A Mechanized Error Analysis Framework for End-to-End Verification of Numerical Programs

by

Mohit Kumar Tekriwal

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Aerospace Engineering)
in the University of Michigan
2023

Doctoral Committee:

Assistant Professor Jean-Baptiste Jeannin, Chair
Professor Andrew W. Appel
Professor Yves Bertot
Professor Karthik Duraisamy
Assistant Professor Manos Kapritsos

Mohit Kumar Tekriwal
tmohit@umich.edu
ORCID iD: 0000-0002-5103-2630

DEDICATION

To my parents, who have supported me and celebrated my achievements throughout this journey.

ACKNOWLEDGEMENTS

The PhD journey has been one of the most challenging and transformative period of my life. Not only did this journey help me grow as a researcher, but it helped me grow personally and has shaped my current personality. This journey would not have been possible without the support of many generous people.

Throughout my time in Michigan, I have had the privilege of being guided and supported by Prof. Jean-Baptiste Jeannin, my advisor. His unwavering support and guidance have been invaluable to me. Under his mentorship, I delved into the realm of formal methods, and he played a crucial role in helping me find my niche in the programming languages community. What truly sets Prof. Jeannin apart is his belief in empowering his students. He granted me the freedom to shape my own PhD projects and foster collaborations with esteemed experts in formal verification and numerical analysis. By treating me as a colleague, he created an environment where I could effectively express and communicate my research ideas, and his reassuring presence helped me get through multiple challenges with resilience.

Throughout my PhD program, I have been fortunate to have Prof. Karthik Duraisamy as my mentor. He has played a pivotal role in shaping my research journey, from the very beginning. Prof. Duraisamy has been instrumental in helping me define my initial PhD projects and has taught me the art of simplifying complex research problems while honing my ability to narrow down research ideas. In addition to his guidance in research, he has been invaluable in assisting me with important decisions such as coursework selection and internships, ensuring a well-rounded academic experience. During the challenging PhD qualification examination, his unwavering support was instrumental in my success. I am truly grateful for his continuous availability, providing valuable feedback and advice whenever I needed it.

I owe a great debt of gratitude to Prof. Yves Bertot, whose technical assistance and warm hospitality during my time at INRIA Sophia Antipolis have been invaluable. Without his continuous support and guidance in navigating the intricacies of the mathematical components library, a significant portion of my PhD research would not have been possible. My personal interactions with Prof. Bertot during my stay at INRIA were truly enlightening, as I had the privilege of learning extensively about formal verification and honing my critical thinking abilities. Moreover, it was a remarkable experience to engage with esteemed

members of the Coq community, including Laurent Théry, Enrico Tassi, Laurence Rideau, Maxime Dénès, and Benjamin Grégoire. Prof. Bertot and the vibrant Coq community have played an integral role in shaping my research journey, and I am grateful for the invaluable knowledge and interactions I have gained from this remarkable experience.

During the final stages of my PhD research, I had the privilege of collaborating closely with Prof. Andrew Appel, Prof. David Bindel, and Ariel Kellison, who have made significant contributions to the development of my work. Prof. Andrew Appel's invaluable insights and assistance with end-to-end correctness proofs have played a crucial role in defining the trajectory of my final PhD project. His meticulous attention to detail and ability to bridge the gap between my research approach and practical applications have greatly enhanced my critical thinking abilities. Prof. David Bindel has been instrumental in providing necessary expertise in numerical analysis. Working alongside Ariel has been an absolute pleasure. She has not only helped me unravel the intricacies of the VCFloat tool and floating-point arithmetic, but has also been a wonderful collaborator and friend. Ariel's insights and technical acumen have allowed me to experiment freely with my project ideas.

Thank you additionally to the other amazing researchers who influenced my research trajectory. These include Prof. Manos Kapritsos, who helped me explore beyond my thesis through a project on distributed controls, Prof. Eva Darulova, who hosted me for an internship in her lab at the Max Planck Insitute for Software Systems (MPI-SWS) and Dr. Heiko Becker, who has been an incredible mentor and friend during my internship at MPI-SWS. Heiko helped me learn the HOL4 theorem prover and introduced me to the verified compilers work. I would also like to thank Geoffrey Huelette, Samuel Pollard, Heidi Thornquist and Randall Lober, who hosted me for an internship at the Sandia National Laboratories.

I am truly indebted to my parents, Ramesh and Anita Tekriwal, and my brother, Rohit Tekriwal, who have been my constant support throughout my PhD journey. They have provided me with an opportunity to grow professionally and personally by helping me with my dream to pursue higher studies in the US, and were a constant source of emotional support.

Thank you to my friends Anshul Kamboj, Vishwas Goel, Kaushal Solanki, Ayush Agarwal, Arshdeep Singh Chahal for bearing with me through my tough times and celebrating my achievements. Michigan would not have been bearable without you guys. I would like to thank my colleagues in the MARVL group for providing me timely feedback on my research. I would also like to thank my friends in France: Swarn Priya, Basavesh Shivakumar, Jean-Christophe Léchenet, who made my stay in INRIA pleasurable and were a family to me during my brief stay in France. I would like to thank my sister, Deepika Tekriwal and my brother-in-law, Manish Agarwal, for hosting me during my visit to Europe and their

guidance through the final stages of my PhD.

Lastly, I would like to thank the National Science Foundation (NSF) grant CCF-2219997 for partially funding my PhD research and the Chateaubriand Fellowship program for providing me an opportunity to visit INRIA Sophia Antipolis.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF APPENDICES

ABSTRACT

The behavior of physical systems is usually modeled by differential equations. For instance, the aerodynamics of airplanes is modeled by the Navier–Stokes equation; problems of optimal control are modeled by the Ricatti differential equation; and the valuation of stock options is modeled by the Black–Scholes equation. Thus, differential equations are pervasive in almost every aspect of science and engineering, and being able to solve them precisely and accurately, but also while trusting that the solutions are accurate, is of utmost importance. Since many of these differential equations are mostly difficult or intractable to solve analytically, they are typically solved numerically. This leads to an accumulation of errors at each approximation step. In the past, this accumulation of errors has led to catastrophic consequences like the failure of the Patriot missile system due to floating-point errors; and the infamous multi-million dollar loss in the Vancouver stock exchange due to accumulation of floating-point errors. It is thus important that we analyze each approximation error rigorously and *formalize* conditions which could lead to unexpected divergent behaviors of numerical solvers.

In my thesis, I propose a mechanized error analysis framework, which treats errors from each approximation step modularly, in a formal setting like the Coq theorem prover. This framework connects a differential equation to the actual implementation of a linear solver for computing solutions to a differential equation. We show convergence of a finite-difference method, which is used to discretize the differential equation; compute an approximated solution to the discretized set of equations using stationary iterative methods, and prove its convergence formally in the field of reals. We then extend this analysis to a concrete implementation of a stationary iterative algorithm: *Jacobi iteration*, and prove correctness, accuracy and convergence of the implementation in the presence of *floating-point errors*. Our floating-point error analysis takes into account exceptional floating-point behaviors including overflow and underflow, and we prove the absence of overflow at each iteration by deriving concrete bounds on the input variables to the algorithm.

Some of the important contributions of this thesis include: the formalization of a generic statement about convergence for finite-difference methods – the *Lax–equivalence theorem*; the formalization of a generic statement about iterative convergence in the field of reals; the

formalization of properties of the l^2 and l^∞ matrix and vector norms; norm-wise forward error bounds for matrix-vector operations in floating-point arithmetic; and the demonstration of a modular approach to achieve program verification on the Jacobi iteration method. This thesis further proposes ways to extend our end-to-end verification framework beyond Jacobi iteration to any stationary iterative method, and proposes ways in which automation could be achieved, as future extensions of this work.

CHAPTER 1

Introduction

1.1 Background and motivation

The behavior of physical systems is usually modeled by differential equations. For instance, the aerodynamics of airplanes is modeled by the Navier–Stokes equation [9]; problems of optimal control are modeled by the Ricatti differential equation [116]; and the valuation of stock options is modeled by the Black–Scholes equation [137]. Thus, differential equations are pervasive in almost every aspect of science and engineering, and being able to solve them precisely and accurately, but also while trusting that the solutions are accurate, is of utmost importance.

Since many of these differential equations are difficult or intractable to solve analytically, they are typically solved numerically. The differential equations are discretized in a finite computational domain, and solved numerically, which leads to an accumulation of errors at each approximation step. These errors are commonly classified [124] into *acknowledged errors*, for which a set procedures exist to identify and possibly remove them, and *unacknowledged errors*, which are commonly computer programming error, as illustrated in Figure 1.1. When a differential equation is discretized in a finite computational domain (grid) using a choice of numerical methods like the finite difference method, finite element method, finite volume method etc., we incur the first layer of approximation error called the *discretization error*. The discretization error is defined as the distance between the numerical solution computed in a grid and the exact solution from the differential equation projected onto this grid. The set of discretized equations we thus obtain from this discretization, which is usually abstracted as a linear system of equations, is then solved for the unknown quantities using a linear solver. This linear solver usually relies on iterative methods to build a sequence of approximations of the "true numerical solution", which is ideally computed by taking an inverse of the coefficient matrix. These iterative methods are less expensive computationally than direct matrix inversion and therefore used in most practical settings, but incur another

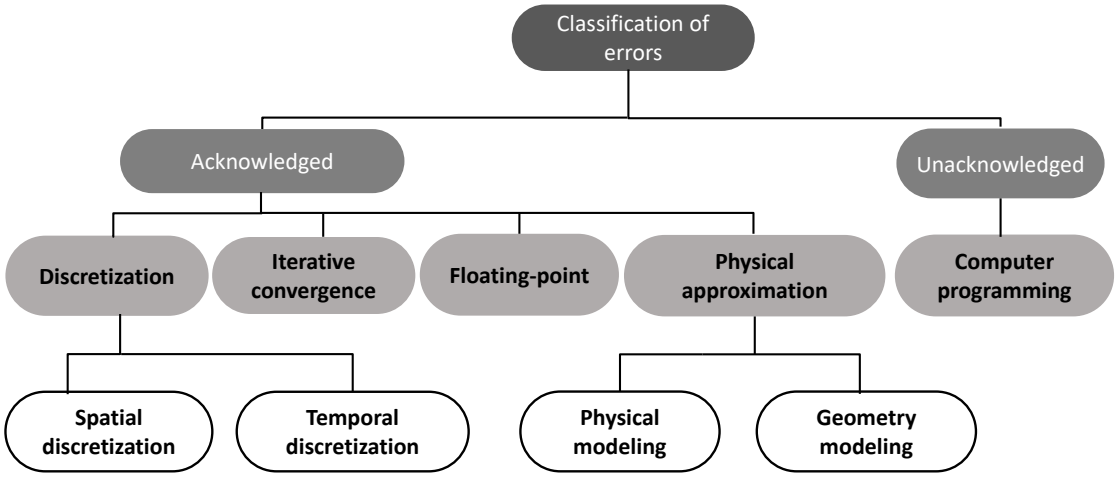

Figure 1.1: Classification of errors in scientific computing. Figure based on [124].

layer of approximation error called the *iterative convergence error*. Finally, since this linear solver is implemented using a programming language in a finite precision machine, the solutions from this solver is corrupted by the *floating-point error* and *computer programming error*. All of these sources of errors have to be analyzed rigorously and bounded carefully to guarantee that the numerical solution obtained from these linear solvers is bounded within a certain range of the exact solution of the differential equation.

The importance of a rigorous analysis of these errors can be motivated by some infamous disasters like the failure of the Patriot missile system [60] due to *floating-point errors*, and the infamous multi-million dollar loss in the Vancouver stock exchange [74] due to *accumulation of floating-point errors.* A case study of the devastating effect of the rounding errors is presented in the book by Nicholas Higham [68]. This book presents an example constructed by Hammarling and Wilkinson [61] for solving a linear system $Ax = b$ numerically using the successive over-relaxation (SOR) method, and the exact solution for this linear system is given by $x_i = 1 - (-2/3)^i$. The coefficient matrix A is a 100×100 lower bidiagonal matrix with $A_{ii} = 1.5$ and $A_{i,i-1} = 1$, and $b_i = 2.5$. The relaxation parameter for this method, $\omega = 0.5$. The SOR method converges in exact arithmetic since the spectral radius of the iteration matrix is $1/2$, but diverges in the presence of rounding error, as illustrated in the Figure 1.2. The forward error, defined as $||\widehat{x_k} - x||_\infty / ||x||_\infty$, levels off after about 100 iterations instead of converging asymptotically to 0, as would be the case in exact arithmetic. Here, $\widehat{x_k}$ denotes the numerical solution, x denoted the exact solution, and $||.||_\infty$ denotes the

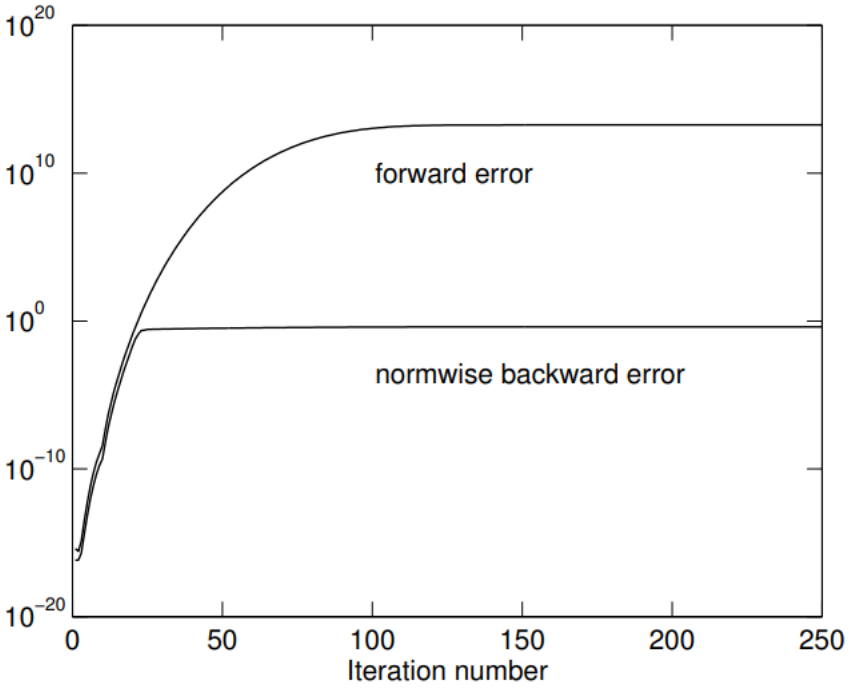

Figure 1.2: Effect of rounding error on convergence of the SOR method. Figure from [68]

ℓ^∞ norm of a vector. This divergence was accounted not because of the ill-conditioning of the matrix A, but due to the rapid growth error resulting from the iteration matrix being far from normal leading the norms of its power to become very large before they ultimately decay by a factor $\approx 1/2$ with each successive power. This subtle issue in theory could turn out to be catastrophic in practice if this method were used deep inside some linear solver used in engineering design. It is thus important that we analyze each approximation error rigorously and *formalize* conditions which could lead to unexpected divergent behaviors of numerical solvers.

