy such that:*

$$\hat{y} + \Delta y = f(x + \Delta x), \quad \|\Delta y\| \leq \varepsilon \|y\|, \quad \|\Delta x\| \leq \eta \|x\|. \tag{1.41}$$

See figure 1.2. If this case, equation (1.41) *is interpreted as saying that $\hat{y}$ is almost the right answer for almost the right data or, alternatively, that the algorithm $\hat{\varphi}$ nearly solves the right problem for nearly the right data. We will also use the relative sense of stability with $\|\delta x\|$ instead.*

In most cases, when we say that an algorithm is *numerically stable* (or just stable for short), we will mean it in the mixed forward-backward sense of (1.41).

The solution to a problem $\varphi(x)$ is often obtained by replacing φ by a finite sequence of simpler problems $\varphi_1, \varphi_2, \ldots, \varphi_n$. In effect, given that the domains and codomains of the simpler subproblems match, this amounts to saying that

$$\varphi(x) = \varphi_n \circ \varphi_{n-1} \circ \cdots \circ \varphi_2 \circ \varphi_1(x). \tag{1.42}$$

For example, if the problem $\varphi(\mathbf{A}, \mathbf{b})$ is to solve the linear equation $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, we might use the LU factoring (*i.e.*, $\mathbf{A} = \mathbf{LU}$ for a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$) factorization to obtain the two

equations

$$\mathbf{Ly} = \mathbf{Pb} \tag{1.43}$$

$$\mathbf{Ux} = \mathbf{y}. \tag{1.44}$$

We have then decomposed $\mathbf{x} = \varphi(\mathbf{A}, \mathbf{b})$ into two problems; the first problem $\mathbf{y} = \varphi_1(\mathbf{L}, \mathbf{P}, \mathbf{b})$ consists in the simple task of solving a lower-triangular system and the second problem $\mathbf{x} = \varphi_2(\mathbf{U}, \mathbf{y})$ consists in the simple task of solving an upper-triangular system (see Corless and Fillion, 201x, chap. 4, "Solving $\mathbf{Ax} = \mathbf{b}$").

> Such decompositions are hardly unique. A good choice of $\varphi_1, \varphi_2, \ldots, \varphi_n$ may lead to a good algorithm for solving φ in this way: solve $\varphi_1(x)$ using its stable algorithm to get $\hat{y}_1$, then solve $\varphi_2(\hat{y}_1)$ using its stable algorithm to get $\hat{y}_2$, and so on. If the subproblems φ_1 and φ_2 are also well conditioned, by theorem 1, it follows that the resulting composed numerical algorithm for φ is numerically stable. (The same principle can be use as a very accurate rule of thumb for the formulations of the condition number not covered by theorem 1).

The *converse* statement is also very useful useful: decomposing a *well-conditioned* φ into two *ill-conditioned* subproblems $\varphi = \varphi_2 \circ \varphi_1$ will result in an *unstable* algorithm for φ, even if stable algorithms are available for each of the subproblems (unless the very unlikely event that the errors in $\hat{\varphi}_1$ and $\hat{\varphi}_2$ cancel each other out obtains[14]).

To a large extent, the problems we examine in this thesis and in Corless and Fillion (201x) are about decomposing problems into subproblems, and examining the correct numerical strategies to solve the subproblems. In fact, if you take any problem in applied mathematics, chances are that it will involve as subproblems things such as evaluating functions, finding roots of polynomials, solving linear systems, finding eigenvalues, interpolating function values, *etc.*

[14]For examples of when this happens, see Higham (2002).

Thus, in each chapter of Corless and Fillion (201x), a small number of "simple" problems are examined, so that the reader can construct the composed algorithm that is appropriate for his own composed problems.

1.6 Complexity and cost of algorithms

So far, we have focused on the accuracy and stability of numerical methods. In fact, this thesis focuses more on accuracy and stability than on cost and complexity. Nonetheless, we will at times need to address issues of complexity. To evaluate the cost of some method, we need two elements: (1) a count of the number of elementary operations required by its execution and (2) a measure of the amount of resources required by each type of elementary operations, or groups of operations. Following the traditional approach, we will only include the first element in our discussion.[15] Thus, when we will discuss cost of algorithms, we will really be discussing the number of floating-point operations required for the termination of an algorithm. The *computational complexity* of a problem is the cost of the algorithm solving this problem with the least cost, *i.e.*, what it would require to solve the problem using the cheapest method.

Typically, we will not be too concerned with the exact flop (*i.e.*, floating-point operation) count. Rather, we will only provide an order of magnitude determined by the highest-order terms of the expressions for the flop count. Thus, if an algorithm taking an input of size n requires $n^2/2 + n + 2$ flops, we will simply say that its cost is $n^2/2 + O(n)$ flops, or even just $O(n^2)$ flops. This way of describing cost is achieved by means of the *asymptotic notation*. The asymptotic notation uses the symbols Θ, O, Ω, o and ω to describe the comparative rate of growth of functions of n as n becomes large. In this thesis, however, we will only use the big-O and small-o notation, which are

[15]The second point in more relevant in computer science, where one might want to consider the relative computer costs of each type of floating-point operation and the memory requirements of methods as a whole.

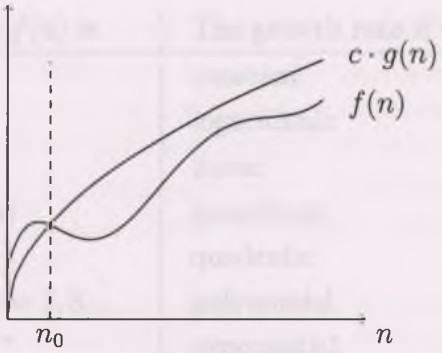


Figure 1.4: Asymptotic notation: $f(n) = O(g(n))$ if, for some c, $cg(n)$ asymptotically bounds $f(n)$ above as $n \to \infty$.

defined as follows:

$$f(n) = O(g(n)) \quad \text{iff} \quad \exists c > 0 \exists n_0 \forall n \geq n_0 \quad \text{such that} \quad 0 \leq f(n) \leq c \cdot g(n)$$
$$f(n) = o(g(n)) \quad \text{iff} \quad \forall c > 0 \exists n_0 \forall n \geq n_0 \quad \text{such that} \quad 0 \leq f(n) \leq c \cdot g(n)$$

Intuitively, a function $f(n)$ is $O(g(n))$ when its rate of growth with respect to n is the same or less than the rate of growth of $g(n)$, as depicted in figure 1.4 (in other words, $\lim_{n\to\infty} {}^{f(n)}/_{g(n)}$ is bounded). A function $f(n)$ is $o(g(n))$ in the same circumstances, except that the rate of growth of $f(n)$ must be strictly less than $g(n)$'s (in other words, $\lim_{n\to\infty} {}^{f(n)}/_{g(n)}$ is zero). Thus, $g(n)$ is an asymptotic upper bound for $f(n)$. However, with the small-o notation, the bound is not tight.

In our context, if we say that the cost of a method is $O(g(n))$, we mean that as n becomes large, the number of flops required will be at worst $g(n)$ times a constant. Let us introduce some standard terminology to qualify cost growth, from smaller to larger growth rate:

The cost $f(n)$ is	The growth rate if the cost is
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	quasilinear
$O(n^2)$	quadratic
$O(n^k)$, $k = 2, 3, \ldots$	polynomial
$O(k^n)$	exponential

We will also use this notation when writing sums. See appendix B.

Bibliography

Batterman, R. W. (2002). *The Devil in the Details: Asymptotic Reasoning in Explanation, Reduction, and Emergence.* Oxford University Press, Oxford.

Bender, C. M. and Orszag, S. A. (1978). *Advanced mathematical methods for scientists and engineers.* McGraw-Hill, New York.

Blum, L., Cucker, F., Shub, M., and Smale, S. (1998). *Complexity and Real Computation.* Springer, New York.

Brassard, G. and Bratley, P. (1996). *Fundamentals of algorithmics.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

Brent, R. P. and Zimmermann, P. (2011). *Modern computer arithmetic*, volume 18. Cambridge University Press, Cambridge, UK; New York.

Corless, R. M. and Fillion, N. (201x). A graduate survey of numerical methods. Forthcoming.

Davis, M. (1982). *Computability & Unsolvability.* Dover Publications.

de Jong, L. (1977). Towards a formal definition of numerical stability. *Numerische Mathematik*, 28(2):211–219.

Deuflhard, P. and Hohmann, A. (2003). *Numerical analysis in modern scientific computing: an introduction*, volume 43. Springer Verlag.

Forsythe, G. E. (1966). How do you solve a quadratic equation. Technical Report CS40, Stanford.

Forsythe, G. E. (1970). Pitfalls in computation, or why a math book isn't enough. *The American Mathematical Monthly*, 77(9):931–956.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, 2nd edition.

Immerman, N. (1999). *Descriptive complexity.* Springer Verlag.

Kahan, W. (2009). Needed remedies for the undebuggability of large-scale floating-point computations in science and engineering. Presented at the Computer Science Dept. Colloquia at the University of California, Berkely.

Muller, J., Brisebarre, N., de Dinechin, F., Jeannerod, C., Melquiond, G., Revol, N., Stehlé, D., and Torres, S. (2009). *Handbook of floating-point arithmetic.* Birkhauser.

Nagle, R., Saff, E., and Snider, A. (2000). *Fundamentals of differential equations and boundary value problems.* Addison-Wesley, 3rd edition.

Pour-El, M. and Richards, J. (1989). *Computability in analysis and physics.* Springer-Verlag, Berlin.

Robidoux, N. (2002). *Numerical solution of the steady diffusion equation with discontinuous coefficients.* PhD thesis, The University of New Mexico.

Trefethen, L. N. (1992). The definition of numerical analysis. *SIAM News*, 25(November 1992):6 and 22.

Trefethen, L. N. (2008). Numerical analysis. In *The Princeton Companion to Mathematics.* Princeton University Press.

Von Neumann, J. and Goldstine, H. (1947). Numerical inverting of matrices of high order. *Bull. Amer. Math. Soc*, 53(11):1021–1099.

Wilkinson, J. H. (1963). *Rounding Errors in Algebraic Processes.* Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs.

Chapter 2

Numerical Solution of ODEs

This chapter has two objectives. The first is to explain the use of basic state-of-the-art codes (such as the MATLAB codes). The second is to introduce the concepts and the perspective on which we base our criteria determining when to trust (and when to distrust) numerical solutions of ODE. Since numerical solution is our principal tool for nonlinear problems, this occupies an important place in the more general practice of scientific modeling. Consequently, we adopt a backward error perspective centered on the concept of *residual*, as explained in chapter 1. The residual of a differential equation is also called the *defect* and the *deviation* in the literature. The use of residuals for ODE is old, going back at least to Cauchy. However, it has not been used as much as it could be in numerical analysis, in spite of its distinguished history. Within the last few decades, however, this has begun to change and we believe that the right approach to the pedagogy and the usage of numerical methods for the solution of ODE includes the residual.

But before we talk about the residual, we should ask why one would use numerical methods at all. As a modest but typical example, consider the following simple-looking initial value problem (we use both x' and $\dot{x}$ to denote dx/dt):

$$\dot{x}(t) = t^2 + x(t) - \frac{x^4(t)}{10}, \qquad x(0) = 0 \tag{2.1}$$

on, say, the interval $0 \leq t \leq 5$. Differential equations of this ilk can easily be

found in applications; although this is a made-up problem, one could imagine that it had to do with population growth, where t is the time, and where x represents a population with spontaneous generation and a power-law death rate. If one tries to find the solution of this equation, then difficulties arise:

- The exact solution of this problem is not expressible in terms of elementary functions or known special functions (for methods to determine when this is the case, see, *e.g.*, Geddes et al., 1992; von zur Gathen and Gerhard, 2003; Bronstein, 2005) and
- High order power series solutions (see section 3.4) have numerical evaluation difficulties for large $(t - a)$ caused by catastrophic cancellation, similar to the difficulties suffered by other functions we have seen before such as AiryAi(x) or e^{-x}, due to "the hump" phenomenon—see chapters 1 and Corless and Fillion (201x, chap. 2).

Even for the problem posed in equation (2.1), which is vastly simpler than problems that occur in real models, the classical solution techniques fail us. In contrast, numerical solution on (for example) $0 \leq t \leq 5$ is simplicity itself when we use state-of-the-art codes such as MATLAB's `ode45` to get a reliable numerical solution. For this reason, in many applications, numerical solution will be considered the solution method *par excellence.*

2.1 Solving Initial Value Problems with `ode45` in MATLAB

MATLAB's `ode45` is an exemplar for easy-to-use state-of-the-art codes. We use it to introduce the reader to the idea of using professional codes. MATLAB's `ode45` requires three arguments:

1. A *function handle* corresponding to $f(t, x)$ in $\dot{x} = f(t, x)$;
2. A *time span*, *i.e.*, a 1×2 vector corresponding to the interval over which we will solve the equation numerically;

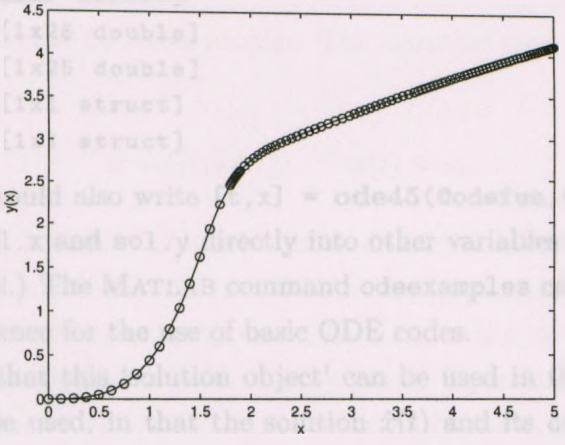


Figure 2.1: Numerical solution of equation (2.1).

3. An *initial value.*

For the problem described in equation (2.1), we could simply execute

```
f = @(t,x) t.^2 + x - x.^4/10;
ode45( f, [0,5], 0 );
```

These commands will dynamically generate the plot displayed in figure 2.1. Note that the solver has apparently produced a continuously differentiable function as a solution and plotted it. In fact, this graph has been generated step by step, using discrete units of time, and between each pair of points lies a separate function, which are then pieced together to produce a piecewise differentiable function on the interval of integration. We denote this continuous approximate solution of our initial value problem by $\hat{x}(t)$. This shows how simple the use of `ode45` can be. For most purposes, one will instead want to execute `ode45` in MATLAB with a left-hand side, *e.g.*,

```
sol = ode45( @odefun, tspan, x0 );
```

For the problem in equation (2.1), the resulting solution `sol` will then be a structured *object* of this kind:

```
sol =
      solver: 'ode45'
```

```
    extdata: [1x1 struct]
          x: [1x25 double]
          y: [1x25 double]
      stats: [1x1 struct]
      idata: [1x1 struct]
```

(Note that one could also write `[t,x] = ode45(@odefun,tspan,x0)` to put the values of `sol.x` and `sol.y` directly into other variables `t` and `x`, which is sometimes useful.) The MATLAB command `odeexamples` can be considered a good go-to reference for the use of basic ODE codes.

We will see that this 'solution object' can be used in the same way that a formula can be used, in that the solution $\hat{x}(t)$ and its derivative $\dot{\hat{x}}(t)$ can be evaluated at any desired point in the interval `tspan`. For now, we note that the points contained in the array `sol.x` are called the 'steps', 'nodes', or 'mesh' t_k for (here) $1 \leq k \leq 25$, and the points in the array `sol.y` are the corresponding values of $\hat{x}(t_k)$ (these were plotted with the circles in figure 2.1). We also note immediately that the value of $\hat{x}$ and $\dot{x}$ are available at off-mesh values of t, by use of the `deval` function, which automatically provides accurate interpolants and their derivatives for all solutions provided by the built-in solvers of MATLAB.

> In older books, a numerical solution to an IVP is considered to consist merely of a discrete mesh of times (*i.e.*, `sol.x`), together with the corresponding values of x at those times (*i.e.*, `sol.y`). In those books, while an analytic solution to an IVP is a function for which we know the rule or formula, a numerical solution is merely a discrete graph. Nowadays it is different: since there are algorithms implemented for evaluating the numerical solution at any point, and its derivative if we choose to ask for it, the distinction between an analytic solution and a numerical solution is not so great.

In the above example, the use of MATLAB's `ode45` required no preparation of the problem. However, it often happens that, if one wants to use standard

codes to solve a problem numerically, one has to rearrange the problem so that it is in a form that the code can process. The *standard form of an initial value problem* is

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t)), \qquad \mathbf{x}(t) = \mathbf{x}_0, \tag{2.2}$$

where $\mathbf{x} : \mathbb{R} \to \mathbb{C}^n$ is the vector-solution as a function of time, $\mathbf{x}_0 \in \mathbb{C}^n$ is the initial condition, and $\mathbf{f} : \mathbb{R} \times \mathbb{C}^n \to \mathbb{C}^n$ is the function equal to $\dot{\mathbf{x}}$. In terms of dynamical systems, $\mathbf{f}$ is a velocity vector field and $\mathbf{x}$ is a curve in phase space that is tangent to the vector field at every point. Equation (2.2) can thus be expanded as a system of coupled initial-value problems as follows:

$$\begin{aligned}
\dot{x}_1 &= f_1(t, x_1(t), x_2(t), \ldots, x_n(t)), & x_1(t_0) &= x_{1,0} \\
\dot{x}_2 &= f_2(t, x_1(t), x_2(t), \ldots, x_n(t)), & x_2(t_0) &= x_{2,0} \\
&\vdots & &\vdots \\
\dot{x}_n &= f_n(t, x_1(t), x_2(t), \ldots, x_n(t)), & x_n(t_0) &= x_{n,0}
\end{aligned}$$

It is also sometimes convenient to write the system in matrix-vector notation,

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t)) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{b}(t),$$

when dealing with linear or linearized systems, since the stability properties will then be partially explained in terms of the eigenvalues or the pseudospectra of $\mathbf{A}(t)$. In this case, the entries $a_{ij}(t)$ of $\mathbf{A}$ are assumed to be continuous in t and, for linear systems, do not depend on any of the x_i. For non-linear systems, the notation is sometimes abused to let the a_{ij}s depend on the x_is. The vector $\mathbf{b}(t)$ corresponds to the non-homogeneous part of the system, *i.e.*, it is a function of t only.

Finally, we observe that the system can always be modified so that it becomes an autonomous system (where $\mathbf{f}$ does not depend on t, *i.e.*, where $\mathbf{f}(t, \mathbf{x}(t)) = \mathbf{f}(\mathbf{x}(t))$). In order to do so, we simply add an $(n+1)$th component to $\mathbf{x}$ if necessary, so that the f_i are now of the form $f_i(x_1(t), \ldots, x_n(t), x_{n+1}(t))$,

and add an $(n+1)$th equation

$$\dot{x}_{n+1} = f_{n+1}(\, x_1(t),\, x_2(t),\, \ldots,\, x_n(t),\, x_{n+1}(t)\,) = 1 \qquad x_{n+1}(t_0) = t_0.$$

It is often convenient to deal with the autonomous form of systems, and we will freely do so. In this notation, we will often simply write $\mathbf{x}$ instead of $\mathbf{x}(t)$ and $\mathbf{A}$ instead of $\mathbf{A}(t)$.

As a first example, we show how to express *systems of first-order equations* in standard form. Consider the Lorenz system, in which we have a system of three first-order differential equations:

$$\begin{aligned} \dot{x} &= yz - \beta x & x(0) &= 27 \\ \dot{y} &= \sigma(z - y) & y(0) &= -8 \\ \dot{z} &= y(\rho - x) - z & z(0) &= 8 \end{aligned} \tag{2.3}$$

We use Saltzman's values of the parameters: $\sigma = 10, \rho = 28$ and $\beta = 8/3$. To express this system in a way that MATLAB can process, we simply make the trivial relabeling of variables $x_1(t) = x(t)$, $x_2(t) = y(t)$, and $x_3(t) = z(t)$:

$$\begin{aligned} \dot{x}_1 &= x_2 x_3 - \beta x_1 \\ \dot{x}_2 &= \sigma(x_3 - x_2) \\ \dot{x}_3 &= x_2(\rho - x_1) - x_3 \end{aligned}$$

We can then, if we need, rewrite the system in matrix-vector notation:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} x_2 x_3 - \beta x_1 \\ \sigma(x_3 - x_2) \\ x_2(\rho - x_1) - x_3 \end{bmatrix} = \begin{bmatrix} -\beta & 0 & x_2 \\ 0 & -\sigma & \sigma \\ -x_2 & \rho & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \mathbf{A}\mathbf{x}$$

Moreover, our initial conditions $x(0)$, $y(0)$, and $z(0)$ now form a vector

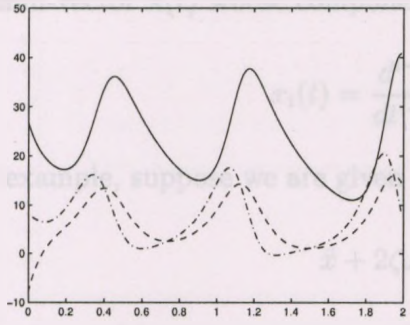
$$\mathbf{x}(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \\ x_3(0) \end{bmatrix} = \begin{bmatrix} 27 \\ -8 \\ 8 \end{bmatrix} = \mathbf{x}_0.$$

To conveniently treat in MATLAB a problem such as the one in our example above, we create an m-file similar to the one below :

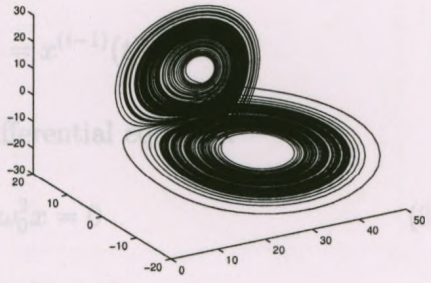
```
function sol=lorenzsys

rho   = 28;
sigma = 10;
beta  = 8/3;
tspan = [ 0, 100 ];
y0    = [ 27, -8, 8 ];
sol = ode45( @(t,y)lorenzeqs(t,y,rho,sigma,beta), tspan, y0 );

tplot = linspace( tspan(1), tspan(end), 1e4 );
yplot = deval( sol, tplot );
% first few points of the time history
plot( tplot(1:200),yplot(1,1:200),'k-', tplot(1:200),yplot
    (2,1:200),'k--',tplot(1:200),yplot(3,1:200),'k-.' )
figure
% phase diagram
plot3( yplot(1,:), yplot(2,:), yplot(3,:), '-k' )

end

function ydot=lorenzeqs(t,y,rho,sigma,beta)
   ydot(1,:) = -beta*y(1,:) + y(2,:).*y(3,:);
   ydot(2,:) = sigma*( y(3,:)-y(2,:) );
   ydot(3,:) = -y(2,:).*y(1,:) + rho*y(2,:)-y(3,:);
end
```

Note the use of a 'curried' in-line function call[1]

[1] A curried function, named after Haskell Curry (see, *e.g*, Curry and Feys, 1958) but first developed by Schönfinkel (1924), a transformation of a multivalued function to treat it as a sequence of single-valued functions.



(a) Time history of $x(t), y(t)$ and $z(t)$.



(b) Phase portrait for $\mathbf{x}(t)$.

Figure 2.2: Plots of the numerical solutions of the Lorenz system.

```
@(t,y)lorenzeqs(t,y,rho,sigma,beta)
```

in line 8 to create a function for `ode45` that does not have explicit reference to the parameters σ, ρ, and β. The first function simply provides the parameters and executes `ode45` (lines 3–8). The second function defines $\mathbf{f}(t, \mathbf{x})$ as described in equation (2.3) (line 20–24). Here, `sol` is a structure containing the $3 \times 1,491$ array `sol.y`; the row `sol.y(i,:)` will be the vector of computed values of $x_i(t_n)$. One can then easily obtain useful plots, such as time histories and phase portraits, using `deval` (lines 10–16). See figure 2.2. Note the use of `deval` to produce a good interpolation of the numerical solution, which is then fed into the functions `plot` and `plot3` to obtain graphical results.

Another trivial change of variables can be used to solve systems of *higher-order differential equations* in MATLAB. This time, the change of variables is used to transform a higher-order differential equation into a system of first-order differential equations;[2] we can then write it in vector notation as above if we like. In this case, a solution to the nth order initial-value problem is a

[2]Instead of this trivial change of variable, it is sometimes better to use physically relevant variables (see Ascher et al., 1988). In this thesis, we usually just use the trivial new set of variables.

column-vector $\mathbf{x}(t)$ whose components are

$$x_i(t) = \frac{d^{i-1}}{dt^{i-1}}x(t) = x^{(i-1)}(t).$$

For example, suppose we are given the differential equation

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \omega_0^2 x = 0 \tag{2.4}$$

for a damped harmonic oscillator. We first form the vector

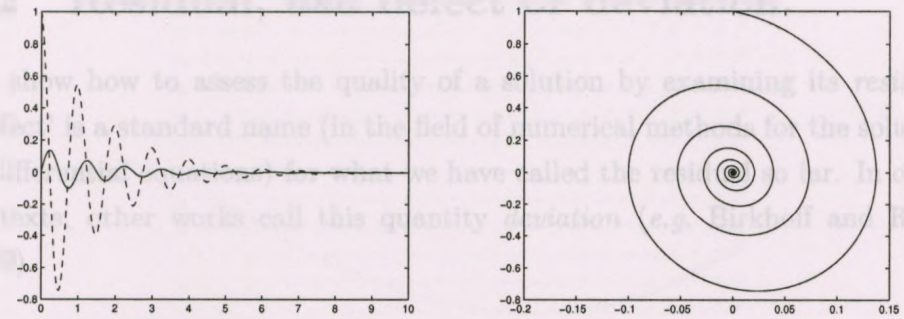
$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix},$$

so that

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix} = \begin{bmatrix} x_2 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -2\zeta\omega_0\dot{x} - \omega_0^2 x \end{bmatrix}$$
$$= \begin{bmatrix} x_2 \\ -2\zeta\omega_0 x_2 - \omega_0^2 x_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \mathbf{A}\mathbf{x} = \mathbf{f}(t, \mathbf{x}).$$

We can then use the MATLAB routine ode45 to find a numerical solution to this initial value problem. Again, we create an m-file similar to the one below:

```
function sol=dampedharmonicoscillator

tspan = [ 0, 10 ];
y0    = [ 0, 1 ]; %this is [x(0),x'(0)]
sol = ode45( @odefun, tspan, y0 );

tplot = linspace( tspan(1), tspan(end), 1000 );
yplot = deval( sol, tplot );
plot( tplot,yplot(1,:),'-k', tplot,yplot(2,:),'--k' )
figure
plot( yplot(1,:), yplot(2,:),'-k' ) %phase portait
end

function f=odefun(t,y)
```



(a) Time History for $x(t)$ and $\dot{x}(t)$. (b) Phase Portrait of $\mathbf{x}$.

Figure 2.3: Damped Harmonic Oscillator

```
15    omega = 2*pi;
16    zeta  = 0.1;  % parameters hard-coded this time
17    f = [ 0, 1; -omega^2, -2*zeta*omega ]*y;
18 end
```

Here, `sol.y(:,1)` will be the computed values of $x(t_n)$ and `sol.y(:,2)` will be the computed values of $\dot{x}(t_n)$. The graphical results are displayed in figure 2.3.

> The question we are generally addressing while attempting to solve a differential equation is: What do the solutions look like, for various initial conditions and parameter values? In the numerical context, we have various methods returning us various answers. The question thus becomes: *Are the numerical solutions faithful representations of solutions to the reference problem?*

In the next section, we look at an effective and efficient test based on computing the residual.

2.2 Residual, *aka* defect or deviation.

We show how to assess the quality of a solution by examining its *residual.* 'Defect' is a standard name (in the field of numerical methods for the solution of differential equations) for what we have called the residual so far. In other contexts, other works call this quantity *deviation* (*e.g.* Birkhoff and Rota, 1989).

For a given initial value problem $\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t))$, if we knew the exact solution $\mathbf{x}(t)$ and its derivative, we would obviously find that $\dot{\mathbf{x}}(t) - \mathbf{f}(t, \mathbf{x}(t)) = 0$. However, the numerical methods do not return us the exact solution $\mathbf{x}(t)$ and its exact derivative $\dot{\mathbf{x}}(t)$, but rather some approximate results $\hat{\mathbf{x}}(t)$ and $\dot{\hat{\mathbf{x}}}(t)$ (this is, as we have seen, what `deval` returns). But then, $\dot{\hat{\mathbf{x}}}(t) - \mathbf{f}(t, \hat{\mathbf{x}}(t))$ will not in general be zero; rather, we will have

$$\Delta(t) = \dot{\hat{\mathbf{x}}} - \mathbf{f}(t, \hat{\mathbf{x}}(t)),$$

where $\Delta(t)$ is what we call *absolute* defect or residual. We can also define the *relative* defect $\delta(t)$ componentwise (provided $\mathbf{f}(t, \hat{\mathbf{x}}(t)) \neq 0$) so that

$$\delta_i(t) = \frac{\dot{\hat{x}}_i - f_i(t, \hat{x})}{f_i(t, \hat{x})} = \frac{\dot{\hat{x}}_i}{f_i(t, \hat{x})} - 1.$$

As before, we can express the original problem in terms of a modified, or perturbed problem, so that our computed solution is an *exact* solution to this modified problem:

$$\dot{\hat{\mathbf{x}}} = \mathbf{f}(t, \hat{\mathbf{x}}) + \Delta(t).$$

The residual vector Δ is then a non-homogenous term added to the function $\mathbf{f}$, and equation (2.5) specifies a *reverse-engineered problem* that has been solved exactly by the numerical method.