The above motivating example raises an important question about the *credibility* of computational results. William L. Oberkampf and Christopher J. Roy define the credibility of a computational result as the result of an analysis being worthy of belief or confidence [108]. They further enumerate fundamental building blocks that build credibility in computational results, which includes: (a) quality of analysts conducting the work, (b) quality of physical modeling, (c) verification and validation practices, and (d) uncertainty quantification and sensitivity analysis. An analyst is mostly concerned with formulating a mathematical model of a physical system, choosing the right discretization scheme, setting up simulations, and analyzing the results of the simulation. The quality of a physical model is judged by the

fidelity and comprehensiveness of the physical detail embodied in the mathematical model representing a physical system. Both (a) and (b) mainly touch upon the modeling aspects of a physical system, which is not the primary interest of this work. We also do not deal with uncertainty quantification (d) in this work, since we only analyze deterministic systems. We are primarily concerned with the verification and validation aspects (c) of a computational result, assuming that we are given the right mathematical model by numerical analysts. Verification and Validation practices are primarily concerned with assessing and quantifying the accuracy of computational results. *Verification* is the process of assessing the correctness and numerical accuracy of the solution to a given mathematical model, while *Validation* is the process of assessing the physical accuracy of a mathematical model based on comparisons between the computational results and experimental data [108]. In other words, verification is about solving the equations right, while validation is about solving the right equation. In this work, we will be mostly discussing about what it means to solve the equation right, i.e., *verification*. The meaning of verification is fuzzy among the practitioners, and most often verification is assumed synonymous to testing. Most verification practitioners use the testing approach to observe the relation between the inputs and output of their algorithmic implementation. This approach however suffers from a major drawback of *state space explosion*, which means that one needs to generate a large number of input vectors to capture the entire behavior of the system and depends on the number of state variables, thereby leading to memory issues in most practical problems. Thus, testing is as good as the input vectors generated and often exposes shallow bugs in the implementation. This issue is well captured by a famous quote by an eminent Dutch computer scientist Edsger Wybe Dijkstra, *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [107]. This issue is alleviated by *formal verification*, which provides a way to identify functional errors in human-engineered designs and proves the absence of bugs. In formal verification, one defines a formal specification for the program implementation, which is a mathematical formula and captures the entire testing space, thereby capturing all the corner cases, which might not be caught by testing. The algorithmic or program implementation is then verified against this formal specification to achieve an end-to-end proof of correctness of the algorithm. *In this work, we will be using the approach of formal verification to develop proof of correctness for numerical programs.*

There are thus two crucial parts in this approach of formal verification – *developing formal specification*, and *developing formal proofs*. Formal specifications are developed using a *logic language* like *ACSL C [10], Linear Temporal Logic (LTL) [36], Gallina [34], Higher Order Logic [58], etc.* The correctness of a program with respect to this formal specification is then evaluated using a program logic, which generates verification goals. These verifica-

tion goals are then discharged with a proof of correctness or a counterexample if no proof is found, either using *automated theorem provers* like Z3 [49], CVC4 [8], Alt-Ergo [40], or *interactive theorem provers* like Coq [34], Isabelle/HOL [58], PVS [110], etc. Automated theorem provers use constraint solvers like SAT solvers for solving the *Boolean satisfiability problem*, or SMT solvers which uses *satisfiability modulo theories* to solve more complex formulas involving real numbers, integers, and/or various data structures like lists, arrays, bit vectors and strings. Interactive theorem provers on the other hand allow users to develop proof scripts interactively by relying on a much more comprehensive theory of domain specific problems, as compared to the automated theorem provers. In our work, we will be using the *Gallina specification language* to develop formal specifications and the *Coq proof assistant* to develop proof scripts interactively. We use the Coq theorem prover over other interactive theorem provers because Coq provides a rich formalization of linear algebra, real analysis and the IEEE-754 floating-point standard, some of which are lacking in other theorem provers. Coq also interacts well with some of the automated tools for program verification like Frama-C [43],the Verifiable Software Toolchain (VST) [33], and semi-automated floating-point analysis tools like VCFloat [114] and VCFloat2 [7]. This interaction with these tools has been important to us in our development of floating-point and program correctness proofs. We do not use the automated theorem provers in our work because of the lack of expressiveness in specifying properties like limits and statement of convergence; underdeveloped libraries on real analysis and functional analysis; and lack of comprehensive formal treatment of the IEEE-754 floating-point standards, even though there has been a recent thrust in this regard in the automated theorem proving community [118].

A number of works have recently emerged in the area of formalization of numerical analysis. This has been facilitated by advancements in automated and interactive theorem proving [109, 28, 54, 103]. Some notable works in the formalization of numerical analysis include the formalization of the Kantorovich theorem by Ioana Pasca [112] to prove convergence properties of the Newton method, the formalization of the matrix canonical forms by Cano et al. [32], and the formalization of the Perron-Frobenius theorem in Isabelle/HOL [133] for determining the growth rate of A^n for small matrices A. In terms of formal analysis for ordinary differential equation (ODE) solvers, Boldo and her colleagues have made significant contributions. Boldo et al [21, 23, 22] proved consistency, stability and convergence of a second-order centered scheme for the wave equation, connecting these formal analysis to the actual implementation of the wave equation in C programming language using the Frama-C tool. Louise Ben Salem-Knapp, Sylvie Boldo and William Weens formalized [122] round-off error bounds for the 1-D and 2-D upwind advection schemes, by taking into account the exceptional behaviors of floating-points. Sylvie Boldo, Florian

Faissole and Alexandre Chapoutot formalized [25] round-off error bounds for the Runge-Kutta (RK4) method, taking into account exceptional behaviors of floating-points, such as overflow and underflow. Besides Coq, numerical analysis of ODEs have also been done in Isabelle/HOL. Fabian Immler and Johannes Hölzl formalize [78] the initial value problem of ODEs and prove the existence of a unique solution using the Picard–Lindelöf theorem. Immler et al. [77, 80, 81] formalized flows, Poincaré map of dynamical systems, and verified rigorous bounds on numerical algorithms in Isabelle/HOL. Immler also formalized [76] a functional algorithm that computes enclosures of solutions of ODEs in Isabelle/HOL.

1.2 Dissertation contributions

Despite several advances in previous work, the existing formalizations of error analysis of ordinary differential equation (ODE) solvers do not address the issue of *iterative convergence*. Iterative methods are extensively used in the ODE solvers to obtain numerical solutions of physical systems. Our work therefore addresses this gap in the current state of the art in the formalization of numerical analysis, by providing a formalization of asymptotic convergence of iterative convergence errors for a general class of iterative methods – *stationary iterative methods* [131]. We further develop a generalized framework for forward error analysis in presence of floating-point errors for stationary iterative methods, and perform an end-to-end verification of the *Jacobi stationary iteration method* by proving accuracy, convergence and correctness of its actual implementation in the C programming language.

Futhermore, we demonstrated an approach to discretize an ordinary differential equation in a finite computational domain, *in a formal setting*. We chose a finite difference method for this discretization, and formalized conditions under which this discretization error converges to zero asympotically, i.e., *convergence* of the solution from a finite-diiference method to the *true solution* of a differential equation. While Boldo et al. proved convergence for a particular finite difference scheme, we formalized [130] a generic convergence statement – the *Lax–equivalence theorem* [94] for a class of finite-difference schemes. This statement can be instantiated with any finite-difference scheme of choice, and we applied the formalization of the Lax–equivalence theorem to show convergence of a centered finite-difference scheme. Thus, my thesis proposes an approach for end-to-end verification for numerical programs, keeping the underlying error-analysis as generic as possible, thereby making important contributions to the Coq and numerical analysis community.

Besides the main projects for my thesis, I have also contributed to the development of a tool called *Dandelion* [13], which generates a certified polynomial approximation of transcendental functions. This tool takes as input a Scala program containing transcendental

functions like $\sin, \cos, \tan$ etc. and replaces its every occurrence with a polynomial approximation generated by the tool *Sollya* [35], which also generates a maximum error bound for approximation in a given interval. The transcendental function, its polynomial approximation, and the maximum error bound forms a certificate, whose soundness is verified using a library for transcendental function approximation in the HOL4 theorem prover, *of which I was the prime developer*. By connecting this certificate verification process with the verified binary generation using the CakeML compiler [93], we develop an end-to-end verification framework for transcendental function approximation. I have also contributed in the formalization of sufficiency and necessary conditions for a set of normal nodes to achieve asymptotic consensus in a control network [132]. This formalization takes into account malicious attacks by adversarial agents for a particular attacker model, and is the first known mechanized proof in the area of distributed controls, to our knowledge.

1.3 Thesis structure

This thesis is structured as follows. In Chapter 2, we discuss the *discretization error*, which arises due to discretization of a continuous system in a finite computational domain . We formalize the *Lax–equivalence theorem* [94], to prove asymptotic convergence of this error for a class of finite difference schemes. We then specialize this formalization to a centered finite-difference scheme to prove *convergence* of the numerical solution x obtained from this scheme, to the *true analytical solution*. In Chapter 3, we compute the numerical solution x using *stationary iterative methods* [119]. We obtain a sequence of iterative solutions, $\{x_k\}$, which are approximations of x. The distance between an iterative solution x_k and x is called the *iterative convergence error* at step k. We formalize the sufficient and necessary conditions for asymptotic convergence of the sequence $\{x_k\}$ to x. We then instantiate this iterative convergence theorem to two classical stationary iterative methods – the *Gauss–Seidel method* and the *Jacobi method* to prove convergence on a model problem. We also formalize an easily verifiable condition for convergence of solutions from the Gauss–Seidel iteration, which just relies on the structure of the original coefficient matrix, instead of explicitly computing eigenvalues of the iteration matrix obtained from this coefficient matrix. The error analysis in this chapter is done in the field of reals, but the actual implementation of an iterative algorithm is done in a finite precision machine. We therefore formalize the effect of finite precision on iterative convergence in Chapter 4. In chapter 4, we define a *floating-point functional model* to compute a floating-point solution for the Jacobi iteration process. We then formalize conditions such that the *residual* $(b - Ax_k)$ computed for this iteration algorithm reaches below a user-specified tolerance τ, within k iterations, for given inputs

- the *coefficient matrix* A and the *right hand side vector* b. We formalize explicit bounds on the inputs such that no overflow occurs at each iteration step, and connect this to an actual implementation of the Jacobi iteration algorithm, implemented in the C programming language, to get a proof of accuracy, convergence and correctness of this iteration algorithm.

CHAPTER 2

Spatial Discretization Error and Lax–equivalence Theorem

2.1 Introduction

Physical systems are usually modeled mathematically using *differential equations*. For example, the aerodynamics of an airplane is modeled by the Navier–Stokes equation [9]; the Riccatti differential equation [116] is used in problems of optimal control; and the Black–Scholes equation [137] is used to model valuation of stock options. Thus, differential equations are pervasive in almost every aspect of science and engineering, and being able to solve them precisely and accurately, but also while trusting that the solutions are accurate, is of utmost importance.

Since *analytical or true* solution of differential equations is intractable for most practical problems of interests, these differential equations are solved numerically in a finite computational domain. In this process, a continuous problem (differential equation) is discretized to obtain a set of discretized equations, which is solved numerically to obtain an approximation of the *true solution*. This approximation error is called the *discretization error*. Since discretization can be done both in time and space, the discretization error is of two kinds– *temporal discretization error* and *spatial discretization error*. A numerical approximation is said to be good if this discretization error decreases as one refines the grid or decreases the discretization step size. This idea is captured by the notion of *convergence* of a numerical scheme. A numerical scheme is said to be *convergent* if the *global discretization error* approaches zero in the limit of infinitesimal discretization. Under these conditions, the numerical solution converges to or approaches the analytical solution. This idea is formally articulated by the Lax–equivalence theorem [94], which states that if a numerical method is *consistent* and *stable*, then it is *convergent*. We will discuss formal definitions of consistency, stability and convergence, and formally state the Lax–equivalence theorem in Section 2.3.1.

Proofs of consistency, stability, and convergence are typically performed by hand, making them prone to possible errors. Formal verification of mathematical proofs provides a much higher level of confidence of the correctness of proofs. Further, formal verification offers a pathway to leverage mathematical constructs therein, and to extend these proofs to more complex scenarios. Since the Lax–equivalence theorem is an essential tool in the analysis of numerical schemes using finite differences, its formalization in the general case opens the door to the formalization and certification of finite difference-based numerical software. The present work will enable the formalization of convergence properties for a large class of finite difference numerical schemes, thereby providing formal proofs of convergence properties usually proved by hand, making explicit the underlying assumptions, and increasing the level of confidence in these proofs.

Contributions: Overall this work makes the following contributions:

- We provide a formalization in the Coq proof assistant of a general form of the Lax–equivalence theorem.
- We prove consistency and stability of a second order accurate finite difference scheme for a model problem – differential equation $\frac{d^2u}{dx^2} = 1$.
- We formally apply the Lax equivalence theorem on this finite difference scheme for our model problem, thereby formally proving convergence for this scheme.
- We also provide a generalized framework for a symmetric tri-diagonal (sparse) matrix in Coq. We define its eigen system and provide an explicit formulation of its inverse in Coq. We show that since the symmteric tri-diagonal matrix is normal, one can perform the stability analysis by just uniformly bounding the eigen values of the inverse. This is important because discretizations of mathematical model of physical systems are usually sparse [90].

This work was published at the 13^{th} *NASA formal methods symposium, 2021* [130] [1]. An important point to note here is that of the two kinds of discretization error, we only deal with the spatial discretization error for our model problem.

2.2 Overview of the Coq proof assistant

Coq [34] is a formal proof management system, which provides a formal language to write specifications of a system and prove properties about the system, interactively.

[1] Our Coq formalization is available at https://github.com/mohittkr/Lax_equivalence.git

2.2.1 Coq standard real library

The Coq standard real library is a general purpose library that contains various developments and axiomatizations of real numbers. This library defines ordering of real numbers, inequalities, maximum, minimum and absolute functions over reals, trigonometic functions, logarithms and powers of real numbers. This library also formalizes some results from analysis like Rolle's theorem, Mean-value theorem, Weierstrass theorem etc.

2.2.2 Coquelicot analysis library

Coquelicot [28] is a library in Coq which defines theory for real analysis like the derivatives, integrals, limits, power series etc. Even though Coquelicot relies on the axiomatic definition of real numbers in the standard real library, it differs from the standard library in that Coquelicot relies on *total functions* to define limits, derivatives, integrals etc. as opposed to their definition using *dependent types* in the standard real library in Coq. Thus, the formalization of real analysis in Coquelicot is intuitive and easy to use as compared to the standard real library . We use the following theory from Coquelicot throughout our formalization:

- **Limits:** We use limits to prove asymptotic convergence of a class of finite-difference schemes in our formalization of the Lax–equivalence in this chapter, and for our proof of *iterative convergence* in the next chapter. Coquelicot defines limits using *filters* [28]. Intuitively, a filter in a partially ordered set P is a subset of P that includes as members the elements which are large enough to satisfy some criterion. Since filters can be interpreted as sets, operations on filters like such as composition, addition, maps, etc. can be viewed as set operations. Therefore, filters provide a nice abstraction to factor numerous proofs that satisfy a given criterion. For instance, the sum of limits of two sequences, and sum of limits of two functions are all instances of same proof about addition, and their proofs can be abstracted using a proof about filters for addition [28].

  Coquelicot defines the limit of a real-valued function $f : \mathbb{R} \to \mathbb{R}$ using the is_lim predicate which is defined in Coq as

  ```
  Definition is_lim (f : R → R) (x l : Rbar) :=
    filterlim f (Rbar_locally' x) (Rbar_locally l).
  ```

  where the filter Rbar_locally x defines an open neighborhood of x in the extended real line: $\mathbb{R} \cup +\infty \cup -\infty$. filterlim maps every point in the open neighborhood of x to a set

which is contained in the open neighborhood of l. i.e.,

$$filterlim(f, F, G) := \ \forall P,\ P \in G \implies f^{-1}(P) \in F$$

Coquelicot also defines the $\epsilon - \delta$ definition of limits (is_lim'), and an equivalence between is_lim and is_lim' to switch between the *filter* definition and the $\epsilon - \delta$ definition of limits at our convenience.

- **Algebraic structures:** Coquelicot also defines algebraic structures like ring, field, abelian group, metric space, module space, normed modules, etc., which we use heavily in our proof of the Lax–equivalence theorem in this chapter. This library also defines iterated sums sum_n_m, and basic matrix operations, which we use in the proof of convergence of a finite difference scheme in this chapter. All of these formalization can be found in the file Coquelicot.Hierarchy.v file.

- **Extended reals and bounds:** Coquelicot defines an extended real set (Rbar): $\mathbb{R} \cup +\infty \cup -\infty$, and operations on Rbar in the file Coquelicot.Rbar.v file.
  Coquelicot also defines least upper bound on Rbar (Lub_Rbar) and the greatest lower bound on Rbar (Glb_Rbar) in the file Coquelicot.Lub.v file.

Even though we discussed only a few theory from the Coquelicot library, more theories on real analysis can be referred to in this library [28].

2.2.3 Lax–Milgram formalization

The Lax–Milgram formalization [19] provided us with necessary tools to define operator norms, linear mapping, normed vector spaces etc. This work by Boldo et al, essentially proved the existence and uniqueness of the solution of a continuous problem and of its discretized counterpart with the aim of proving correctness of a finite-element method. This work also formalized important results from geometry, linear algebra, functional analysis and Hilbert spaces. Some of these results that we used in our formalization are as follows

- Properties about continuous linear maps and operator norms can be found in the file continuous_linear_map.v in the formalization [20, 19]. It is noteworthy to discuss their formalization of an *operator norm*. An operator norm is defined formally in this file as

```
Definition operator_norm (f:E → F) : Rbar :=
match ls_only_zero_set_dec E with
| left _ ⇒ Lub_Rbar (fun x ⇒ ∃ u:E, u <> zero ∧
                x = norm (f u) / norm u)
```

```
| right _ ⇒ 0
end.
```

corresponding to the mathematical definition

$$||f||_\phi = \sup_{u \neq 0_E \wedge \phi(u)} \frac{||f(u)||_F}{||u||_E} \tag{2.1}$$

f is a mapping from a *Normed module* E to another *Normed module* F, and $\phi : E \to Prop$ is such that the operator norm of f denoted by $||f||_\phi$ is defined on a subset of ϕ. A module is a structure of an abelian group with respect to addition over a ring, while a vector space is the same but with respect to a field. Since every field is a ring, a vector space is a module. Thus, a module is a generalization of a vector space. A Normed module is a module equipped with norms. Therefore, in the process of defining operator_norm over a normed module, Boldo et al have also defined operator norm over a normed vector space. The definition operator_norm states that if ϕ is $\{0_E\}$, i.e., ϕ is empty, we do not expect a *supremum*. In this case, the default value of an operator norm is set to 0, since the operator norm is defined as a *total function* in Coq. However, when ϕ is not empty, the definition of operator_norm corresponds to 2.1. The supremum is defined in Coquelicot using Lub_Rbar, where Lub stands for the least upper bound. An important point to note here is that the supremum is defined in the extended real line Rbar, which includes $+\infty$ and $-\infty$.

- Properties and definition of a linear map are defined in the linear_map.v file in [20, 19]. This linear map is defined from a *Module space* E to another *Module space* F.

We use all of the above infrastructure provided by Coq to develop our proofs for the Lax–equivalence theorem and proof of convergence of a finite-difference scheme.

2.3 Proof the Lax–equivalence theorem

2.3.1 Consistency, stability, and convergence

Definition 1 (The Continuous Problem [123])**.** Let $\mathbf{X}$ (the space of solutions) and $\mathbf{Y}$ (the space of data) be normed spaces, both real or both complex. We consider a linear operator $\mathcal{A}$ with domain $\mathbf{D} \subset \mathbf{X}$ and range $\mathbf{R} \subset \mathbf{Y}$. The problem to be solved is of the form

$$\mathcal{A}u = f, \qquad f \in \mathbf{Y} \tag{2.2}$$

Here $\mathcal{A}$ is not assumed to be bounded, so that unbounded differential operators are included. The problem (2.2) is assumed to be well-posed, i.e., there exists a *bounded, linear operator*, $\mathcal{E} \in B(\mathbf{Y}, \mathbf{X})$, such that $\mathcal{E}\mathcal{A} = I$ in $\mathbf{D}$, and that for $f \in \mathbf{Y}$, equation (2.2) has a unique solution, $u = \mathcal{E}f$. Furthermore, the solution u depends continuously on the data.

Definition 2 (The Approximate Problem [123])**.** Let H be a set of positive numbers such that 0 is the unique limit point of H. For each $h \in H$, let X_h, Y_h be normed spaces and consider the approximate or discretized problem

$$A_h u_h = f_h, \qquad f_h \in Y_h \tag{2.3}$$

where A_h is a linear operator $A_h : X_h \longrightarrow Y_h$.

We assume that for each $h \in H$, problem (2.3) is well-posed and there exists a solution operator, $E_h = A_h^{-1}$, i.e. $u_h = E_h f_h$. The true solution u and the approximate solution u_h can be related with each other by defining a *bounded, linear operator*, $r_h : \mathbf{X} \to X_h$ for each $h \in H$. Similarly, data $f \in Y$ can be related to data in a discrete space, $f_h \in Y_h$ by defining a restriction operator s_h. For each $h \in H$, $s_h : \mathbf{Y} \to Y_h$ is also a *bounded, linear operator*. We assume that the operator norms can be *uniformly* bounded:

$$||r_h|| \leq C_1, \qquad ||s_h|| \leq C_2, \tag{2.4}$$

where the constants C_1, C_2 are independent of h. The true solution $u = \mathcal{E}f$ is compared with the discrete solution $u_h = E_h s_h f$ corresponding to the discretized datum f. The family $(X_h, Y_h, A_h, r_h, s_h)$ defines a *method* for the solution of (2.2) [123].

Definition 3 (Convergence [123])**.** Let f be a given element in $\mathbf{Y}$. The method $(X_h, Y_h, A_h, r_h, s_h)$ is convergent for the problem (2.2) if

$$\lim_{h \to 0} ||r_h \mathcal{E} f - E_h s_h f||_{X_h} = 0 \tag{2.5}$$

We say that the method is convergent if it is convergent for each problem (2.2) for any f in $\mathbf{Y}$.

Intuitively, this means that in the limit of the discretization step, h, tending to zero, the numerical solution $E_h s_h f$ approaches the analytical solution $r_h \mathcal{E} f$. The analytical solution $r_h \mathcal{E} f$ is the restriction of the true (analytical) solution, $u = \mathcal{E}f$, onto the grid of size $N = 1/h$, and $E_h s_h f$ is the discrete solution, $u_h = E_h f_h$ computed on the grid of size N.

Definition 4 (Consistency [123])**.** Let u be a given element in $\mathbf{D}$. The method is consistent at u if

$$\lim_{h \to 0} ||A_h r_h u - s_h \mathcal{A} u||_{Y_h} = 0 \tag{2.6}$$

A method is consistent if it is consistent at each u in a set D_o such that the image $\mathcal{A}(D_o)$ is dense in $\mathbf{Y}$.

Intuitively, this means that in the limit of the discretization step, h, tending to zero, the finite difference scheme $A_h u_h = f_h$ approaches the differential equation $\mathcal{A}u = f$, i.e., we are discretizing the right differential equation.

Definition 5 (Stability [123])**.** The method is stable if there exists a constant K such that

$$||E_h||_{B(Y_h,X_h)} \leq K \tag{2.7}$$

Intuitively, stability of the numerical scheme means that a small numerical perturbation does not allow the solution to blow up. Uniform boundedness of the inverse $E_h = A_h^{-1}$ is a check on the conditioning of matrices (sensitivity to small perturbations), i.e., it ensures that the matrix A_h is not ill-conditioned. Thus, if the numerical problem (2.3) were unstable, even though we were trying to solve the right differential equation, we would never converge to the true solution. Hence, both stability and consistency are sufficient for proving convergence of the numerical scheme.

The quantities within the norms (2.5) and (2.6) are, respectively, the *global* and *local* discretization errors.

2.3.2 Coq formalization

The Lax–equivalence theorem, which is a statement of convergence for a family of finite-difference schemes is stated as

Theorem 6 (Lax–equivalence theorem [123])**.** *Let* $(X, Y, A, X_h, Y_h, A_h, r_h, s_h)$ *be as above. If the method is consistent and stable, then it is convergent.*

Proof. We start with the definition of *convergence* in (2.5),

$$
\begin{aligned}
&\lim_{h\to 0} ||r_h\mathcal{E}f - E_h s_h f||_{X_h} \\
&= \lim_{h\to 0} ||r_h u - E_h s_h f||_{X_h} \quad (u \triangleq \mathcal{E}f) \\
&= \lim_{h\to 0} ||r_h u - E_h s_h A u||_{X_h} \quad (f \triangleq \mathcal{A}u) \\
&= \lim_{h\to 0} ||I r_h u - E_h s_h \mathcal{A}u||_{X_h} \quad (r_h u = I r_h u) \\
&= \lim_{h\to 0} ||E_h A_h r_h u - E_h s_h \mathcal{A}u||_{X_h} \quad (E_h A_h \triangleq I) \\
&\leq \lim_{h\to 0} ||E_h||_{B(Y_h,X_h)} ||(A_h r_h u - s_h \mathcal{A}u)||_{Y_h} \\
&\leq K \lim_{h\to 0} ||(A_h r_h u - s_h \mathcal{A}u)||_{Y_h} \quad \text{(From stability: (2.7))} \\
&= 0 \quad \text{(From Consistency: (2.6))}
\end{aligned}
$$

□

We use the mathematical structures from the `Coquelicot` library [28] for implementing the proof. *Since we use the Coquelicot and standard reals libraries which are based on classical axiomatization of reals [28], our proofs are also non-constructive.* Since s_h and r_h have to be bounded linear operator, we define a bounded linear operators in Coq as

```
Definition is_bounded_linear (E F: CompleteNormedModule R_AbsRing) (phi:E→ F):=
    is_linear_mapping E F phi ∧ (∃ K:R, 0<=K ∧ (∀ x:E, norm(phi x) <= K* norm x)).
```

where we have used the predicate `is_linear_mapping` from the Lax–Milgram formalization [19] to assert a linear map from E to F. Since the spaces ($\mathbf{X}$, $\mathbf{Y}$, X_h, Y_h) have to be *Banach spaces* or *complete normed vector spaces* [92], we define the linear mapping from a `CompleteNormedModule` E to another `CompleteNormedModule` F in Coq. We augment the definition of a linear mapping with condition of boundedness of the linear operator between two *Banach spaces* [92], i.e.

$$||\phi(x)|| \leq K\ ||x||; \quad \phi : E \to F$$

where K is a constant independent of the choice of $x \in E$.

The definition of *consistency* (2.6) and *convergence* (2.5) hold in the limit of h tending to zero. Thus, an important step in the proof is to express these limits in Coq. Formally, the notion of f tending to l at the limit point x requires, for any $\epsilon > 0$, to find a neighborhood V of x such that any point u of V satisfies $|f(u) - l| < \epsilon$ [28]. This notion has been formalized in `Coquelicot` [28] using the concept of *filters*. In topology, a filter is a set of sets, which is nonempty, upward closed, and closed under intersection [39]. It is commonly used to express

the notion of convergence in topology. We have used a filter, `locally x` [96] to denote an open neighborhood of x, and predicate `filterlim` [96] to formalize the notion of convergence (in the context of limits) of f towards l at limit point x, i.e. $\lim_{x \to a} f(x) = l$. Therefore, the definition of consistency (2.6) is expressed as:

```
(is_lim (fun h:R ⇒ norm (minus (Ah h (rh h u)) (sh h (A u)))) 0 0
```

where the limits of functions is expressed using the predicate is_lim [28].

We next discuss the formalization of the statement of convergence of a finite difference scheme in Coq. We note that from Theorem 6, *consistency* and *stability* imply *convergence*. This notion is expressed in Coq as follows:

```
(is_lim (fun h:R ⇒ norm (minus (Ah h (rh h u)) (sh h (A u)))) 0 0 (*Consistency*) ∧
(∃ K:R , ∀(h:R), operator_norm(Eh h)<=K ) (* Stability*) →
is_lim(fun h:R⇒ norm (minus (rh h (E(f))) (Eh h (sh h (f))))) 0 0) (*Convergence*).
```

where the *operator norm* is defined as $||f||_\phi = sup_{u \neq 0_E \wedge \phi(u)} \frac{||f(u)||_F}{||u||_E}$ and has been formally defined in [19].

The basic idea is that we bound the *global discretization error* ($||r_h \mathcal{E} f - E_h s_h f||$) above using the stability criterion, i.e. $||r_h \mathcal{E} f - E_h s_h f|| \leq K \; ||A_h r_h u - s_h \mathcal{A} u||$, and then prove that as the *local discretization error* ($||A_h r_h u - s_h \mathcal{A} u||$) tends to zero in the limit of h tending to zero, the upper bound on the global discretization error tends to zero (using the property of limits). Using the property of norm , i.e. $0 \leq ||r_h \mathcal{E} f - E_h s_h f||$, we arrive at the inequality

$$0 \leq ||r_h \mathcal{E} f - E_h s_h f|| \leq K \; ||A_h r_h u - s_h \mathcal{A} u||$$

In Coq, we define the lower bound of the inequality as a constant function with value 0 as: `fun _ ⇒ 0`. Since the limit of a constant function is the constant itself, i.e. $\lim_{h \to 0} 0 = 0$, and $\lim_{h \to 0} ||A_h r_h u - s_h \mathcal{A} u|| = 0$ (Consistency), using the *sandwich theorem* for limits, $\lim_{h \to 0} ||r_h \mathcal{E} f - E_h s_h f|| = 0$. The *sandwich theorem* [125] states that if we have functions obeying the inequality: $f(x) \leq g(x) \leq h(x)$ and $\lim_{x \to a} f(x) = L \quad \wedge \quad \lim_{x \to a} h(x) = L$ on some open neighborhood of $x = a$, then $\lim_{x \to a} g(x) = L$. This proves the convergence 2.5 and completes the proof of the Lax–equivalence theorem.

We state the statement of Lax–equivalence theorerm in Coq as

```
Theorem is_convergent:
∀(u:X) (f:Y) (h:R) (uh: Xh h) (rh: ∀(h:R), X → (Xh h)) (sh: ∀(h:R), Y→ (Yh h))
 (E: Y→ X) (Eh:∀ (h:R), (Yh h)→ (Xh h)),
 is_linear_mapping X Y Aop →
 (* Hypothesis that A is a linear mapping from X to Y*)
    f=Aop u→
```

```
(* Differential equation: A u = f *)
 (∀ (h:R), is_linear_mapping (Xh h) (Yh h) (Ah_op h) )→
 (* Hypothesis that Ah is a linear mapping from Xh to Yh for each h*)
 (∀ (h:R), is_bounded_linear X (Xh h) (rh h))→
 (* Hypothesis that rh is a bounded linear operator (restriction) from X to Xh for each h*)
 (∀ (h:R), is_bounded_linear Y (Yh h) (sh h))→
 (* Hypothesis that sh is a bounded linear operator (restriction) from Y to Yh*)
 is_bounded_linear Y X E →
 (* Hypothesis that E is a bounded linear operator from Y to X*)
 u= E f→
 (* Defining solution in continuous space (true solution) *)
 (∀ (h:R), is_bounded_linear (Yh h) (Xh h) (Eh h)) →
 (* Hypotheis that Eh is a bounded linear operator from Yh to Xh for each h*)
 (∀ h:R, is_finite (operator_norm(Eh h))) →
 (* Hypothesis that ||Eh|| is finite*)
  (uh= Eh h (sh h f))→
  (* Defining a discrete solution uh *)
  ( Ah_op h uh = sh h f)→
  (* Discretized set of equation: Ah uh = fh *)
 (∀ (h:R), rh h u= Eh h (Ah_op h (rh h u)))→
 (* uh =Eh Ah uh, where Eh Ah=I *)
 (∀ h:R, minus (Ah_op h (rh h u)) (sh h (Aop u)) <> zero )→

 (is_lim (fun h:R ⇒
      norm (minus (Ah_op h (rh h u)) (sh h (Aop u)))) 0 0 (*Consistency*) ∧
 ( ∃K:R , ∀(h:R), operator_norm(Eh h)<=K ) (* Stability*)→
 is_lim (fun h:R⇒
      norm (minus (rh h (E(f))) (Eh h (sh h (f))))) 0 0) (*Convergence*).
```

2.4 Proof of convergence of a finite difference scheme

2.4.1 Overview of finite difference scheme

A finite difference scheme (FD) approximates a differential equation with a difference equation. The derivatives are expressed in terms of function values at finite number of points in the discretized domain. Thus, an analytical k^{th} derivative, $\frac{d^k u}{dx^k}$ at point x_o is approximated

numerically as

$$\widehat{\frac{d^k u}{dx^k}}\bigg|_{x_o} \approx \sum_{i=l}^{r} a_i u_i$$

where l is the number of points to the left of x_o, and r is the number of points to the right of x_o, as illustrated in the Figure 2.1. Let us consider the example of approximating

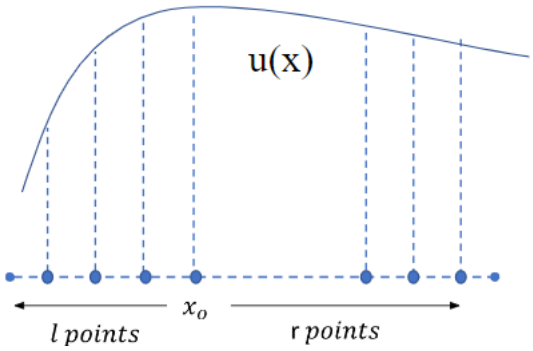

Figure 2.1: Discretization of a continuous function is a discrete domain.

a first derivative at point x_i as illustrated in the Figure 2.2. A *forward difference scheme*

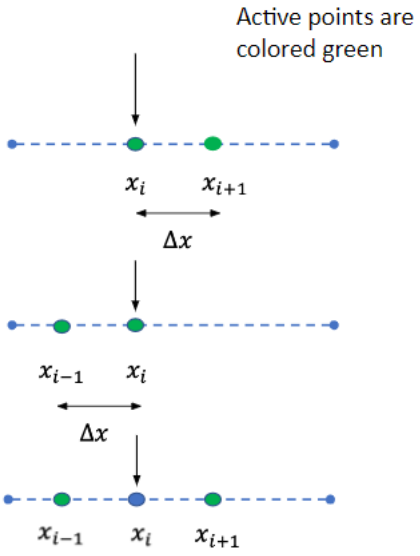

Figure 2.2: Illustration of common finite-difference schemes

(illustrated in the top of Figure 2.2) would be written as

$$\widehat{\frac{du}{dx}}\Big|_{x_i} \approx \frac{u(x_{i+1}) - u(x_i)}{\Delta x}$$

A *backward difference scheme* would be written as

$$\widehat{\frac{du}{dx}}\Big|_{x_i} \approx \frac{u(x_i) - u(x_{i-1})}{\Delta x}$$

Similarly, a *central difference scheme* would be written as

$$\widehat{\frac{du}{dx}}\Big|_{x_i} \approx \frac{u(x_{i+1}) - u(x_{i-1})}{2\Delta x}$$

Thus, each of these schemes approximate a first derivative with function values at discrete set of points, i.e., $u(x_{i-1}),\ u(x_i),\ u(x_{i+1})$ in a given domain, depending on the choice of direction, i.e., *forward, backward, or central*. The value of *coefficient* a_i depends on the order of accuracy for a given scheme. In the case of a *forward* scheme (*first order accurate*) to approximate the first derivative, $a_i = -1,\ a_{i+1} = 1$. Similarly, for the *backward* scheme (*first order accurate*), $a_i = 1,\ a_{i-1} = -1$, and for the *central* scheme (*second order accurate*), $a_{i+1} = 1/2,\ a_i = 0,\ a_{i-1} = -1/2$. A more detailed perspective on developing FD schemes for a differential equation can be referred to in the book [119].