Let us look at the residual from the point of view of dynamical systems. As

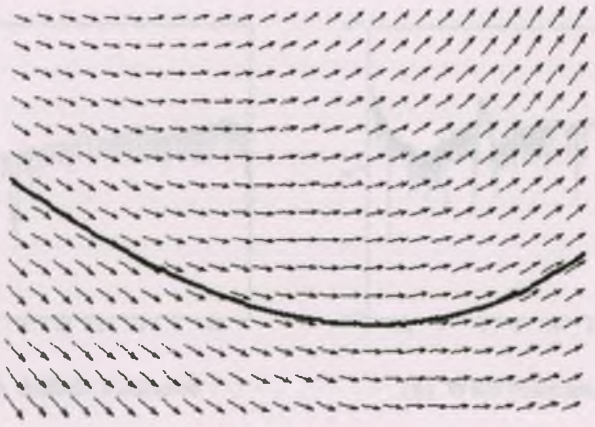


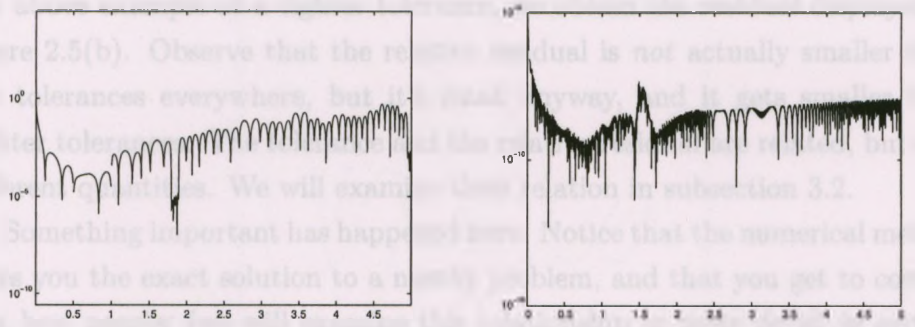
Figure 2.4: A vector field with a nearly tangent computed solution.

we have seen, in an ODE of the form $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t)$, the function $\mathbf{f}$ determines a velocity vector and a solution $\mathbf{x}(t)$ is then a curve in the phase space that is tangent to the vector field at every point (see figure 2.4). By computing the residual $\Delta(t) = \dot{\hat{\mathbf{x}}}(t) - \mathbf{f}(t, \hat{\mathbf{x}}(t))$, we are in effect measuring how far from satisfying the differential equation our computed trajectory $\hat{\mathbf{x}}(t)$ is, *i.e.*, how close it is to be tangent to the vector field. Alternatively, we can then say that the computed trajectory $\hat{\mathbf{x}}$ is tangent to a perturbed vector field $\mathbf{f}(t, \mathbf{x}) + \Delta(t)$.

Note that the residual is easily computed in MATLAB. The ODE solver returns a structure `sol` containing the evaluation points t_k (determined by the step sizes) as well as the values of $\hat{\mathbf{x}}(t_k)$. Using a suitable interpolant, we can then find a continuous and differentiable function $\hat{\mathbf{x}}(t)$, as well as its derivative $\dot{\hat{\mathbf{x}}}(t)$. But this is all one needs to compute the defect, since the function $\mathbf{f}$ is known from the beginning, being the definition of our initial value problem.

In practice, the computation of the defect is even simpler, since we usually don't have to worry about numerically interpolating and differentiating the interpolant.[3] For a given selection of points t_k (defined, *e.g.*, by `linspace`), the MATLAB command

[3]Nonetheless, Corless and Fillion (201x) examines examples where the built-in interpolant and its derivative have some difficulties.



(a) With default tolerance (b) With tolerance `1e-11`.

Figure 2.5: Easily computed residual of the solution of (2.1) with `ode45`. Notice the scale difference.

```
[ xhat , dotxhat ] = deval( sol , t )
```

returns us the values $\hat{\mathbf{x}}(t)$ and $\dot{\hat{\mathbf{x}}}(t)$ at any point in the interval. The defect is thus obtained almost automatically. As an example, consider again the problem in equation (2.1). One can obtain the residual as follows:

```
f   = @(t,x) t.^2 + x - x.^4/10;
sol = ode45( f, [0,5], 0 );
% Compute and plot the relative residual on a lot of points
t = linspace( 0,5,1001 );
[xhat,dotxhat] = deval(sol,t);
deltat = dotxhat ./ f(t,xhat)-1;
semilogy( t, abs(deltat), 'k-' )
```

In figure 2.5(a), one finds the residual for this problem. Observe that the maximum residual over the interval is quite large, *i.e.*, about 0.5.

To reduce the size of the residual, MATLAB offers the user the possibility of specifying a tolerance. To be specific, it allows the user to specify a *relative* and an *absolute tolerance*. The above example can be modified as follows in order to specify the tolerance:

```
opts = odeset( 'RelTol',1.0e-11, 'AbsTol',1.0e-11 );
sol  = ode45( f, [0,5], 0, opts );
```

The default absolute tolerance in MATLAB for `ode45` is 1.0×10^{-6}. If we run

our above example at a tighter tolerance, we obtain the residual displayed in figure 2.5(b). Observe that the relative residual is *not* actually smaller than the tolerances everywhere, but it's small anyway, and it gets smaller with tighter tolerances. The tolerance and the relative residual are related, but still different quantities. We will examine their relation in subsection 3.2.

Something important has happened here. Notice that the numerical method gives you the exact solution to a nearby problem, and that you get to control just how nearby (we will examine this relationship in more detail in section 3.2.2). You can look at the residual if you choose to do more computations.

> As we do throughout this thesis, from our *a posteriori* backward error analysis point of view, we then use our computation to approximate a backward error, and henceforth say that our numerical solution gives us the exact solution of of a nearby problem.

In our first example, we can then claim that we have provided an exact solution to

$$\dot{x} = t^2 + x - \frac{x^2}{10} + 10^{-6}v(t), \qquad 0 \leq t \leq 5,$$

where $v(t)$ is some noisy function such that $|v(t)| \leq 1$. The value $\varepsilon = 10^{-6}$ has been chosen based on the maximum computed value of the residual on the interval $[t_0, t_f]$.

In the example of the Lorenz system, we can find the residual over the last few points as follows (see figure 2.6):

```
t = linspace( sol.x(end-3), sol.x(end), 301 );
[yhat,dotyhat] = deval( sol, t );
Deltat = dotyhat - lorenzeqs( t,yhat, 28,10,8/3 );
plot( t, Deltat, 'k-' )
```

As a result, we say that our numerical solution is the exact solution of the nearby problem

$$\dot{\mathbf{x}} = \text{lorenzeqs}(\mathbf{x}) + 10^{-6}\mathbf{v}(t)$$

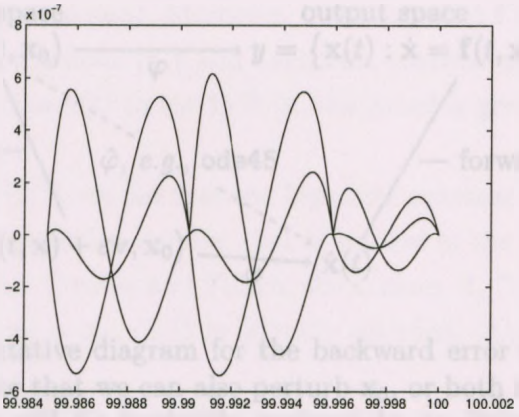


Figure 2.6: Residual components of the Lorenz system with `1e-9` tolerance with `ode45`.

where each component of the vector $\mathbf{v}(t) = [v_1(t), v_2(t), v_3(t)]^T$ is less than 1, *i.e.*, the vector satisfies $\|\mathbf{v}(t)\|_\infty \leq 1$.

2.3 Conditioning Analysis

In the previous section, we have defined the residual and we have seen how to compute it without pain. We then use this computed value to estimate a backward error. Now, from the general perspective developed in chapter 1, an initial value problem can be seen as a functional map

$$\varphi : \Big(\mathbf{f}(t, \bullet), \mathbf{x}_0\Big) \to \Big\{\mathbf{x}(t) : \dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t))\Big\}, \tag{2.5}$$

where $\mathbf{f}$ is a functional $\mathbb{R} \times \mathbb{C}^n \to \mathbb{C}^n$ (the tangent vector field) and $\mathbf{x}_0$ is the initial condition. We can then study the effects of three cases of backward errors: where we perturb $\mathbf{f}$, where we perturb $\mathbf{x}_0$, and where we perturb both. In the previous section, we have given two examples of perturbation of $\mathbf{f}$ with $\varepsilon\mathbf{v}$, where the magnitude of ε is the maximum computed residual and $\mathbf{v}$ is a noisy function with $\|\mathbf{v}\|_\infty \leq 1$ ($\mathbf{v}$ will sometimes be assumed to be a function of t only, as in subsection 2.3.2, and will sometimes be allowed to be a nonlinear

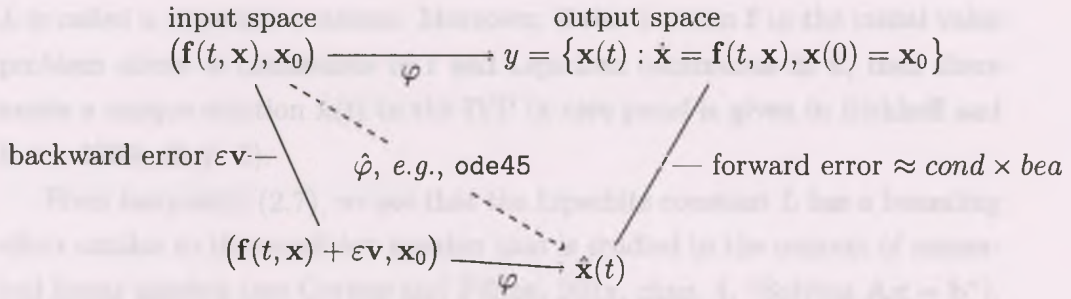


Figure 2.7: Commutative diagram for the backward error analysis of initial value problems. Note that we can also perturb $\mathbf{x}_0$, or both $\mathbf{x}_0$ and $\mathbf{f}$. In some cases, this diagram will be implicitly replaced by an "almost commutative diagram", as defined in chapter 1.

function of t and $\mathbf{x}$, as in subsection 2.3.3). This situation is represented in figure 2.7.

The question we ask in this section is: what effect do perturbations of $\mathbf{f}$, $\mathbf{x}_0$, or both have? That is what the *conditioning* of the initial value problem tells us. We begin by examining the effects of a perturbation of the initial condition only. Next, we examine the effects of a perturbation of the functional $\mathbf{f}$.

2.3.1 Conditioning & Lipschitz constants

A first assessment of the conditioning of an IVP can be obtained from Lipschitz constants. Consider the initial value problem

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t)) \qquad \mathbf{x}(0) = \mathbf{x}_0. \tag{2.6}$$

The function $\mathbf{f}(t,\mathbf{x})$ is said to be Lipschitz continuous in $\mathbf{x}$ with respect to a norm $\|\cdot\|$ over the interval $t \in [a,b]$ if there is a constant L such that for any $t \in [a,b]$ and for any two $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$,

$$\|\mathbf{f}(t,\mathbf{x}_1) - \mathbf{f}(t,\mathbf{x}_2)\| \leq L\|\mathbf{x}_1 - \mathbf{x}_2\|. \tag{2.7}$$

L is called a *Lipschitz constant.* Moreover, if the function $\mathbf{f}$ in the initial value problem above is continuous in t and Lipschitz continuous in $\mathbf{x}$, then there exists a unique solution $\mathbf{x}(t)$ to the IVP (a nice proof is given in Birkhoff and Rota, 1989, chap. 6).

From inequality (2.7), we see that the Lipschitz constant L has a bounding effect similar to the condition number that is studied in the context of numerical linear algebra (see Corless and Fillion, 201x, chap. 4, "Solving $\mathbf{Ax} = \mathbf{b}$"). Observe that when $\mathbf{x}_1 = \mathbf{x}_2$, (2.7) is trivially satisfied. Also, if $\mathbf{x}_1 \neq \mathbf{x}_2$, we have

$$\frac{\|\mathbf{f}(t, \mathbf{x}_1) - \mathbf{f}(t, \mathbf{x}_2)\|}{\|\mathbf{x}_1 - \mathbf{x}_2\|} \leq L.$$

Thus, L provides an upper bound on the effect that changes in $\mathbf{x}$ can have on $\mathbf{f}(t, \mathbf{x})$. Also, by the mean-value theorem, we can find that, for some $\overline{\mathbf{x}}$ between $\mathbf{x}_1$ and $\mathbf{x}_2$,

$$\|\mathbf{f}(t, \mathbf{x}_1) - \mathbf{f}(t, \mathbf{x}_2)\| = \|\mathbf{f}'(t, \overline{\mathbf{x}}) \cdot (\mathbf{x}_1 - \mathbf{x}_2)\| \leq \|\mathbf{J}_\mathbf{f}(\overline{\mathbf{x}})\| \|\mathbf{x}_1 - \mathbf{x}_2\|,$$

and so we can use the maximum norm of the Jacobian of $\mathbf{f}$ as Lipschitz constant.

However, we are not so much interested in the effects of a perturbation of $\mathbf{x}(t_0)$ on $\mathbf{f}$ as in its effect on $\mathbf{x}(t)$, the solution of the initial-value problem. To examine this problem, we first need a lemma.

Lemma 1 (Gronwall's Lemma). *If* $\mathbf{x}(t)$ *satisfies*

$$\dot{x}(t) \leq ax(t) + b, \qquad x(0) = x_0,$$

with $a, x_0 > 0$, b *constant, and,* $t > t_0$, *then*

$$x(t) \leq x_0 e^{at} + \frac{b}{a}(e^{at} - 1). \tag{2.8}$$

Proof. Rearranging the terms of the assumption and multiplying by e^{-at}, we

get

$$e^{-at}\dot{x}(t) - ae^{-at}x(t) \leq be^{-at}.$$

By Leibniz' rule, we then obtain

$$\frac{d}{dt}x(t)e^{-at} \leq be^{-at}.$$

Integrating both sides on $[0, t]$, and with the help of the fundamental theorem of calculus, we obtain

$$x(t)e^{at} - x(0) \leq \frac{b}{a}(1 - e^{-at}),$$

from which we get the lemma by rearranging the terms. □

Now, suppose that $\mathbf{y}(t)$ and $\mathbf{z}(t)$ are solutions of

$$\begin{aligned} \dot{\mathbf{y}}(t) &= \mathbf{f}(t, \mathbf{y}(t)), & \mathbf{y}(0) &= \mathbf{y}_0 \\ \dot{\mathbf{z}}(t) &= \mathbf{f}(t, \mathbf{z}(t)), & \mathbf{z}(0) &= \mathbf{z}_0 \end{aligned}$$

for a Lipschitz continuous $\mathbf{f}$. Then, a bound on $\|\mathbf{y}(t) - \mathbf{z}(t)\|$ in terms of $\|\mathbf{y}_0 - \mathbf{z}_0\|$ would give us an estimate of the sensitivity to perturbation of the initial condition. For practical purpose, we will typically be interested to establish this bound for the 2-norm. First, observe that for the 2-norm, a useful relation holds between the derivative of the norm of a function and the norm of the derivative of a function:

$$\frac{d}{dt}\|g(t)\|_2^2 = \frac{d}{dt}\langle g(t), g(t)\rangle = 2g(t)\frac{d}{dt}g(t) = 2\left\langle g(t), \frac{d}{dt}g(t)\right\rangle$$

So, by the Cauchy-Schwartz inequality, we have

$$\frac{d}{dt}\|g(t)\|_2^2 = 2\left\langle g(t), \frac{d}{dt}g(t)\right\rangle \leq 2\left|\left\langle g(t), \frac{d}{dt}g(t)\right\rangle\right|^2 \leq 2\|g(t)\|_2\|\frac{d}{dt}g(t)\|_2.$$

Consequently,

$$\frac{d}{dt}\|g(t)\|_2 = \frac{1}{2\|g(t)\|_2}\frac{d}{dt}\|g(t)\|_2 \leq \frac{1}{2\|g(t)\|_2}2\|g(t)\|_2\|\frac{d}{dt}g(t)\|_2 = \|\frac{d}{dt}g(t)\|_2.$$

Now, let $g(t) = \mathbf{y}(t) - \mathbf{z}(t)$ in the above argument. Since $\mathbf{f}$ is Lipschitz continuous, we find that

$$\begin{aligned}\frac{d}{dt}\|\mathbf{y}(t) - \mathbf{z}(t)\|_2 &\leq \|\frac{d}{dt}(\mathbf{y}(z) - \mathbf{z}(t))\|_2 = \|\mathbf{f}(t, \mathbf{y}(t)) - \mathbf{f}(t, \mathbf{z}(t))\|_2 \\ &\leq L\|\mathbf{y}(t) - \mathbf{z}(t)\|_2.\end{aligned}$$

Thus, $\|\mathbf{y}(t) - \mathbf{z}(t)\|_2$ satisfies the hypothesis of Gronwall's lemma with $a = L$ and $b = 0$.

Therefore,

$$\|\mathbf{y}(t) - \mathbf{z}(t)\|_2 \leq \|\mathbf{y}_0 - \mathbf{z}_0\|_2 e^{Lt}.$$

Consequently, if $\mathbf{f}$ is Lipschitz continuous and we know the Lipschitz constant L, we can find an upper bound on the effect of a perturbation of the initial condition on the trajectories. However, since $L > 0$, there may be exponential separation of the trajectories.

It is important to understand that L normally gives a pessimistic evaluation of the quality of the solution. Thus, if we want to use the Lipschitz constant as a condition number, it is important to consider the smallest Lipschitz constant for the problem. Moreover, even if the Lipschitz constant provides a tight bound, it is possible to tighten it further using the Dahlquist constant instead (Söderlind, 1984).

2.3.2 Condition *via* the variational equation

Consider the autonomous initial-value problem

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t)), \qquad \mathbf{x}(0) = \mathbf{x}_0 \tag{2.9}$$

and the corresponding perturbed autonomous initial value problem

$$\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}(t)) + \varepsilon\mathbf{v}(t), \qquad \mathbf{z}(0) = \mathbf{z}_0. \tag{2.10}$$

We denote their solutions $\mathbf{x}(t)$ and $\mathbf{z}(t)$, respectively, and we let $\mathbf{x}_0 = \mathbf{z}_0$. Notice that since $\mathbf{z}(t)$ is the solution of the perturbed problem, we can also write $\mathbf{z}(t) = \mathbf{x}(t, \varepsilon)$ as a regular perturbation series, using the notation of appendix B. In this context, ε is a small number and $\varepsilon\mathbf{v}$ a small perturbation, and we will accordingly investigate $\|\mathbf{x}(t) - \mathbf{z}(t)\|$ as $\varepsilon \to 0$. The question is then: as $\varepsilon \to 0$, does $\mathbf{x}(t, \varepsilon)$ converge to $\mathbf{x}(t)$ as $t \to \infty$? We will examine the Gröbner-Alexeev approach in the next subsection. However, we take an easier approach here, and linearize the exact solution about the computed solution.[4] This type of reasoning is standard in the theory of dynamical systems (see, *e.g.*, Nagle et al., 2000; Lakshmanan and Rajasekar, 2003); in effect, we will study the relation between $\mathbf{x}$ and $\mathbf{z}$ in the *tangent space*. The information we obtain will be valid only insofar as the tangent space trajectories represent well the original trajectories (*i.e.*, in a small neighbourhood).

Consider the asymptotic expansion of $\mathbf{z}(t)$—which, remember, is just $\mathbf{x}(t, \varepsilon)$—in powers of the perturbation ε:

$$\mathbf{z}(t) = \mathbf{x}_0(t) + \varepsilon\mathbf{x}_1(t) + O(\varepsilon^2).$$

By formula (B.9), since the limit of $\mathbf{x}(t, \varepsilon)$ as $\varepsilon \to 0$ is just $\mathbf{x}(t)$, we have $\mathbf{x}_0(t, \varepsilon) = \mathbf{x}(t)$, giving us

$$\mathbf{z}(t) = \mathbf{x}(t) + \mathbf{x}_1(t)\varepsilon + O(\varepsilon^2). \tag{2.11}$$

[4] We again assume that we have interpolated the numerical solution so that it has a sufficient number of derivatives.

We want to solve for $\mathbf{x}_1(t)$ to determine the first-order effect of the perturbation on the solution, since $\mathbf{z}(t) - \mathbf{x}(t) \doteq \varepsilon \mathbf{x}_1(t)$.

The derivative of equation (2.11) is

$$\begin{aligned}\dot{\mathbf{z}}(t) &= \dot{\mathbf{x}}(t) + \dot{\mathbf{x}}_1(t)\varepsilon + O(\varepsilon^2) \\ &= \mathbf{f}(\mathbf{x}(t)) + \dot{\mathbf{x}}_1(t)\varepsilon + O(\varepsilon^2).\end{aligned}$$

Since it follows from equation (2.11) that $\mathbf{x}(t) = \mathbf{z}(t) - \mathbf{x}_1(t)\varepsilon - O(\varepsilon^2)$, we can substitute and expand $\mathbf{f}$ about the computed solution $\mathbf{z}(t)$:

$$\begin{aligned}\dot{\mathbf{z}}(t) &= \mathbf{f}\Big(\mathbf{z}(t) - \mathbf{x}_1(t)\varepsilon - O(\varepsilon^2)\Big) + \dot{\mathbf{x}}_1(t)\varepsilon + O(\varepsilon^2) \\ &= \mathbf{f}(\mathbf{z}(t)) + \mathbf{f}'(\mathbf{z}(t))\Big((\mathbf{z}(t) - \varepsilon\mathbf{x}_1(t) - O(\varepsilon^2)) - \mathbf{z}(t)\Big) + \varepsilon\dot{\mathbf{x}}_1(t) + O(\varepsilon^2) \\ &= \mathbf{f}(\mathbf{z}(t)) + \mathbf{f}'(\mathbf{z}(t))(-\varepsilon\mathbf{x}_1(t)) + \varepsilon\dot{\mathbf{x}}_1(t) + O(\varepsilon^2)\end{aligned}$$

Now, by equation (2.10), we obtain

$$\dot{\mathbf{z}}(t) - \mathbf{f}(\mathbf{z}(t)) = \varepsilon\mathbf{v}(t) = \varepsilon\dot{\mathbf{x}}_1(t) - \varepsilon\mathbf{x}_1(t)\mathbf{J}_\mathbf{f}(\mathbf{z}(t),$$

where the partial derivative in the Jacobian $\mathbf{J}_\mathbf{f}$ are with respect to $\mathbf{z}$.

Neglecting the higher powers of ε and rearranging the terms, we finally obtain

$$\dot{\mathbf{x}}_1(t) = \mathbf{J}_\mathbf{f}(\mathbf{z}(t))\mathbf{x}_1(t) + \mathbf{v}(t), \qquad \mathbf{x}_1(t_0) = \mathbf{0}. \tag{2.12}$$

This is the *first variational equation* (Bender and Orszag, 1978).

The exact, analytic solution of this equation involves the machinery for linear non-homogeneous equations with (possibly) variable coefficients. In what follows, we only briefly sketch how to solve the equation for $\mathbf{x}_1$ in the matrix-vector notation (for more details, see Nagle et al. (2000) or Kaplan (2002)). This is an aside that does not have much to do with numerics, but

it will help to fix the notation and to explain the mathematical objects we're dealing with in the codes. We will see at the end of this section how to implement all of this numerically.

Consider the homogenous part of the variational equation (2.12), $\dot{\mathbf{x}}_1 = \mathbf{J_f}(\mathbf{z})\mathbf{x}_1$ (we'll drop the subscript "1" below since the method applies generally to linear systems). If $\mathbf{x}_1(t), \mathbf{x}_2(t), \ldots, \mathbf{x}_n(t)$ are solutions of $\dot{\mathbf{x}} = \mathbf{J_f}(\mathbf{z})\mathbf{x}$ and the Wronskian is non-zero, *i.e.*, if

$$W(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \det \begin{bmatrix} \mathbf{x}_1 & \cdots & \mathbf{x}_n \end{bmatrix} \neq 0,$$

then the solutions are linearly independent, so that the general solutions of $\dot{\mathbf{x}} = \mathbf{J_f}(\mathbf{z})\mathbf{x}$ are

$$\mathbf{x}_h(t) = \mathbf{x}_1(t)c_1 + \mathbf{x}_2(t)c_2 + \ldots + \mathbf{x}_n(t)c_n = \mathbf{X}(t)\mathbf{c},$$

and $\mathbf{X}(t)$ is then called a *fundamental solution matrix.* As one can easily verify from the above definition, the fundamental solution matrix satisfies the matrix differential equation $\dot{\mathbf{X}}(t) = \mathbf{J_f}(\mathbf{z})\mathbf{X}(t)$. Moreover, we can always choose a fundamental matrix whose initial conditions will be $\xi(0) = \mathbf{I}$ by applying a transformation $X(t) = \xi(t)\mathbf{C}$ ($\mathbf{C}$ constant), so that $\mathbf{I} = \mathbf{X}^{-1}(0)\mathbf{C}$. Then,

$$\dot{\xi}(t) = \mathbf{J_f}(\mathbf{z})\xi(t), \qquad \xi(0) = \mathbf{I} \tag{2.13}$$

is called the *associated matrix variational equation.* Note that there is no general method to identify the fundamental solutions filling up the matrix $\mathbf{X}(t)$, when the components of $\mathbf{J_f}(\mathbf{z})$ are not constant, in terms of elementary functions, but it is known for some classes of problems.[5]

Whenever $\mathbf{X}(t)$ is a fundamental matrix, it is non-singular, and as a result the coefficients c_i are uniquely identifiable for a given initial-value problem as $\mathbf{c} = \mathbf{X}^{-1}(t_0)\mathbf{x}(t_0)$. Therefore, we find that the solution of the homogeneous

[5] Also, note that there *are* algorithms that will find solutions when they are findable, and prove that they are not when they are not. See, *e.g.*, Bronstein and Lafaille (2002).

system is

$$\mathbf{x}_h(t) = \mathbf{X}(t)\mathbf{X}^{-1}(t_0)\mathbf{x}(t_0).$$

Once we know a set of fundamental solutions for the homogeneous part of the system, we can find a particular solution $\mathbf{x}_p(t)$ to the non-homogenous system by variation of parameters, obtaining a general solution with the superposition principle $\mathbf{x}(t) = \mathbf{x}_h(t) + \mathbf{x}_p(t)$. The solution of the inhomogeneous system, $\dot{\mathbf{x}} = \mathbf{J_f}(\mathbf{z})\mathbf{x} + \mathbf{v}$, is obtained by letting the coefficients c_i be functions of t, so that $\mathbf{x}_p(t) = \mathbf{X}(t)\mathbf{c}(t)$. Note that the derivative of $\mathbf{x}_p(t)$ is

$$\dot{\mathbf{x}}_p(t) = \dot{\mathbf{X}}(t)\mathbf{c}(t) + \mathbf{X}(t)\dot{\mathbf{c}}(t),$$

so that, by substituting in $\dot{\mathbf{x}} = \mathbf{J_f}(\mathbf{z})\mathbf{x} + \mathbf{v}$, we obtain

$$\dot{\mathbf{X}}(t)\mathbf{c}(t) + \mathbf{X}(t)\dot{\mathbf{c}}(t) = \mathbf{J_f}(\mathbf{z})\mathbf{X}(t)\mathbf{c}(t) + \mathbf{v}(t)$$

Since the fundamental solution matrix satisfies $\dot{\mathbf{X}}(t) = \mathbf{J_f}(\mathbf{z})\mathbf{X}(t)$, we find that $\dot{\mathbf{c}}(t) = \mathbf{X}^{-1}(t)\mathbf{v}(t)$ (since $\mathbf{X}$ is invertible). By integration of $\dot{\mathbf{c}} = \mathbf{X}^{-1}\mathbf{v}$, we finally identify the variable coefficients and the particular solution:

$$\mathbf{x}_p(t) = \mathbf{X}(t)\int_{t_0}^{t} \mathbf{X}^{-1}(\tau)\mathbf{v}(\tau)d\tau.$$

Therefore, the general solution of the variational equation is (we reintroduce the subscript "1"):

$$\mathbf{x}_1(t) = \mathbf{X}_1(t)\mathbf{X}_1(t_0)\mathbf{x}_1(t_0) + \int_{t_0}^{t} \mathbf{X}_1(t)\mathbf{X}_1^{-1}(\tau)\mathbf{v}(\tau)d\tau. \tag{2.14}$$

But since $\mathbf{x}_1(t_0) = \mathbf{0}$, the homogeneous term is just $\mathbf{0}$. This gives us the following expression for $\mathbf{z}(t) - \mathbf{x}(t)$,

$$\mathbf{z}(t) - \mathbf{x}(t) = \varepsilon\mathbf{x}_1(t) = \varepsilon\int_{t_0}^{t} \mathbf{X}_1(t)\mathbf{X}_1^{-1}(\tau)\mathbf{v}(\tau)d\tau,$$

where again we ignored the higher powers of ε. Now, it follows that

$$\|\mathbf{z}(t) - \mathbf{x}(t)\| = \varepsilon \left\| \mathbf{X}_1(t) \int_{t_0}^{t} \mathbf{X}_1^{-1}(\tau)\mathbf{v}(\tau)d\tau \right\| \leq \varepsilon \|\mathbf{X}_1(t)\| \int_{t_0}^{t} \|\mathbf{X}_1^{-1}(\tau)\| \|\mathbf{v}(\tau)\| d\tau$$

by the submultiplicativity of the norm.