Since we are computing a numerical approximation of the exact derivatives, we are interested in knowing the order of the discretization error.

Definition 7 (Discretization error)**.** Let $D(u)$ denote the true derivative of a function $u : \mathbb{R} \to \mathbb{R}$ and $N(u)$ denote the finite difference approximation of the true derivative. The discretization error (commonly referred to as the truncation error) (τ) is then defined as:

$$\tau \triangleq D(u) - N(u) \tag{2.8}$$

If the function u is *analytic*, it can be expressed as a *Taylor series expansion* at the point of evaluation. The truncation error is then evaluated by expressing the numerical derivatives in terms of a truncated Taylor polynomial and then taking a difference of the true derivative and the numerical derivative. This gives us an upper bound on the discretization error. If a numerical method is consistent, the truncation error can be expressed as:

$$\tau = \mathcal{O}(\Delta x)^n$$

when Δx tends to zero, and where n is the order of the truncated Taylor polynomial. Thus, if a finite difference scheme is second order accurate, then the truncation error (τ) would

be of the order of $\mathcal{O}(\Delta x)^2$. We use this idea to formalize the proof of consistency of a finite difference scheme. This requires the use of an important theorem from calculus, the Taylor–Lagrange theorem.

Theorem 8 (Taylor–Lagrange theorem)**.** *Suppose that f is $n+1$ times differentiable on some interval containing the center of convergence c and x, and let $P_n(x) = f(c) + \frac{f^{(1)}(c)}{1!}(x - c) + \frac{f^2(c)}{2!}(x-c)^2 + .. + \frac{f^{(n)}(c)}{n!}(x-c)^n$ be the n^{th} order Taylor polynomial of f at $x = c$. Then $f(x) = P_n(x) + E_n(x)$ where $E_n(x)$ is the error term of $P_n(x)$ from $f(x)$. i.e. $E_n = f(x) - P_n(x)$, and for ξ between c and x, the Lagrange remainder form of the error E_n is given by the formula $E_n(x) = \frac{f^{n+1}(\xi)}{(n+1)!}(x-c)^{(n+1)}$.*

2.4.2 Example problem

We will consider a $1-$D differential equation,

$$\frac{d^2u}{dx^2} = 1 \tag{2.9}$$

in a domain $x \in (0, L)$ with a simple boundary condition, $u(0) = 0$ and $u(L) = 0$, where L is the length of the domain. We will use a second order accurate central finite-difference scheme to approximate 2.9 to obtain a discretized equation

$$\frac{u(x + \Delta x) - 2u(x) + u(x - \Delta x)}{(\Delta x)^2} = 1 \tag{2.10}$$

where Δx is the discretization step and x is the point at which the difference equation is evaluated. We will assume a *uniform discretization step* through out the domain. We will refer this 2.10 as a numerical scheme $\mathcal{N}_h$.

2.4.3 Proof of convergence

Theorem 6 requires that we prove *consistency* and *stability* of a FD scheme to prove its *convergence*. We will therefore discuss the formalization of consistency and stability of the scheme $\mathcal{N}_h$.

2.4.3.1 Proof of consistency

Stacking the point-wise discretized equation $\mathcal{N}_h$ 2.10 in a $1-$D domain, we obtain the following linear system

$$\underbrace{\frac{1}{h^2}\begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ 0 & \dots & 1 & -2 & 1 & 0 \\ 0 & \dots & 0 & 1 & -2 & 1 \\ 0 & \dots & 0 & 0 & 0 & 1 \end{bmatrix}}_{A_h} \underbrace{\begin{bmatrix} u_o \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{bmatrix}}_{r_h u} = \underbrace{\begin{bmatrix} 0 \\ 1 \\ \vdots \\ 1 \\ 1 \\ 0 \end{bmatrix}}_{s_h A u} \tag{2.11}$$

Instantiating the definition of consistency 2.6 with equation 2.11, we want to prove

$$\lim_{h\to 0} \left\| \frac{1}{h^2}\begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ 0 & \dots & 1 & -2 & 1 & 0 \\ 0 & \dots & 0 & 1 & -2 & 1 \\ 0 & \dots & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_o \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 1 \\ 1 \\ 0 \end{bmatrix} \right\|$$

$$= \lim_{h\to 0} \left\| \begin{bmatrix} \frac{u_o}{h^2} \\ \frac{u_o - 2u_1 + u_2}{h^2} - 1 \\ \frac{u_1 - 2u_2 + u_3}{h^2} - 1 \\ \vdots \\ \frac{u_{N-2} - 2u_{N-1} + u_N}{h^2} - 1 \\ \frac{u_N}{h^2} \end{bmatrix} \right\|$$

$$= 0 \tag{2.12}$$

Theorem 9 (Scheme consistency)**.** *$\forall h : \mathbb{R},\ h > 0$, the scheme $\mathcal{N}_h$ is consistent in the domain $x \in (0, L)$ if the l^1 vector norm of equation 2.12*

$$\lim_{h\to 0} \left[\left| \frac{u_o}{h^2} \right| + \left| \frac{u_o - 2u_1 + u_2}{h^2} - 1 \right| + .. + \left| \frac{u_{N-2} - 2u_{N-1} + u_N}{h^2} - 1 \right| + \left| \frac{u_N}{h^2} \right| \right] = 0 \tag{2.13}$$

Proof. Since, we are computing the limit of a finite summation, equation 2.13 can be written

as a finite of sum of individual limits using the addition rule for limits [125] as

$$\lim_{h\to 0}\left|\frac{u_o}{h^2}\right| + \sum_{i=1}^{N-1}\lim_{h\to 0}\left|\frac{u_{i-1}-2u_i+u_{i+1}}{h^2}-1\right| + \lim_{h\to 0}\left|\frac{u_N}{h^2}\right| = 0 \tag{2.14}$$

We can prove the equivalence between equation 2.13 and equation 2.14 by mathematical induction on the summation index i. $\lim_{h\to 0}\left|\frac{u_o}{h^2}\right| = 0$ and $\lim_{h\to 0}\left|\frac{u_N}{h^2}\right| = 0$, trivially because of the boundary conditions we imposed, i.e. $u_o = 0$ and $u_N = 0$. *The norm used in (2.12) are in the space* Y_h*, i.e.,* $||.||_{Y_h}$. Thus, 2.14 reduces to proving:

$$\sum_{i=1}^{N-1}\lim_{h\to 0}\left|\frac{u_{i-1}-2u_i+u_{i+1}}{h^2}-1\right| = 0 \tag{2.15}$$

To prove 2.15, we need to prove

$$\lim_{h\to 0}\left|\frac{u_{i-1}-2u_i+u_{i+1}}{h^2}-1\right| = 0; \quad \forall\ i,\ 1\le i\le N-1 \tag{2.16}$$

The limit 2.16 can be proved using the *sandwich theorem* for limits [125]. We will show using the point-wise consistency analysis, as will be discussed next, that the function

$$f(h) = \left|\frac{u_{i-1}-2u_i+u_{i+1}}{h^2}-1\right|$$

can be bounded by a function $g(h)$ such that $\lim_{h\to 0} g(h) = 0$. Since $f(h)$ is bounded below trivially by 0, by sandwich theorem for limits,

$$\sum_{i=1}^{N-1}\lim_{h\to 0}\left|\frac{u_{i-1}-2u_i+u_{i+1}}{h^2}-1\right| = 0.$$

□

Proving point-wise consistency: The point-wise consistency analysis of the scheme $\mathcal{N}_h$ is the application of the Taylor–Lagrange Theorem 8 to $\mathcal{N}_h$. We will specifically prove that for the scheme $\mathcal{N}_h$, 2.10, the truncation error τ is quadratic in Δx:

$$\tau = \left|\frac{d^2u}{dx^2} - \frac{u(x+\Delta x)-2u(x)+u(x-\Delta x)}{(\Delta x)^2}\right| = \mathcal{O}(\Delta x)^2 \tag{2.17}$$

By invoking the definition of Big-O notation, the theorem for point-wise consistency analysis is stated as

Theorem 10. *For a given $x \in (a,b)$, $\exists\ \gamma > 0$ and $\exists\ \Gamma > 0$, such that $\forall\ \Delta x > 0$, $(x+\Delta x) \in (a,b)$, $(x-\Delta x) \in (a,b)$ and $\Delta x < \gamma$,*

$$\left| \frac{d^2u}{dx^2} - \frac{u(x+\Delta x) - 2u(x) + u(x-\Delta x)}{(\Delta x)^2} \right| \leq \Gamma(\Delta x)^2$$

Note that both γ and Γ are independent of the discretization step size Δx. We state the Theorem 10 in Coq as

```
Theorem taylor_FD (x:R): Oab x →
∃ gamma:R, gamma >0 ∧ ∃ G:R, G>0 ∧
∀ dx:R, dx>0 → Oab (x+dx) → Oab (x−dx)→ (dx< gamma →
Rabs((D 0 (x+dx)− 2∗(D 0 x) + D 0 (x−dx)) / (dx ∗ dx)− D 2 x) <= G∗(dx^2)).
```

where `Oab x` represents an open interval, i.e., $a < x < b$ and `D k x` denotes k^{th} derivative of u with respect to x. Thus, the function $u(x)$ will be denoted as `D 0 x`.

We start by introducing the following lemmas required to complete the proof.

Lemma 11 ($|F(x)| \sim \mathcal{O}(\Delta x)^4$)**.** $\forall x \in (a,b), \exists\ \eta \in \mathbb{R}, \eta > 0 \wedge \exists\ M \in \mathbb{R}, M > 0 \wedge$
$\forall \Delta x \in \mathbb{R}, \Delta x > 0 \rightarrow (x+\Delta x) \in (a,b) \rightarrow \Delta x < \eta \rightarrow |F(x)| \leq M(\Delta x)^4$.

Here, $F(x)$ is the Lagrange remainder in the expansion of $u(x+\Delta x)$ up to degree 3 and is defined as:

$$F(x) \triangleq u(x+\Delta x) - u(x) - \Delta x \frac{du}{dx}\Big|_x - \frac{1}{2!}(\Delta x)^2 \frac{d^2u}{dx^2}\Big|_x - \frac{1}{3!}(\Delta x)^3 \frac{d^3u}{dx^3}\Big|_x \tag{2.18}$$

Thus, Lemma 11 states that the Lagrange remainder $F(x) = \frac{1}{4!}(\Delta x)^4 \frac{d^4u(\xi)}{dx^4}$ is of order $(\Delta x)^4$ for all $\xi \in (x, x+\Delta x)$.

Lemma 12 ($|G(x)| \sim \mathcal{O}(\Delta x)^4$)**.** $\forall x \in (a,b), \exists\ \delta \in \mathbb{R}, \delta > 0 \wedge \exists\ K \in \mathbb{R}, K > 0 \wedge$
$\forall \Delta x \in \mathbb{R}, \Delta x > 0 \rightarrow (x-\Delta x) \in (a,b) \rightarrow \Delta x < \delta \rightarrow |G(x)| \leq K(\Delta x)^4$.

Here, $G(x)$ is the Lagrange remainder in the expansion of $u(x-\Delta x)$ up to degree 3 and is defined as:

$$G(x) \triangleq u(x-\Delta x) - u(x) + \Delta x \frac{du}{dx}\Big|_x - \frac{1}{2!}(\Delta x)^2 \frac{d^2u}{dx^2}\Big|_x + \frac{1}{3!}(\Delta x)^3 \frac{d^3u}{dx^3}\Big|_x \tag{2.19}$$

Thus, Lemma 12 states that the Lagrange remainder $G(x) = \frac{1}{4!}(\Delta x)^4 \frac{d^4u(\xi)}{dx^4}$ is of order $(\Delta x)^4$ for all $\xi \in (x-\Delta x, x)$.

Both the lemmas are straightforward applications of the Taylor–Lagrange theorem (Theorem 8), and are crucial to the formalization of the proof of consistency of a finite difference scheme.

Next, we present an informal proof of the Theorem 10 (taylor_FD) followed by a discussion on its formalization.

Proof.

$$|F(x)| \leq M(\Delta x)^4 \quad \text{[From Lemma 11]} \tag{2.20}$$

$$|G(x)| \leq K(\Delta x)^4 \quad \text{[From Lemma 12]} \tag{2.21}$$

Adding equation (2.20) and (2.21), we get:

$$\begin{aligned} & |F(x)| + |G(x)| \leq (M+K)(\Delta x)^4 \\ \implies & |F(x) + G(x)| \leq (M+K)(\Delta x)^4 \\ & \text{[Using the triangle inequality, } (|F(x)+G(x)| \leq |F(x)| + |G(x)|) \text{]} \\ \implies & |F(x) + G(x)| \leq \Gamma(\Delta x)^4 \quad \text{(Instantiating } \Gamma := M+K) \end{aligned} \tag{2.22}$$

Unfolding the definitions $F(x)$ and $G(x)$, and doing the algebra we get:

$$\begin{aligned} & \left| u(x+\Delta x) - 2u(x) + u(x-\Delta x) - (\Delta x)^2 \frac{d^2u}{dx^2} \right| \leq \Gamma(\Delta x^4) \\ \implies & \left| \frac{u(x+\Delta x) - 2u(x) + u(x-\Delta x)}{(\Delta x)^2} - \frac{d^2u}{dx^2} \right| \leq \Gamma(\Delta x^2) \quad \textbf{[QED]} \end{aligned} \tag{2.23}$$

□

An important point to note is that the condition $|F(x)| + |G(x)| \leq M(\Delta x)^4 + K(\Delta x)^4$ holds when $0 < |\Delta x| < \gamma$, where γ is as defined in 10. We therefore choose, $\gamma = min(\eta, \delta)$, where η is such that, $|F(x)| \leq M(\Delta x)^4$ holds when $0 < \Delta x < \eta$, and δ is such that, $|G(x)| \leq K(\Delta x)^4$ holds when $0 < \Delta x < \delta$.

Formalization in Coq: We followed the proof above and formalized it in the Coq proof assistant. To apply the Taylor–Lagrange theorem [103] to the consistency analysis of a central difference approximation, we broke down the Theorem 10 into two lemmas 11, and 12. Therefore, we next discuss the proof of these lemmas.