Therefore, we obtain the inequality

$$\|\mathbf{z}(t) - \mathbf{x}(t)\| \leq \|\mathbf{X}_1(t)\| \max_{t_0 \leq \tau \leq t} \|\mathbf{X}_1^{-1}(\tau)\| \int_{t_0}^{t} \varepsilon \|\mathbf{v}(\tau)\| d\tau, \tag{2.15}$$

where

$$\kappa(\mathbf{X}_1) = \|\mathbf{X}_1(t)\| \max_{t_0 \leq \tau \leq t} \|\mathbf{X}_1^{-1}(\tau)\| \tag{2.16}$$

acts as a condition number and the integral of $\varepsilon\|\mathbf{v}\|$ is a norm of $\widetilde{\varepsilon\mathbf{v}}$. Thus, if the fundamental solution matrix of the variational equation $\mathbf{X}_1$ is well-conditioned, then we can expect an accurate numerical solution since the term $\mathbf{v}(t)$ will be damped, or at least won't grow too much.

As we show in Corless and Fillion (201x), the norm of a matrix $\mathbf{A}$ equals the largest singular value σ_1 of $\mathbf{A}$ and the norm of $\mathbf{A}^{-1}$ is the inverse of the smallest singular value of $\mathbf{A}$, denoted σ_n^{-1}. Accordingly, we can write the condition number as

$$\kappa(\mathbf{X}) = \frac{\sigma_1(t)}{\max\limits_{t_0 \leq \tau \leq t} \sigma_n(\tau)}.$$

It is not an accident that the singular-value decomposition plays a role in this context. A standard tool for the evaluation of the effect of $\mathbf{X}$ on $\|\mathbf{z}(t) - \mathbf{x}(t)\|$ is the computation of the Lyapunov exponents λ_i of $\mathbf{X}(t)$ (see, *e.g.*, Geist et al., 1990). These λ_i are defined as the *logarithms of the eigenvalues of* Λ, where

$$\Lambda = \lim_{t \to \infty} \left(\mathbf{X}^T\mathbf{X}\right)^{\frac{1}{2t}}. \tag{2.17}$$

In other words, the Lyapunov exponents are closely related to the eigenvalues of $\mathbf{X}^T\mathbf{X}$. But as we show in Corless and Fillion (201x, chap. 4, "Solving $\mathbf{Ax} = \mathbf{b}$"), the eigenvalues of $\mathbf{X}^T\mathbf{X}$ are just the squares of the singular values of $\mathbf{X}$ (since $\mathbf{X}^T\mathbf{X} = (\mathbf{U}\Sigma\mathbf{V}^H)^H\mathbf{U}\Sigma\mathbf{V}^H = \mathbf{V}\Sigma^2\mathbf{V}^H$). Note that we use the analytic SVD in this case (Bunse-Gerstner et al., 1991). If we take a small displacement in $\mathbf{x}$ (so far, that's what we labeled $\varepsilon\mathbf{x}_1$), the singular value decomposition gives us a nice geometrical interpretation of equation (2.17). The SVD factoring $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^H$ guarantees that $\mathbf{XV} = \Sigma\mathbf{U}$ for some unitary (in the real case, orthogonal) set of vectors $\{\mathbf{v}_i\}$ and $\{\mathbf{u}_i\}$. We can apply it to the displacement vector $\varepsilon\mathbf{x}_1$ to get $\mathbf{XV}\varepsilon\mathbf{x}_1 = \Sigma\mathbf{U}\varepsilon\mathbf{x}_1$; since the unitary transformations $\mathbf{V}\varepsilon\mathbf{x}_1$ and $\mathbf{U}\varepsilon\mathbf{x}_1$ of $\varepsilon\mathbf{x}_1$ preserve 2-norm, we see that the singular values show how much the matrix $\mathbf{X}(t)$ stretches the displacement vector $\varepsilon\mathbf{x}_1$ (here, since the original equation is nonlinear, the singular values will be function of t). Now, if we take the logarithmic average of the singular values when $t \to \infty$ (because we are interested with average exponential growth), we get

$$\lim_{t\to\infty} \frac{1}{t} \ln \sigma_i(t) = \lim_{t\to\infty} \ln \left(\sigma_i^2(t)\right)^{\frac{1}{2t}} = \lim_{t\to\infty} \ln \mathrm{eig}_i(\mathbf{X}^T\mathbf{X})^{\frac{1}{2t}} = \ln \mathrm{eig}_i(\Lambda) = \lambda_i.$$

Thus, if the Lyapunov exponents of $\mathbf{X}$ are positive, the average exponential growth of the displacement $\varepsilon\mathbf{x}_1$ will be *positive*, and so our initial-value problem will be exponentially ill-conditioned. However, remember that this is the behaviour in the tangent space, since we linearized the initial value problem. Thus, this does *not* imply that the displacement $\varepsilon\mathbf{x}_1$ is unbounded, since the non-linear terms might have an effect as we move away from the point about which we linearized. In fact, when it remains bounded, we will often face chaotic systems. We will return on this issue in subsection 2.5.

Now, all of these tools somehow require to compute $\mathbf{X}_1(t)$ in some way, or at least a computation of $\mathbf{X}_1(t_k)$ on a sufficient number of nodes t_k.[6] An alternative to solving for $\mathbf{x}_1(t)$ analytically consists in solving the variational equation numerically.

[6]It would also be possible to use a strictly qualitative approach consisting in finding the stable/unstable equilibria and to determine whether our trajectory is attracted/repulsed by it.

However, the reader might want to ask: how do we know the numerical solution of the variational equation is reliable? It may seem that in order to determine whether it is or not, we would need to find *its* variational equation, and solve it numerically. Does it not lead to a regress *ad infinitum*? No. The first observation we need to make is that whereas the reference initial-value problem is in general nonlinear, the variational equation is linear. As a result, the variational equation is its own variational equation, and so there is no regress.

Moreover, our numerical solution of the variational equation can either show that the reference problem is ill-conditioned or not. On the one hand, if the computed solution shows that the components of $\mathbf{X}_1$ are large, then the variational equation is claimed to be ill-conditioned. This must be the case: if it were well-conditioned, then because we are using a stable method we will find a small forward error and thus a small $\mathbf{X}_1$). We may well not identify $\mathbf{X}_1$ very accurately, but we will see that it is large. On the other hand, if the computed solution shows that the entries of $\mathbf{X}_1$ are not large, then the variational equation is claimed to be well-conditioned. However, obtaining a computed solution with a small $\mathbf{X}_1$ is consistent with an ill-conditioned solution. How likely is it that an ill-conditioned $\mathbf{X}_1$ will have all its errors arranged in such a way that the computed answer is small and thus the equation appears well-conditioned? This can indeed happen, but then:

> Anyone unlucky enough to encounter this sort of calamity has probably already been run over by a truck. (Higham, 2002, p. 242)

As a result, we will typically not worry about "false negatives."

Let us examine an example in detail. Consider this initial value problem:

$$\begin{aligned} \dot{x_1} &= -x_1^3 x_2, \qquad x_1(0) = 1 \\ \dot{x_2} &= -x_1 x_2^2, \qquad x_2(0) = 1 \end{aligned} \tag{2.18}$$

We may solve this simple nonlinear autonomous system analytically to find

that

$$x_1(t) = \frac{1}{1+W(t)} \qquad \text{and} \qquad x_2(t) = e^{-W(t)},$$

where W is the principal branch of the Lambert W function (see Corless et al., 1996). This function has a derivative singularity at $t = -1/e \doteq -0.3679$. Now, if we didn't know all this, we would solve the problem numerically. As we show, we can also use the theory presented in this section to track the condition number of the problem numerically. Since we can also track the residual numerically as shown before, we have all the ingredient for an *a posteriori* error analysis of our solution.

The trick is to simultaneously solve the associated matrix equation of the variational equation (2.13), $\dot{\xi}(t) = \mathbf{J_f}(\mathbf{z})\xi(t)$ with $\xi(0) = \mathbf{I}$, and the system above. So, we first compute the Jacobian by hand (or with MAPLE),

$$\mathbf{J} = \begin{bmatrix} -3z_1^2 z_2 & -z_1^3 \\ -z_2^2 & -2z_1 z_2 \end{bmatrix}$$

and then expand the matrix product $\mathbf{J_f}(\mathbf{z})\xi(t)$. We can then create an m-file solving for all the components of

$$\mathbf{y} = \begin{bmatrix} x_1 & x_2 & \xi_{11} & \xi_{12} & \xi_{21} & \xi_{22} \end{bmatrix}^T$$

simultaneously. The function $\mathbf{f}(t, \mathbf{y})$ will then be as follows:

```
function yp=bothodes(t,y)
    yp=zeros(size(y));
    %yp=[x1;x2;xi11;xi12;xi21;xi22];
    yp(1,:)=-y(1,:).^3.*y(2,:);
    yp(2,:)=-y(1,:).*y(2,:).^2;
    yp(3,:)=-3*y(1,:).^2.*y(2,:).*y(3,:)-y(1,:).^3.*y(5,:);
    yp(4,:)=-3*y(1,:).^2.*y(2,:).*y(4,:)-y(1,:).^3.*y(6,:);
    yp(5,:)=-y(2,:).^2.*y(3,:)-2*y(1,:).*y(2,:).*y(5,:);
    yp(5,:)=-y(2,:).^2.*y(4,:)-2*y(1,:).*y(2,:).*y(6,:);
end
```

With this function defined, we can then use the code below to do all these things:

1. Solve our initial value problem for x_1 and x_2 (lines 2-11);
2. Solve for the components of the fundamental solution matrix ξ_{ij} (lines 2-11 again, since they are solved simultaneously);
3. Find the absolute and relative residual (lines 23-27);
4. Estimate the condition number of the problem (lines 36-46);
5. Plot the results on the appropriate scale (lines 29-34 and 47-48).

In addition, the lines 13-21 are refining the mesh size to obtain more significant graphical output (the reader is invited to use `deval` on a mesh ignoring adaptive stepsize selection to see that the result does not look right).

```
1  function variationaleq_ex
2  % Simple nonlinear autonomous example
3  % dotx1 = -x1^3x2  and dotx2 = -x1x2^2, with x1(0)=x2(0)=1.
4  % J = Jacobian matrix, so the associated matrix variational
       equation is dotXI = J XI, XI(0) = eye(2).
5  tspan = [0,-0.36];
6  % Initial conditions for x_1, x_2, and xi = eye(2)
7  Y0 = [1,1,1,0,0,1];
8  % Integrate to reasonably tight tolerances
9  opts = odeset( 'reltol', 1.0e-8, 'abstol', 1.0e-8 );
10 % Put the solution in a structure, to compute the residual.
11 sol = ode45( @bothodes, tspan, Y0, opts );
12
13 % We refine the mesh ourselves so as to be sure that our
       residual computation reflects the actual changes in the
       solution as found by ode45.
14 nRefine = 9;
15 n = length( sol.x );
16 size( sol.x )
17 h = diff( sol.x );
18 tRefine = repmat( sol.x(1:end-1).', 1, nRefine ) ;
```

```
tRefine = (tRefine + (h.')*[0:nRefine-1]/nRefine).' ;
tRefine = tRefine(:);
numpoints = length(tRefine);

% Now compute the residual.
[yhat,yphat] = deval(sol,tRefine);
Resid = yphat - bothodes(tRefine,yhat);
% The residual relative to the size of the rhs is also of
   interest.
RResid = yphat./bothodes(tRefine,yhat) - 1;

figure(1),plot(tRefine,Resid(1,:),'k-')
title('Residual in x_1')
figure(2),plot(tRefine,Resid(2,:),'k-')
title('Residual in x_2')
figure(3), semilogy(tRefine,abs(RResid), 'k-')
title('Relative Residual')

% Now look at the condition number
sigma1 = zeros(1,n);
sigma2 = zeros(1,n);
cond   = zeros(1,n);
for k=1:n,
    Xt=[sol.y(3,k),sol.y(4,k);sol.y(5,k),sol.y(6,k)];
    sigma = svd(Xt);
    sigma1(k) = sigma(1);
    sigma2(k) = sigma(2);
    cond(k) = sigma1(k)/min(sigma2(1:k));
end
figure(4), semilogy(sol.x,cond,'k')
title('Condition number')
end
```

The results of this numerical *a posteriori* analysis of the numerical solution of the initial-value value problem are presented in figure 2.8. As we observe in figure 2.8(a), 2.8(b) and 2.8(c), the absolute and relative residuals of all the components on this interval of integration are smaller than 10^{-6}. So, we have

computed the exact solution of

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) + 10^{-6}\mathbf{v}(t), \qquad \|\mathbf{v}\|_\infty \leq 1.$$

Moreover, as we observe in figure 2.8(d), the condition number increases significantly as we approach the singularity $-1/e$ (reaching $\approx 10^5$), which is as expected from what a condition number does. Consequently, as we approach the singularity, we can expect that the exact solution and the computed solution differ as

$$\|\mathbf{z} - \mathbf{x}\| \leq 10^5 \int_{t_0}^{t} 10^{-6}\mathbf{v}(\tau)d\tau = 10^{-1} \int_{t_0}^{t} \mathbf{v}(\tau)d\tau.$$

Naturally, we would obtain better than 10^{-1} for tighter tolerance.

2.3.3 Condition analysis based on the Gröbner-Alexeev approach

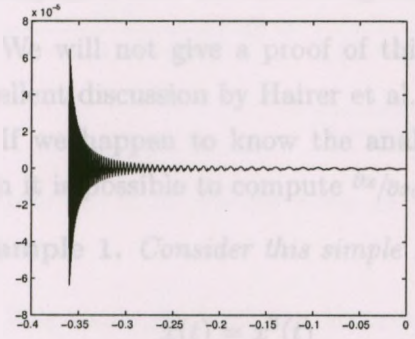
In the previous section, we have assumed that $\mathbf{v}$ depends only on t. Now, we allow it to be a nonlinear function of $\mathbf{x}$ and t. This is important for cases when the residual is correlated with the solution of the initial-value problem, as indeed it is usually for numerical methods.

Theorem 2 (Gröbner-Alexeev nonlinear variation-of-constants formula)**.** *Let* $\mathbf{x}(t)$ *and* $\mathbf{z}(t)$ *be the solutions of*

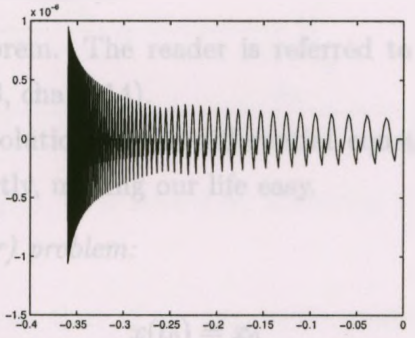
$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t)), \qquad \mathbf{x}(t_0) = \mathbf{x}_0$$
$$\dot{\mathbf{z}}(t) = \mathbf{f}(t, \mathbf{z}(t)) + \varepsilon\mathbf{v}(t, \mathbf{z}) \qquad \mathbf{z}(t_0) = \mathbf{x}_0,$$

respectively. If $\mathbf{J_f}$ *exists and is continuous, then*

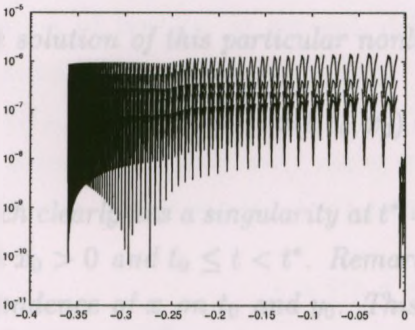
$$\mathbf{z}(t) - \mathbf{x}(t) = \varepsilon \int_{t_0}^{t} \mathbf{G}(t, \tau, \mathbf{z}(\tau))\mathbf{v}(\tau, \mathbf{z}(\tau))d\tau \tag{2.19}$$



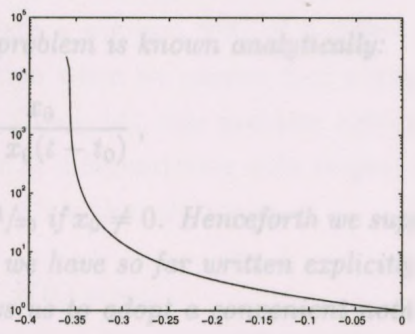
(a) Absolute resididual in x_1.



(b) Absolute residual in x_2.



(c) Relative residual in the six components of $\mathbf{y}$.



(d) Condition number of the problem.

Figure 2.8: Numerical, *a posteriori* analysis of the numerical solution of the initial-value value problem (2.18) with `ode45`.

where the matrix $\mathbf{G}$ *is given by*

$$G_{ij}(t,\tau,\mathbf{z}(\tau)) := \frac{\partial x_i}{\partial x_{0,j}}(t,\tau,z(\tau))$$

and acts as a condition number, i.e., *as a quantity dictating how* $\varepsilon\mathbf{v}(t,\mathbf{z})$ *will be magnified over the interval of integration* $[t_0,t_f]$.

We will not give a proof of this theorem. The reader is referred to the excellent discussion by Hairer et al. (1993, chap. 14).

If we happen to know the analytic solution of the differential equation, then it is possible to compute $\partial x/\partial x_0$ directly, making our life easy.

Example 1. *Consider this simple (scalar) problem:*

$$\begin{aligned} \dot{x}(t) &= x^2(t) & x(t_0) &= x_0 \\ \dot{z}(t) &= z^2(t) - \varepsilon z(t) & z(t_0) &= x_0. \end{aligned}$$

The solution of this particular nonlinear problem is known analytically:

$$x(t,t_0,x_0) = \frac{x_0}{1 - x_0(t-t_0)},$$

which clearly has a singularity at $t^* = t_0 + 1/x_0$ *if* $x_0 \neq 0$. *Henceforth we suppose that* $x_0 > 0$ *and* $t_0 \leq t < t^*$. *Remark that we have so far written explicitly the dependence of* x *on* t_0 *and* y_0. *This allows us to adopt a convenient notation for* $\partial x/\partial x_0$, *a differentiation with respect to a parameter. In what follows, we will use instead the notation* $\partial x/\partial x_0 = D_3x(t,t_0,x_0)$. *Then, we directly find that*

$$D_3x(t,t_0,x_0) = \frac{1}{(1-x_0(t-t_0))^2} = \frac{1}{(1-z(t_0)(t-t_0))^2}.$$

By theorem (2), *the difference between* z *and* x *can be written as*

$$z(t) - x(t) = -\varepsilon \int_{t_0}^{t} \frac{z(\tau)}{(1-z(\tau)(t+\tau))^2}\, d\tau.$$

Here again, our life is easy since we can solve $\dot{z} = z^2(t) - \varepsilon z$ *analytically, and*

we find that

$$z(t) = \frac{\varepsilon x_0}{x_0 - (x_0 - \varepsilon)e^{\varepsilon(t-t_0)}}$$

We can then compute the resulting integral:

$$\begin{aligned}\int_{t_0}^{t} \frac{z(s)}{(1 - z(s)(t-s))^2}\, ds &= \left(\frac{-x_0}{(\varepsilon - x_0)\, e^{\varepsilon\,(s-t_0)} + x_0\,(1 - \varepsilon\, t + \varepsilon\, s)} \right)\Bigg|_{t_0}^{t} \\ &= \frac{-x_0}{(\varepsilon - x_0)\, e^{\varepsilon\,(t-t_0)} + x_0} + \frac{x_0}{\varepsilon - x_0 + x_0\,(1 - \varepsilon\, t + \varepsilon\, t_0)}.\end{aligned}$$

It turns out to be just the same as $z(t) - y(t)$, as it should be, according to the theorem. □

This example (and the theorem) shows that *if we can differentiate the solution of our differential equation with respect to the initial conditions*, then we can account for perturbations to the differential equation. This raises the question of how to, in general, differentiate a solution $\mathbf{x}(t)$ of a differential equation with respect to an initial condition when we cannot find a formula for $\mathbf{x}(t)$. As shown in Hairer et al. (1993, Chap. 14), this problem reduces to the seemingly conceptually easier question of differentiating with respect to a *parameter*. Suppose $\mathbf{x}(t) = \mathbf{x}(t, t_0, \mathbf{x}_0, p) \in \mathbb{C}^n$ is the solution to

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, p), \qquad \mathbf{x}(t_0) = \mathbf{x}_0,$$

where $p \in \mathbb{C}$ is a parameter, constant for the duration of the solution process. We want to find what $D_{3,j}\mathbf{x}(t, t_0, \mathbf{x}_0, p) = {}^{\partial x_j}/_{\partial p}$ is.

Suppose $\mathbf{y} = \mathbf{x}(t, t_0, \mathbf{x}_0, p)$ solves the given problem, and $\mathbf{z} = \mathbf{x}(t, t_0, \mathbf{x}_0, p + \Delta p)$ solves the same problem with a slightly different parameter. Then, by taking the Taylor expansion about $\mathbf{y}$ and $p + \Delta p$, we find that the difference $\dot{\mathbf{z}} - \dot{\mathbf{y}}$ satisfies

$$\begin{aligned}\dot{\mathbf{z}} - \dot{\mathbf{y}} &= \mathbf{f}(t, \mathbf{z}, p + \Delta p) - \mathbf{f}(t, \mathbf{y}, p) \\ &= \mathbf{J}_{\mathbf{f}}(t, \mathbf{y}, p)(\mathbf{z} - \mathbf{y}) + D_3(\mathbf{f})(t, \mathbf{y}, p)\Delta p + O(\Delta p)^2. \qquad (2.20)\end{aligned}$$

As a result, if

$$\phi_i = \frac{\partial x_i(t, t_0, \mathbf{x}_0, p)}{\partial p} = \lim_{\Delta p \to 0} \frac{x_i(t, t_0, \mathbf{x}_0, p + \Delta p) - x_i(t, t_0, \mathbf{x}_0, p)}{\Delta p},$$

then taking the limit of equation (2.20) divided by Δp, as $\Delta p \to 0$, gives

$$\dot{\phi}(t) = \mathbf{J}_\mathbf{f}(t, \mathbf{x}, p)\phi(t) + \mathbf{f}_p(t, \mathbf{x}, p)\,. \tag{2.21}$$

Examining the initial solutions of $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{x}, p)$, for example by Euler's method or a higher-order method, shows that $\phi(0) = 0$. That is, in order to take the derivative of $\mathbf{x}(t, t_0, \mathbf{x}_0, p)$ with respect to p, we *differentiate the differential equation* and then solve that resulting equation. Of course, we may solve this differential equation numerically, along with the solution of the original equation. This is sometimes called the variational method of computing sensitivity.

Now let us adapt this idea to the differentiation of $\mathbf{x}(t)$ with respect to the initial condition, which does not appear in the differential equation (usually). A little thought shows that, instead of equation (2.21), we can formulate an equation for $\Phi(t)$, which is a matrix, each entry of which is $\partial x_i(t, t_0, x_0)/\partial x_{0,j}$, the derivative with respect to *one component* of the initial condition. It is convenient to package them all at once:

$$\dot{\Phi}(t) = \mathbf{J}_\mathbf{f}(t, \mathbf{x})\Phi(t) \tag{2.22}$$

where now the initial condition is $\Phi(t_0) = \mathbf{I}$, the identity matrix. This is, in fact, exactly the first associated matrix variational equation.

In more complicated examples, of course one must solve for $\mathbf{x}(t)$ and $\Phi(t)$ simultaneously, by some numerical scheme. It has been our experience that MATLAB is perfectly satisfactory for the process, once a Jacobian $\mathbf{J}_\mathbf{f}(t, \mathbf{x})$ has been coded; of course, Computer Algebra systems help with that, as we show in the next section's extended example.

2.4 An extended example: the Restricted Three-Body Problem

What follows is a discussion of a much-used example of numerical solution of initial-value problems for ordinary differential equations. The model in question dates back to Arenstorf (1963), where it was derived as a *perturbation* model intended to analyze part of the three-body problem in the limit of when one of the masses goes to zero. Further discussions of this model can be found in Hairer et al. (1993) and in Shampine and Gordon (1975), and in MATLAB it is taken up in the `orbitode` demo.

In spite of its simplistic nature (or perhaps because of it), the example is a good traditional one, in that it shows the value of adaptive step-size strategies for efficiency. Since the example has been well-studied, there are a great many solutions existing in the literature that the reader's computation can be compared to. Besides that, it is interesting, and produces surprising pictures. However, it is (nowadays) a curious problem, in that there seems little point in the idealizations that give rise to the model: one might as well integrate Newton's equations of motion directly, and indeed direct simulations of the solar system are of great interest, and are highly advanced.[7] Nonetheless, we use the model equations and extend the discussion a bit to show that the example is also excellent for showing how residual (defect) computations can be interpreted in terms of the physical modelling assumptions: in particular, we will see that if we have the solution of a model with a small residual, then we have just as good a solution as an exact solution would be. We also take the discussion a bit further and exhibit the conditioning of the initial-value problem.

In the Arenstorf model, there are three bodies, moving under their mutual gravitational influence[8]. Two of the bodies have nontrivial mass, and move

[7] See for example the JPL simulator, at `http://space.jpl.nasa.gov/`.

[8] We do not claim to be computational astronomers. This discussion is by no means complete and should not be considered authoritative. The discussion is intended only to motivate the model, and to remind the reader of some of the idealizations made, for comparison with the numerical effects of simulation.

about their mutual center of gravity in *circular* orbits. If we are thinking of these bodies as the Earth and the Moon, then already this is an idealization: the Earth-Moon system more nearly has the Moon moving elliptically, with eccentricity about 0.05, about the center of gravity of the Earth-Moon system. If we are thinking about the Sun-Jupiter pair, then again Jupiter's orbit is not circular but rather eccentric. So, again, a circular orbit is an idealization. The third body in the model is taken to be so small that its influence on the two larger bodies can be supposed negligible. One thinks of an artificial satellite, with mass less than a thousand tonnes (10^6 kg). The mass of the Moon is $M_M = 7.3477 \times 10^{22}$ kg, so the satellite's mass is less than 10^{-16} of that (coincidentally, not too different from the machine epsilon of roundoff error in MATLAB). Therefore, supposing that this body does not affect the other two bodies is a reasonable assumption. By making this assumption, the actual mass of the satellite drops out of the computation.

Another assumption, common in gravitational models since Newton, is that the bodies act as *point masses*. Neither the Earth, the Moon, the Sun, nor Jupiter, nor even the satellite, are point masses. In fact, the radius of the Earth is about $1/60$ the Earth-Moon distance, and while the gravitational effects of a uniform spherical body are indeed, when outside the body, identical in theory to those of a point mass, the Earth is neither uniform nor exactly spherical (it's pretty close, though, with a radius of 6,378.1km at the equator and 6,356.8km at the poles). Similarly for Jupiter, which departs from sphericity by more than the Earth does (71,492km radius at the equator and 66,854km radius at the poles). This departure from ideal point-mass gravity has a potentially significant effect on the satellite's orbit.

Finally, we neglect the influence of the other bodies in the Solar System. For the Earth-Moon system, neglecting the influence of the Sun, which differs at different points of the satellite's orbit around the Earth-Moon pair, means that we are neglecting forces about 10^{-11} of the base field; a smaller influence than the eccentricity of the orbit, and smaller than the effects of departure from the point-mass idealization, but larger than the effects of the trivial mass of the satellite. For the Sun-Jupiter pair, this is not a bad assumption—the

most significant other body is Saturn, and Saturn is far enough away that its effects are detectable only cumulatively.

Once these assumptions are made, then we put M equal to the mass of the largest body, and m equal to the mass of the smaller nontrivial body; the total mass of the system is then $M + m$, and we place the origin of our coordinate system at the center of gravity. The larger mass is treated as a point at distance

$$-\mu = -\frac{m}{M+m} \tag{2.23}$$

from the origin, in units of the diameter of the orbit, and the smaller mass is then at $\mu^* = 1 - \mu$.

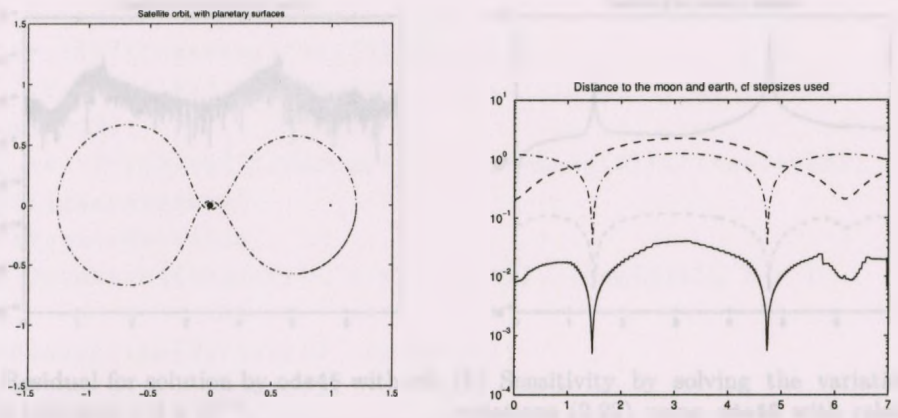
In the *rotating coordinate system* that fixes the large body at $-\mu$ and the small body at $1 - \mu$, the equations of motion of the tiny satellite are given in Arenstorf (1963) as follows:

$$\begin{aligned} \ddot{x} &= 2\dot{y} + x - \mu^* \frac{x+\mu}{R_{13}^3} - \mu \frac{x-\mu^*}{R_{23}^3} \\ \ddot{y} &= -2\dot{x} + y - \mu^* \frac{y}{R_{13}^3} - \mu \frac{y}{R_{23}^3} \end{aligned} \tag{2.24}$$

where the distances $R_{13} = \sqrt{(x+\mu)^2 + y^2}$ and $R_{23} = \sqrt{(x-\mu^*)^2 + y^2}$ between the tiny satellite and the massive body and the minor body, respectively, are consequences of Newton's law (and an assumption of smallness of μ). The problem parameters used by the MATLAB demo `orbitode` are:

```
% Problem parameters
mu = 1 / 82.45;
mustar = 1 - mu;
y0 = [1.2; 0; 0; -1.04935750983031990726];
tspan = [0 7];
```

Note that $1/82.45 \doteq 0.01213$. The value of μ used by Hairer et al. (1993) is, however, $\mu = 0.012277471$, and they use 30 decimals in their initial conditions for their periodic orbits. On the other hand, if we take the values of the Earth's mass and the Moon's mass given in Wikipedia, we get $\mu = 0.0121508\ldots$, different in the 4th significant figure from either of these two. While these

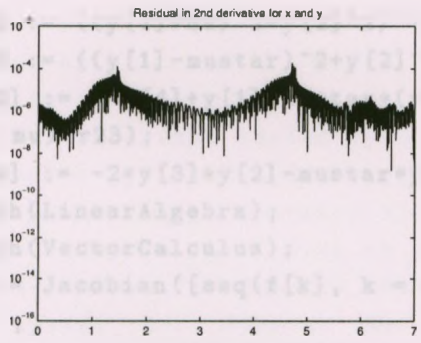


(a) Restricted three-body problem orbit (b) Distances and stepsizes

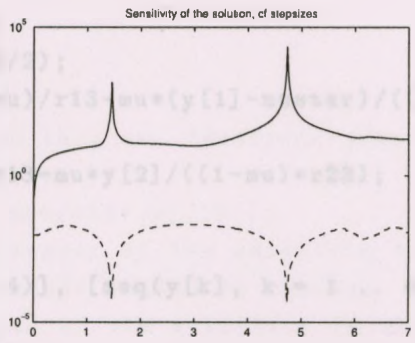
Figure 2.9: Solution of the Arenstorf model restricted three-body problem for the same parameters and initial conditions as are in `orbitodedemo`, intended to be similar to a small satellite orbiting the Earth-Moon system in a coordinate system rotating with the Earth and Moon. When distances to either massive body are small, stepsizes (computed as speed times delta t) also get small.

differences in parameters are alarming, if we take the model equations at all seriously, they are not terribly significant given that the model's derivation has neglected things like the eccentricity of size about 0.05. Finally, Arenstorf's derivation has at one point replaced $\mu/(1-\mu)$ with just μ, a simplification from the point of view of perturbation theory but no simplification at all for numerical solution; this makes a further difference of about 1.2% in the terms of the equation.

Computing the solution to this problem, with the parameters given above and analyzing the results, we find figures 2.9, 2.9(b) and 2.10(b). We see that the step sizes are not uniform—indeed, they change over the orbit by a factor of about 30. As seems physically reasonable, the steps are small when the satellite is near one of the massive bodies (and thus experiencing relatively large forces and accelerations). We also see, from figure 2.10(b), that the residual is small, never larger than 10^{-4}, approximately. Since we have neglected terms in the equations whose magnitude is 10^{-2} or so, we see that



(a) Residual for solution by ode45 with relative tolerance 1.0×10^{-6}.



(b) Sensitivity by solving the variational equations (2.22) using ode45 with relative tolerance 1.0×10^{-6}.

Figure 2.10: Measured residual and sensitivity in the Arenstorf restricted three-body problem.

we have found a perfectly adequate model solution: this plot tells us precisely as much as an exact solution to the Arenstorf restricted three-body problem would. Finally, we need to know how sensitive these orbits are to changes in μ or to the initial conditions are. Using the first variational equation and plotting the norm of the fundamental solution matrix in figure 2.9(b) we see that the orbit gains sensitivity as the satellite plunges towards the Earth, but shows a rapid decrease in sensitivity as the satellite moves away; both of these observations agree with intuition.

We have pointed out the different values of μ used in the various books. How much difference does this make to the orbits? We can use our variational equation

$$\dot{\Psi} = \mathbf{J_f}(\mathbf{x})\Psi + \mathbf{f}_\mu$$

where $\Psi(0) = 0$ and the Jacobian matrix $\mathbf{J_f}(\mathbf{x})$ and the partial derivative $\mathbf{f}_\mu$ are easily computed by a computer algebra system. Here are the MAPLE commands used to do so for this example.

```
1 f[1] := y[3];
2 f[2] := y[4];
3 mustar := 1-mu;
```

```
r13 := ((y[1]+mu)^2+y[2]^2)^(3/2);
r23 := ((y[1]-mustar)^2+y[2]^2)^(3/2);
f[3] := 2*y[4]+y[1]-mustar*(y[1]+mu)/r13-mu*(y[1]-mustar)/((1-
    mu)*r23);
f[4] := -2*y[3]+y[2]-mustar*y[2]/r13-mu*y[2]/((1-mu)*r23);
with(LinearAlgebra);
with(VectorCalculus);
J := Jacobian([seq(f[k], k = 1 .. 4)], [seq(y[k], k = 1 .. 4)])
    ;
CodeGeneration[Matlab](J, optimize);
fp := map(diff, [seq(f[k], k = 1 .. 4)], mu);
CodeGeneration[Matlab](Vector(fp), optimize);
```

The output of these commands are then bundled up into MATLAB m-files and used in the call to ode45. We explain below how we created a modified version of MATLAB's orbitode. We first define the parameters of the problem:

```
function t=orbitodejac

% Problem parameters
mu = 1 / 82.45; % Shampine & Gordon
mustar = 1 - mu;
y0 = [1.2; 0; 0; -1.04935750983031990726; 0; 0; 0; 0]; %
    Shampine & Gordon
tspan = [0 7]; % Shampine & Gordon
```

As before, we solve the problem with the command

```
[t,y,te,ye,ie] = ode45(@f,tspan,y0,options);
```

The command is slightly different than in the previous examples, since we use the "event location" feature. We then plot the results using the following list of commands:

```
% Plotting circles representing earth and moon
rade = 6371/384403;
radm = 1737/384403;
th=linspace(-pi,pi,101);
esx = rade*cos(th)-mu;
esy = rade*sin(th);
msx = radm*cos(th)+1-mu;
```

```
msy = radm*sin(th);
close all
figure
% Plot timesteps scaled by speed so they are "distance-steps"
avevel = (y(1:end-1,3:4)+ y(2:end,3:4) )/2;
avespeed = sqrt( avevel(:,1).^2 + avevel(:,2).^2 );
%Plots Below: 'k--' line is the distance of the satellite to
    the moon; 'k' line is the time-steps scaled by average
    speed; 'k-.' line is the distance of the satellite to the
    earth
semilogy( t, sqrt( (y(:,1)-mustar).^2 + y(:,2).^2 ), 'k--',t(2:
    end), diff(t).*avespeed, 'k', t, sqrt( (y(:,1)+mu).^2 + y
    (:,2).^2 ), 'k-.' )
title('Distance to the moon and earth, cf stepsizes used')
figure
plot(y(:,1),y(:,2),'k-.',esx,esy,'k',msx,msy,'k');
hold on
% Estimate of the size of filled circle was obtained by trial
    and error on actual radius of earth and moon figures, later
     not needed.
scatter( -mu, 0, 50, 'k', 'filled');
scatter( 1-mu, 0, 50*(radm/rade)^2, 'k', 'filled' );
axis([-1.5,1.5,-1.5,1.5])
axis('square')
title('Satellite orbit, with planetary surfaces')
hold off
figure
semilogy(t(2:end),diff(t), 'k--', t,sqrt(y(:,5).^2+y(:,6).^2+y
    (:,7).^2+y(:,8).^2),'k')
title('Sensitivity of the solution, cf stepsizes');
figure

% Inefficiently, solve it again, so we may compute the residual
     (should have done this the first time)
sol = ode45(@f,tspan,y0,options);
np = 4003;
tt = linspace( tspan(1), tspan(end), np);
[yb,ypb] = deval( sol, tt );
res = zeros(1,np);
```

```
for i=1:np,
    rr = ypb(:,i) - f(tt(i), yb(:,i) );
    res(i) = sqrt(rr(3)^2+rr(4)^2);
end;
semilogy( tt, res, 'k' )
title( 'Residual in 2nd derivative for x and y' )
```

It remains to describe the code for the function handle `@f` passed to the command `ode45`. The code begin as

```
function dydt = f(t,y)
% Derivative function -- mu and mustar shared with the outer
    function.
    r13 = ((y(1) + mu)^2 + y(2)^2) ^ 1.5;
    r23 = ((y(1) - mustar)^2 + y(2)^2) ^ 1.5;

    % Jacobian computed by Maple
    cg0 = zeros(4,4);
    cg2 = zeros(4,1);
    t1 = 1 - mu;
t2 = y(1) + mu;
```

and then it is followed by the MAPLE-generated MATLAB code. Using these, we finally define our function `dydt=f(t,y)`:

```
  dydt = [ y(3)
           y(4)
           2*y(4) + y(1) - mustar*((y(1)+mu)/r13) - mu/mustar
              *((y(1)-mustar)/r23)
          -2*y(3) + y(2) - mustar*(y(2)/r13) - mu/mustar*(y
             (2)/r23)
           cg2(1)
           cg2(2)
           cg2(3)
           cg2(4)];
```

Thus, the use numerical solutions in combination with computer algebra systems is a very effective way to estimate the condition of an initial value problem (or, in other words, the "sensitivity" of the orbits).

2.5 What good are numerical solutions of chaotic problems?

Chaotic systems can be studied from many perspectives and, accordingly, chaotic motion can be defined in many ways.[9] In all cases, some intuitions drawn from physics motivate the various approaches:

> The concepts of 'order' and 'determinism' in the natural sciences recall the predictability of the motion of simple physical systems obeying Newton's laws: the rigid plane pendulum, a block sliding down an inclined plane, or motion in the field of a central force are all examples familiar from elementary physics. In contrast, the concept of 'chaos' recalls the erratic, unpredictable behavior of elements of a turbulent fluid or the 'randomness' of Brownian motion as observed through a microscope. For such chaotic motions, knowing the state of the system at a given time does not permit one to predict it for all later times. (Campbell and Rose, 1983, vii)

The idea is that a chaotic motion $\mathbf{x}(t)$ satisfying a deterministic nonlinear differential equation $\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t))$ is *bounded* (*i.e.*, $\mathbf{x}(t)$ does not go to ∞ as $t \to \infty$), *aperiodic* (*i.e.*, for no T does $\mathbf{x}(t) = \mathbf{x}(t+T)$) and extremely *sensitive to initial conditions*. Now, if two trajectories $\mathbf{x}(t)$ and $\mathbf{z}(t)$ were uniformly diverging (*i.e.*, if the distance between the two trajectories were continuously increasing with t), at least one of them would be unbounded. But because of the non-linearity of the equation, the distance between the two curves varies in very erratic ways. It is thus practically impossible to track how close our two trajectories are from one another in the long run (and often even in the short run!). To establish sensitivity to initial condition, the important thing is that, *on average*, for finite time, the trajectories diverge from each other. This is exactly what positive Lyapunov exponents (as defined in equation 2.17, and

[9] Martelli et al. (1998, p. 112) claim amusingly, "with a bit of exaggeration, that there are as many definitions of chaos as experts in this new area of knowledge." Concerning some of the conceptual issues involved with different definitional attempts, see Batterman (1993).

interpreted in terms of the SVD factoring) show. But then, it also follows that solutions of chaotic initial-value problems are exponentially ill-conditioned.

This situation raises two related questions regarding the reliability of numerical methods:

1. are the computed trajectories satisfactory, and in what sense are they to be regarded satisfactory?

2. are the numerical methods introducing spurious chaos or suppressing actual chaos?

These questions turn out to be quite tricky. We only mention the most important aspects here and refer the reader to Corless (1992, 1994a,b) for a more complete answer. The first and foremost observation to make is that, because chaotic problem are exponentially ill-conditioned (they have positive Lyapunov exponents), one cannot hope to obtain numerical solutions with small forward error. Accordingly, errors in the initial conditions, discretization error, and even rounding error will be exponentially magnified as time increases. As a result, one can obtain huge forward error, *i.e.*, important lack of correlation between the projected motion and the actual motion. But is this a reason to claim that our algorithm is a bad one, *i.e.*, that it is unstable?

No. The solutions can be satisfactory in the backward sense, since it is unreasonable to ask of a numerical method more than the problem allows for. For chaotic problem, good methods such as the one implemented in `ode45` *do solve a problem near the reference problem.* As we explained in chapter 1, for genuine (as opposed to artificial) problems, we will have to consider physical perturbations anyway. For the Lorenz system, which is already a truncation of a much more complex fluid model, physical perturbations are important, and the detailed trajectory is very sensitive to perturbations. This explains why the results of our simulations of chaotic systems agree with experiments.

To sum up, if we concern ourselves primarily with position in the phase space, the computed trajectories of chaotic systems can be satisfactory in

the backward sense (if computed with a backward stable method). However, if one is asking whether they are satisfactory in the forward sense, the answer is that they are not, because of the ill-conditioning of the problems. Consequently, in answer to such concerns, the analysis we provide is literally the same as the one presented in section 2.3.

However, there are other aspects of chaotic systems that are not so easily explained by the backward error perspective. Even if $\mathbf{x}(t)$ is sentive to initial conditions, some features of chaotic systems may not be, such as the dimension of the attractor, or the measure on the attractor, and numerical solution can be expected to give us good information about those things. But in this case, the situation is tricky. Consider for example the Gauss map

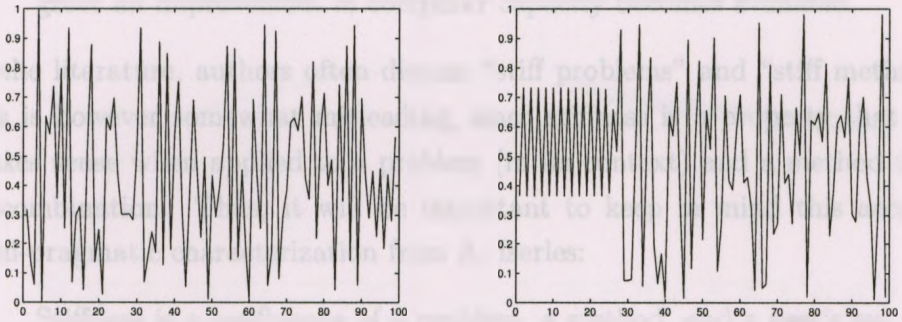
$$x_{k+1} = G(x_k),$$

where

$$G(x) = \begin{cases} 0 & \text{if } x = 0 \\ x^{-1} \mod 1 & \text{otherwise} \end{cases}$$

It can be shown that this map is ultimately periodic for rational numbers and solutions of quadratics, but that it is chaotic for other real numbers. Now, would this be revealed by a simulation using floating-point operations? To begin with, observe that since the set of floating-point numbers $\mathbb{F}$ is a *finite* subset of $\mathbb{Q}$, any floating-point implementation iterated n times as $n \to \infty$ *will necessarily be periodic.* As a result, we see that the method appears to suppress chaos. Nonetheless, as we see from picture 2.11(a), if we use $x_0 = \hat{\pi}$ as initial condition, the trajectory seem to be completely erratic. This would suggest chaos. At the same time, if we take we take $x_0 = -1/2 + \sqrt{3}$, as illustrated in figure 2.11(b), we are supposed to have a periodic orbit (in exact arithmetic) but, as we see, the floating-point operations introduce spurious chaos.

What are we to do with such results? To begin with, such examples show that it is healthy to entertain an initial skepticism of chaos known only through



(a) Gauss map with $x_0 = \pi$. Aperiodic behavior.

(b) Gauss map with $x_0 = -1/2 + \sqrt{3}$ being the solution of a quadratic. Initially periodic behavior becoming aperiodic.

Figure 2.11: Time histories for the floating-point Gauss map, linearly interpolated.

numerical solutions. As we mentioned in chapter 1, "a useful backward error-analysis is an explanation, not an excuse, for what may turn out to be an extremely incorrect result" (Kahan, 2009).

2.6 Solution of stiff problems

Stiffness is an important concept in scientific computation. So far, we have seen that standard methods such as the one implemented in MATLAB's `ode45` can be relied upon to accurately and efficiently solve many problems. Even for chaotic problems, we have seen that such methods can be relied upon to give stable results, in the backward sense. With stiff problems, however, we are facing a different computational phenomenon that does not fall under the themes treated so far. As Shampine and Gear (1979, p. 1) put it,

> [t]he problems called 'stiff' are too important to ignore, and they are too expensive to overpower. They are too important to ignore because they occur in many physically important situations. They are too expensive to overpower because of their size and the inherent difficulty they present to classical methods, no matter how

great an improvement in computer capacity becomes available.

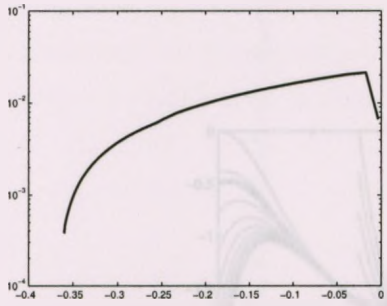
In the literature, authors often discuss "stiff problems" and "stiff methods;" this is however somewhat misleading, since stiffness is a property that only makes sense when applied to a problem (in its context) and a method taken in combination. Thus, it will be important to keep in mind this accurate semi-pragmatic characterization from A. Iserles:

> Stiffness is a confluence of a problem, a method, and a user's expectations.

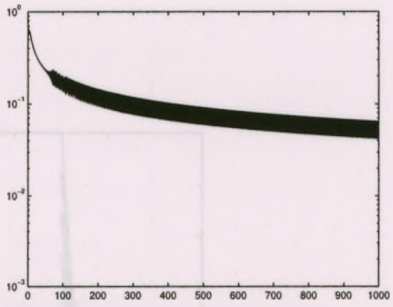
Even if there is no generally accepted rigorous definition of stiffness, there is a widely shared consensus as to what stiffness involves in practice. In this section, we explain the problem-specific aspect of stiffness, and delay the method-specific aspects to chapter 3, insofar as this is possible.

When a problem is moderately conditioned and we try to solve it with a reasonably tight tolerance, we will usually observe that the stepsize automatically selected by the program decreases. This is because state-of-the-art codes ultimately control the residual of the computed solution by reducing the stepsize when necessary. Let us reconsider the example from equation (2.18), but let us now turn our attention to the stepsize as we approach the singular point $-1/e$. As we can see in figure 2.12(a), `ode45` takes 71 steps to solve the problem. The largest step size, near the beginning, is 2×10^{-2}. The smallest step size, occurring at the very end when the problem is the most ill-conditioned, is 3×10^{-4}. The difference would be even larger if we required a tighter tolerance. As we see, the amount of work required increases with the condition number.

Now, what would happen if we had a very well-conditioned problem instead? Our *user expectation* would be that the stepsize would remain large and that we would obtain a cheap solution to the problem. Even if this will often be the case, stiff problems are such that this is precisely what fails. A stiff problem is extremely well-conditioned; loosely speaking, its Jacobian matrix has *no* eigenvalue with large positive real part, but it has *at least one* eigenvalue with large negative real part. Accordingly, it is not an accident that we examine stiff problems immediately after chaotic problems since, in a sense,



(a) Reduction of the stepsize in the solution of (2.18) as the problem becomes increasingly ill-conditioned.



(b) Reduction of the stepsize in the solution of (2.25) when the problem is very well-conditioned.

Figure 2.12: Stepsize adaptation of ill-conditioned and well-conditioned problems with respect to step number.

they are opposite on the same spectrum. Chaotic problems are badly conditioned: since they have large, positive real Lyapunov exponents, the nearby trajectories diverge very quickly from the trajectory exactly solving the initial value problem. The difficulty with stiff problems comes from the fact that they are *too well-conditioned.*

Consider an example:

$$\dot{x} = x^2 - t, \qquad x(0) = -\frac{1}{2}. \tag{2.25}$$

We will look at two intervals: $I_1 = 0 \leq t \leq 1$ and $I_2 = 0 \leq t \leq 10^3$. The solution of this problem is displayed in figure 2.13, together with many other solutions using different initial values. We see that the trajectories all converge to the same one extremely fast. Our expectation would thus be that our program would find the numerical solution without being too strict about stepsize, since even relatively large errors on each step would quickly be damped. In fact, for the first time span I_1, `ode45` takes 11 steps only and has residual $\leq 8 \cdot 10^{-7}$ (see figure 2.14(a)). This agrees with our expectations: `ode45` does just fine. However, as we see in figure 2.12(b), the stepsizes required to go over I_2 remain very small, the minimum one being 3×10^{-4}. For this

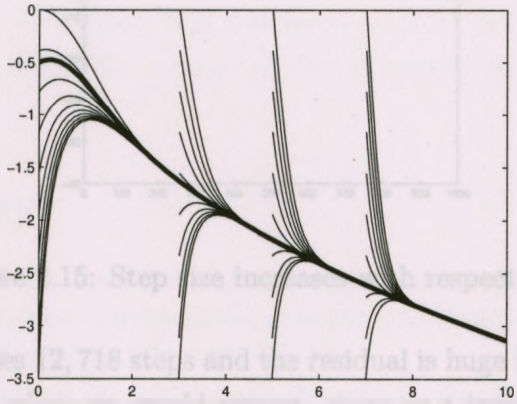
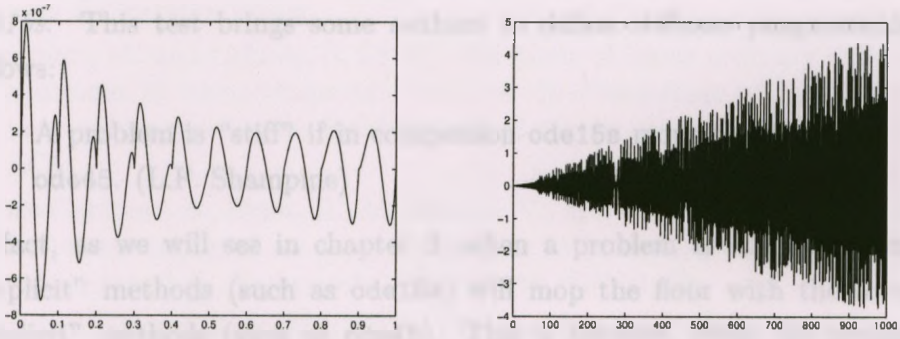


Figure 2.13: Extreme well-conditioning of a stiff problem.



(a) Absolute residual on I_1, 11 steps (b) Absolute residual on I_2, 12,718 steps

Figure 2.14: Residual for the solution of (2.25) on two intervals.

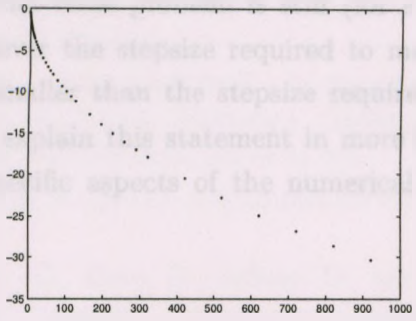


Figure 2.15: Step size increases with respect to t.

interval, `ode45` takes 12,718 steps and the residual is huge (see figure 2.14(b)). This goes against what we would expect, since as t increases, the problem becomes increasingly better conditioned. On the other hand, on the same problem for the interval I_2, `ode15s` takes only 57 steps! If we examine the step sizes (see figure 2.15), we see that as t increases and the problem becomes better conditioned, `ode15s` takes bigger and bigger steps.

This situation is not unique to this problem. In fact, in practice, a common diagnosis tool to identify stiff problems consists in trying to solve the problem with a standard ODE solver, *e.g.*, `ode45`. If it has a hard time, try it with *ode15s*. This test brings some authors to define stiffness pragmatically as follows:

> A problem is "stiff" if in comparison `ode15s` mops the floor with `ode45`. (L.F. Shampine)

In fact, as we will see in chapter 3, when a problem is stiff, the so-called "implicit" methods (such as `ode15s`) will mop the floor with the so-called "explicit" methods (such as `ode45`). This is because, when the problem is very well-conditioned, explicit methods become unstable for large step sizes. In this spirit, Higham and Trefethen (1993) claim that

> It is generally agreed that the essence of stiffness is a simple idea,
>
> *Stability is more of a constraint than accuracy.*

More precisely, an initial-value problem is stiff (for a given interval of very good condition) whenever the stepsize required to maintain the stability of the method is much smaller than the stepsize required to maintain a small residual. However, to explain this statement in more detail, we will need to look at the method-specific aspects of the numerical solutions of IVPs for ODEs.

Bibliography

Arenstorf, R. F. (1963). Periodic solutions of the restricted three body problem representing analytic continuations of keplerian elliptic motions. *American Journal of Mathematics*, 85(1):pp. 27–35.

Ascher, U. M., Mattheij, R. M. M., and Russell, R. D. (1988). *Numerical Solution of Boundary Problems for Ordinary Differential Equations.* Prentice Hall.

Batterman, R. (1993). Defining chaos. *Philosophy of Science*, 60(1):43–66.

Bender, C. M. and Orszag, S. A. (1978). *Advanced mathematical methods for scientists and engineers.* McGraw-Hill, New York.

Birkhoff, G. and Rota, G.-C. (1989). *Ordinary Differential Equations.* Wiley, 4th edition.

Bronstein, M. (2005). *Symbolic integration I: transcendental functions*, volume 1. Springer Verlag.

Bronstein, M. and Lafaille, S. (2002). Solutions of linear ordinary differential equations in terms of special functions. In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 23–28. ACM.

Bunse-Gerstner, A., Byers, R., Mehrmann, V., and Nichols, N. (1991). Numerical computation of an analytic singular value decomposition of a matrix valued function. *Numerische Mathematik*, 60(1):1–39.

Campbell, D. and Rose, H. (1983). Preface. In Campbell, D. and Rose, H., editors, *Order in Chaos: Proceedings of the International Conference on Order in Chaos held at the Center for Nonlinear Studies*, pages vii–viii, Los Alamos.

Corless, R. (1994a). What good are numerical simulations of chaotic dynamical systems? *Computers & Mathematics with Applications*, 28(10-12):107–121.

Corless, R. M. (1992). Continued fractions and chaos. *The American Mathematical Monthly*, 99(3):203–215.

Corless, R. M. (1994b). Error backward. In Kloeden, P. and Palmer, K., editors, *Proceedings of Chaotic Numerics, Geelong, 1993*, volume 172 of *AMS Contemporary Mathematics*, pages 31–62.

Corless, R. M. and Fillion, N. (201x). A graduate survey of numerical methods. Forthcoming.

Corless, R. M., Gonnet, G., Hare, D., Jeffrey, D., and Knuth, D. E. (1996). On the Lambert W function. *Advances in Computational Mathematics*, 5(1):329–359.

Curry, H. and Feys, R. (1958). Combinatory logic.

Geddes, K. O., Czapor, S. R., and Labahn, G. (1992). *Algorithms for computer algebra*. Kluwer Academic, Boston.

Geist, K., Parlitz, U., and Lauterborn, W. (1990). Comparison of different methods for computing lyapunov exponents. *Prog. Theor. Phys*, 83(5):875–893.

Hairer, E., Nørsett, S. P., and Wanner, G. (1993). *Solving ordinary differential equations: Nonstiff problems*. Springer.

Higham, D. and Trefethen, L. (1993). Stiffness of ODEs. *BIT Numerical Mathematics*, 33(2):285–303.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 2nd edition.

Kahan, W. (2009). Needed remedies for the undebuggability of large-scale floating-point computations in science and engineering. Presented at the Computer Science Dept. Colloquia at the University of California, Berkely.

Kaplan, W. (2002). *Advanced Calculus*. Addison Wesley, Boston, 5 edition.

Lakshmanan, M. and Rajasekar, S. (2003). *Nonlinear dynamics: integrability, chaos, and patterns*. Springer Verlag.

Martelli, M., Dang, M., and Seph, T. (1998). Defining chaos. *Mathematics magazine*, 71(2):112–122.

Nagle, R., Saff, E., and Snider, A. (2000). *Fundamentals of differential equations and boundary value problems*. Addison-Wesley, 3rd edition.

Schönfinkel, M. (1924). On the building blocks of mathematical logic. In Van Heijenoort, J., editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, page 355. Harvard University Press.

Shampine, L. and Gear, C. (1979). A user's view of solving stiff ordinary differential equations. *SIAM review*, pages 1–17.

Shampine, L. F. and Gordon, M. (1975). *Computer solution of ordinary differential equations : the initial value problem.* Freeman.

Söderlind, G. (1984). On nonlinear difference and differential equations. *BIT Numerical Mathematics*, 24(4):667–680.

von zur Gathen, J. and Gerhard, J. (2003). *Modern computer algebra.* Cambridge University Press.