Proof of Lemma 11: We state the lemma 11 formally in Coq as

```
Lemma taylor_uupper (x:R): Oab x→ ∃eta: R, eta>0 ∧
  ∃M :R, M>0 ∧ ∀dx:R, dx>0 → Oab (x+dx) →
  (dx<eta → Rabs(D 0 (x+dx)− Tsum 3 x (x+dx))<=M∗(dx^4)).
```

In the proof of lemma taylor_uupper, existential quantification associated with η and M has to be addressed. We chose η as $b - x$, since the interval in which we are studying Taylor–Lagrange for $u(x + \Delta x)$ is $[x, b]$. Since $\Delta x \in (x, b)$ and $\Delta x < \eta$, it seems logical to chose

$\eta = b - x$. For the choice of M, we obtained extreme bounds in the interval. Since the function u and its derivatives are continuous in a compact set $[x, b]$, we are guaranteed to get maximum and minimum values [cite]. In Coq, we applied the lemma continuity_ab_max to obtain a maximum value, $\left(\frac{d^4u}{dx^4}\right)_{max} = \frac{d^4u(F)}{dx^4}$ such that $\frac{d^4u(\xi)}{dx^4} \leq \frac{d^4u(F)}{dx^4}, \forall \xi \in [x, b]$. Similarly, we apply the lemma continuity_ab_min to obtain a minimum value, $\left(\frac{d^4u}{dx^4}\right)_{min} = \frac{d^4u(G)}{dx^4}$ such that $\frac{d^4u(G)}{dx^4} \leq \frac{d^4u(\xi)}{dx^4}, \forall \xi \in [x, b]$. Thus, $M := \max\left(\left|\frac{d^4u(G)}{dx^4}\right|, \left|\frac{d^4u(F)}{dx^4}\right|\right)$. With this choice of M, we can bound the Lagrange remainder or the truncation error from above and thus prove Lemma 11.

Proof of Lemma 12: Formally Lemma 12 is stated in Coq as:

```
Lemma taylor_ulower (x:R): Oab x → ∃delta: R, delta>0 ∧
    ∃K :R, K>0 ∧ ∀dx:R, dx>0 → Oab (x−dx) →
    (dx<delta → Rabs(D 0 (x−dx)−Tsum 3 x (x−dx))<=K∗(dx^4)).
```

The proof of Lemma 12 follows the same approach as that of Lemma 11. Here, we chose δ as $x - a$, since the interval in which we are studying Taylor–Lagrange theorem for $u(x - \Delta x)$, $\Delta x \in (a, x)$, and $\Delta x < \delta$. We chose K in the same way as we chose M in Lemma 11 except that the interval in which we obtain maximum and minimum values for $\frac{d^4u}{dx^4}$ is $[a, x]$ in this case. Thus, $\left(\frac{d^4u}{dx^4}\right)_{min} = \frac{d^4u(G)}{dx^4}$, $\left(\frac{d^4u}{dx^4}\right)_{max} = \frac{d^4u(F)}{dx^4}$, and $K := \max\left(\left|\frac{d^4u(G)}{dx^4}\right|, \left|\frac{d^4u(F)}{dx^4}\right|\right), \forall c \in [a, x]$.

We can then instantiate $\Gamma := M + K$, and $\gamma := \min(\eta, \delta)$ in Theorem 10, where (M, η) and (K, δ) have been defined as in Lemma 11 and 12 respectively. To implement this instantiation, we have to carefully *destruct* the lemmas introduced in the theorem statement. Then, we simply apply lemma 11 and 12, to complete the proof of taylor_FD.

Proving the main consistency theorem: We formalized the main theorem statement 9 in Coq as

```
Theorem consistency_inst: ∀(U:X) (f:Y) (h:R) (uh: Xh h)
 (rh: ∀(h:R), X → (Xh h)) (sh: ∀(h:R), Y→ (Yh h))
 (E: Y→ X) (Eh: ∀(h:R),(Yh h)→ (Xh h)),
 is_lim (fun h:R ⇒ norm (minus (Ah h (rh h U)) (sh h (A U)))) 0 0.
```

As discussed earlier, we prove Theorem 9 by conveniently reducing its proof to point-wise consistency analysis – application of the Taylor–Lagrange theorem at each point x in the 1−D domain. Thus, to integrate this Taylor–Lagrange analysis (taylor_FD) into the main theorem for consistency (consistency_inst), we prove the following lemma 2.16 in Coq

```
Lemma lim_sum: is_lim (fun h:R ⇒
  sum_n_m (fun i:nat ⇒ Rabs (( D 0 (x i −h) −2∗ (D 0 (x i))
```

+ D 0 (x i +h))∗/(h^2) −1)) 1%nat (pred N)) 0 0.

where sum_n_m defines $\sum_{i=n}^{m}$ in Coq. In order to represent $x_i,\ i = 0, \ldots, N$, we define x of type: nat → R. The boundary conditions for $\mathcal{N}_h$ are imposed as hypothesis statements:

Hypothesis u_0 : (D 0 (x 0))= 0.
Hypothesis u_N: (D 0 (x N)) =0.

The differential equation is defined as

Hypothesis u_2x: ∀ i:nat, (D 2 (x i)) =1.

We note here that the above-mentioned formalization (consistency_inst) is not unique to the second order scheme that we discussed. The approach we discuss can easily be generalized to verify consistency of any finite difference scheme. The crucial step in such a generalization is the appropriate instantiation of the A_h matrix and the vectors $r_h u$ and $s_h A u$.

2.4.3.2 Proof of stability

In this section we discuss the stability of the scheme $\mathcal{N}_h$. We will be treating stability from a spectral viewpoint – reason stability using the eigenvalue-eigenvector pair of the matrix A_h. From section 2.3.1, stability of a numerical scheme requires the solution operator $E_h = A_h^{-1}$ to be uniformly bounded.

Theorem 13 (Scheme stability)**.**

$$\exists\ K \in \mathbb{R},\ \forall\ h \in \mathbb{R}, ||E_h||_{op} \leq K$$

where $||.||_{op}$ *is the operator norm.*

For our proof of stability of $\mathcal{N}_h$, we consider $||.||_{op}$ as the $||.||_2$ of a matrix. We prove Theorem 13 by bounding the eigenvalues of E_h uniformly. Eigenvalues of E_h are just inverse of the eigenvalues of A_h. We prove this fact using the following lemma statement in Coq

Lemma inverse_eigen (m N:nat) (a b:R) :
(2< N)%nat → (0<=m<N)%nat → 0<a →
((invertible N (Ah N a b a) (inverse_A N a b)) ∧ (LHS m N a b a= RHS m N a b a)) →
(Eigen_vec m N a b a) =
Mmult (inverse_A N a b) (Mmult (Eigen_vec m N a b a) (Lambda m N a b a)).

where, $LHS \triangleq A_h s_m$ and $RHS \triangleq s_m \lambda_m$ for the eigenvalue-eigenvector pair (*eigen-system*) (λ_m, s_m). We use the precondition $N > 2$ in the lemma inverse_eigen and any other lemmas related to the matrix A_h because we need at least three points in the 1−D domain to define a matrix system corresponding to the scheme $\mathcal{N}_h$.

Proof. We start with the definition of eigen-system (λ_m, s_m),

$$A_h s_m = \lambda_m s_m$$

Multiplying by A_h^{-1} on both sides and using the definition :$A_h^{-1}A_h = I$,

$$A_h^{-1}A_h s_m = A_h^{-1}\lambda_m s_m \implies s_m = \lambda_m A_h^{-1} s_m \implies \frac{s_m}{\lambda_m} = A_h^{-1} s_m$$

□

We have developed a generalized framework for the formalization of stability for a symmetric tri-diagonal matrix in Coq. We denote this matrix with $A_h(a, b, c)$ with $c = a$ for symmetry. This notation means that b is on the diagonal, c is on the upper diagonal and a is on the lower diagonal. All the other entries are zero. We define the matrix $A_h(a, b, c)$ in Coq as

```
Definition Ah (m:nat) (a b c: R) := mk_matrix m m
    (fun i j ⇒ if (eqb i j) then b else
      if (eqb (sub i j) one) then a else
        if (eqb (sub j i) one) then c else 0).
```

Ah takes the dimension m, and $a,\ b,\ c$ as parameter and constructs a tri-diagonal matrix using the matrix constructor mk_matrix defined in Coquelicot. We will use A_h interchangeably with $A_h(a, b, c)$ for the sake of brevity. Since A_h is tri-diagonal, we can define a closed form expression for its eigen-value λ_m (Lambda), and the corresponding eigenvector s_m (Eigen_vec) as

$$\lambda_m = b + 2\sqrt{ac}\cos\left[\frac{m\pi}{N+1}\right]; \quad s_m = (s_j)_m = \left[\frac{a}{c}\right]^{j-1/2}\sqrt{\frac{2}{N+1}}\sin\left[j\frac{m\pi}{N+1}\right] \tag{2.24}$$

$\forall m, j = 1, \ldots, N$. The corresponding Coq definitions are

Definition Eigen_vec (m N:nat) (a b c:R):= mk_matrix N 1%nat (fun i j ⇒
sqrt (2 / INR (N+1))∗(Rpower (a ∗/c) (INR i +1 −1∗/2))∗
sin(((INR i +1)∗INR(m+1)∗PI)∗/INR (N+1))).

Definition Lambda (m N:nat) (a b c:R):= mk_matrix 1%nat 1%nat (fun i j ⇒
b + 2∗ sqrt(a∗c)∗ cos ((INR (m+1) ∗ PI)∗/INR(N+1))).

Since naturals in Coq start with zero, we write INR (m+1) and INR i+1, where INR is an injection from naturals to reals in Coq.

Lemma to verify that the eigenvalues and eigenvectors belong to the spectrum of $A_h(a, b, a)$: We then formally verify that the analytical expressions for the pair (λ_m, s_m) indeed belong to the spectrum of A_h. In Coq, we state this formally as:

Lemma eigen_belongs (a b c: R): ∀(m N:nat),
(2 < N)%nat → (0 <= m < N)%nat → a=c ∧ 0<c→
(LHS m N a b c) = (RHS m N a b c).

Here we used the definition of eigenvalue-eigenvector, i.e., $A_h s_m \triangleq \lambda_m s_m$. Formalizing the proof of the lemma eigen_belongs was challenging due to the structure of the matrix A_h. A_h is a tri-diagonal matrix with non-zero entries on the diagonal, sub-diagonal and super-diagonal. The other entries are zero and hence the matrix is sparse.

$$\therefore \underbrace{\sum_{j=0}^{N-1} A_h(i,j)s_m(i)}_{A_h(i,j)\neq 0} + \underbrace{\sum_{j=0}^{N-1} A_h(i,j)s_m(i)}_{A_h(i,j)=0} = \lambda_m s_m(i); \quad 0 \leq i \leq N-1 \tag{2.25}$$

In Coq, we had to carefully destruct the matrix A_h to separate the non-zero and zero sums in the LHS of equation (2.25). The idea is to do a case analysis on the row-index i as illustrated in Figure 2.3. The lemma mat_prop_1 asserts that all elements in the row $i = 0$ are zero except for the first and second element, i.e. $j = 0, 1$. The lemma mat_prop_2 asserts that all elements of the second row, $i = 1$ are zero except for the first three elements. The lemma mat_prop_3 asserts that all elements to the right of the *super diagonal* entries of the matrix A_h for all rows, $2 \leq i < N-2$ are zero. Similarly, the lemma mat_prop_6 asserts that all elements to the left of the *sub diagonal* entries of the matrix A_h for all row, $2 \leq i < N-1$ are zero. The lemma mat_prop_4 asserts that $A_h(2, 0) = 0$, and the lemma mat_prop_5 asserts that all elements in the last row except the last two elements are zero.

Now that we have formally defined the eigen-system, we will discuss the formalization of boundedness of the matrix norm of $E_h = A_h^{-1}$ to prove stability of $\mathcal{N}_h$. We have used an

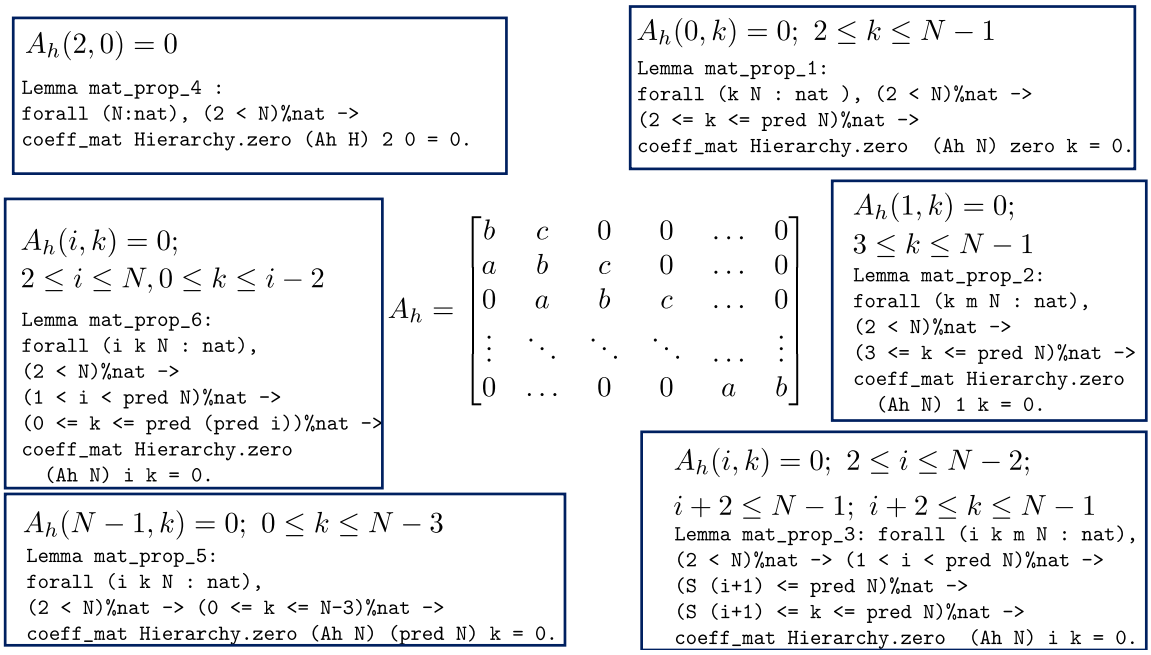

Figure 2.3: Formalizing the tri-diagonal structure of the matrix. This formalization can be used for any tri-diagional system. coeff_mat _ A i j is `Coquelicot`'s definition for $A_{i,j}$

explicit formulation of A_h^{-1} [73] in our formalization and we verify this formally using the definition: $A_h^{-1}A_h = I \ \wedge \ A_hA_h^{-1} = I$. In Coq, we state the following lemma to verify the invertibility of A_h:

Lemma invertible_check (a b:R) : ∀(N:nat), (2<N)%nat → 0<a →
Mk N (b/a) <> 0 → invertible N (Ah N a b a) (inverse_A N a b).

Here, M_k is the determinant of A_h of size k. We used the recurrence relation [73]: $M_k = D \times M_{k-1} - M_{k-2}, \quad D = \frac{b}{a}$. Overall, the approach is similar to the proof of the lemma eigen_belongs, i.e. we exploit the tridiagonal structure of A_h. The proof required us to formalize some properties about combinatorics.

For the scheme that we are considering, $D = -2$. Two important steps that were required to complete the proof of $M_k \neq 0$ for the scheme $\mathcal{N}_h$ were:

1. Proving that $M_k = (-1)^k \times (k+1)$: We proved this using strong induction on k and the recurrence relation described above. To get an intuition of why it is true, we observe the values of M_k for initial values of k: $M_0 = 1, \quad M_1 = -2, \quad M_2 = 3, \quad M_3 = -4 \cdots M_k = (-1)^k \times (k+1)$

2. Proving that the determinant, $M_k \neq 0$

Lemma on the boundedness of the matrix norm for scheme $\mathcal{N}_h$: Here, we have used the definition of the spectral (2-norm): $||A||_2 = \rho(A)$, where $\rho(A)$ is the spectral radius of A and is defined as the maximum eigen-value of A, i.e. $\rho(A) = \max_m |\lambda_m(A)|$. For the symmetric tri-diagonal matrix A_h, $A = E_h$ and $\lambda_m(E_h) = 1/\lambda_m(A_h)$. Since $\lambda_m(A_h) < 0$, $\max_m |\lambda_m(E_h)| = 1/|\lambda_{min}(A_h)|$. Hence, we define the matrix norm in Coq as follows:

Definition matrix_norm (N:nat):= 1/ Rabs (Lambda_min N).