Chapter 3

Numerical Methods for ODEs

In the previous chapter, we have investigated what a numerical solution to an initial-value problem is as well as how to use state-of-the-art codes to obtain one. We have re-introduced the concept of condition number in the context of IVPs; here, as for all numerical methods, we have seen that conditioning is an intrinsic property of a problem rather than a property of a particular numerical method to solve it. We have also explained how to make a residual-based *a posteriori* error analysis resulting from the use of a given code to solve a given problem. Together with condition analysis, this gives us a general method to determine if our numerical method has exactly (or nearly exactly) solved a nearby problem, and enables us to estimate the influence changes to the problem have. Moreover, the analysis follows the outline of chapter 1 and parallels all the other applications of this methods developed in Corless and Fillion (201x).

Now, it's time to investigate what's under the hood, *i.e.*, what the codes are actually doing, and why they work or fail for certain types of problems. We will begin our investigation with a venerable method, namely *Euler's method.* Through the presentation of this method, we will introduce some key concepts such as the distinction between *implicit* and *explicit* method, *local* and *global error*, as well as their relations to the residual, *adaptive stepsize selection*, *etc.* On that that basis, we will show that Euler's method turns out to be a particular case of more general methods. Generalizing in one way, we will obtain the *Taylor series methods* whereas, if we go another way, we obtain

the *Runge-Kutta methods*. From there, Corless and Fillion (201x) turn to yet another generalization, *multistep methods*, particularly Adams', who is very important in practice. It is there shown how John Butcher introduced the notion of General Linear Method to unify these generalizations. Moreover, as we will see, these methods will come in two flavours: explicit and implicit (the latter being used for the solution of stiff problems). In this thesis chapter, however, we will only include the Taylor series method and the continuous Runge-Kutta methods. The point we wish to make is that numerical methods do in fact produce continuous solution, *contra* the standard presentation of those methods. Showing that it is so for Runge-Kutta methods should suffice, since they are the most popularly used in broadly distributed software suites. Consequently, it is hoped that the idea of using residual control as an error control strategy will appear natural.

3.1 Euler's method and basic concepts

Euler's method is in some sense the most fundamental method for numerically solving initial-value problems. In its barest form, it is of limited use because it is not very accurate.[1] Nonetheless, since it is theoretically and pedagogically important, it is worth looking at it in some details. We will use it to introduce a number of fundamental concepts that will reappear in our study of more refined methods. We will also see that such methods correspond to different ways of generalizing Euler's method.

The basic idea of Euler's method is one shared by many important methods of calculus. Consider the initial-value problem $\dot{x}(t) = f(t, x(t))$, $x(t_0) = x_0$, and let $x(t)$ be a solution. Since we are given the values t_0, x_0, and $\dot{x}(t_0)$ (because we can compute $f(t_0, x(t_0))$, we can make the construction shown in figure (3.1). Now, consider a forward time step h so that $t_1 = t_0 + h$ and $x_1 = x_0 + hf(x_0)$. In general, we won't have the exact equality $x(t_1) = x_1$. Nonetheless, if h is small, we will have the approximation $x(t_1) \approx x_1$.

[1] It should be noted, however, that there are special circumstances where, until recently, it was the most efficient method known (see Christlieb et al., 2010).

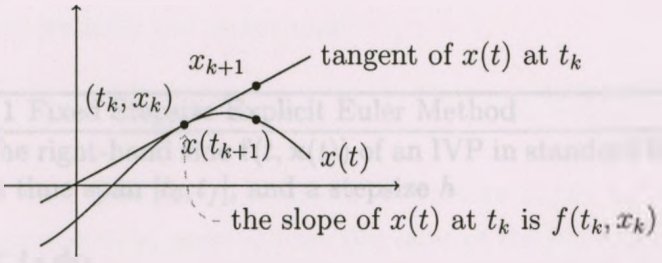


Figure 3.1: A step with Euler's method, where $x_k = x(t_k)$ is assumed to be known exactly.

Euler's method precisely consists in replacing the computation of $x(t_1)$ by the computation of x_1. Since it is enough to have the point (t_0, x_0) and the value of $f(t_0, x_0)$ to find the equation of the tangent of $x(t)$ at (t_0, x_0), we have:

$$x_1 = x_0 + hf(t_0, x_0).$$

In other words, we treat the problem based on the fact that the value of x changes linearly as $h \to 0$, with f as its slope. Then, if x_1 is close to $x(t_1)$, we can *pretend* that this point is on the curve $x(t)$, and repeat the same operation, with $f(t_1, x_1)$ as the slope. We will thus get a point (t_2, x_2), which will hopefully satisfy $x(t_2) \approx x_2$ to a sufficient degree. Using the map

$$x_{n+1} = x_n + hf(t_n, x_n),$$

we can then generate a sequence of values $x_0, x_1, x_2, \ldots, x_N$ at the mesh points $t_0, t_1, t_2, \ldots, t_N$ that approximates $x(t_0), x(t_1), x(t_2), \ldots, x(t_N)$. This iterative process gives rise to algorithm (1). Analytically, as we approach the limit $h \to 0$, it is expected that the approximation will become arbitrarily good. An example is presented in figure (3.2). Numerically, however, it is important to be careful, since for very small values of h, floating-point error may prevent us from obtaining the desired convergence.

Euler's method is not limited to scalar problems. For an IVP posed in

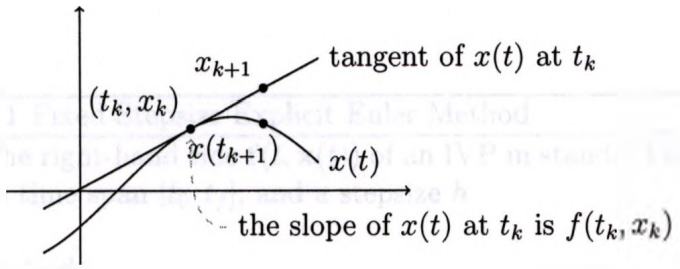


Figure 3.1: A step with Euler's method, where $x_k = x(t_k)$ is assumed to be known exactly.

Euler's method precisely consists in replacing the computation of $x(t_1)$ by the computation of x_1. Since it is enough to have the point (t_0, x_0) and the value of $f(t_0, x_0)$ to find the equation of the tangent of $x(t)$ at (t_0, x_0), we have:

$$x_1 = x_0 + hf(t_0, x_0).$$

In other words, we treat the problem based on the fact that the value of x changes linearly as $h \to 0$, with f as its slope. Then, if x_1 is close to $x(t_1)$, we can *pretend* that this point is on the curve $x(t)$, and repeat the same operation, with $f(t_1, x_1)$ as the slope. We will thus get a point (t_2, x_2), which will hopefully satisfy $x(t_2) \approx x_2$ to a sufficient degree. Using the map

$$x_{n+1} = x_n + hf(t_n, x_n),$$

we can then generate a sequence of values $x_0, x_1, x_2, \ldots, x_N$ at the mesh points $t_0, t_1, t_2, \ldots, t_N$ that approximates $x(t_0), x(t_1), x(t_2), \ldots, x(t_N)$. This iterative process gives rise to algorithm (1). Analytically, as we approach the limit $h \to 0$, it is expected that the approximation will become arbitrarily good. An example is presented in figure (3.2). Numerically, however, it is important to be careful, since for very small values of h, floating-point error may prevent us from obtaining the desired convergence.

Euler's method is not limited to scalar problems. For an IVP posed in

Algorithm 1 Fixed Stepsize Explicit Euler Method

Require: The right-hand side $\mathbf{f}(t, \mathbf{x}(t))$ of an IVP in standard form, an initial value $\mathbf{x}_0$, a time span $[t_0, t_f]$, and a stepsize h

$n = 0$

while $t_n < t_f$ **do**

$x_{n+1} = x_n + hf(t_n, x_n)$

$t_{n+1} = t_n + h$

$n := n + 1$

end while

Obtain a continuous function $\hat{x}(t)$ by somehow interpolating on $\mathbf{t}, \mathbf{x}$, typically piecewise.

return $\hat{x}(t)$

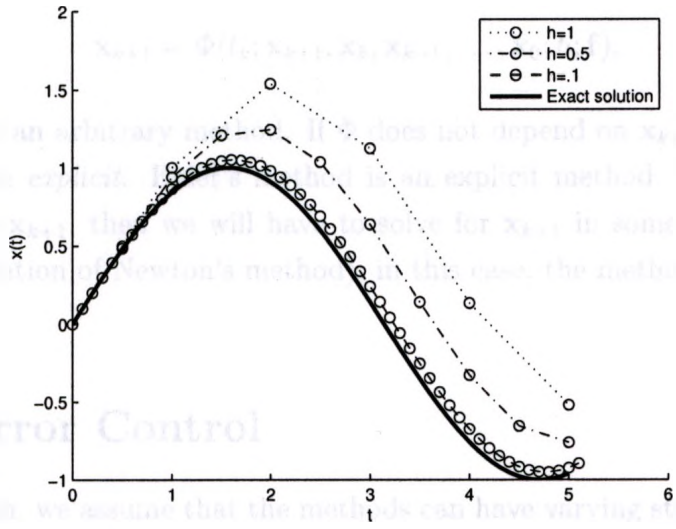


Figure 3.2: Effect of reducing the step size in Euler's method

standard form, we have the vector map

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}(t_n, \mathbf{x}_n) \qquad \mathbf{x}_0 = \mathbf{x}(t_0). \tag{3.1}$$

The method then produces a sequence of vectors $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k, \ldots, \mathbf{x}_N$ whose ith components x_k^i approximate the value of the ith equation at time k, *i.e.*, $x_k^i \approx x^i(t_k)$. Again, we can interpolate those points to obtain a continuous approximation $\hat{\mathbf{x}}(t)$ to $\mathbf{x}(t)$. Algorithm (1) can be rewritten in vector form in an obvious way.

Observe the notation x_k^i. In this chapter, we use superscript i to denote the ith component of a vector—typically $\mathbf{x}$ or $\mathbf{f}$—and subscript k to denote the time step t_k. In context, there should not be confusion with exponents.

Because $\hat{\mathbf{x}}(t)$ is iteratively generated by a discrete time variable in order to approximate $\mathbf{x}(t)$, Euler's method (and all the later explicit methods) is often called a *marching method.* In general, such methods will produce the value of $\mathbf{x}_{k+1}$ by means of the values of $\mathbf{x}_k, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_1, \mathbf{x}_0$. Thus, we can introduce the notation

$$\mathbf{x}_{k+1} = \Phi(t_k; \mathbf{x}_{k+1}, \mathbf{x}_k, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_0; h; \mathbf{f}), \tag{3.2}$$

to represent an arbitrary method. If Φ does not depend on $\mathbf{x}_{k+1}$, the method is said to be *explicit.* Euler's method is an explicit method. However, if Φ depends on $\mathbf{x}_{k+1}$, then we will have to solve for $\mathbf{x}_{k+1}$ in some way (perhaps using a variation of Newton's method); in this case, the method is said to be *implicit.*

3.2 Error Control

From now on, we assume that the methods can have varying stepsizes. So, we will add a subscript k to h to indicate $h_k = t_{k+1} - t_k$. This is important since all good methods use an adaptive stepsize. In fact, the programs implementing such methods will automatically increase or decrease the stepsizes on the basis of some assessment of the error. Accordingly, let us turn to error estimation.

3.2.1 The Residual

We have seen in chapter 2 that the residual of a numerical solution is

$$\Delta(t) = \dot{\hat{\mathbf{x}}} - \mathbf{f}(t, \hat{\mathbf{x}}(t)),$$

where $\hat{\mathbf{x}}(t)$ is *differentiable* (or at least piecewise differentiable). That the numerical solution be differentiable is very important for the use of the concept of residual in error control, since its evaluation requires the derivative $\dot{\hat{\mathbf{x}}}$ of the numerical solution.

What is the residual for the Euler method? As we have seen, the Euler method generates a sequence of point $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ based on the rule $\mathbf{x}_{k+1} = \mathbf{x}_k + h_k \mathbf{f}(t_k, \mathbf{x}_k)$. The numerical solution is then generated from those points by interpolation.

> We then say that the numerical solution $\hat{\mathbf{x}}$ is the interpolant of those points on the given mesh. The residual will naturally be different depending on the choice of an interpolant.

Given the rudimentary character of the Euler method, it makes sense to choose the equally rudimentary piecewise linear interpolation, which is the one used in figure 3.2. Between the point t_k and t_{k+1}, the interpolant will then be

$$\hat{\mathbf{x}}_k(t) = \mathbf{x}_k + (t - t_k)\frac{\mathbf{x}_{k+1} - \mathbf{x}_k}{t_{k+1} - t_k} = \mathbf{x}_k + (t - t_k)\mathbf{f}(t_k, \mathbf{x}_k). \tag{3.3}$$

Moreover, if we let

$$\theta_k = \frac{t - t_k}{t_{k+1} - t_k} = \frac{t - t_k}{h_k},$$

we can write the pieces of the interpolant as

$$\hat{\mathbf{x}}_k(\theta_k) = \mathbf{x}_k + \theta_k(\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{x}_k + h_k\theta_k \mathbf{f}(t_k, \mathbf{x}_k). \tag{3.4}$$

Then, the argument θ_k ranges over the interval $[0, 1[$. As a result, for the mesh

points $t_0, t_1, \ldots, t_N$ and the generated points $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N$, the interpolant is then defined piecewise as

$$\hat{\mathbf{x}}(t) = \begin{cases} \hat{\mathbf{x}}_0(t) & t_0 \le t < t_1 \\ \hat{\mathbf{x}}_1(t) & t_1 \le t < t_2 \\ \vdots & \\ \mathbf{x}_k(t) & t_k \le t < t_{k+1} \\ \vdots & \\ \mathbf{x}_{N-1}(t) & t_{N-1} \le t \le t_n \end{cases},$$

or it can equivalently be written as a function of θ.

Theorem 3. *The Euler method has an $O(h)$ residual.*

We will give a detailed proof that includes the typical series manipulations and introduces some notation that we will use in the whole chapter.

Proof. Without loss of generality, we choose an interval $t_k \le t < t_{k+1}$. We can substitute the explicit expression of (3.3) for $\hat{\mathbf{x}}$ in the definition of residual:

$$\begin{aligned} \Delta(t) &= \frac{d}{dt}\left(\mathbf{x}_k + (t - t_k)\mathbf{f}(t_k, \mathbf{x}_k)\right) - \mathbf{f}\big(t, \mathbf{x}_k + (t - t_k)\mathbf{f}(t_k, \mathbf{x}_k)\big) \\ &= \mathbf{f}(t_k, x_k) - \mathbf{f}\big(t, \mathbf{x}_k + (t - t_k)\mathbf{f}(t_k, \mathbf{x}_k)\big). \end{aligned} \tag{3.5}$$

Now, we will expand $\mathbf{f}(t, \mathbf{x}_k + (t - t_k)\mathbf{f}(t_k, \mathbf{x}_k))$ in a Taylor series about the point $(t_k, \mathbf{x}_k)$:

$$\begin{aligned} &\mathbf{f}\big(t, \mathbf{x}_k + (t - t_k)\mathbf{f}(t_k, \mathbf{x}_k)\big) = \\ &\quad \mathbf{f}(t_k, x_k) + (t - t_k)\mathbf{f}_t(t_k, \mathbf{x}_k) + (t - t_k)\mathbf{f}_{\mathbf{x}}(t_x, \mathbf{x}_k)\mathbf{f}(t_k, \mathbf{x}_k) + O((t - t_k)^2) \end{aligned}$$

Here, $\mathbf{f}_t(t_k, \mathbf{x}_k)$ is the vector of partial derivatives of $\mathbf{f}$ with respect to t and $\mathbf{f}_{\mathbf{x}}(t_k, \mathbf{x}_k)$ is the Jacobian matrix with partial derivatives with respect $\mathbf{x}$, both evaluated at $(t_k, \mathbf{x}_k)$. From now on, we will not explicitly write the point of evaluation of $\mathbf{f}$ and its derivatives when it is $(t_k, \mathbf{x}_k)$, and simply write $\mathbf{f}, \mathbf{f}_t$ and $\mathbf{f}_{\mathbf{x}}$, for otherwise the expressions would quickly become unreadable. Adding

the fact that $\theta_k h_k = t - t_k$, we obtain the much more neat

$$\mathbf{f}\big(t, \mathbf{x}_k + \theta_k h_k \mathbf{f}\big) = \mathbf{f} + \theta_k h_k (\mathbf{f}_t + \mathbf{f}_\mathbf{x}\mathbf{f}) + O(h_k^2)$$

Putting this in equation (3.5), we find that

$$\begin{aligned}\Delta(t) &= \mathbf{f} - \big(\mathbf{f} + \theta_k h_k (\mathbf{f}_t + \mathbf{f}_\mathbf{x}\mathbf{f}) + O(h_k^2)\big) \\ &= -\theta_k h_k (\mathbf{f}_t + \mathbf{f}_\mathbf{x}\mathbf{f}) + O(h_k^2),\end{aligned}$$

which is $O(h)$ since k was arbitrarily chosen. □

Note that, in this proof, we have also given an explicit expression for the first term of the residual of the numerical solution on an arbitrary subinterval. It can be expanded in vectors and matrices as follows:

$$\Delta(t) \doteq -\theta_k h \left(\begin{bmatrix} \partial f_1/\partial t \\ \partial f_2/\partial t \\ \vdots \\ \partial f_n/\partial t \end{bmatrix}_{\substack{t=t_k \\ \mathbf{x}=\mathbf{x}_k}} + \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \dots & \partial f_1/\partial x_n \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \dots & \partial f_2/\partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \partial f_n/\partial x_2 & \dots & \partial f_n/\partial x_n \end{bmatrix}_{\substack{t=t_k \\ \mathbf{x}=\mathbf{x}_k}} \overline{\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}_{\substack{t=t_k \\ \mathbf{x}=\mathbf{x}_k}}} \right)$$

Note that, instead of giving an explicit expression for the residual in terms of its Taylor expansion, there are situations in which we are satisfied with only a bound for it. Suppose that $\mathbf{f}$ satisfies a Lipschitz condition with Lipschitz constant L. Moreover, without loss of generality, suppose $\mathbf{f}$ is autonomous. The residual of a numerical solution generated with the Euler method is then

$$\begin{aligned}\Delta(t) = \dot{\hat{\mathbf{x}}}(t) - \mathbf{f}(\hat{\mathbf{x}}) &= \frac{d}{dt}(\mathbf{x}_k + (t - t_k)\mathbf{f}(\mathbf{x}_k)) - \mathbf{f}(\mathbf{x}_k + (t - t_k)\mathbf{f}(\mathbf{x}_k)) \\ &= \mathbf{f}(\mathbf{x}_k) - \mathbf{f}(\mathbf{x}_k + (t - t_k)\mathbf{f}(\mathbf{x}_k))\end{aligned}$$

and so

$$\begin{aligned}\|\Delta(t)\| &\leq L\|\mathbf{x}_k - \mathbf{x}_k - (t-t_k)\mathbf{f}(\mathbf{x}_k)\| \\ &\leq Lh_k\|\mathbf{f}(\mathbf{x}_k)\|.\end{aligned}$$

This is a natural inequality given the connection between the Jacobian and the Lipschitz constant.

We can now define the important concept of the order of a method.

Definition 3 (Order of a method Φ). *If a method Φ has residual $O(h^p)$, then it is said to be a pth order method.*

In general, the higher the order of a method is, the more accurate it is for smooth problems. For instance, the MATLAB code `ode45` implements an order 4 method. In the literature on the numerical solutions of ODEs, the concept of accuracy is usually formulated using the following definition:

Definition 4 (Global error). *The global error* $\mathbf{ge}(t)$ *for the numerical solution from t_0 to t is simply*

$$\mathbf{ge}(t) = \hat{\mathbf{x}}(t) - \mathbf{x}(t). \tag{3.6}$$

As we see, the global error is simply what we have so far called *forward error*. In fact, it is much preferable to use the term "forward error," since it makes explicit the uniformity of the methods of error analysis about differential equations and about other things, such as matrix equations, roots of polynomials, interpolation, *etc.*.

Note that, since the global error is nothing but the forward error, we can use the formulas of section 2.3 for it, such as the Gröbner-Alexeev formula

$$\mathbf{z}(t) - \mathbf{x}(t) = \int_{t_0}^{t} \mathbf{G}(t,\tau,\mathbf{z}(\tau))\Delta(\tau)d\tau.$$

We can also use the inequality

$$\|\mathbf{z}(t) - \mathbf{x}(t)\| \leq \kappa(\mathbf{X}_1)\|\Delta(t)\|,$$

provided that the correlation between the solution and the residual does not matter too much (which, in practice, is always a correct assumption). Moreover, in chapter 2, we simply assumed that the residual was some function $\varepsilon\mathbf{v}$ with $\|\mathbf{v}\|_\infty \leq 1$. Now, if we know what method has been used for the computation of the numerical solution, we can actually find an expression for $\varepsilon\mathbf{v}$, if we wish to do so.

This shows that the order of the residual and the order of the global error are the same, so the order of a method can be characterized by one or the other interchangeably.

3.2.2 Local error: an old-fashioned control of error

An old-fashioned way to control the error in the numerical solution of ODEs is based on the concept of local error.

Definition 5 (Local truncation error). *The* local truncation error *or, for short,* local error, *is the error* **le** *incurred by the method* Φ *over a single time step* $[t_k, t_{k+1}]$ *of length* h_k *as* $h_k \to 0$. *Accordingly, assuming that* $\mathbf{x}(t_k) = \mathbf{x}_k$ *holds exactly at the beginning of the interval, the local error is*

$$\mathbf{le} = \mathbf{x}(t_{k+1}) - \mathbf{x}_{k+1} = \mathbf{x}(t_{k+1}) - \Phi(t_k; \mathbf{x}_{k+1}, \mathbf{x}_k, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_0; h; \mathbf{f}) \qquad (3.7)$$

as $h \to 0$. *Associated with* **le** *is the local error per unit step,* **lepus**, *which is just*

$$\mathbf{lepus} = \frac{\mathbf{le}}{h_k}.$$

The concept of 'local error' has traditionally been used in construction of numerical methods for the solution of differential equations, but has been used nowhere else. We strongly believe (following Shampine, Enright, and others) that the concept of *residual* is more general, more physically intuitive, and more understandable. In fact, as the reader will see, there's no reasonable interpretation of the local error in terms of the modeling context in which the computation occurs. However, since local error is commonly used, we will

explain it briefly here, and show how it relates to our approach in terms of residual.

To begin with, let us find the local error of Euler's method.

Theorem 4. *The local error in Euler's method is $O(h^2)$.*

Proof. Suppose $\mathbf{x}(t_k)$ is known exactly at t_k, *i.e.*, suppose $\mathbf{x}(t_k) = \mathbf{x}_k$. If we expand the solution $\mathbf{x}(t)$ about t_k, we obtain

$$\mathbf{x}(t) = \sum_{k=0}^{\infty} \frac{\mathbf{x}^{(k)}(t_k)}{k!}(t - t_k)^k = \mathbf{x}(t_k) + \dot{\mathbf{x}}(t_k) + \frac{\ddot{\mathbf{x}}(t_k)}{2}(t - t_k)^2 + O\left((t - t_k)^3\right) \tag{3.8}$$

Because $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x})$, we have

$$\mathbf{x}(t) = \mathbf{x}(t_k) + \mathbf{f}(t_k, \mathbf{x}_k)(t - t_k) + \frac{\dot{\mathbf{f}}(t_k, \mathbf{x}_k)}{2}(t - t_k)^2 + O\left((t - t_k)^3\right)$$

Note that the derivative of $\dot{\mathbf{f}}$ at (t_k, x_k) is just $\mathbf{f} = \mathbf{f}_t + \mathbf{f}_{\mathbf{x}}\dot{\mathbf{x}} = \mathbf{f}_t + \mathbf{f}_{\mathbf{x}}\mathbf{f}$, by the chain rule. Accordingly, we can rewrite equation (3.8) in this way:

$$\mathbf{x}(t) = \mathbf{x}(t_k) + (t - t_k)\mathbf{f} + \frac{(t - t_k)^2}{2}(\mathbf{f}_t + \mathbf{f}_{\mathbf{x}}\mathbf{f}) + O\left((t - t_k)^3\right)$$

Now, we evaluate $\mathbf{x}(t)$ at $t = t_{k+1}$ using this series:

$$\begin{aligned}\mathbf{x}(t_{k+1}) &= \mathbf{x}(t_k) + (t_{k+1} - t_k)\mathbf{f} + \frac{(t_{k+1} - t_k)^2}{2}(\mathbf{f}_t + \mathbf{f}_{\mathbf{x}}\mathbf{f}) + O\left((t_{k+1} - t_k)^3\right) \\ &= \mathbf{x}(t_k) + h_k\mathbf{f} + \frac{h_k^2}{2}(\mathbf{f}_t + \mathbf{f}_{\mathbf{x}}\mathbf{f}) + O\left(h_k^3\right)\end{aligned}$$

So, the local error $\mathbf{le} = \mathbf{x}(t_{k+1}) - \mathbf{x}_{k+1}$ is:

$$\mathbf{le} = \mathbf{x}(t_{k+1}) - \mathbf{x}_{k+1} = \frac{h_k^2}{2}(\mathbf{f}_t + \mathbf{f}_{\mathbf{x}}\mathbf{f}) + O\left(h_k^3\right) = O\left(h_k^2\right) \tag{3.9}$$

Therefore, the local error is $O(h^2)$. □

In our reasoning about local truncation error, we have assumed that $\mathbf{x}(t_k) =$

$\mathbf{x}_k$ holds exactly to find out the error over a single step. However, when we use the method, we are only guaranteed that $\mathbf{x}(t_0) = \mathbf{x}_0$; the further points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ will not in general be exactly on the curve $\mathbf{x}(t)$. Thus, for each step $k > 0$, we based our calculation on approximate initial points, and we must ask how the error accumulates as we go through the interval. This cumulative local truncation error gives another expression for the global error:

$$\begin{aligned}\mathbf{ge} &= \mathbf{x}(t_k) - \mathbf{x}_k \\ &= \mathbf{x}(t_k) - (\mathbf{x}_0 + \Phi(t_0; \mathbf{x}_1, \mathbf{x}_0; h; \mathbf{f}) + \ldots + \Phi(t_{k-1}; \mathbf{x}_k, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_0; h; \mathbf{f}))\end{aligned}$$

This formula is to a large extent why, traditionally, the error analysis of numerical solution to differential equations has focused on the control of local error. By controlling the local error, we obtain a way to keep a certain control on the global error, which is what we really want to control. This formula, however, involves a lot of bookkeeping and obfuscates the relation between error control in numerical analysis for ODEs and other fields of numerical analysis.[2]

Now, controlling the residual gives a more direct control of accuracy, and it also characterized the order of the method.[3] As an error-control strategy, controlling the local error provides satisfactory results because it gives an indirect control on the residual. We make this precise below with a theorem improving the one in Stetter (1973).

Theorem 5. *Controlling the local error (specifically, the local error per unit*

[2]The concept of local error is also potentially deceitful in another way. Many users think that setting `rtol=1.0e-6` in MATLAB means that the code will attempt to guarantee that

$$\|\hat{\mathbf{x}}(t) - \mathbf{x}(t)\| \approx 10^{-6}\|\mathbf{x}(t)\|. \tag{3.10}$$

Instead, it only tries to make it so on a given interval h_k, assuming that $\hat{\mathbf{x}}(t_k) = \mathbf{x}(t_k)$ holds exactly. The relationship to global error and the residual is more remote than many users think. Consequently, local error adds to the difficulty of interpretation of the quality of the numerical solution.

[3]Traditionally, the order of a method is said to be p when the local error is of order $p+1$. This is awkward; the definition in terms of the residual is more natural.

step) indirectly controls the residual. Moreover, if $h_k L_k < B$ *and* **lepus** $< \varepsilon$,

$$\|\Delta(t)\| \leq (4+B)(2+B)\varepsilon. \tag{3.11}$$

Proof. Without loss of generality, we assume that the problem $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ is autonomous. Suppose also that the underlying numerical solution method used the mesh $t_0 < t_1 < \ldots < t_k < t_{k+1} < \ldots < t_N$ and that the mesh has been chosen in such a way as to ensure that the **lepus** is less than or equal to a given tolerance $\varepsilon > 0$ on each subinterval.

Moreover, let $\mathbf{x}_k(t)$ be the *local solutions* on intervals $t_k \leq t < t_{k+1}$, so that

$$\dot{\mathbf{x}}_k(t) = \mathbf{f}(\mathbf{x}), \quad \mathbf{x}(t_k) = \mathbf{x}_k, \quad t_k \leq t < t_{k+1}.$$

As defined above, we also have $\mathbf{le} = \hat{\mathbf{x}}_{k+1} - \mathbf{x}(t_{k+1})$ and $\mathbf{lepus} = \mathbf{le}/h_k$.

Now, consider the following theoretical interpolant satisfying the conditions $\hat{\mathbf{x}}(t_k) = \mathbf{x}_k(t_k), \hat{\mathbf{x}}'(t_k) = \mathbf{f}(\mathbf{x}_k(t_k)), \hat{\mathbf{x}}(t_{k+1}) = \mathbf{x}_k(t_{k+1})$ and $\hat{\mathbf{x}}'(t_{k+1}) = \mathbf{f}(\mathbf{x}_k(t_{k+1}))$:

$$\hat{\mathbf{x}}(t) = \mathbf{x}_k(t) - \frac{(t-t_k)^2}{h_k^2}\mathbf{le} + \left(\mathbf{f}(\mathbf{x}_k(t_{k+1})) - \mathbf{x}_k'(t_{k+1}) + \frac{2}{h_k}\mathbf{le}\right)\frac{(t-t_k)^2(t-t_{k+1})}{h_k^2}$$

Another interpolant could have been chosen, but this one is good enough. Note that the derivative of this interpolant is

$$\begin{aligned}\hat{\mathbf{x}}'(t) = \mathbf{x}_k'(t) &- \frac{2(t-t_k)}{h_k^2}\mathbf{le} \\ &+ \left(\hat{\mathbf{x}}'(t_{k+1}) - \mathbf{x}_k'(t_{k+1}) + \frac{2}{h_k}\mathbf{le}\right)\left(\frac{2(t-t_k)(t-t_{k+1}) + (t-t_k)^2}{h_k^2}\right).\end{aligned}$$

So, the residual $\Delta(t) = \hat{\mathbf{x}}' - \mathbf{f}(\hat{\mathbf{x}})$ is

$$\begin{aligned}\Delta(t) &= \mathbf{x}_k'(t) - \mathbf{f}(\hat{\mathbf{x}}(t)) - \frac{2(t-t_k)}{h_k^2}\mathbf{le} \\ &\quad + \left(\hat{\mathbf{x}}'(t_{k+1}) - \mathbf{x}_k'(t_{k+1}) + \frac{2}{h_k}\mathbf{le}\right)\left(\frac{2(t-t_{k+1})+(t-t_k)}{h_k^2}\right)(t-t_k) \\ &= \mathbf{f}(\mathbf{x}_k(t)) - \mathbf{f}(\hat{\mathbf{x}}(t)) - \frac{2}{h_k^2}(t-t_k)\mathbf{le} \\ &\quad + \left(\mathbf{f}(\hat{\mathbf{x}}(t_{k+1})) - \mathbf{f}(\mathbf{x}_k(t_{k+1})) + \frac{2}{h_k}\mathbf{le}\right)\left(\frac{2(t-t_{k+1})+(t-t_k)}{h_k^2}\right)(t-t_k).\end{aligned}$$

Therefore, using the fact that $(t-t_k)/h = \theta \leq 1$, we find that

$$\|\Delta(t)\| \leq \|\mathbf{f}(\mathbf{x}_k(t)) - \mathbf{f}(\hat{\mathbf{x}}(t))\| + 8\frac{\|\mathbf{le}\|}{h_k} + 3\|\mathbf{f}(\hat{\mathbf{x}}(t_{k+1})) - \mathbf{f}(\mathbf{x}_k(t_{k+1}))\|.$$

By the definition of this theoretical interpolant, we also have

$$\mathbf{x}_k(t) - \hat{\mathbf{x}}(t) = \frac{(t-t_k)^2}{h_k^2}\mathbf{le} - \left(\mathbf{f}(\hat{\mathbf{x}}(t_{k+1})) - \mathbf{f}(\mathbf{x}_k(t_{k+1})) + \frac{2}{h_k}\mathbf{le}\right)\frac{(t-t_n)^2(t-t_{k+1})}{h_k^2},$$

from which, using the Lipschitz condition, it follows that

$$\|\mathbf{x}_k(t) - \hat{\mathbf{x}}(t)\| \leq \|\mathbf{le}\| + h_k\left(L\|\mathbf{le}\| + \frac{2}{h_k}\|\mathbf{le}\|\right) = (3 + h_kL)\|\mathbf{le}\|.$$

As a result, we finally obtain

$$\begin{aligned}\|\Delta(t)\| &\leq L(3+h_kL)\|\mathbf{le}\| + \frac{8}{h_k}\|\mathbf{le}\| + 3L\|\mathbf{le}\| \\ &= \left(\frac{8}{h_k} + L(3+h_kL) + 3L\right)\|\mathbf{le}\|.\end{aligned}$$

Alternatively, we have the following expression In terms of **lepus**:

$$\|\Delta(t)\| \leq (8 + h_kL_k(3+h_kL_k) + 3h_kL_k)\,\|\mathbf{lepus}\|, \tag{3.12}$$

where $L_k \leq L$ is the local Lipschitz constant on this subinterval. Moreover, if

we let $h_k L_k = B$, we obtain

$$\|\Delta(t)\| \leq (4 + B)(2 + B)\varepsilon. \tag{3.13}$$

This completes the proof. □ □

This bound is apt to be pessimistic in most cases; admittedly the bound $h_n L_n \leq B$ might be inconvenient in practice as well. Nonetheless, it gives a clear rationale for expecting that controlling the local error per unit step will also control (up to a problem-and-method-dependent constant B) the residual. In addition, note that this analysis is independent of the method used to generate the mesh or the solutions $\mathbf{x}_k$ at the mesh points or the method used to guarantee that $\|\mathbf{lepus}\| \leq \varepsilon$. This analysis works for one-step methods and for multisptep methods (indeed for Taylor series methods and general multistep methods too).

3.2.3 Convergence and consistency of methods

We begin our discussion of convergence with the simplest initial-value problem, namely

$$\dot{x} = f(t, x) = \lambda x, \qquad x(0) = x_0. \tag{3.14}$$

Then, the analytic solution is $x(t) = x_0 e^{\lambda t}$. If we use Euler's method (with fixed stepsize) to tackle this problem, we can then give a general expression for the x_ks:

$$x_{k+1} = x_k + hf(t_k, x_k) = x_k + h\lambda x_k = (1 + h\lambda)x_k = (1 + h\lambda)^k x_0. \tag{3.15}$$

The region of convergence for this method is then a disk of radius 2 centered at -1 in the complex plane. If $|1 + h\lambda| < 1$, *i.e.*, if $h\lambda$ is inside the disk, the computed solution $\hat{x}(t)$ will go to x_0 as $t \to \infty$. If this condition fails, *i.e.*, if $h\lambda$ is outside the disk, the computed solution $\hat{x}(t)$ will go to ∞ as $t \to \infty$. Now, suppose the problem is ill-conditioned, *i.e.*, suppose $\lambda > 0$. Then, since

$h > 0$, the condition will fail and the computed solution $\hat{x}(t)$ will go to ∞. This, however, is unproblematic from the point of view of convergence, since $x(t) = x_0 e^{\lambda t}$ will also go to ∞ as $t \to \infty$ (in this case, however, it might be problematic from the point of view of accuracy, as we have seen).

Now what happens if $\lambda < 0$? In this case, convergence requires a relation of inverse proportionality: the larger $|\lambda|$ is, the smaller h will have to be. Accordingly, as the problem becomes increasingly well-conditioned, we have to make the stepsize $h \to 0$ to guarantee convergence. In practice, however, this means that, as the condition number increases, the *cost* of the solution increases, since the program will have to take more steps to go through an interval. We will come back to this point in the next section.

For now, let us formally introduce the concepts involved in order to precisely establish a key connection between them:

Definition 6 (Consistency of a method). *A method is said to be* consistent *if*

$$\lim_{h \to 0} \Delta(t) = 0,$$

i.e., *if the residual of the method tends to 0 as* $h \to 0$.

Obviously, any method whose residual is the product of some power of h by some factor independent of h will be consistent. As we have seen in theorem 3, this is the case for Euler's method.

Definition 7 (Convergence of a method). *A method is said to be* convergent *if*

$$\lim_{h \to 0} \|\mathbf{x}(t) - \hat{\mathbf{x}}(t)\| = 0,$$

i.e., *if the forward error of the method tends to 0 as* $h \to 0$.

Note that, as we have explained in chapter 2, the forward error is bounded by the product of the condition number and the norm of the residual, *i.e.*

$$\|\mathbf{x}(t) - \hat{\mathbf{x}}(t)\| \leq \kappa \|\Delta(t)\|.$$

In this case, if the method is consistent then it will also be convergent, provided that the problem is well-conditioned (*e.g.*, a badly conditioned problem with $\kappa = 1/\|\Delta(t)\|$ could make the condition fail).

3.3 Stiffness and Implicitness

As shown in section 2.6, stiff problems are very well-conditioned problems for which numerical methods have to reduce the stepsize in order to maintain stability, even if accuracy would allow for large stepsizes. In the example of equation (3.14) in the last section, we have seen just that: when λ is large and negative, we have to take very small stepsizes to maintain convergence. As a result, the method becomes extremely inefficient. In section 2.6, we also suggested that the cure to this problem is the use of implicit methods. So, let us consider a first-order implicit method, namely Euler's implicit method:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{f}(t_{k+1}, \mathbf{x}_{k+1}) \tag{3.16}$$

If we again consider the example from equation (3.14), we obtain

$$x_{k+1} = x_k + hf(t_{k+1}, x_{k+1}) = x_k + h\lambda x_{k+1},$$

so that

$$x_{k+1} = \frac{1}{1-h\lambda}x_k = \left(\frac{1}{1-h\lambda}\right)^k x_0.$$

The region of convergence for this method is then the entire complex plane minus a disk of radius 2 centered at 1. For any well-conditioned problem, $h\lambda$ will be in the left half-plane, and thus Euler's implicit method will converge without restriction. The software will thus be able to take the steps as large as accuracy permits.

However, we should note that the much larger region of stability does not justify using implicit methods by default. For non-linear problems with vector functions $\mathbf{f}$, the cost of solving the system for the implicit value will be very

high. Only seriously stiff problems justify the use of implicit methods from the point of view of efficiency.[4]

Note that this example gives us grounds to draw practically general conclusions since, in practice, the numerical solution of a problem involves linearizing the equation, fixing the variable coefficients to some value of interest, and finally diagonalizing the system to obtain decoupled differential equations (Higham and Trefethen, 1993). But then, the decoupled equations will have the form of the equation in this example.

The story above, explaining the success of implicit methods in terms of regions of stability is, however, not the whole story. Let us once again assume that we have linearized our problem, that we have fixed the coefficients, and that we have decoupled the equations. In this common context, fundamental solutions of differential equations have the form $e^{\lambda t}$, and the general solutions of the homogeneous part are linear combinations of such terms (as we have seen in section 2.3.2). Now, we have seen that the explicit Euler method in fact corresponds to the two leading terms of a Taylor expansion. Moreover, as we will see later in the chapter, higher-order methods are also constructed so that they match the leading terms of the Taylor series. Accordingly, we will examine the accuracy of Taylor polynomials to approximate the exponential function.

To begin with, we observe the asymmetric relative accuracy δ_{exp} of truncated Taylor series for e^x:

$$p(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120}$$

$$\delta_{exp}(x) = \frac{p(x) - e^x}{e^x} = p(x)e^{-x} - 1.$$

Note $\delta_{exp}(-2.0) \doteq 0.508$ is more than 30 times as large as $\delta_{exp}(2.0) \doteq 0.016$. Accuracy is quite a bit better to the right, with a relative error of less than 2%, but more than 50% on the left. See figure 3.3. The more terms we keep in the series, the bigger the factor is. Lower order approximations are not

[4]There is also a significant practical difficulty: solving the nonlinear systems for large h may be difficult, or even impossible.

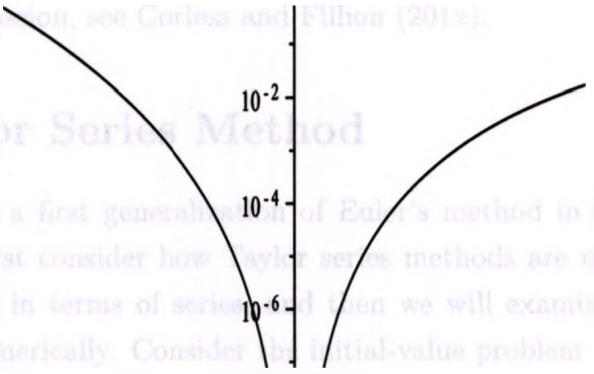


Figure 3.3: Approximation of e^x by a Taylor polynomial has a larger relative error for negative x than it does for positive x.

quite as bad; if we only keep $p(x) = 1 + x + x^2/2 + x^3/3$, we have a factor of 24 difference. However, the asymmetry persists at all orders, as can be seen by series expansion of $\delta_{exp}(x)$, showing that the series has alternating signs and thus must be larger on the left, with negative x, when all the terms have the same sign. Naturally, if we examine e^{-x} and its corresponding Taylor polynomials, the 'left' and 'right' above will be interchanged.

What does this asymmetry mean? It means that Taylor polynomials are good at growing, but that they are not so good at flattening out. Now, e^x grows to the right (faster than polynomials can, it's true), and flattens out very quickly to the left. Fitting a polynomial at one point only, as we are doing with Taylor series, reflects this asymmetry. This phenomenon can be observed no matter about what point we expand e^x, as one can easily check.

What does that mean for the example we have been examining? Suppose we integrate forward for positive time. On the one hand, if the problem is ill-conditioned, then λt is going to be positive. In this case, the Taylor polynomial will have a better relative residual on the right-hand side, as shown in figure 3.3. Thus, explicit methods based on Taylor series will do better than implicit methods. On the other hand, if the problem is well-conditioned, then λt will be negative. Accordingly, the situation will be reversed and the Taylor polynomial will have a better residual on the left-hand side. In this case, implicit methods based on Taylor series will do better than explicit methods.

For a fuller discussion, see Corless and Fillion (201x).

3.4 Taylor Series Method

We now look at a first generalization of Euler's method in terms of Taylor series. Let us first consider how Taylor series methods are used analytically to find solutions in terms of series, and then we will examine how they are implemented numerically. Consider the initial-value problem

$$\dot{x}(t) = x(t)^2 - t, \qquad x(1) = 2. \tag{3.17}$$

Now, we suppose the existence of a solution $x(t)$ that can be written as a Taylor series about $t_0 = 1$:

$$\begin{aligned} x(t) &= x(t_0) + \dot{x}(t_0)(t - t_0) + \frac{\ddot{x}(t_0)}{2!}(t - t_0)^2 + \frac{\dddot{x}(t_0)}{3!}(t - t_0)^3 + \cdots \\ &= \sum_{k=0}^{\infty} \frac{x^{(k)}(t_0)}{k!}(t - t_0)^k \end{aligned}$$

The Taylor series method consists in determining the coefficients $x^{(k)}(t_n)/k!$—we will denote them by $x_{n,k}$—in a recursive way. That is, given that we know $x(t_0)$ (henceforth denoted $x_{0,0}$) and that we know how to differentiate the differential equation, we can find all the coefficients $x_{n,k}$ automatically. Coming back to our example, we obtain $\dot{x}(t_0) = \dot{x}(1)$ by direct substitution in (3.17):

$$\dot{x}(1) = x(1)^2 - 1 = 2^2 - 1 = 3.$$

We can then differentiate the differential equation as many times as needed and then evaluate at $t_0 = 1$, *e.g.*,

$$\begin{aligned} \ddot{x}(t) &= 2x(t)\dot{x}(t) - 1 & \ddot{x}(1) &= 2 \cdot 2 \cdot 3 - 1 = 11 \\ \dddot{x}(t) &= 2(x(t)\ddot{x}(t) + \dot{x}(t)^2) & \dddot{x}(1) &= 2(2 \cdot 11 + 3^2) = 62, \end{aligned}$$

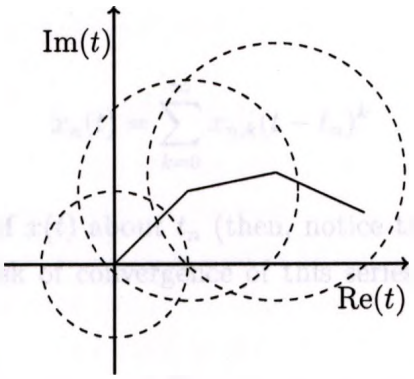


Figure 3.4: Analytic Continuation, *i.e.*, successive application of the Taylor series method.

Accordingly, the solution can be written as

$$x(t) = 2 + 3(t-1) + \frac{11}{2}(t-1)^2 + \frac{62}{6}(t-1)^3 + \cdots . \tag{3.18}$$

Now, associated with the Taylor series about t_n is a radius of convergence. Within the radius of convergence, we can use a Taylor polynomial containing the first N terms of the the series to approximate $x(t)$ with a residual which is at most $O((t-t_n)^N)$, but there is no such guarantee outside the radius of convergence. Accordingly, any time step within the radius of convergence will give valid approximating results. So, if we let t_n be in the radius of convergence, we can find $x(t_n)$ to the order of accuracy desired. Moreover, we can then use this as a new initial value and expand $x(t)$ in a Taylor series about this point. If this new expansion has a convergence disk not entirely overlapping with the original one, this allows us to advance outside the original convergence disk. See figure 3.4.

More generally, for an initial-value problem

$$\dot{x} = f(t, x(t)), \qquad x(t_0) = x_0, \qquad t_0 \le t \le t_N, \tag{3.19}$$

let us denote by

$$x_n(t) = \sum_{k=0}^{\infty} x_{n,k}(t - t_n)^k \tag{3.20}$$

the Taylor expansion of $x(t)$ about t_n (then, notice that $x_{0,0} = x_0(t_0) = x_0$). For any t_{n+1} in the disk of convergence of this series, we may then find (in theory)

$$x_n(t_{n+1}) = \sum_{k=0}^{\infty} x_{n,k}(t_{n+1} - t_n)^k \tag{3.21}$$

The *analytic continuation* idea is just to repeat this procedure: let $x_n(t_{n+1}) = x_{n+1,0}$ be the new initial value and find a new Taylor series of the problem

$$\dot{x} = f(t, x(t)), \qquad x(t_{n+1}) = x_{n+1,0} \tag{3.22}$$

and repeat. By piecing together these series, we find a complete solution along the path from t_0 to t_N (Barton et al., 1971).

In numerical practice, the Taylor series method uses not series but polynomials:

$$\hat{x}_n(t) = \sum_{k=0}^{N} x_{n,k}(t - t_n)^k. \tag{3.23}$$

The resulting (absolute) residual about $t = t_n$ is then

$$\Delta_n(t) = \dot{\hat{x}}_n - f(\hat{x}_n) \tag{3.24}$$

$$= \sum_{k=1}^{N} k x_{n,k}(t - t_n)^{k-1} - f\left(\sum_{k=0}^{N} x_{n,k}(t - t_n)^k\right) \tag{3.25}$$

$$= r_N(t - t_n)^N + r_{N+1}(t - t_n)^{N+1} + \cdots \tag{3.26}$$

To obtain an automatic numerical procedure, we need a way find an iterative numerical substitute to repeatedly differentiating the differential equation.

This is done by using the algebra of power series studied in Corless and Fillion (201x, chap. 2).

Observe that we could think about this method in a dual way, which is the one used for computation. The presentation above *assumes* that the coefficients in the series representation of the solution are Taylor series coefficients (as in equation (3.20)), from which it results as a *matter of fact* that the first N coefficients of the residual are zeros. Instead, we can merely assume that the solution is represented a a formal power series, and we make no assumption as to whether they are Taylor coefficients. Then, we can impose the *constraint* that the first N coefficients of the residual are zeros, and conclude that the coefficients of the formal power series for the solution are indeed Taylor coefficients. This dual perspective would be better named "minimal residual formal power series method." But in any case, it is the one used in practice.

Let us examine an example we discussed in chapter 2 (but we don't use x_1, x_2 to avoid ambiguity in the indices):

$$\begin{aligned} \dot{S} &= -S^3 I, \qquad S(0) = 1 \\ \dot{I} &= -SI^2, \qquad I(0) = 1. \end{aligned} \tag{3.27}$$

Let us examine it with the series method. To begin with, we let

$$S_n(t) = \sum_{k=0}^{N} S_{n,k}(t - t_n)^k \qquad \text{and} \qquad I_n(t) = \sum_{k=0}^{N} I_{n,k}(t - t_n)^k \tag{3.28}$$

be truncated power series approximating $S(t)$ and $I(t)$. We define auxiliary quantities to deal with SI, SI^2, S^2, and $S^3 I$, since we will encounter these quantities in the evaluations of the derivatives $\dot{S}(t_n), \ddot{S}(t_n), \dddot{S}(t_n), \ldots$, and

$\dot{I}(t_n), \ddot{I}(t_n), \dddot{I}(t_n), \ldots$ required to find the series coefficients. So, let

$$C = SI \doteq \sum_{k=0}^{N} C_{n,k}(t - t_n)^k$$

$$D = SI^2 = CI \doteq \sum_{k=0}^{N} D_{n,k}(t - t_n)^k$$

$$E = S^2 \doteq \sum_{k=0}^{N} E_{n,k}(t - t_n)^k$$

$$F = S^3 I = CE \doteq \sum_{n,k}^{N} F_{k=0}(t - t_n)^k,$$

where the coefficients satisfy the conditions for series multiplication:

$$C_{n,k} = \sum_{j=0}^{k} I_{n,j} S_{n,k-j}$$

$$D_{n,k} = \sum_{j=0}^{k} C_{n,j} I_{n,k-j}$$

$$E_{n,k} = \sum_{k=0}^{k} S_{n,j} S_{n,k-j}$$

$$F_{n,k} = \sum_{j=0}^{k} C_{n,j} E_{n,k-j}.$$

Observe that these sums are just inner products, so they pose no numerical difficulty, as we will see below. The residuals in S and I, denoted Δ_S and Δ_I, are defined to be

$$\Delta_S = \dot{S} + S^3 I = \sum_{k=0}^{N} \Delta_{S,k}(t - t_n)^k + O(t - t_n)^{N+1} \tag{3.29}$$

$$\Delta_I = \dot{I} + SI^2 = \sum_{k=0}^{N} \Delta_{I,k}(t - t_n)^k + O(t - t_n)^{N+1}, \tag{3.30}$$

where the coefficients then satisfy

$$\Delta_{S,k} = (k+1)S_{n,k+1} + F_{n,k} \tag{3.31}$$

$$\Delta_{I,k} = (k+1)I_{n,k+1} + D_{n,k} \tag{3.32}$$

for all k, $0 \leq k \leq N$. Note that $S_{n,N+1} = I_{n,N+1} = 0$ because we truncate this series. As a result, for $0 \leq k \leq N-1$, we may set both $\Delta_{S,k}$ and $\Delta_{I,k}$ to zero as follows: because both D_k and F_k are known once all $S_{n,j}$ and $I_{n,j}$ with $0 \leq j \leq k$ are known, we may simply set

$$S_{n,k+1} = -\frac{F_{n,k}}{k+1} \quad \text{and} \quad I_{n,k+1} = -\frac{D_{n,k}}{k+1}. \tag{3.33}$$

Then, the first N coefficients of Δ_S and Δ_I are zeros and, consequently, the $S_{n,k}$ and $I_{n,k}$ in the truncated power series are the coefficients of the Taylor polynomials. So, we have a recursive method in terms of power series to find the coefficients of the Taylor series without having to differentiate the differential equation directly. Note also that, since

$$\Delta_{S,k} = F_{n,k} \qquad k \geq N \tag{3.34}$$

$$\Delta_{I,k} = D_{n,k} \qquad k \geq N \tag{3.35}$$

our procedure for computing Taylor series also automatically computes Taylor series for the residual. Thus, with little extra effort we will have an error estimate at hand. Of course, we may evaluate both $\Delta_S(t)$ and $\Delta_I(t)$ directly, just as easily, and this is usually best.

For our example, we can easily implement the scheme numerically in MATLAB. Firstly, the coefficients are computed in a straightforward way with this compact code:

```
1 function [I,S] = tsw(I0,S0,N,sg)
2 % Taylor series coeffs of solution of S' = -S^3*I, I'= -S*I^2
3 % S(tn)=S0, I(tn)=I0; sg is the direction of integration.
4 % if tn=0, I0=S0=1, then I=exp(-W(t)) and S = 1/(1+W(t)).
5     I = zeros(1,N+1);
```

```
    S = I; C = I; D = I; E = I; F = I;
    I(1) = I0; S(1) = S0;
    for k=1:N,
        C(k) = S(1:k)*I(k:-1:1).';
        D(k) = C(1:k)*I(k:-1:1).';
        E(k) = S(1:k)*S(k:-1:1).';
        F(k) = C(1:k)*E(k:-1:1).';
        S(k+1) = -sg*F(k)/k;
        I(k+1) = -sg*D(k)/k;
    end
end
```

Then, the residual is also easily computed:

```
function [I,S,r,dt,It,St]=tswresid4text(In,Sn,N,sg)
[I,S]=tsw(In,Sn,N,sg); %Find the coefficients.
t=linspace(0,.5,256);
It=polyval(I(end:-1:1),t);
St=polyval(S(end:-1:1),t);
Std=polyval([N:-1:1].*S(end:-1:2),t);
Itd=polyval([N:-1:1].*I(end:-1:2),t);
r=[Itd+sg*St.*It.^2;Std+sg*St.^3.*It];
rsq = sqrt(r(1,:).^2+r(2,:).^2);
semilogy(t,abs(r(1,:)),'k-',t,abs(r(2,:)),'k-.')
%Find the numerical value dt at which the residual start to be
    bigger than the tolerance, here 1.0e-6, and evaluate the
    Taylor polynomial at this point to have new initial values.
ii=find(abs(rsq)>1.0e-6);
if numel((ii))>0,
    ig=ii(1)-1;
    if ig==0,
        error('failure',rsq(1))
    end
    dt = t(ig);
    It = It(ig);
    St = St(ig);
else
    dt = 1.0;
    It = It(end);
    St = St(end);
```

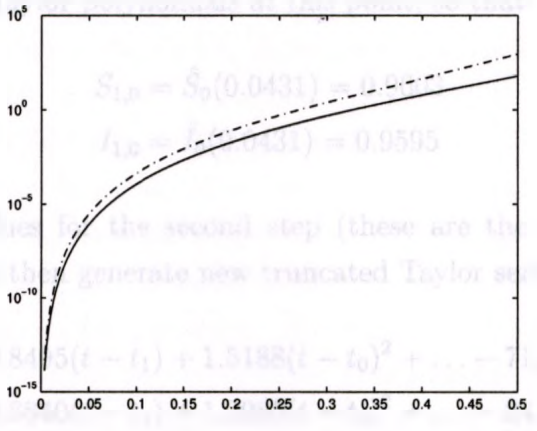


Figure 3.5: Residual of the Taylor polynomials of S (dotted line) and I (hard line).

```
25 end
26 end
```

Let us take two steps explicitly with this method. For no particular reason, take $N = 7$ and start with $n = 0$ and $t_n = t_0 = 0$. Here $S(0) = s_{0,0} = 1$ and $I(0) = I_{0,0} = 1$. The program above gives

$$\hat{S}_0(t) = 1 - t + 2t^2 - 4.5t^3 + 10.6667t^4 - 26.0417t^5 + 64.8t^6 - 163.4014t^7$$
$$\hat{I}_0(t) = 1 - t + 1.5t^2 - 2.667t^3 + 5.2083t^4 - 10.8t^5 + 23.3431t^6 - 52.0127t^7$$

where as usual we have printed only the usual short MATLAB representation of the coefficients; more figures are used than shown. By inspection of the graphs of

$$\Delta_S(t) = \dot{\hat{S}} + \hat{S}^3\hat{I} \quad \text{and} \quad \Delta_I(t) = \dot{\hat{I}} + \hat{S}\hat{I}^2, \tag{3.36}$$

we see that $\sqrt{\Delta_S^2 + \Delta_I^2} \leq 10^{-6}$ if $0 \leq t \leq 0.0431$. We thus take $t_1 = 0.0431$

and evaluate the Taylor polynomials at this point, so that

$$S_{1,0} = \hat{S}_0(0.0431) = 0.9603 \tag{3.37}$$

$$I_{1,0} = \hat{I}_0(0.0431) = 0.9595 \tag{3.38}$$

are our initial values for the second step (these are the `St` and `It` in the program). We can then generate new truncated Taylor series about t_1:

$$\hat{S}_1(t) = 0.9603 - 0.8495(t - t_1) + 1.5188(t - t_0)^2 + \ldots - 71.184(t - t_0)^7$$

$$\hat{I}_1(t) = 0.9595 - 0.8840(t - t_1) + 1.2085(t - t_0)^2 + \ldots - 24.6896(t - 0.0545)^7.$$

Again, the program indicates that $\sqrt{\Delta_{I_1}^2 + \Delta_{S_1}^2}$ is smaller than 10^{-6} if $0 \leq t - 0.0431 \leq 0.0490$.

This process can obviously be repeated. *Provided that we can always take t_{n+1} so that $h_n = t_{n+1} - t_n$ is bounded below* by some minimum stepsize, say $\varepsilon_M t_n$, so that $t_{n+1} > t_n$ in floating-point arithmetic, we may integrate $\dot{x} = f(x)$ from t_0 to some fixed t_N by taking a finite number of steps of this method. At the end, we will have a mesh $t_0 < t_1 < t_2 < \ldots < t_{N-1} < t_N$ and a collection of polynomials $\hat{x}_k(t)$ with residuals $\Delta_k(t)$ on $t_k \leq t \leq t_{k+1}$, where $\|r_k\|$ is at most our tolerance ε. All together, we will have a continuous (but not continuously differentiable) piecewise function $\hat{x}(t)$ solving $\dot{x} = f(x) + \varepsilon v(t)$ and $x(t_0) = y_0$, with $\|v\|_\infty \leq 1$.

The caveat, that we must be able to make progress, *i.e.*, $t_{n+1} > t_n$ in floating-point arithmetic, turns out to be interesting. For tight tolerances, we are *not* be able to get much past $t = -1/e \doteq 0.3679$. This is because the solution is singular there (more precisely, it has a derivative singularity). Location (or merely just detection) of singularities is an interesting topic, and useful in and of itself. We remark that the Taylor series method offers a way to detect such singularities essentially for free; if there is one, then keeping $\|\Delta_n\| \leq \varepsilon$ ensures $h_n \to 0$.