To show that the matrix norm is uniformly bounded, we need to show that $1/|\lambda_{min}(A_h)|$ is uniformly bounded. This is where we instantiate the tri-diagonal matrix A_h with the scheme $\mathcal{N}_h$. Thus, we prove the following lemma in Coq:

Lemma spectral: ∀ (N:nat),(2<N)%nat → 1/Rabs(Lambda_min N) <= L^2/4.

where L is the length of the domain, independent of h, and is constant throughout. Lambda_min is the minimum eigenvalue for the instantiated matrix, $A'_h = A_h(\frac{1}{h^2}, \frac{-2}{h^2}, \frac{1}{h^2})$. A proof of the uniform boundedness of the eigenvalues of the scheme $\mathcal{N}_h$ is as follows

Proof.

$\lambda_{min}(A'_h) = \dfrac{2}{h^2}\left[-1 + \cos\left(\dfrac{\pi}{N+1}\right)\right]$ [For m=1 in the expression of λ_m]

Since all eigenvalues are negative, $\min|\lambda_m(A'_h)| = |\lambda_{min}(A'_h)|$,

$$\therefore \frac{1}{|\lambda_{min}(A_{h'})|} = \frac{1}{\left|\frac{2}{h^2}\left[-1+\cos\left(\frac{\pi}{N+1}\right)\right]\right|} \implies \frac{1}{|\lambda_{min}(A'_h)|} = \frac{h^2}{4\sin^2\left(\frac{\pi}{2(N+1)}\right)}$$

[Using the identity: $-1 + \cos(2x) = -2\sin^2(x)$]

Using the definition, $h \triangleq \dfrac{L}{N+1}$, where L is the domain length,

$$\therefore \frac{1}{|\lambda_{min}(A'_h)|} = \frac{L^2}{4(N+1)^2\sin^2\left(\frac{\pi}{2(N+1)}\right)} = \frac{L^2}{\pi^2}\frac{\pi^2}{4(N+1)^2\sin^2\left(\frac{\pi}{2(N+1)}\right)} = \frac{L^2}{\pi^2}\frac{x^2}{\sin^2(x)}$$

where, $x = \dfrac{\pi}{2(N+1)}$

Using the relation, $\forall x \in (0, \pi/2]$, $\dfrac{2x}{\pi} \leq \sin(x)$, or, $\dfrac{x}{\sin(x)} \leq \dfrac{\pi}{2}$, we get : $\dfrac{x^2}{\sin^2(x)} \leq \dfrac{\pi^2}{4}$

$$\therefore \frac{1}{|\lambda_{min}(A'_h)|} \leq \frac{L^2}{4}$$

□

We prove the relation $\forall x \in (0, \pi/2]$, $\frac{x}{\sin(x)} \leq \frac{\pi}{2}$, by using the concavity of $\sin(x)$ in $[0, \pi/2]$. We define a concave function $f : \mathbb{R} \to \mathbb{R}$ in Coq as follows:

Definition concave (f:R→ R) (x y c:R):=

0<=c<=1 → f(c∗x + (1−c) ∗ y) >= c∗ f x + (1−c) ∗ f y.

The proof for $\frac{x^2}{\sin^2(x)} \leq \frac{\pi^2}{4},\ \forall x \in (0, \pi/2]$ is formalized as the following lemma statement in Coq:

Lemma spectral_intermed:∀(x:R),0<x<=PI/2 → (x^2)/(sin x)^2 <=(PI^2)/4.

To show that all the eigenvalues have the same bound, we prove that $\frac{1}{\lambda_{min}(A'_h)}$ is the maximum eigenvalue of E'_h. The lemma statement is as follows:

Lemma eigen_relation: ∀(i N:nat), (2<N)%nat → (0<=i<N)%nat →

Rabs (lam i N) <= 1/ Rabs(Lambda_min N).

This completes the proof on the boundedness of the eigenvalues of E'_h. The lemma, eigen_relation also shows that the spectral radius of E'_h is $\frac{1}{|\lambda_{min}(A'_h)|}$, and justifies the defintion of matrix_norm.

We note that the definition of the matrix norm of A_h^{-1} is valid only if A_h^{-1} is a normal matrix . We therefore verify that A_h^{-1} is normal. This lemma is stated as:

Lemma inverse_is_normal (a b:R): ∀(N:nat),

Mmult (inverse_A N a b) (mat_transpose N (inverse_A N a b)) =

Mmult (mat_transpose N (inverse_A N a b)) (inverse_A N a b).

We also provide a proof that A_h is diagonalizable, i.e. $A_h = S\Lambda S^T$, where S is the matrix of eigenvectors and Λ is a diagonal matrix of Eigen-values of A_h.

Proof. We start with the definition of an Eigensystem:

$$A_h S = S\Lambda \implies A_h S S^T = S\Lambda S^T \implies A_h = S\Lambda S^T \quad [SS^T = I]$$

□

Here, we use the fact that $S^{-1} = S^T$, since S is orthonormal. We verify this by using the definition of inverse of matrices, i.e. $SS^T = S^T S = I$. In Coq, we prove the following lemma:

Lemma Scond:∀ (N:nat) (a b:R), (2<N)%nat → 0<a →

Mmult (Sm N a b) (Stranspose N a b) = identity N ∧

Mmult (Stranspose N a b) (Sm N a b) = identity N.

To prove the lemma `Scond`, we split the proof into two sub-proofs:

1. i = j,

2. $i \neq j$

For the first case, we have the condition that $\vec{s}_i \cdot \vec{s}_i = 1$, i.e. $||\vec{s}_i||^2 = 1$. This reduces to proving that the sum of the following sine-squared series is 1.

$$\sum_{m=1}^{N} \frac{2}{N+1} \sin^2 \left[j \frac{m\pi}{N+1} \right] = 1 \tag{2.26}$$

In Coq, we prove the following lemma to verify (2.26):

```
Lemma sin_sqr_sum: ∀(i N:nat), (2<N)%nat ∧ (0<=i<N)%nat →
  sum_n_m (fun l:nat ⇒ (2/(INR(N+1)))*
    sin(((INR l+1)* INR (i+1)*PI)*/ INR (N+1)) ^2) 0 (pred N)=1.
```

Here, we make use of the following theorem from [91]:

Theorem 14. *If $a\ b \in \mathbb{R}$ and $d \neq 0$ and n is a positive integer,*
$\sum_{k=0}^{n-1} cos(a+kd) = \frac{\sin nd/2}{\sin d/2} cos\left(a + \frac{(n-1)d}{2}\right)$; *where* $\sin^2(\theta) = (1 - \cos(2\theta))/2$.

We state the Theorem 14, using the following hypothesis statement in Coq:

```
Hypothesis cos_series_sum: ∀(a d:R) (N:nat), d <>0→
  sum_n_m (fun l:nat ⇒ cos (a+(INR l)*d)) 0 (pred N)=
  sin(INR N*d/2)* cos(a+INR(N−1)*d/2)*/ sin(d/2).
```

We then use the hypothesis `cos_series_sum` to prove the lemma `sin_sqr_sum`.
For the second case, we have the orthogonality condition $\vec{s}_i \cdot \vec{s}_j = 0,\ i \neq j$. This reduces to proving:

$$\sum_{k=0}^{N-1} \sin \left[(k+1) \frac{(i+1)\pi}{N+1} \right] \sin \left[(k+1) \frac{(j+1)\pi}{N+1} \right] = 0 \tag{2.27}$$

since, $\frac{2}{N+1}$ is a constant, it can be taken outside the summation.
Using the trigonometric identity,

$$\sin A \sin B = \frac{1}{2} \left[\cos(A-B) - \cos(A+B) \right]$$

we can reduce (2.27) into sums of cosines as follows:

$$\frac{1}{2} \sum_{k=0}^{N-1} \cos \left[(k+1) \frac{(i-j)\pi}{N+1} \right] - \frac{1}{2} \sum_{k=0}^{N-1} \cos \left[(k+1) \frac{(i+j+2)\pi}{N+1} \right] = 0 \tag{2.28}$$

Using Theorem (14), we can further reduce each sum in equation (2.28) into the product of sine and cosine. By doing some algebra, we prove that if $(i-j)$ and $(i+j+2)$ are

simultaneously even or they are simultaneously odd, the sums in equation (2.28) cancel out. We further note that it is always the case that $(i-j)$ and $(i+j+2)$ are simultaneously even or they are simultaneously odd. We provide an informal proof of this fact as follows:

Proof. Case 1: $(i-j)$ is even:

$$\begin{aligned}
&\exists m : nat, (i-j) = 2m \\
&\implies i = 2m + j \\
&\implies i + j + 2 = 2m + j + j + 2 \\
&\implies i + j + 2 = 2 * (m + j + 1) \quad \therefore \text{ Even}
\end{aligned}$$

Case 2: $(i-j)$ is odd:

$$\begin{aligned}
&\exists m : nat, (i-j) = 2m + 1 \\
&\implies i = j + 2m + 1 \\
&\implies i + j + 2 = j + 2m + 1 + j + 2 \\
&\implies i + j + 2 = 2 * (j + m + 1) + 1 \quad \therefore \text{ Odd}
\end{aligned}$$

□

and vice-versa for each cases. This completes the proof of orthogonality of the Eigen vectors. In Coq, we prove the following lemma to verify (2.28):

```
Lemma cos_sqr_sum: ∀(i j N:nat),
(2<N)%nat ∧ (0<=i<N)%nat ∧ (0<=j<N)%nat ∧ (i<>j) →
sum_n_m (fun l:nat ⇒ mult(/INR(N+1))
    (cos((INR(i) − INR(j)) ∗ PI / INR (N + 1) +
        INR l ∗ (INR(i) − INR(j)) ∗ PI / INR (N + 1)) −
    cos(INR(i+j+2)∗PI ∗/ INR(N+1) +
        INR l ∗ INR(i+j+2)∗PI ∗/ INR(N+1)))) 0 (pred N)=0.
```

This helps us to formally establish that the eigen vectors are orthogonal and hence the eigen space is complete.

Main stability theorem: We integrate all of the previous lemmas to prove the main stability Theorem (13).

```
Theorem stability: ∀(u:X) (f:Y) (h:R) (uh: Xh h)
    (rh: ∀(h:R), X → (Xh h))(sh: ∀(h:R), Y→ (Yh h))
    (E: Y→ X) (Eh:∀ (h:R), (Yh h)→ (Xh h)),
```

∃ K:R , ∀ (h:R), operator_norm(Eh h)<=K.

where the operator norm is instantiated with the matrix norm, $||.||_2$ using the following hypothesis:

Hypothesis mat_op_norm: ∀ (u:X) (f:Y) (h:R) (uh: Xh h)
(rh: ∀ (h:R), X → (Xh h))(sh: ∀ (h:R), Y→ (Yh h))
(E: Y→ X) (Eh:∀ (h:R),(Yh h)→ (Xh h)),
operator_norm (Eh h) = matrix_norm m.

2.4.3.3 Application of the Lax–equivalence theorem

In this section, we apply the Lax equivalence theorem is_convergent that we proved in Section 2.3.1 to a concrete differential equation $\frac{d^2u}{dx^2} = 1$ and the numerical scheme $\mathcal{N}_h$. We recall that the proof of convergence using the Lax equivalence theorem requires that the difference scheme is consistent with respect to the differential equation and is stable. We discussed the proof of consistency of the scheme in Section 2.4.3.1 and the stability in Section 2.4.3.2. Thus, we apply these proofs to complete the proof of convergence for the scheme $\mathcal{N}_h$.

2.5 Conclusion

This work investigated the formalization of convergence, stability and consistency of a finite difference scheme in the Coq proof assistant. Any continuously differentiable function can be approximated by a Taylor polynomial. The Lagrange remainder of a Taylor series provides an estimate of the *truncation* error and we formally proved that this error can be bound by n^{th} power of the discretization step, Δx, where $n-1$ is the order of the Taylor polynomial. We implemented the proof of the consistency of a finite difference scheme by breaking down the theorem statement into lemmas, each corresponding to function values at points neighboring the point of evaluation. These lemmas were proved individually by applying the Taylor–Lagrange theorem, the proof of which is already formalized in the `Coq.Interval` library [103]. Consistency and stability guarantees convergence as stated by the Lax–equivalence theorem. Following the proof of the the Lax–equivalence theorem, we formally proved convergence of a specific finite difference scheme. Specifically, we proved that the global discretization error could be bounded above by a constant factor of the local discretization error. Then, by applying the sandwich theorem for limits, we proved that the convergence condition is satisfied in the limit $\Delta x \to 0$. In the process of formalizing the proof of stability for the numerical scheme, we also developed tools for linear algebra and spectral theory, for the Coquelicot definition of matrices in Coq, which can be reused. As noted earlier, the approach

we follow is not specific to the sample numerical scheme, but can be easily extended to other numerical schemes with appropriate *instantiation* of the matrix A_h, and vectors, $r_h u$, $s_h A u$. Formalization of the proof of orthogonality of the eigenvectors helped us report the missing constant $\sqrt{\frac{2}{N+1}}$ in s_m that occurs in most textbooks/literature on numerical analysis.

We want to highlight the fact this work relied on Coquelicot formalization of matrix and finite sums. However, as we explored later in our work, the mathcomp library [100] provides better infrastructure for matrix operations and finite sums. The finite sums can be defined as an instance of the iterated big operations, which includes both iterated finite sums and products, using the infrastructure provided by the bigop [16] library in mathcomp. Thus, the bigop library provides a unified abstraction for defining iterated finite operations, which would help us deal with *sparsity* in matrix operations in a better way. Similarly, the matrix formalization in mathcomp abstracts matrix and vectors over a generic *ring* type, and provides a unified approach for treating any instance of ring type. The Coquelicot definition of matrix is however defined over real numbers, and by extension for complex numbers as well, since complex numbers are treated as a pair of reals in Coquelicot. The matrix formalization in mathcomp library also provides a rich formalization of linear algebra, and includes definitions and properties of *determinants, trace, block matrices* etc, which is missing in the matrix formalization of Coquelicot. We therefore used the mathcomp formalization of linear algebra in our subsequent developments, after our initial experiments with Coquelicot. The formalization done as part of this work could be ported easily to mathcomp.

So far in this work, we have just proved the *existence* of a *unique numerical solution*, and proved that this numerical solution converges to the *true analytical solution* under certain conditions, for a choice of a class of numerical methods. But, we also need to solve for this numerical solution. This solution will incur another set of approximation errors like *method error* and *floating-point errors*, which we will discuss in next chapters.

CHAPTER 3

Iterative Convergence Error

3.1 Introduction

In the previous chapter, we saw how to mathematically model a physical system and formulate a system of discretized equations. We also formalized a set of conditions under which the solution of these discretized set of equations converge to some "true" solution. The next logical step is to solve this discretized set of (linear) equations. For the linear system of the form, $Ax = b$, one approach (direct) to obtain the solution x is to invert the matrix A to obtain $x := A^{-1}b$. Let us denote x as the *true numerical solution*. However, such direct approaches which typically involves matrix inversion are computationally expensive. The computational (time) complexity of matrix inversion is of the order $\mathcal{O}(n^3)$, where n is the dimension of the matrix. Thus, as the dimension of the linear system increases, which is typically the case for most practical physical problems, direct approaches become intractable. We therefore need better approaches for solving a linear system of equations. Fortunately, the classical numerical analysis literature is abundant with low cost methods for solving the linear system. One such class of methods is called the *iterative methods* [119].

The goal of an iterative method is to build a sequence of approximations of the *true numerical solution*. One starts with an initial guess vector x_o, and builds a sequence of approximate solutions: $\{x_1, x_2, \ldots, x_{k-1}, x_k\}$ for k iterations, with the hope that x_k is close to the true numerical solution. The distance between x_k and the true numerical solution is called the *iterative convergence error*. To ensure the asymptotic convergence of these approximate solutions to the true numerical solution, we need to bound the iterative convergence error, and further show that this error decreases as we increase the number of iterations.

Many general purpose ordinary differential equation (ODE) solvers use some kind of iterative method to solve the linear system. For instance, ODEPACK [71], which is a collection of FORTRAN solvers for initial value problem for ODEs, uses iterative (preconditioned Krylov) methods instead of direct methods for solving linear systems. Since most codes in scientific

computing are still written and maintained in FORTRAN, these solvers are being widely used. Another widely used suite of ODE solvers is the SUNDIALS [72]. SUNDIALS has support for a variety of direct and Krylov iterative methods for solving the system of linear equations. SUNDIALS solvers are used by the mixed finite element (MFEM) package for solving nonlinear algebraic systems and by NASA for spacecraft trajectory simulation [72]. Because those iterative methods are widely used, it is important to obtain formal guarantees for the convergence of iterative solutions to the "true" solutions of differential equations. In this work we use the Coq theorem prover to formalize the convergence guarantees for a class of iterative methods called the *Stationary iterative methods*.

Contributions: We provide an overview of the *Stationary iterative methods* in Section 3.2, followed by formalization of a generalized iterative convergence theorem in Coq, and its specialization to two classical iterative methods in Section 3.4. Overall, this work [1] makes the following contributions:

- We provide a formalization of the necessary and sufficient conditions for iterative convergence in the Coq proof assistant;
- We formalize conditions for convergence of the Gauss–Seidel classical iterative method for a specific matrix structure and prove convergence on an example problem;
- We then apply the main theorem on iterative convergence to an example of the Jacobi iteration, another classical iterative method, to prove its convergence;
- During our formalization, we develop libraries for dealing with complex matrices and vectors on top of the `mathcomp` complex formalization;
- We also formalize the properties of the 2-norm of a matrix and its spectral properties.

3.2 Discussion about Stationary iterative methods

We will provide an overview of the Stationary iterative methods adopted from the book [119].

3.2.1 Theory

Let x be the true numerical solution, or the direct solution obtained by inverting the linear system $Ax = b$ as

$$x \triangleq A^{-1}b \tag{3.1}$$

[1] Our Coq formalization is available at https://github.com/mohittkr/iterative_convergence.git

Here, the *coefficient* matrix $A \in \mathbb{R}^{n \times n}$ and the *right hand side* vector $b \in \mathbb{R}^n$ are known to us and we are computing for the unknown vector $x \in \mathbb{R}^n$. We assume that the matrix A is non-singular. Thus, there exists a unique solution x of the linear system $Ax = b$. For any iterative algorithm, we start with an initial guess vector x_o and obtain a sequence of numerical solutions which are an approximation of the solution x. Let x_k be the iterative solution obtained after k iterations obtained by solving the iterative system

$$Mx_k + Nx_{k-1} = b \tag{3.2}$$

for some choice of initial solution vector x_o. The vector x_{k-1} is the iterative solution obtained after $k-1$ iterations. At the k^{th} iteration step, x_{k-1} is known to us. The matrices M and N are obtained by splitting (*regular splitting* [119]) the original coefficient matrix A such that M is easily invertible. Therefore,

$$A = M + N \tag{3.3}$$

The choice of matrices M and N define the choice of an iterative method. For instance, if we choose M to be the diagonal entries of matrix A and N to be the strictly lower and upper triangular entries of A, we obtain the Jacobi method. Thus, for the Jacobi method, $M = D$, and $N = L + U$. We discuss the Jacobi method in detail in section 3.2.3. If we choose M to be the lower triangular entries of A and N to be the strictly upper triangular entries of A, we get the Gauss–Seidel iterative method. Thus, for the Gauss–Seidel method, $M = L + D$, and $N = U$. We discuss the Gauss–Seidel method in detail in section 3.2.2. Therefore, the

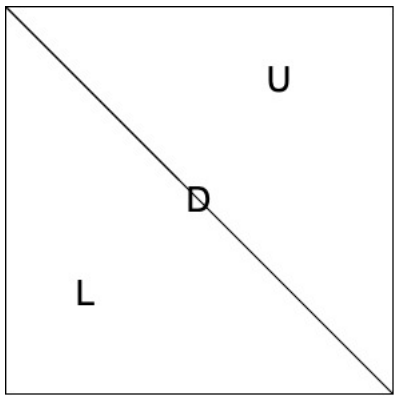

Figure 3.1: Initial partitioning of matrix $A = L + D + U$. L is the strictly lower triangular matrix. D is the diagonal matrix. U is the strictly upper triangular matrix.

matrices M and N are also known to us based on the choice of an iterative method. The right hand vector b is also known to us. Thus, the unknown solution vector x_k at k^{th} step is

obtained by rearranging terms in the iterative system (3.2) as

$$x_k = M^{-1}(b - Nx_{k-1}) \tag{3.4}$$

Since we are computing the iterative solution in the field of reals, equation 3.4 can be equivalently written as

$$x_k = (-M^{-1}N)x_{k-1} + M^{-1}b \tag{3.5}$$

It is important to note that when we compute the iterative solution in finite precision using the *floating-point arithmetic*, equation 3.4 is not equivalent to the equation 3.5 for the reasons explained in the next chapter. We will choose equation 3.4 to define our floating-point iterative solution, as will be discussed in the next chapter. In this chapter we will use equation 3.5 to define the iterative system since we work in the field of reals, and this representation makes it easier to define and work with the error recurrence relation 3.7 in Coq.

The quantity $(-M^{-1}N)$ in equation 3.5 is called an *iterative matrix* and we will denote it as S. Therefore,

$$S \triangleq -M^{-1}N \tag{3.6}$$

The iterative convergence error after k iterations is defined as

$$e^k_{iterative} \triangleq x_k - x = S^k e^o \tag{3.7}$$

The iterative solution x_k is said to converge to x if and only if

$$\lim_{k\to\infty} ||e^k_{iterative}|| = \lim_{k\to\infty} ||x_k - x|| = 0 \tag{3.8}$$

where $||.||$ denotes a vector norm. In this chapter, we will be using the l^2 vector norm defined as

$$||x||_2 = \sqrt{\sum_{j=1}^{n} |x_i|^2}.$$

We will next discuss two classical iterative methods – the Gauss–Seidel method and the Jacobi methods. Both of these methods are instances of stationary iterative methods.

3.2.2 Gauss–Seidel method

The Gauss–Seidel iterative method is an instance of stationary iterative methods to solve the system of linear equations $Ax = b$ or $\sum_{j=1}^{n} A_{ij}x_j = b_i,\ \forall i = 1, \ldots, n$ using the following

update formula [119]

$$x_{k+1,i} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_{k+1,j} - \sum_{j=i+1}^{n} A_{ij} x_{k,j} \right), \quad i = 1, 2, \ldots, n. \tag{3.9}$$

where $x_{k+1,i}$ is the value of x_i after $k+1$ iterations, and $x_{k,i}$ is the value of x_i after k iterations. We can expand equation 3.9 to obtain a system of simultaneous equations as

$$\begin{aligned} A_{11}x_{k,1} + A_{12}x_{k-1,2} + A_{13}x_{k-1,3} + \ldots + A_{1n}x_{k-1,n} &= b_1 \\ A_{21}x_{k,1} + A_{22}x_{k,2} + A_{23}x_{k-1,3} + \ldots + A_{2n}x_{k-1,n} &= b_2 \\ A_{31}x_{k,1} + A_{32}x_{k,2} + A_{33}x_{k,3} + \ldots + A_{3n}x_{k-1,n} &= b_3 \\ &\vdots \\ A_{n1}x_{k,1} + A_{n2}x_{k,2} + A_{n3}x_{k,3} + \ldots + A_{nn}x_{k,n} &= b_n \end{aligned} \tag{3.10}$$

By comparing equation 3.10 with equation 3.2, the matrix M and N can be written as [119]:

$$(M)_{ij} = \begin{cases} A_{ij} & \text{if } i \geq j \\ 0 & \text{if } i < j \end{cases}; \qquad (N)_{ij} = \begin{cases} A_{ij} & \text{if } i < j \\ 0 & \text{if } i \geq j \end{cases} \tag{3.11}$$

The iteration matrix S_G would then be:

$$S_G \triangleq -M^{-1}N = -(L+D)^{-1}U$$

where D is the diagonal matrix, L is strictly lower triangular matrix, and U is strictly upper triangular matrix of A.

3.2.3 Jacobi method

The Jacobi method is another instance of stationary iterative methods to solve the system of linear equations $Ax = b$ or $\sum_{j=1}^{n} A_{ij}x_j = b_i,\ \forall i = 1, \ldots, n$ by successive approximations. We solve for the values x_i at the k^{th} iteration, by keeping the other elements of the vector x fixed at the value obtained after $k-1$ iterations. This gives us the following update formula [119]:

$$x_{k,i} = \frac{b_i - \sum_{j \neq i} A_{ij} x_{k-1,j}}{A_{ii}} \tag{3.12}$$

Comparing the system of equations (3.12) with the iterative system (3.2), we can define the matrix M and N as [119]:

$$(M)_{ij} = \begin{cases} A_{ij} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}; \qquad (N)_{ij} = \begin{cases} A_{ij} & \text{if } i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{3.13}$$

The iterative matrix S_J, would then be:

$$S_J \triangleq -M^{-1}N = -D^{-1}(A - D) = I - D^{-1}A \tag{3.14}$$

where D is the diagonal matrix, L is strictly lower triangular matrix and U is strictly upper triangular matrix of A.

We will next discuss the existing state-of-the art in Coq, and formalization of the iterative convergence error in Coq.

3.3 Current state-of-the art in Coq

We have formalized the iterative convergence theorem in Coq theorem prover. Before we delve deeper into the formalization, it is worth providing an overview of the infrastructure provided by Coq. We primarily use two libraries in Coq on top of the standard reals library – mathcomp library [100] for its formalization of linear algebra and ssreflect language for proof automation, and the Coquelicot library [28] for its formalization of real analysis.

3.3.1 Discussion about the mathematical components library

We work extensively with *matrices, finite sequences, and canonical forms* in our formalization. Thanks to the the mathcomp [100] formalization of linear algebra, we have all the tools to work with these components.

- **Matrix and vectors:** A matrix is defined as a function from a finite set to an appropriate ring type.

  ```
  Variant matrix : predArgType := Matrix of {ffun 'I_m x 'I_n → R}.
  ```

  where 'I_m and 'I_n are *ordinal types* and represent a finite set of natural numbers from $\{0, \ldots, (m-1)\}$ and $\{0, \ldots, (n-1)\}$, respectively. A matrix denoted in mathcomp as

  ```
  Notation "\matrix_(i,j) E" := (matrix_of_fun (fun i j ⇒ E)).
  ```

  For instance, a 2×2 real valued matrix, $A = [1, 2; 3, 4]$ can be defined as

```
Definition A := \matrix_(i< 2, j < 2)
  (if (i == 0%N :> nat) then
  (if (j == 0%N :> nat) then 1%Re else 2%Re) else
  (if (j == 0%N :> nat) then 3%Re else 4%Re)).
```

where :> nat is a coercion to natural numbers. Since ordinal types is composed of a *base type*, which in this case is nat, and a *proof* that the set of naturals is less than the maximum ordinal specified, we just need the base value for comparison between natural numbers. Thus, we need to specify this explicit coercion to naturals, in the *if* clause.

- **Finite sequences:** The seq library in mathcomp provides a formalization of finite sequences. In our formalization, we use sequences to define eigenvalues from the roots of the characteristic polynomial of a matrix A. The following notation defines a map for each element x in the sequence s

  ```
  [seq E | x <- s] := map (fun x ⇒ E) s.
  ```

  An instance where we use mapping is in the definition of absolute values of eigenvalues for defining the spectral radius of an iteration matrix. We can then extract an i^{th} element in the sequence s using the notation: nth x0 s i, where x0 is the default value of the sequence. Since we switch between the set notation and sequences, we will use the enum function provided by mathcomp , which allows us to translate from the set notation to sequences.

- **Iterated sums and products:** The bigop [16] formalization in mathcomp provides necessary infrastructure for working with iterated summation and product. The notation

  ```
  Notation "\big[op/idx] (i <- r \ P) F" :=
   (bigop idx r (fun i ⇒ BigBody i op P%B F)) : big_scope.
  ```

  allows us to define iterated sums and products by instantiating the op operator and the appropriate identity idx. Here, F is a function of i chosen from a finite sequence s when the predicate P holds true. The matrix operations like matrix-vector multiplication, dot products, traces, etc. are defined in term of these big operations.

- **Jordan canonical forms:** The *canonical forms* [32] defines the Jordan canonical forms which we use extensively in our proof of iterative convergence. In particular, we will be using the definition of a block diagonal matrix which is defined in Coq as

```
Definition diag_block_mx s F :=
 if s is x :: l return 'M_((size_sum s).+1) then diag_block_mx_rec x l F else 0.
```

where s is a sequence of natural numbers and is a sequence of algebraic multiplicity of the eigenvalues of a matrix. The function (F : (∀ n, nat → 'M[R]_n.+1)) defines a block in the block diagonal matrix. It takes the size of a block, and the location of the block in the block diagonal matrix and returns a matrix of size $n + 1$. The block diagonal matrix is defined recursively using the following definition

```
Fixpoint diag_block_mx_rec k (s : seq nat) (F : (∀ n, nat → 'M[R]_n.+1)) :=
 if s is x :: l return 'M_((size_sum_rec k s).+1)
 then block_mx (F k 0%N) 0 0 (diag_block_mx_rec x l (fun n i ⇒ F n i.+1))
 else F k 0%N.
```

The definition block_mx generates a matrix using four block matrices. size_sum defines the size of a sequence s recursively.

3.3.2 Discussion about the Coquelicot library

Coquelicot [28] is an extension of the standard reals library and formalizes important results in real analysis like limits, derivatives, integrals, etc. We discussed the Coquelicot library in the previous chapter. In this chapter, we will discuss briefly the formalization of limits of sequences in Coquelicot. Limit of a sequence is defines in Coquelicot using the is_lim_seq predicate. This predicate is implemented using filters, as discussed in the previous chapter, and is defined in Coquelicot as

```
Definition is_lim_seq (u : nat → R) (l : Rbar) := filterlim u eventually (Rbar_locally l).
```

We use the is_lim_seq predicate to define the limit of iterative solutions, i.e., $\lim_{k\to\infty} ||x_k||$. There exists an equivalent $\epsilon - \delta$ definition of limits in Coquelicot [28] which states that a sequence u_n has a limit l if and only if

$$\forall \epsilon > 0, \exists N \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq N \Rightarrow |u_n - l| < \epsilon$$

Formally, this is defined in `Coquelicot` [28] as

```
Definition is_lim_seq' (u : nat → R) (l : Rbar) :=
  match l with
    | Finite l ⇒ ∀ eps : posreal, eventually (fun n ⇒ Rabs (u n − l) < eps)
    | p_infty ⇒ ∀ M : R, eventually (fun n ⇒ M < u n)
    | m_infty ⇒ ∀ M : R, eventually (fun n ⇒ u n < M)
  end.
```

where eventually is a topology on natural numbers and states that given a property P, there exists a natural number N such that a property $P\ n$, holds for all $n \geq N$. In Coq,

```
Definition eventually (P : nat → Prop) := ∃ N : nat, ∀ n, (N <= n)%nat → P n.
```

In our formalization, we often had to switch between the *filter* definition of limits and the $\epsilon - \delta$ definition of limits. This was possible due to the following equivalence lemma in Coquelicot [28]

```
Lemma is_lim_seq_spec : ∀ u l, is_lim_seq' u l ↔is_lim_seq u l.
```

3.3.3 Missing formalization in mathcomp

While there exists a rich formalization of real analysis, linear algebra, finite sets and sequences in Coq, generic properties about complex vectors and matrices were lacking. In this section, we will discuss about how we addressed these gaps in our formalization.

3.3.3.1 Formalizing properties of complex matrices and vectors

The `complex` theory in the `real_closed` [37] library in `mathcomp` defines complex numbers and basic operations on them. They define complex numbers as a real closed field, thereby allowing us to instantiate a generic field with a complex field. This was useful when we used the `eigenvalue` definition from `mathcomp` matrix algebra library and the canonical forms library by Cano et al [32]. However, since the basic properties like modulus of a complex number, conjugates, properties of complex matrices and vectors were lacking, we added them in our formalization.