There are widely-distributed Taylor series codes. For instance, MAPLE offers `dsolve/numeric` with the `taylorseries` optional method, written by Allan Wittkopf, works very well. A code for DAE, called DAETS, by Nedialkov and Pryce, is available for industrial applications. The package ATOMFT by Corliss and Chang is freely available. Nonetheless most industrial or other high-quality codes use other methods. A set of historical, evolutionary traps locking in the results of earlier decisions have likely played a role, as follows.

In the early days of computation, there was neither computing time nor memory available to compute or represent interpolants: numerical methods for ODE were expected to produce only a table of values at selected points (and that was hard enough, given limited hardware). There was also little understanding of the code generation needed for what is now called automatic differentiation—and symbolic differentiation, done badly, generates exponential growth in the length of expressions. Finally, and perhaps most important, many problems are not smooth, such as

$$\dot{x} = |1 - x^2|, \qquad x(0) = 0 \tag{3.39}$$

and so derivatives were set aside, as being too much work for too little gain. A more interesting objection is that not all interesting functions have differential equations associated with them. *E.g.*, solving

$$\dot{y}(t) = \Gamma(t) + \frac{y(t)}{1 + \Gamma(t)} \tag{3.40}$$

by Taylor series needs special treatment because the derivatives of Γ are themselves special. However, in practice this seems not to be an issue.

Nonetheless, before moving on, let us recount the advantages of the Taylor series method:

1. It provides a free piecewise interpolant on the whole interval of integration;

2. It provides an easy estimate of the residual from the leading few terms in $\dot{x} - f(x)$;

3. It is a good tool for singularity detection and location;

4. It is flexible as to order of accuracy[5] $N \geq 1$ and adaptive stepsize $h_n = t_{n+1} - t_n$;

5. And finally, but perhaps most importantly, it is now understood how to make program to implement them.

All this is by the way. The fact is that people *did* turn away from Taylor series methods, not realizing their advantages, and invented several classes of beautiful alternatives. Using Taylor series methods as the underlying standard, we now discuss one such class of alternatives, the Runge-Kutta methods.

3.5 Runge Kutta methods

Marching methods, including Euler's method and explicit Runge-Kutta methods, share the following structure: start at $\mathbf{x}_k$ and more to $\mathbf{x}_{k+1}$ by making a step of length h_k along a line whose slope approximated the slope of the secant connecting $\mathbf{x}_k$ and $\mathbf{x}(t_{k+1})$. In the case of Euler's method, we simply used $\mathbf{f}(t_k, \mathbf{x}_k)$ as our approximate value for the slope of the secant. The idea of Runge-Kutta methods is to evaluate the function $\mathbf{f}(t, \mathbf{x})$ more than once, at different points, and to use a weighted average of the values thus obtained as an approximation of the slope of the secant. Then, depending on how many evaluations of $\mathbf{f}$ the method use and on how well the weights of the average have been chosen, the methods so constructed will have a higher or lower order of accuracy. Let us examine a few examples.

[5]We have high accuracy if the solution is smooth and N is large, at a reasonable cost. It has been shown that Taylor series method has cost polynomial in the number of bits of accuracy requested (Ilie et al., 2008; Corless et al., 2006)

3.5.1 Examples of 2nd-, 3rd- and 4th-order RK methods

If we consider adding a second evaluation of the function $\mathbf{f}$, then the natural thing to do is to compute it at the point $(t_{k+1}, \mathbf{x}(t_{k+1}))$. This is exactly what the *Improved Euler method* does:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\left(\frac{1}{2}\mathbf{f}(t_k, \mathbf{x}_k) + \frac{1}{2}\mathbf{f}\big(t_{k+1}, \mathbf{x}_k + h\mathbf{f}(t_k, \mathbf{x}_k)\big)\right). \tag{3.41}$$

This method just takes the arithmetic mean (*i.e.*, the weights are $1/2$) of the slope at the beginning of the interval and at the end of it. At the end of it we substitute the exact point $(t_{k+1}, \mathbf{x}(t_{k+1}))$ by the approximation $(t_{k+1}, \mathbf{x}_k + h\mathbf{f}(t_k, \mathbf{x}_k))$. Then, we need to find a way to interpolate the data generated in an appropriate manner, so as to make it possible to define and compute the residual.

As it turn out, we gain an order of accuracy with this method:

Theorem 6. *The Improved Euler method is a 2nd-order.*

We will examine later how to construct *continuous* Runge-Kutta methods, and we will then be able to find the order of methods based on their residual. But since we do not have this available yet, we will show instead that the order of the local error of the Improved Euler method is $O(h^3)$, which implies that this is an order 2 method, since this approach does not require an interpolant. Moreover, in this section, we will assume without loss of generality that the functions $\mathbf{f}$ are autonomous, since we can always rewrite a non-automous system as an autonomous system of higher dimension using the trick presented in chapter 2. This assumption simplifies very much the Taylor series expansion. Moreover, we will continue to simply write $\mathbf{f}, \mathbf{f}_\mathbf{x}$, *etc*, to denote the evaluation of those derivatives of $\mathbf{x}$ at $\mathbf{x}_k$.

Proof. First, notice that

$$\mathbf{f}(\mathbf{x}_k + h\mathbf{f}) = \mathbf{f} + h\mathbf{f}_\mathbf{x}\mathbf{f} + O(h^2),$$

so that the solution computed by the improved Euler method can be expanded as

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k + \frac{h}{2}\left(\mathbf{f} + \mathbf{f}(\mathbf{x}_k + h\mathbf{f})\right) = \mathbf{x}_k + \frac{h}{2}\left(\mathbf{f} + h\mathbf{f}_{\mathbf{x}}\mathbf{f} + O(h^2)\right) \\ &= \mathbf{x}_k + h\mathbf{f} + \frac{h^2}{2}\mathbf{f}_{\mathbf{x}}\mathbf{f} + O(h^3).\end{aligned}$$

Moreover, the exact solution $\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k + h)$ can be expanded about t_k as

$$\begin{aligned}\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k + h) &= \mathbf{x}(t_k) + h\dot{\mathbf{x}}(t_k) + \frac{h^2}{2}\ddot{\mathbf{x}}(t_k) + O(h^3) \\ &= \mathbf{x}_k + h\mathbf{f} + \frac{h^2}{2}\mathbf{f}_{\mathbf{x}}\mathbf{f} + O(h^3).\end{aligned}$$

As, a result, the local error $\mathbf{le} = \mathbf{x}(t_{k+1}) - \mathbf{x}_{k+1}$ is $O(h^3)$. Therefore, the method is second order. □

As we will examine higher-order method, we will need more machinery to deal with the Taylor series. Expanding vector-valued scalar functions in Taylor series does not require much notation beyond what is found in vector and matrix analysis. However, doing so for vector-valued vector functions requires the introduction of tensors. Even for this simple second-order method, if one tries to find the explicit expression for the first term of the local truncation error, we that the matrix-vector notation is cumbersome (but it can be done, with a lot of patience). For higher-order terms, however, it becomes essential to use tensor notation. So, for now, we will simply present a 3rd- and a 4th-order method without providing any proof.

Here's is a standard third-order RK method works. We use k_i for the

various values of $\mathbf{f}$ we compute:

$$t_1 = t_0 + h \tag{3.42}$$

$$k_1 = \mathbf{f}(t_0, \mathbf{x}_0) \tag{3.43}$$

$$k_2 = \mathbf{f}(t_0 + h, \mathbf{x}_0 + hk_1) \tag{3.44}$$

$$k_3 = \mathbf{f}(t_0 + \frac{h}{2}, \mathbf{x}_0 + \frac{h}{2}\frac{k_1 + k_2}{2}) \tag{3.45}$$

$$x_1 = \mathbf{x}_0 + h\frac{k_1 + k_2 + 4k_3}{6} \tag{3.46}$$

The computations of $\mathbf{f}$ are called *stages*. Each stage of this method is illustrated in figure 3.6. Remark that in the case in which $\mathbf{f}$ depends only on t, and not on $\mathbf{x}$, the problem amounts to $\mathbf{x}(t) = \int_{t_0}^{t_1} \mathbf{f}(t)dt$, so that the method in effect is the very same as Simpson's rule for quadrature.

Here is a fourth-order Runge-Kutta method:

$$t_1 = t_0 + h \tag{3.47}$$

$$k_1 = \mathbf{f}(t_0, \mathbf{x}_0) \tag{3.48}$$

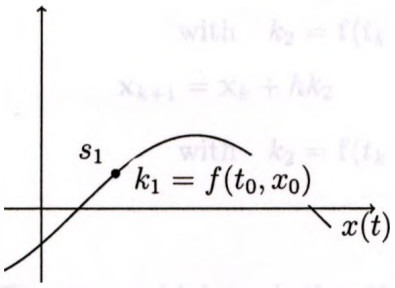
$$k_2 = \mathbf{f}(t_0 + \frac{h}{2}, \mathbf{x}_0 + \frac{h}{2}k_1) \tag{3.49}$$

$$k_3 = \mathbf{f}(t_0 + \frac{h}{2}, \mathbf{x}_0 + \frac{h}{2}k_2) \tag{3.50}$$

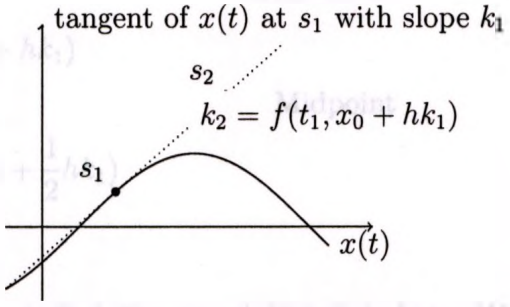
$$k_4 = \mathbf{f}(t_0 + h, \mathbf{x}_0 + hk_3) \tag{3.51}$$

$$x_1 = \mathbf{x}_0 + h\frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \tag{3.52}$$

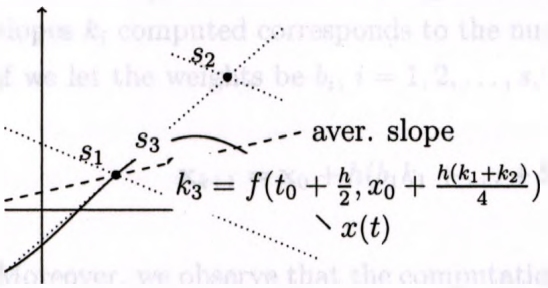
In fact, this method is one of the most popular Runge-Kutta methods. It is often simply referred to as RK4 or "the classical Runge-Kutta method." With these representative examples, it should be clear what strategy Runge-Kutta methods exploit. To continue our investigation, we will now introduce a general notation for the Runge-Kutta methods, and then return to the mechanics of constructing the methods.



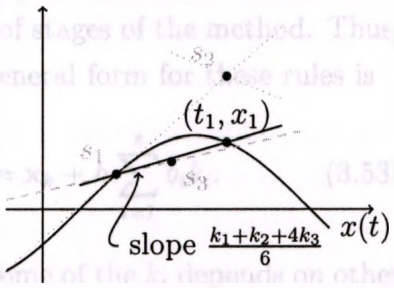
(a) Stage 1. We compute the stage $k_1 = f(t_0, x_0)$ at $s_1 = (t_0, x_0)$.



(b) Stage 2. We move h along the tangent, whose slope is k_1, to the point $s_2 = (t_1, x_0 + hk_1)$. Then, we compute $k_2 = f(t_1, x_0 + hk_1)$.



(c) Stage 3. The slope at s_2 is k_2. We come back to s_1 and move $h/2$ on a straight line whose slope is $k_1 + k2/2$, the average of k_1 and k_2 to the point $s_3 = (t_0 + h/2, x_0 + (h/2)(k_1 + k2/2))$.



(d) We make the step h with the weighted average of k_1, k_2 and k_3. The resulting point is (t_1, x_1)

Figure 3.6: RK3.

3.5.2 Generalization and Butcher Tableaux

We have seen two examples of Runge-Kutta methods above, in (3.46) and (3.52). Other simple methods include the following:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{f}(t_k, \mathbf{x}_k) = \mathbf{x}_k + hk_1 \qquad \text{Euler}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h(\frac{1}{2}k_1 + \frac{1}{2}k_2) \qquad \text{Trapezoidal}$$

$$\text{with} \quad k_2 = \mathbf{f}(t_k + h, \mathbf{x}_k + hk_1)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + hk_2 \qquad \text{Midpoint}$$

$$\text{with} \quad k_2 = \mathbf{f}(t_k + \frac{1}{2}h, \mathbf{x}_k + \frac{1}{2}hk_1)$$

The trapezoidal method is the one we called "improved Euler" before. We observe that each of these 5 methods consists in computing a number of slopes at different points and take a weighted average of them, where the number s of slopes k_i computed corresponds to the number of stages of the method. Thus, if we let the weights be b_i, $i = 1, 2, \ldots, s$, the general form for these rules is

$$\mathbf{x}_{k+1} = \mathbf{x}_0 + h(b_1k_1 + \ldots + b_sk_2) = \mathbf{x}_k + h\sum_{i=1}^{s} b_ik_i. \tag{3.53}$$

Moreover, we observe that the computation of some of the k_i depends on other values k_j (in the explicit cases considered here, only for $j < i$). For instance, in the trapezoidal rule, k_2 depends on k_1 for the value of the second variable in $\mathbf{f}(t, \mathbf{x}(t))$. The parameters indicating how much weight have the previous steps j in finding the new point of evaluation of $\mathbf{f}$ to determine k_i are denoted a_{ij}. Moreover, as we have seen in the midpoint rule, for instance, there is a constant dictating how big a time step we take. Thus, for explicit methods,

we get the general form:

$$k_1 = \mathbf{f}(t_k, \mathbf{x}_k) \tag{3.54}$$

$$k_2 = \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + a_{21} k_1) \tag{3.55}$$

$$k_3 = \mathbf{f}(t_k + c_3 h, \mathbf{x}_k + a_{31} k_1 + a_{32} k_2) \tag{3.56}$$

$$\vdots \tag{3.57}$$

$$k_s = \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + a_{s1} k_1 + a_{s2} k_2 + \ldots + a_{s,s-1} k_{s-1}) \tag{3.58}$$

In general, for explicit methods, we have

$$k_i = \mathbf{f}(t_k + c_i h, \mathbf{x}_k + h \sum_{j=1}^{i-1} a_{ij} k_j). \tag{3.59}$$

Since we always begin with an evaluation of the slope at $(t_k, \mathbf{x}_k)$, we have $c_1 = 0$. Moreover, since the evaluation of k_1 cannot depend on previously computed values of k_i in an explicit method, we have $a_{1,i} = 0, i = 1, 2, \ldots, s$. In the Trapezoidal rule, we have $c_2 = 1$ and $a_{21} = 1$. In the midpoint rule, we have $c_2 = 1/2$ and $a_{21} = 1/2$.

As we see, the weights b, the size of time steps c and the weights a_{ij} of previously computed values of k_j, $j < i$, fully determines a Runge-Kutta method. This information for explicit methods can conveniently be summarized in a tableau, called *Butcher tableau*, having the following form:

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array} = \begin{array}{c|ccccc} 0 & 0 & 0 & 0 & \cdots & 0 \\ c_2 & a_{21} & 0 & 0 & \cdots & 0 \\ c_3 & a_{31} & a_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ c_2 & a_{s1} & a_{s2} & \ldots & a_{s,s-1} & 0 \\ \hline & b_1 & b_2 & b_3 & \cdots & b_s \end{array}, \tag{3.60}$$

where A is a lower-triangular matrix with zeros as diagonal entries. We typically leave the 0s out, leaving the cell blank. As one can expect, an implicit

Runge-Kutta method of the form

$$k_1 = \mathbf{f}(t_k + c_1 h, \mathbf{x}_k + a_{s1}k_1 + a_{s2}k_2 + \ldots + a_{s,s-1}k_{s-1} + a_{ss}k_s) \tag{3.61}$$

$$k_2 = \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + a_{s1}k_1 + a_{s2}k_2 + \ldots + a_{s,s-1}k_{s-1} + a_{ss}k_s) \tag{3.62}$$

$$k_3 = \mathbf{f}(t_k + c_3 h, \mathbf{x}_k + a_{s1}k_1 + a_{s2}k_2 + \ldots + a_{s,s-1}k_{s-1} + a_{ss}k_s) \tag{3.63}$$

$$\vdots \tag{3.64}$$

$$k_s = \mathbf{f}(t_k + c_s h, \mathbf{x}_k + a_{s1}k_1 + a_{s2}k_2 + \ldots + a_{s,s-1}k_{s-1} + a_{ss}k_s) \tag{3.65}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\sum_{i=1}^{s} b_i k_i. \tag{3.66}$$

would have a full Butcher tableau:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array} \tag{3.67}$$

Let us illustrate this notation by giving the Butcher tableau of some methods we have encountered. The midpoint rule gives

$$\begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \end{array} \tag{3.68}$$

The trapezoidal rule gives

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array} \tag{3.69}$$

Finally, the classical Runge-Kutta method RK4 has the following Butcher

tableau:

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \tag{3.70}$$

3.5.3 How to construct a discrete method

We now introduce the traditional strategy to construct a discrete Runge-Kutta method, *i.e.*, a method that returns a discrete set of points $\mathbf{x}_0, \mathbf{x}, \ldots, \mathbf{x}_N$ and the corresponding mesh points. However, this is just a transition stage we make in order to introduce the strategy for the construction of a *continuous* Runge-Kutta method in a didactically acceptable way. Ideally, we would prefer to skip this entirely and move directly to the continuous approach, but we have not yet found an acceptable way to do so.

The traditional strategy goes as follows. We begin by expanding the exact solution $x(t_{k+1})$ about t_k

$$\mathbf{x}(t_{k+1}) = \mathbf{x}_k + h\mathbf{f} + \frac{h^2}{2}(\mathbf{f}_t + \mathbf{f}_\mathbf{x}\mathbf{f}) + O(h^3) \tag{3.71}$$

up to the desired order. We then specify the order p of the method we want to build, as well as the number s of stages we allow (there is, however, a minimum number of stages required to build a method of given order). Then, we solve for the b_i, c_i, and a_{ij}. Let us give the simplest non-trivial example: $p = 2$ and $s = 2$. For $s = 2$, the Runge-Kutta method is given by

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\sum_{i=1}^{s} b_i k_i = \mathbf{x}_k + h(b_1 k_1 + b_2 k_2), \tag{3.72}$$

where the stages are $k_1 = \mathbf{f}(t_k, \mathbf{x}_k) = \mathbf{f}$ and

$$k_2 = \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + h a_{2,1} k_1). \tag{3.73}$$

The first step in the construction is to expand the k_i (here, we have only one) in Taylor series about t_k:

$$k_2 = \mathbf{f} + hc_2\mathbf{f}_t + ha_{21}\mathbf{f}_\mathbf{x}k_1 + O(h^2) = \mathbf{f} + hc_2\mathbf{f}_t + ha_{21}\mathbf{f}_\mathbf{x}\mathbf{f} + O(h^2). \quad (3.74)$$

We then substitute in equation (3.72):

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k + h\left(b_1\mathbf{f} + b_2\left(\mathbf{f} + hc_2\mathbf{f}_t + ha_{21}\mathbf{f}_\mathbf{x}\mathbf{f} + O(h^2)\right)\right) \\ &= \mathbf{x}_k + h\mathbf{f}(b_1 + b_2) + h^2(b_2c_2\mathbf{f}_t + b_2a_{21}\mathbf{f}_\mathbf{x}\mathbf{f}) + O(h^3). \quad (3.75)\end{aligned}$$

Finally, we match the coefficients of the exact solution (3.71) and of the numerical solution (3.75) (assuming that $\mathbf{x}(t_k) = \mathbf{x}_k$), to get a set of constraints for the b_i, c_i, and a_{ij}. In this case, we find that:

$$1 = b_1 + b_2 \quad (3.76)$$

$$\frac{1}{2} = b_2c_2 \quad (3.77)$$

$$\frac{1}{2} = b_2a_{21} \quad (3.78)$$

These equations are called *order conditions*. Now, since we have only three equations for four unknowns, there is a free degree of freedom, *i.e.*, there is an infinite family of two-stage order two Runge-Kutta method. The popular second-order methods we have examined before fall in this category. Setting $b_2 = 1/2$ gives the trapezoidal method. Setting $b_2 = 1$ gives the midpoint method. Many more naturally exist. To go into the details of higher-order methods, we need tensor notation, which we introduce later to discuss the *general* theory of order conditions.

3.5.4 Investigation of Continuous Explicit Runge-Kutta Methods

The idea of a continuous Runge-Kutta method, CERK for short, is quite natural. We already have the idea of generating a discrete set of points

$\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$ on a mesh $t_0, t, \ldots, t_N$ with a Runge-Kutta method. Moreover, we have already seen the importance of somehow interpolating our discrete set of points in order to be able to evaluate and interpret the results. The idea of CERKs is then: instead of doing this in two stages, why not combining them?

How are we to do this? Following Hairer et al. (1993), we simply do as we did in the last subsection, *i.e.*, expand expressing in Taylor series and match coefficients to find order conditions restricting the parameters of the method, to the exception that we let the *weights* b_i be variable and ranging over a subinterval $[t_k, t_{k+1}]$. We have already used this notation before to describe Euler's method interpolated piecewise linearily in equation (3.4). Adapting this equation to our current Runge-Kutta notation, we find:

$$\hat{\mathbf{x}}_k(t) = \mathbf{x}_k + h_k \theta_k \mathbf{f}(t_k, \mathbf{x}_k) \quad \xrightarrow{CERKing} \quad \hat{\mathbf{x}}(\theta) = \mathbf{x}_k + h b_1(\theta) k_1,$$

where $b_1(\theta) = \theta$. In general, for the construction of a CERK, as opposed to a discrete Runge-Kutta method, the rule generating the points $\mathbf{x}_i$ will have the form

$$\hat{\mathbf{x}}(\theta) = \mathbf{x}_k + h \sum_{i=1}^{s} b_i(\theta) k_i. \tag{3.79}$$

Note also that, where we used the notation $\hat{\mathbf{x}}_k(t)$, a function of t indexed for subintervals, we now use the variable θ and drop the index, since it is already built in $\theta = (t - t_k)/h_k$. Also, note that in addition to the order conditions found by the process described in the last section, we also impose other constraints on the constants a_{ij}, such as

$$c_i = \sum_{j=1}^{i-1} a_{ij}, \tag{3.80}$$

where, remember, $c_1 = 0$.

When constructing a continuous Runge-Kutta method, we will again choose an order p for the method and a number of stages s. In this context, the resid-

ual is

$$\Delta(t) = \frac{d}{dt}\hat{\mathbf{x}}(\theta) - \mathbf{f}(\hat{\mathbf{x}}(\theta)). \tag{3.81}$$

Since $\theta = (t - t_n)/h$, this gives

$$\Delta(\theta) = \frac{1}{h}\hat{\mathbf{x}}'(\theta) - \mathbf{f}(\hat{\mathbf{x}}(\theta)), \tag{3.82}$$

where the prime now denotes differentiation with respect to θ. By definition of order, the choice of order imposes a constraint on the residual of the method defined in (3.79), namely is required that $\Delta(t) = O(h^p)$.

As an example, consider again the case of a second-order two-stage method:

$$k_1 = \mathbf{f}(\mathbf{x}_k) \tag{3.83}$$

$$k_2 = \mathbf{f}(t_k + c_2 h, \mathbf{x}_k + h a_{21} k_1) \tag{3.84}$$

$$\hat{\mathbf{x}}(\theta) = \mathbf{x}_k + h\left(b_1(\theta)k_1 + b_2(\theta)k_2\right). \tag{3.85}$$

To begin with, note that the Taylor series of the exact solution is

$$\mathbf{x}_k(t + \theta h) = \mathbf{x}_k + h\theta k_1 + \frac{h^2\theta^2}{2}(\mathbf{f}_t + \mathbf{f}_{\mathbf{x}}\mathbf{f}) + O(h^3). \tag{3.86}$$

In order to match the coefficients of $\hat{\mathbf{x}}(\theta)$ with the exact solution, we first expand k_2:

$$k_2 = \mathbf{f} + c_2 h \mathbf{f}_t + h a_{21} \mathbf{f}_{\mathbf{x}} k_1 + O(h^2) \tag{3.87}$$

As a result, the computed continuous solution has the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h k_1 (b_1(\theta)k_1 + b_2(\theta)) + h^2(b_2(\theta)c_2\mathbf{f}_t + b_2(\theta)a_{21}\mathbf{f}_{\mathbf{x}}k_1) + O(h^3)$$

We can now match the coefficients, thus finding the order conditions:

$$\begin{aligned} b_1(\theta) + b_2(\theta) &= \theta \\ \frac{\theta^2}{2} &= b_2(\theta)c_2 \\ \frac{\theta^2}{2} &= b_2(\theta)a_{21} \end{aligned}$$

Consequently, we can use $c_2 = a_{21} = \alpha$ as a free parameter, so that

$$b_2(\theta) = \frac{\theta^2}{2\alpha} \tag{3.88}$$

$$b_1(\theta) = \theta - b_2(\theta) = \theta - \frac{\theta^2}{2\alpha}. \tag{3.89}$$

We have accordingly generated a family of two-stage second-order continuous explicit Runge-Kutta methods

$$\hat{\mathbf{x}} = \mathbf{x}_k + h\theta k_1 - \frac{h\theta^2}{2\alpha}k_1 + \frac{h\theta^2}{2\alpha}k_2 \tag{3.90}$$

whose Butcher tableaux are

$$\begin{array}{c|cc} 0 & & \\ \alpha & \alpha & \\ \hline & b_1(\theta) & b_2(\theta) \end{array} \quad = \quad \begin{array}{c|cc} 0 & & \\ \alpha & \alpha & \\ \hline & \theta - \theta^2/2\alpha & \theta^2/2\alpha \end{array}. \tag{3.91}$$

This shows how we can solve for the order condition to construct continuous Runge-Kutta methods. If we let $\alpha = 1$, the we have a continuous trapezoidal method. If we let $\alpha = 1/2$, then we have a continuous midpoint method. Now, finding the higher-order terms to construct methods of higher order would require new notation. However, it appear to be a good place to end our discussion of CERKs.

Bibliography

Barton, D., Willers, I., and Zahar, R. (1971). The automatic solution of systems of ordinary differential equations by the method of taylor series. *The Computer Journal*, 14(3):243.

Christlieb, A., Macdonald, C. B., and Ong, B. (2010). Parallel high-order integrators. *SIAM J. Sci. Comput.*, 32(2):818–835.

Corless, R., Ilie, S., and Reid, G. (2006). Computational complexity of numerical solution of polynomial systems. *Proc. Transgressive Computing*, pages 405–408.

Corless, R. M. and Fillion, N. (201x). A graduate survey of numerical methods.

Hairer, E., Nørsett, S. P., and Wanner, G. (1993). *Solving ordinary differential equations: Nonstiff problems.* Springer.

Higham, D. and Trefethen, L. (1993). Stiffness of ODEs. *BIT Numerical Mathematics*, 33(2):285–303.

Ilie, S., Söderlind, G., and Corless, R. (2008). Adaptivity and computational complexity in the numerical solution of ODEs. *Journal of Complexity*, 24(3):341–361.

Stetter, H. (1973). *Analysis of discretization methods for ordinary differential equations.* Springer.

Conclusion

This thesis has delineated a generally applicable perspective on numerical methods for scientific computation called residual-based *a posteriori* backward error analysis, based on the concepts of condition, backward error, and residual. The basic underpinning of this perspective, that a numerical method's errors should be analyzable in the same terms as physical and modelling errors, is readily understandable across scientific fields, and it thereby provides a view of mathematical tractability readily interpretable in the broader context of mathematical modelling. As a result, we maintain that this perspective arrives at the desideratum of integrating the study of numerical methods as a part of a more general practice of mathematical modelling as is found in applied mathematics and engineering

Based on the pioneering work of Turing and Wilkinson, and on the many works produced in the last decades, we have described backward error analysis at a level of generality seldom reached in the literature. In particular, we have provided a new general definition of the concept of residual that shows explicitly what is common to the applications of this concepts to all numerical methods, whether it is about floating-point arithmetic, numerical linear algebra, function evaluation and root finding, numerical solutions of differential equations, or other topics. As we went along, we have maintained a high standard of rigour regarding the distinction between the properties of problems and the properties of numerical methods. As a result, we obtain a better insight as to when aspects of numerical solutions are genuinely representative of the solution of a problem, and when it is induced by the choice of a particular method.

A shortcoming of the thesis, due to limitation of space, is that we have not effectively shown how the general method applies to all numerical methods, despite claiming it a number of times. The forthcoming book by Corless and Fillion (201x), however, does just that. In this thesis, the residual-based *a posteriori* backward error analysis perspective is applied mainly to numerical solution of differential equations, with some fragmentary applications to floating-point arithmetic. Chapter 2 has provided an introduction to the use of state-of-the-art codes without explaining the mechanics of the methods they implement. The idea was to allow the reader to be able to obtain numerical methods, and to examine properties of problems without paying attention to the particular details of the methods providing numerical results. As a matter of fact, measuring the error in a numerical solution of a differential equation *a posteriori* by the residual precisely disregards what particular method has been used to generate it. Following this line of thought, we have presented three different mathematical ways of characterizing the most important problem-specific property of initial-value problems, namely conditioning. We have shown how this allows us to understand the numerical solution of chaotic problems as being satisfactory in the backward sense. The second chapter has also introduced the problem-specific aspects of the numerical phenomenon known as stiffness in a way that show a duality with chaotic problems.

Our analysis of the properties of problems has been geared toward the use of the concept of residual across the board. This analysis, however, demands that one adopt the unusual perspective that numerical methods for the solution of differential equations produce continuous, even continuously differentiable, solutions and not merely a discrete set of solution values. Accordingly, we introduced method-specific concepts of error analysis such as convergence, consistency, order of a method, local and global error in terms of residual control. This habit of thought is possible nowadays because professional-quality solvers already provide access to accurate interpolants together with the discrete solution values. In order to concretely show that it is practically feasible to treat numerical methods as such, we have presented Taylor series methods and Runge-Kutta methods in a way that was, once again, based on residual

control. We also discussed, *en passant*, how the success of implicit methods for stiff problems can be enlightened by examining the size of the residual of Taylor polynomials for the approximation of exponential growth.

The main advantage—for a large class of initial-value problems for ODEs—of thinking continuously is that numerical methods can be seen to deliver an exact solution, and in many cases precisely as good a solution as an analytic exact solution of the reference problem, when physical perturbations of the original problem are taken into account. In other words, thinking continuously about solutions allow us to use the perspective provide backward error analysis. Consequently, we believe that the general perspective we have developed represents a notion of effective computation complementing in many respect the notion of effective computation developed by Turing that is used in metamathematics. Instead of being based of the notion of effective computability, it takes into account the modelling context to impose standards of *mathematical tractability*.

To conclude this thesis with good conscience, however, we have to add a dissonant note. We have devoted our effort to showing when we *can* use this type of backward error analysis fruitfully. Further work will be required in order to investigate when we can expect the perspective we promote to encounter serious problems. In this respect, it is important to bear in mind the words of Kahan that we cited before: A useful backward error-analysis is an explanation, not an excuse, for what may turn out to be an extremely incorrect result.

Bibliography

Corless, R. M. and Fillion, N. (201x). A graduate survey of numerical methods.

Appendix A

Floating-Point Arithmetic

James Gosling (1998), creator of Java, claimed that "95% of the folks out there are completely clueless about floating-point." Nonetheless, one distinctive feature of computer-assisted mathematics is that, instead of making computations that operate on elements of continuous fields (*e.g.*, the real or the complex numbers), one operates on a discrete, finite set of digital objects. The real numbers, for instance, can be exactly represented by infinite strings of digits, and the operations on them can be seen as acting on those infinite strings. However, computation on a digital computer is not making such operations, since only finite strings of digits are manipulated. Many nice number-theoretical properties—such as associativity of operations—are typically not satisfied in floating-point arithmetic. One sees, therefore, that it is a quite different type of mathematics; key concepts such as roundoff error, underflow, and overflow emerge when we switch to floating-point operations. As a result, floating-point arithmetic can be seen as an independent mathematical theory that explains how we can accurately represent and operate on real numbers with finite strings of digits. One of the main challenges is to guarantee that floating-point operations are correctly rendering results, where "correctly" is measured on the basis of standard arithmetic.

Many systems of digital arithmetic have been developed. This appendix will introduce you to the central concepts used to characterize a number system, and to the IEEE Standard 754, which is nowadays used by most computers. The reader interested in more details is invited to consult Goldberg

(1991), Ercegovac and Lang (2004) and Overton (2001).

A.1 Number Representation

A digital number system associates digital representations with numbers. To take a simple example, we can associate the number 'four' with the decimal representation "4," with the binary representation "100," or with the roman numeral "IV." Such digital representations are also called *numerals.* Thus, a digital number system is composed of a set of numbers N, a set of numerals W, and a representation mapping $\phi : N \to W$. If the association is one-one, we can similarly write $\phi^{-1} : W \to N$. In this case, we could write, *e.g.*, $\phi(\text{four}) = \text{IV}$ and $\phi^{-1}(\text{IV}) = \text{four}$. As we see, ϕ associates the number four—a uniquely identifiable element of a number structure (*e.g.*, a ring or a field)—with its digital representation in roman numerals.

Integer and rational representation A non-negative integer $x \in \mathbb{N}_0$ is represented by a *digit-vector*

$$[d_{n-1},\, d_{n-2},\, \ldots\, ,\, d_1,\, d_0]\,,$$

from which we obtain the more standard $\phi(x) = d_{n-1}d_{n-2}\ldots d_1 d_0$ by concatenating the elements of the digit-vector. The concatenated digit-vector is what we called a *numeral.* The number of digits n is called the *precision.* Each d_i belongs to a set D, the set of digits available for the representation (*e.g.*, $\{0, 1\}$ in the binary case). If D contains m elements, it will then only be possible to form $|W| = m^n$ distinct numerals. Since m and n are finite, $|W|$ is also finite. This is much less than the $\aleph_0$ integers or rationals that we want to represent! In fact, each numeral will be used to represent many numbers.

The cases we are interested with here are the so-called *weighted* or *positional* representations of *fixed radix.* If we let r be the radix, we associate a

number $x \in N$ with a numeral $w \in W$ by a mapping ϕ such that

$$\phi(x) = \phi\left(\sum_{i=0}^{n-1} d_i r^i\right) = d_{n-1}d_{n-2}\ldots d_1 d_0 = w. \tag{A.1}$$

The most familiar system—the decimal system—has $r = 10$ and $D = \{0, 1, 2, \ldots, 9\}$. For instance, in the decimal representation, we have

$$\phi(\text{nine hundred eighty-four}) = 9 \cdot 10^2 + 8 \cdot 10^1 + 4 \cdot 10^0 = 984.$$

Note that, for computer implementation, $r = 2$ and $D = \{0, 1\}$ is usually favored.

From the definitions above, it follows that the range of non-negative integers that can be exactly represented with a precision-n and radix-r number system is $[0, r^n - 1]$:

$$0 = \sum_{i=0}^{n-1} 0 \cdot r^i \leq x \leq \sum_{i=0}^{n-1} (r-1) \cdot r^i = r^n - 1$$

To represent both positive and negative integers—*i.e.*, to represent *signed* integers—we need a way to determine the sign of the integer represented by a given numeral. There are two main types of representations:

1. use a bit for the sign and the rest for the magnitude;
2. use all the bits for the magnitude and add a bias;

In the former case, we reserve a digit in the word to determine the sign. 0 usually represents '+,' while 1 represents '−.' Then, for an n-bit word, the range is

$$[-r^{n-1} + 1,\ r^{n-1} - 1]. \tag{A.2}$$

In the latter case, all the bits are determining the magnitude; the value represented is then the value it would represent under the standard positional

representation, minus a certain bias B. A standard bias for an n-bit radix-r representation is $B = r^{n-1} - 1$.[1] Then, for an n-bit word, the range is $[-r^{n-1} + 1,\ r^n - 1 - (r^{n-1} - 1)]$, *i.e.*,

$$[-r^{n-1} + 1,\ r^{n-1}(r-1)]. \tag{A.3}$$

In the binary case $r = 2$, it just results in $[-r^{n-1} + 1,\ r^{n-1}]$. In comparison to the sign-and-magnitude representation, we see that it provides us with one additional value.

Note that rational numbers can be written in the form

$$d_{n-1}d_{n-2}\ldots d_2 d_1 d_0 \,.\, d_{-1}d_{-2}\ldots d_{-f-2}d_{-f-1}d_f.$$

Hence, we see that it is simply a pair of words with respective precisions n and f. The first word is the *integer part*, and the second is the *fractional part*. Provided that the radix is the same for both words,

$$x = \sum_{i=-f}^{n-1} d_i \cdot r^i. \tag{A.4}$$

It is usually assumed that the integer part is a signed integer, whereas the fractional part is a positive integer.

Floating-point representation We now introduce the notion of *floating-point number*. A floating-point number is any real number that has an *exact* floating-point representation. Formally, if we let $\mathbb{F}$ be the set of floating-point numbers, W the set of floating-point words (to be defined), and $\phi : \mathbb{F} \to W$ some floating-point representation mapping (to be defined), we have

$$\mathbb{F} = \left\{ x \in \mathbb{R} \;\middle|\; \phi(x) = w \text{ for some } w \in W \right\}. \tag{A.5}$$

[1] We will always use this bias, unless otherwise specified.

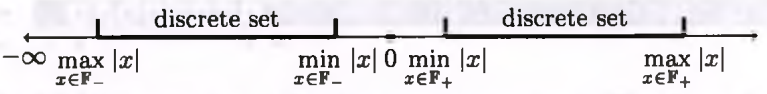


Figure A.1: Floating-point number line. The intervals $(-\infty, \max_{x\in\mathbb{F}_-}|x|)$ and $(\max_{x\in\mathbb{F}_+}|x|), \infty+)$ are called, respectively, negative and positive overflow. Similarly, the intervals $(\min_{x\in\mathbb{F}_-}|x|, 0)$ and $(0, \min_{x\in\mathbb{F}_+}|x|)$ are called negative and positive underflow.

Since there will be, once again, only a finite number of exactly representable numbers, $\mathbb{F}$ is finite. As a result, $\mathbb{F}$ has unique minimal and maximal elements, both for the positive and negative numbers. Consequently, we find that the floating-point number "line" can be represented as in figure A.1.

An n-bit floating-point representation has two components:

1. An m-bit word ($0 < m < n$) called *mantissa* or *significand*,[2] representing a signed rational number M with sign S_M (using a sign-and-magnitude representation);
2. An $n-m$-bit word called *exponent*, representing a signed integer E (using a biased representation).

The choice of type of representations for the signed integers M and E follows the IEEE standard. We will assume that the significand and the exponent have the same radix r. The corresponding floating-point number in a base-b system is then

$$M \times b^E. \tag{A.6}$$

The IEEE standard requires that M be normalized, *i.e.*, of the form $\pm 1.F$ where F is the fractional part. It is then not required to use a bit for the integer part (the "1" is said to be a *hidden bit*), and the m bits for the significand

[2] m is often used for the length of the unsigned significand. It is of course just a notational convention; one must simply keep track of all the $+1$ and -1 in the exponents to have agreeing results.

Figure A.2: 32-bit word for a floating-point number. The biased exponent occupies 8 bits and the significand occupies 24 bits (one for the sign, 23 for the fractional part or the normalized significand).

significand \ exp.	-11	-10	-01	±00	01	10	11
1.000	0.125	0.25	0.5	1	2	4	8
1.001	0.140625	0.28125	0.5625	1.125	2.25	4.5	9
1.010	0.15625	0.3125	0.625	1.25	2.5	5	10
1.011	0.171875	0.34375	0.6875	1.375	2.75	5.5	11
1.100	0.1875	0.375	0.75	1.5	3	6	12
1.101	0.203125	0.40625	0.8125	1.625	3.25	6.5	13
1.110	0.21875	0.4375	0.875	1.75	3.5	7	14
1.111	0.234375	0.46875	0.9325	1.875	3.75	7.5	15

Figure A.3: $\mathbb{F}(7,4,2,2)$: represented values when $n = 7, m = 4, r = 2, b = 2$. The words for the significand and the exponent are written in binary. For convenience, the values are in decimal. Only the positive values of the significand have been listed.

are used for the sign and the fractional part.[3] In the rest of this appendix, we will deal with such normalized numbers. In figure A.2, one can see the bits partition for a representation of a floating-point number with 32 bits.

Values represented As we have seen, the set of (normalized) floating-point representations W is determined by three numbers n, m, r. The set of floating point numbers $\mathbb{F}$ is, in turn, determined by the set of representations W and by a basis b. We can then write $\mathbb{F}(n, m, r, b)$. For a given number system $\mathbb{F}(n, m, r, b)$, one can list all the values that can be represented in a finite table, as in table A.3. The values computed in table A.3 are represented on

[3]We should note that restricting representation to normalized M makes it impossible to represent some very small numbers. The IEEE standard also defines *unnormalized* representations (also called denormals or subnormals) to deal with those numbers. However, we will ignore this refinement in this appendix.

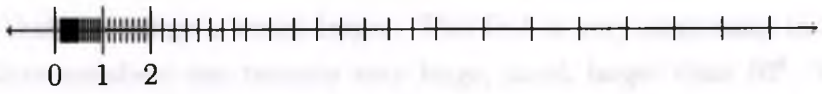


Figure A.4: The $\mathbb{F}(7,4,2,2)$ positive number "line".

the "$\mathbb{F}(7,4,2,2)$ number line" in figure A.4. One sees at a glance that the numbers are not uniformly distributed. In fact, they are much more densely distributed close to 0. As is easy to see, the distribution depends on n, m, b and r.

Range and machine epsilon The main advantage of floating-point numbers is their *range*, which is much larger than the range of fixed-point numbers. As we have seen, the largest number that can be represented by an n-bit radix-r word is $r^n - 1$, resulting in the range $[0, r^n - 1]$. However, the largest floating-point number in the set $\mathbb{F}(n, m, r, b)$ is $M_{\max} \cdot b^{E_{\max}}$, where $M_{\max}$ and $E_{\max}$ are, respectively, the largest significand and the largest exponent. In the normalized case, since $M_{\max} = 1.F_{\max}$, we obtain

$$\begin{aligned} M_{\max} &= 1.F_{\max} = 1 + \frac{r^{m-1} - 1}{r^{m-1}} \\ E_{\max} &= r^{n-m-1}(r-1) \\ \max \mathbb{F} &= \left(1 + \frac{r^{m-1} - 1}{r^{m-1}}\right) \cdot b^{r^{n-m-1}(r-1)} \end{aligned} \tag{A.7}$$

As an example, the largest 32-bit radix-2 fixed-point word is $2^{32} - 1 \approx 4 \cdot 10^9$. If we make it signed, so that it includes negative numbers, the largest one will be $2^{31} - 1 \approx 2 \cdot 10^9$. However, the largest base-2 floating-point number represented by 32-bit words partitioned as in figure A.2 will have significand $+1$ followed by twenty-three '1' for the fractional part and biased exponent with eight '1', *i.e.*,

$$\left(1 + \frac{2^{23} - 1}{2^{23}}\right) \cdot 2^{2^7} \approx 7 \cdot 10^{38}$$

We see that the range is much larger. This fact is very important in applications, since numbers can become very large, much larger than 10^9. Without extended range, computations would produce lots of overflows and underflows. This is why floating-points are preferred for scientific computing, despite the sacrifice in precision involved.

An important notion for the analysis of floating-point error is the *unit in the last place*, or *ulp*, which is the difference of two consecutive values of the significand. Since the values of the significand are uniformly distributed, *ulp* is a constant. For given values of m and r, we find that

$$ulp = \left(1 + \frac{r^{m-1} - x}{r^{m-1}}\right) - \left(1 + \frac{r^{m-1} - (x+1)}{r^{m-1}}\right) = r^{-m+1}. \tag{A.8}$$

Assuming $b = r$, it follows immediately that the difference between two floating-point numbers x_1 and x_2 with the same exponent E will be given by

$$\Delta x = x_1 - x_2 = (M_1 - M_2)r^E = r^{-m+1}r^E = r^{E-m+1}. \tag{A.9}$$

For normalized floating-point numbers between 1 and 2, where $E = 0$, we simply obtain *ulp*.

The spacing of floating-point numbers when $E = 0$ is called the *machine epsilon*, which is denoted 'ε_M.' Each number system has its value of ε_M, and they generally differ.

In the case of $\mathbb{F}(7, 4, 2, 2)$ discussed above, $\varepsilon_M = 2^{-4+1} = 0.125$.

An important related value is the maximum relative error due to the floating-point representation, called the *roundoff level*, which is just $\varepsilon_M/2$. We will denote this quantity by '$\varepsilon_M/2$' or 'μ_M' interchangeably. Since roundoff is the main source of arithmetic error, the machine epsilon will be used throughout this book as a unit of error.

The IEEE Standard 754 Some additional constraints are given by different floating-point representations. The current standard for floating-point arithmetic has been developed by the IEEE (Institute of Electrical and Electronics Engineers). The *single* precision format represents numbers with 32 bits, while the *double* precision format represents them with 64 bits. MATLAB uses the double-precision format by default, and so we will present this format in this section.

The number system associate with the IEEE standard 754 double precision format is basically just $\mathbb{F}(64, 53, 2, 2)$ with a few tweaks. The radix and the basis are both 2, which is the standard practice for binary arithmetic. The 64 bits are partitioned as follows:

- Significand: $m = 53$;
- Exponent: $n - m = 11$.

As mentioned before, the significand is a normalized signed integer with a sign-and-magnitude representation and the exponent is a signed integer with a biased representation. Thanks to the hidden bit, this format has precision $p = 53$. The range of the values represented by the significand is

$$\left[1,\ 1 + \frac{r^{m-1} - 1}{r^{m-1}}\right] = \left[1,\ 1 + \frac{2^{52} - 1}{2^{52}}\right] \approx [1, 2). \tag{A.10}$$

The bias of the exponent is the standard bias:

$$B = r^{n-m-1} - 1 = 2^{10} - 1 = 1023 \tag{A.11}$$

Consequently, the range of the values represented by the exponent is

$$[-r^{n-m-1} + 1,\ r^n - 1] = [-2^{10} + 1,\ 2^{10}] = [-1023,\ 1024]. \tag{A.12}$$

However, -1023 and 1024 are reserved to denoted negative and positive infinity, *i.e.*, `-Inf` and `Inf` in MATLAB. As a result, the range of the exponent is $[-1022, 1023]$.

Consequently, the range of the positive double-precision floating-point numbers is

$$\left[M_{\min} \cdot 2^{E_{\min}},\ M_{\max} \cdot 2^{E_{\max}}\right] = \left[2^{-1022},\ \left(1 + \frac{2^{52} - 1}{2^{52}}\right) \cdot 2^{1023}\right]$$
$$\approx \left[2.2 \cdot 10^{-308},\ 1.8 \cdot 10^{308}\right] \qquad \text{(A.13)}$$

MATLAB calls the limit points of this range `realmin` and `realmax`. Finally, the machine epsilon—MATLAB calls this value `eps`—is

$$\varepsilon_M = r^{-m+1} = 2^{-52} \approx 2.2 \cdot 10^{-16}. \qquad \text{(A.14)}$$

Finally, since zero has no direct representation in this system (due to normalization), the word with $M = 1.0$ and $E = 0$ is used to represent it.

A.2 Operations and Roundoff

The IEEE standard also defines the result of floating-point arithmetic operations (called *flops*). It is easy to understand the importance of having such standards! In the last section, the reader might have noticed that floating-point representation works just like the scientific notation of real numbers—in which numbers are written in the form $a \cdot 10^b$—to the exception that we mostly use base 2 and that both a and b have length restrictions. The same analogy will hold true for the four basic operations.

The four basic operations on the real numbers are functions $* : \mathbb{R}^2 \to \mathbb{R}$, with

$$* \in \{+, -, \times, /\}$$

In floating-point arithmetic, we use similar operations, but using floating-point numbers. The four basic operations on floating-point numbers are thus

functions $\circledast : \mathbb{F}^2 \to \mathbb{F}$, with

$$\circledast \in \{\oplus, \ominus, \otimes, \oslash\}$$

These operations have many different implementations. Given our expectation that floating-point operations return us results that are very close to what the real operations would return, the important question is: how do $\oplus, \ominus, \otimes, \oslash$ relate to their counterparts $+, -, \times, /$ in the real numbers? The best-case scenario would be the following: given a rounding procedure converting a real number into a floating-point number, the floating-point operations always return the floating-point number which is closest to the real value of the real operation. The good news is, if we are only interested with the impact of floating-point arithmetic in applications, there is no need to examine the detailed implementations of the floating-point operations. The IEEE standard guarantees that, for the four basic operations $\oplus, \ominus, \otimes, \oslash$, the best-case scenario obtains.

Let us formulate this more rigorously. A *rounding* operation $\square : \mathbb{R} \to \mathbb{F}$ is a procedure converting real numbers into floating-point numbers satisfying the following properties:

1. $\square x = x$ for all $x \in \mathbb{F}$;
2. $x \leq y \Rightarrow \square x \leq \square y$ for all $x, y \in \mathbb{R}$;
3. $\square(-x) = -\square x$ for all $x \in \mathbb{R}$.

There are many rounding operations satisfying this definition. In what follows, we will use the rounding to the nearest floating-point number (with ties towards $+\infty$), denoted '$\bigcirc$.' If we let f_1, f_2 be two consecutive floating-point

numbers and $x \in \mathbb{R}$ such that $f_1 \leq x \leq f_2$, then $\bigcirc$ is defined by[4]

$$\bigcirc x = \begin{cases} f_1 & \text{if } |x - f_1| < |x - f_2| \\ f_2 & \text{if } |x - f_1| \geq |x - f_2| \end{cases} . \tag{A.15}$$

Then the IEEE standard guarantees that the following equations holds:

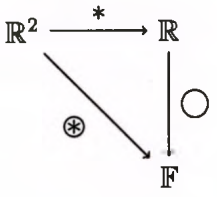
$$x \oplus y = \bigcirc(x + y) \tag{A.16}$$

$$x \ominus y = \bigcirc(x - y) \tag{A.17}$$

$$x \otimes y = \bigcirc(x \times y) \tag{A.18}$$

$$x \oslash y = \bigcirc(x/y) \tag{A.19}$$

i.e., for $* : \mathbb{R}^2 \to \mathbb{R}$ and $\bigcirc : \mathbb{R} \to \mathbb{F}$, we obtain $\circledast : \mathbb{R}^2 \to \mathbb{F}$ such that



These equations jointly mean that the result of a floating-point operation is the correctly rounded result of a real operation.

However, if things are so nice, why do we need error analysis? We need error analysis precisely because it is not always so nice for sequences of operations. For example,

$$((((x_1 \oplus x_2) \ominus x_3) \oplus x_4) \ominus x_5) = \bigcirc(x_1 + x_2 - x_3 + x_4 - x_5) \tag{A.20}$$

does not hold generally. So, the big question is: when are compound operations reliable? When no result of guaranteed validity exists, the error analysis must

[4]To consider the cases where x does not lie within the range of the floating-point number system, we need to specify that if $|\bigcirc x| > \max\{|y| : y \in \mathbb{F}\}$ or $0 < |\bigcirc x| < \min\{|y| : 0 \neq y \in \mathbb{F}\}$, the rounding procedure returns, respectively, overflow and underflow.

be left to the hands of the user. This leads us to chapter 1.

Bibliography

Ercegovac, M. D. and Lang, T. (2004). *Digital Arithmetic.* Elsevier Science.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, March.

Gosling, J. (1998). Extensions to java for numerical computing. Keynote address.

Overton, M. L. (2001). *Numerical computing with IEEE floating point arithmetic.* SIAM.

Appendix B

Asymptotic Series in Scientific Computation

Niels Henrik Abel (1802-1829) wrote

> The divergent series are the invention of the devil, and it is a shame to base on them any demonstration whatsoever. By using them, one may draw any conclusion he pleases and that is why these series have produced so many fallacies and paradoxes [...]. (cited in Hoffman, 1998, p. 218)

Nowadays, the attitude is different, and closer to what Heaviside meant when he said

> The series is divergent; therefore we may be able to do something with it. (cited in Hardy, 1949)

In fact, asymptotic series will be used a lot in this book, and we will often not care too much whether they converge. This is because, in many contexts, the first few terms contain all the numerical information one needs; there's no need to ponder on what happens in the tail end of the series.

The key to understanding asymptotic series is to realize that there are *two* limits to choose from, with a series. Suppose we have for example

$$f(z) = \sum_{k=0}^{N} \frac{f^{(k)}(a)}{k!}(z-a)^k \quad + \quad R_N(z)(z-a)^{N+1} \tag{B.1}$$

as the usual truncated Taylor series for $f(z)$ near $z = a$. We can take the first limit, $N \to \infty$, to get the familiar mathematical object of the *infinite series*. This only makes sense if the limit exists. [There is some freedom to alter the definition of limit that we use in this case; we do not pursue this here.] If that limit exists, we say the series is *convergent*. However, there is another limit to be considered here, which often leads to very useful results. Namely, do *not* let $N \to \infty$, but rather keep it fixed (perhaps even at $N = 1$ or $N = 2$). Instead, consider the limit as $z \to a$. Even if the series is divergent in the first sense, this second limit often gives enormously useful information, typically because $R_N(z)$ (as it is written above) is well-behaved near $z = a$, and so the term $(z-a)^{N+1}$ ensures that the remainder term vanishes more quickly than do the terms that are kept. The rest of this section explores that simple idea.

We often want to consider the behavior of a function $y(x)$ in the presence of some perturbations. Then, instead of studying the original function $y(x)$, we study the asymptotic behavior of a 2-parameter function $y(x, \varepsilon)$, where ε is considered "small".

An asymptotic expansion for the function $y(x, \varepsilon)$ has the form

$$y(x,\varepsilon) = y_0(x)\phi_0(\varepsilon) + y_1(x)\phi_1(\varepsilon) + y_2(x)\phi_2(\varepsilon) + \ldots = \sum_{k=0}^{\infty} y_k(x)\phi_k(\varepsilon), \tag{B.2}$$

where $\phi_k(\varepsilon)$ are referred to as *gauge functions*, *i.e.*, they are a sequence of functions $\{\phi_k(\varepsilon)\}$ such that, for all k,

$$\lim_{\varepsilon \to 0} \frac{\phi_{k+1}(\varepsilon)}{\phi_k(\varepsilon)} = 0. \tag{B.3}$$

The type of gauge function we will use the most often are the power of the perturbation ε, *i.e.*, $\phi_k(\varepsilon) = \varepsilon^k$, in which case we simply have a formal power series:

$$y(x,\varepsilon) = y_0(x) + y_1(x)\varepsilon + y_2(x)\varepsilon^2 + \ldots = \sum_{k=0}^{\infty} y_k(x)\varepsilon^k. \tag{B.4}$$

We then have to solve for the $y_k(x)$, $k = 0, 1, \ldots, N$. To find the first coefficient $y_0(x)$, divide equation (B.2) by $\phi_0(\varepsilon)$, and then take the limit as $\varepsilon \to 0$:

$$\frac{y(x,\varepsilon)}{\phi_0(\varepsilon)} = y_0(x) + \frac{1}{\phi_0(\varepsilon)} \sum_{k=1}^{\infty} y_k(x)\phi_k(\varepsilon) \tag{B.5}$$

$$\lim_{\varepsilon \to 0} \frac{y(x,\varepsilon)}{\phi_0(\varepsilon)} = y_0(x). \tag{B.6}$$

All the higher-order terms vanish since $\phi_k(\varepsilon)$ is a gauge function. This gives us $y_0(x)$. Now, subtract $y_0(x)\phi_0(\varepsilon)$ from both sides in equation (B.2); we then divide both sides by $\phi_1(\varepsilon)$ and take the limit as $\varepsilon \to 0$:

$$\frac{y(x,\varepsilon) - y_0(x)\phi_0(\varepsilon)}{\phi_1(\varepsilon)} = y_1(x) + \frac{1}{\phi_1(\varepsilon)} \sum_{k=2}^{\infty} y_k(x)\phi_k(\varepsilon) \tag{B.7}$$

so

$$\lim_{\varepsilon \to 0} \frac{y(x,\varepsilon) - y_0(x)\phi_0(\varepsilon)}{\phi_1(\varepsilon)} = y_1(x)\,. \tag{B.8}$$

As we see, we will in general have

$$y_k(x) = \lim_{\varepsilon \to 0} \frac{1}{\phi_k(\varepsilon)} \left(y(x,\varepsilon) - \sum_{\ell=0}^{k-1} y_\ell(x)\phi_\ell(\varepsilon) \right). \tag{B.9}$$

Convergence of a series is all about the tail, which requires an infinite amount of work. What we want instead is gauge functions that go to zero very fast, i.e., that the speed at which they go to zero is asymptotically faster from one term to the next.

Example 2. *Consider the (convergent) integral and the (divergent) asymptotic series*

$$\int_0^\infty \frac{e^{-t}}{1+xt} dt = \sum_{k=0}^{n} (-1)^k k! x^k + O(x^{n+1}) \,. \tag{B.10}$$

One can discover that series by replacing $1/(1+xt)$ *(exactly) with the finite sum* $1 - xt + x^2t^2 + \cdots (xt)^n + (-xt)^{n+1}/(1+xt)$*, giving*

$$\int_0^\infty \frac{e^{-t}}{1+xt} dt = \sum_{k=0}^{n} (-1)^k x^k \int_0^\infty t^k e^{-t}\, dt + (-1)^{n+1} x^{n+1} \int_0^\infty \frac{t^{k+1} e^{-t}}{1+xt}\, dt \tag{B.11}$$

which gives a perfectly definite meaning to each of the entries in the asymptotic series. Notice that the series diverges for any $x \neq 0$*, if we take the limit as* $n \to \infty$*. Nonetheless, taking (say)* $n = 5$ *allows us to evaluate the integral perfectly accurately for small enough* x*, say* $x = 0.03$*: summing the six terms gives* 0.9716545240 *whereas the exact value begins* 0.9716549596*, which differs by about* $5 \cdot 10^{-7}$*.*

That is, we have used a *divergent* series to give us a good approximation to the correct answer. Heaviside was right, and this often happens. The reason this works is that it is the limit as $x \to 0$ that is dominating, here: if we had wanted an accurate answer for $x = 10$, we would have been out of luck. Asymptotic series are extremely useful in numerical analysis. We will often be concerned with the asymptotic properties of the error as the *average mesh width* (call it h) goes to zero, for example, and methods will be designed to be accurate in that limit.

Bibliography

Hardy, G. (1949). *Divergent series.* Clarendon Press.

Hoffman, P. (1998). *The man who loved only numbers: The story of Paul Erdös and the search for mathematical truth.* Hyperion, New York.