We define the modulus of a complex number as

```
Definition C_mod (x: R[i]):= sqrt ( (Re x)^+2 + (Im x)^+2).
```

Here, the type `complex` is denoted by `R[i]`, and `Re x` and `Im x` denote the real and imaginary part of `x`, respectively. We proved some basic properties of the modulus, which we enumerate in Table 3.1. The `complex` theory in `mathcomp` defines the conjugate $\bar{x}$ of a complex number x as

```
Definition conjc {R : ringType} (x : R[i]) := let: a +i* b := x in a -i* b.
```

The Table 3.2 lists the missing formalization of complex conjugates that we added for this formalization. We define the conjugate transpose of a complex matrix in Coq as

```
Definition conjugate_transpose (m n:nat) (A: 'M[complex R]_(m,n)):=
  transpose_C (conjugate A).
```

where `transpose_C` is the transpose of a complex matrix, which we define in Coq as

Mathematical properties	Formalization in Coq
$\|\|0\|\| = 0$	**Lemma** C_mod_0: C_mod 0 = 0%Re.
$0 \leq \|\|x\|\|$	**Lemma** C_mod_ge_0: ∀(x: complex R), (0<= C_mod x)%Re.
$\|\|xy\|\| = \|\|x\|\| \ \|\|y\|\|$	**Lemma** C_mod_prod: ∀(x y: complex R), C_mod (x * y) = C_mod x * C_mod y.
$\|\|\frac{x}{y}\|\| = \frac{\|\|x\|\|}{\|\|y\|\|}, y \neq 0$	**Lemma** C_mod_div: ∀(x y: complex R), y <> 0 → C_mod (x / y) = (C_mod x) / (C_mod y).
$\|\|x\|\| \neq 0, \quad \text{if. } x \neq 0$	**Lemma** C_mod_not_zero: ∀(x: complex R), x <> 0 → C_mod x <> 0.
$\|\|1\|\| = 1$	**Lemma** C_mod_1: C_mod 1 = 1.
$\|\|x^n\|\| = \|\|x\|\|^n$	**Lemma** C_mod_pow: ∀(x: complex R) (n:nat), C_mod (x^+ n) = (C_mod x)^+n.
$\|\|x + y\|\| \leq \|\|x\|\| + \|\|y\|\|$	**Lemma** C_mod_add_leq : ∀(a b: complex R), C_mod (a + b) <= C_mod a + C_mod b.
$\|\|\frac{1}{x}\|\| = \frac{1}{\|\|x\|\|}, \quad \text{if } x \neq 0$	**Lemma** C_mod_inv : ∀x : complex R, x <> 0 → C_mod (invc x) = Rinv (C_mod x).
$\|\|xy\|\|^2 = \|\|x\|\|^2 \|\|y\|\|^2$	**Lemma** C_mod_sqr: ∀(x y : complex R), Rsqr (C_mod (x * y)) = (Rsqr (C_mod x)) * (Rsqr (C_mod y)).
$\|\| - x\|\| = \|\|x\|\|$	**Lemma** C_mod_minus_x: ∀(x: complex R), C_mod (−x) = C_mod x.
$\|\| \sum_{j=0}^{n} u(j)\|\| \leq \sum_{j=0}^{n} \|\|u(j)\|\|$	**Lemma** C_mod_sum_rel: ∀(n:nat) (u : 'I_n.+1 → (complex R)), (C_mod (\big[+%R/0]_j (u j))) <= \big[+%R/0]_j ((C_mod (u j))).

Table 3.1: Formalization of properties of complex modulus in Coq

Mathematical properties	Formalization in Coq
$\overline{xy} = \bar{x}\ \bar{y}$	**Lemma** Cconj_prod: ∀ (x y: complex R), conjc (x∗y)%C = (conjc x ∗ conjc y)%C.
$\overline{x+y} = \bar{x} + \bar{y}$	**Lemma** Cconj_add: ∀ (x y: complex R), conjc (x+y) = conjc x + conjc y.
$\|\|x\|\| = \|\|\bar{x}\|\|$	**Lemma** Cconjc_mod: ∀ (a: complex R), C_mod a = C_mod (conjc a).
$x = \bar{\bar{x}}$	**Lemma** conj_of_conj_C: ∀ (x: complex R), x = conjc (conjc x).
$\bar{x}x = \|\|x\|\|^2$	**Lemma** conj_prod: ∀ (x:complex R), ((conjc x)∗x)%C = RtoC (Rsqr (C_mod x)). [1]
$Re[x] + Re[\bar{x}] = 2Re[x]$	**Lemma** Re_conjc_add: ∀ (x: complex R), Re x + Re (conjc x) = 2 ∗ (Re x).
$\overline{\sum_{j=0}^{n} f(i)} = \sum_{j=0}^{n} \overline{f(i)}$	**Lemma** Cconj_sum: ∀ (p:nat) (x: 'I_p → complex R), conjc (\big[+%R/0]_(j < p) x j)= \big[+%R/0]_(j < p) conjc (x j).

Table 3.2: Formalization of properties of complex conjugates in Coq

[a] Here, `RtoC` is a coercion from reals to complex.

Definition transpose_C (m n:nat) (A: 'M[complex R]_(m,n)):= \matrix_(i<n,j<m) A j i.

and `conjugate` is the conjugate of a rectangular matrix. In Coq,

Definition conjugate (m n:nat) (A: 'M[complex R]_(m,n)):= \matrix_(i<m,j<n) conjc (A i j).

The lemma

Lemma conj_scal_mat_mul:
∀ (m n : nat) (l:complex R) (x: 'M[complex R]_(m,n)),
conjugate_transpose (scal_mat_C l x) = scal_mat_C (conjc l) (conjugate_transpose x).

proves the scaling property of a complex matrix, A

$$(\overline{lA})^T = \bar{l}(\bar{A})^T$$

Lemma conj_matrix_mul :
∀ (m n p:nat) (A: 'M[complex R]_(m,p)) (B: 'M[complex R]_(p,n)),
conjugate_transpose (mulmx A B) = (conjugate_transpose B) ∗m (conjugate_transpose A).

The lemma `conj_matrix_mul` states that the conjugate transpose of the product of matrices A and B equals the product of conjugate transpose of the matrices, i.e., $\overline{AB} = \bar{B}\bar{A}$.

Lemma conj_of_conj: ∀ (m n:nat) (x: 'M[complex R]_(m,n)),

```
conjugate_transpose (conjugate_transpose x) = x.
```

The lemma `conj_of_conj` states the idempotent property of the conjugate transpose of a complex vector, v, i.e., $\overline{\overline{v}} = v$.

3.3.3.2 Formalization of vector and matrix norms

Another missing piece in the existing formalization was the norm of a vector and a matrix. In this work, we formalize the l^2-norm of a vector and its induced matrix norm. In Coq, we define the l^2-norm of a matrix as

```
Definition matrix_norm (n:nat) (A: 'M[complex R]_n.+1) :=
Lub_Rbar (fun x⇒
    ∃v: 'cV[complex R]_n.+1, v != 0 ∧ x = (vec_norm_C (mulmx A v))/ (vec_norm_C v)).
```

where `vec_norm_C` is the l^2-norm of a complex vector, which we define in Coq as

```
Definition vec_norm_C (n:nat) (x: 'cV[complex R]_n.+1):=
  sqrt (\big[+%R/0]_l (Rsqr (C_mod (x l 0)))).
```

The definition `Lub_Rbar` is the least upper bound and is already defined in the `Coquelicot` [28] library. Mathematically, `matrix_norm` formalizes the following definition of a matrix norm

$$||A||_i = \sup_{x \neq 0} \frac{||Ax||}{||x||}$$

for a given vector norm $||.||$, which in this case is the l^2 vector norm and is mathematically defined as

$$||x|| = \sqrt{\sum_{j=1}^{n} |x_j|^2}$$

In Table 3.3 and Table 3.4, we enumerate the properties of matrix and vector norms that we formalized.

An important point to note here is that since we are using the `Coquelicot` definition of an extended real line, `Rbar`, coercion of a quantity of type `Rbar` to real requires us to prove finiteness of that quantity. We therefore have to prove that the matrix norm is finite, which we state as the following lemma in Coq

```
Lemma matrix_norm_is_finite: ∀(n:nat) (A: 'M[complex R]_n.+1),
  is_finite (matrix_norm A).
```

Since we prove asymptotic convergence of component-wise limit of the elements of a Jordan matrix, we have to work with the Frobenius norm of a matrix, which we define in Coq as

Mathematical properties	Formalization in Coq
$0 \leq \|\|A\|\|_i$	**Lemma** matrix_norm_ge_0: ∀(n:nat) (A: 'M[complex R]_n.+1), (0 <= matrix_norm A)%Re.
$\|\|Ax\|\| \leq \|\|A\|\|_i \|\|x\|\|, \quad x \neq 0$[1]	**Lemma** matrix_norm_compat: ∀(n:nat) (x: 'cV[complex R]_n.+1) (A: 'M[complex R]_n.+1), x != 0 → vec_norm_C (mulmx A x) <= ((matrix_norm A) * vec_norm_C x)%Re.
$\|\|AB\|\|_i \leq \|\|A\|\|_i \|\|B\|\|_i$	**Lemma** matrix_norm_prod: ∀(n:nat) (A B: 'M[complex R]_n.+1), (matrix_norm (A *m B) <= (matrix_norm A) * (matrix_norm B))%Re.
$0 \leq \|\|A\|\|_i \leq \|\|A\|\|_F$[2]	**Lemma** mat_2_norm_F_norm_compat: ∀(n:nat) (A: 'M[complex R]_n.+1), (0 <= matrix_norm A <= mat_norm A)%Re.

Table 3.3: Formalization of properties of matrix norm in Coq

[a]Here, x is a vector and the relation proves compatibilty relation between a matrix norm and its induced vector norm.

[b]Here, $||A||_F$ is the Frobenius norm and we prove that the 2-norm of a matrix is bounded above by the Frobenius matrix norm. The Frobenius norm of a matrix is defined as

$$||A||_F = \sqrt{\sum_{j=1}^{n}\sum_{j=1}^{n}|A_{ij}|^2}$$

Mathematical properties	Formalization in Coq
$0 \leq \|\|v\|\|$	**Lemma** vec_norm_C_ge_0: ∀(n:nat) (v: 'cV[complex R]_n.+1), (0<= vec_norm_C v)%Re.
$\|\|av\|\| = \|a\| \, \|\|v\|\|, \quad a$ is scalar	**Lemma** ei_vec_ei_compat: ∀(n:nat) (x:complex R) (v: 'cV[complex R]_n.+1), vec_norm_C (scal_vec_C x v) = C_mod x * vec_norm_C v.
$\|\|v_1 + v_2\|\| \leq \|\|v_1\|\| + \|\|v_2\|\|$	**Lemma** vec_norm_add_le: ∀(n:nat) (v1 v2 : 'cV[complex R]_n.+1), vec_norm_C (v1 + v2) <= vec_norm_C v1 + vec_norm_C v2.
$v \neq 0 \implies \|\|v\|\| \neq 0$[1]	**Lemma** non_zero_vec_norm: ∀(n:nat) (v: 'cV[complex R]_n.+1), vec_not_zero v → (vec_norm_C v <> 0)%Re.

Table 3.4: Formalization of properties of vector norm in Coq

[a] `vec_not_zero` is Coq's definition of $v \neq 0$.

```
Definition mat_norm (n:nat) (A: 'M[complex R]_n.+1) : R:=
  sqrt (\sum_i (\sum_j (Rsqr (C_mod (A i j))))).
```

We will next discuss how we will use all of the above infrastructure to prove iterative convergence in Coq.

3.4 Discussion about the Coq formalization

In this section we will provide a Coq formalization of the asymptotic convergence of the iterative convergence error in the field of reals and show its application to two classical iterative methods– the *Gauss-Seidel method* and the *Jacobi method*. We will use the example of the 1−D differential equation that we used in the previous chapter and show convergence of the Gauss–Seidel and Jacobi methods on this example. For the Gauss–Seidel method, we also prove a sufficient condition [115] for iterative convergence for a specific matrix structure. This condition provides an easy check for convergence and does not require us to compute the eigenvalues of an iterative matrix to show iterative convergence.

3.4.1 Problem set-up: Discuss the model problem

In our work, the linear differential equation $\mathcal{A}u = f$ that we chose was $\frac{d^2u}{dx^2} = 1$ for $x \in (0, 1)$ and the boundary conditions being $u(0) = u(1) = 0$. Here, the differential operator $\mathcal{A}$ is $\frac{d^2}{dx^2}$,

and f is the constant function 1. We chose a uniform grid with P points in the interior of the 1−D domain. The grid has a uniform spacing h. We will be using a central difference scheme [130] for discretizing the differential equation. Therefore, the difference equation at point x_i in the interior of the 1−D domain is given by

$$\frac{-u(x_{i+1}) + 2u(x_i) - u(x_{i-1})}{h^2} = -1; \quad h = x_{i+1} - x_i = \frac{1}{P+1} \tag{3.15}$$

When we stack the equation (3.15) for all points in the interior of the 1−D domain, we get a linear matrix system

$$\underbrace{\frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & 0 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & \dots & 0 & 0 & -1 & 2 & -1 \\ 0 & \dots & 0 & 0 & 0 & -1 & 2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_N - 1 \\ u_N \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} -1 \\ -1 \\ \vdots \\ -1 \\ -1 \end{bmatrix}}_{b} \tag{3.16}$$

Here, A is the coefficient matrix, b is the right hand side vector and x is the unknown solution vector, which can be exactly obtained by inverting the matrix A, i.e., $x = A^{-1}b$. But, we will obtain an approximation of x using iterative algorithms. We will instantiate two classical iterative algorithms: Gauss–Seidel and Jacobi, with this example problem and apply Theorem 15 to prove convergence of the approximate solutions, obtained using these algorithms, to the exact solution.

3.4.2 Generic iterative convergence theorem

The following theorem provides necessary and sufficient conditions for iterative convergence

Theorem 15. *Let an iterative matrix be defined as (3.6) for the iterative system (3.2). The sequence of iterative solutions $\{x_k\}$ converges to the direct solution x for all initial values x_o, if and only if the spectral radius of the iterative matrix S is less than* 1.

Spectral radius of a matrix is defined as the maximum eigenvalue in magnitude. We next discuss the proof of Theorem 15 followed by its formalization in the Coq proof assistant. It is noteworthy that while such proofs have been discussed in numerical analysis literature, we found several missing pieces during the formalization. Most facts about intermediate steps in the proof have just been stated in the numerical analysis literature without a rigorous proof. We therefore spent considerable time developing those proofs during our formalization. In

that regard, a contribution of this work is to provide a clean machine-checked proof of the main theorem and any intermediate lemma or fact that was required to close the proof of the Theorem 15.

3.4.2.1 Proof of Theorem 15

To prove Theorem 15, we first need to obtain a recurrence relation for the iterative convergence error at k^{th} step in terms of the initial iteration error $(x_o - x)$. Therefore,

Proof.

$$\begin{aligned} x_k - x &= -M^{-1}Nx_{k-1} + M^{-1}b - x \\ &= -M^{-1}Nx_{k-1} + M^{-1}(Ax) - x \quad [Ax \triangleq b] \\ &= -M^{-1}Nx_{k-1} + M^{-1}(M+N)x - x \qquad [M+N=A] \\ &= -M^{-1}Nx_{k-1} + M^{-1}Mx + M^{-1}Nx - x \\ &= -M^{-1}N(x_{k-1} - x) \quad [M^{-1}M \triangleq I] \end{aligned}$$

Taking norm of the vector on both sides, the iterative convergence error at the k^{th} step can be written in terms of the iterative convergence error at $(k-1)^{th}$ step as

$$||x_k - x|| = ||(-M^{-1}N)(x_{k-1} - x)|| \tag{3.17}$$

Since, the system is linear, equation (3.17) can be written in terms of the initial iteration error as

$$||x_k - x|| = ||(-M^{-1}N)^k(x_o - x)|| \tag{3.18}$$

□

Taking limits of the vector norms on both sides of equation (3.18),

$$\lim_{k\to\infty} ||x_k - x|| = \lim_{k\to\infty} ||(-M^{-1}N)^k(x_o - x)|| \tag{3.19}$$

If $x_o = x$, the iterative convergence error is zero trivially. The case $x_o \neq x$ is interesting and we can prove Theorem 15 by splitting it into two lemmas

Lemma 16. *For given matrices $M \in \mathbb{R}^{n\times n}$ and $N \in \mathbb{R}^{n\times n}$ respecting the regular splitting, i.e., $A = M + N$, the sequence of iterative solutions $\{x_k\}$ converges to x for any given initial vector x_0 if and only if the l^2 matrix norm of the iterated product of the iteration matrix,*

$(-M^{-1}N)^k$ *approaches zero as* $k \to \infty$*, i.e.,*

$$(\forall x_o, \lim_{k\to\infty} ||(-M^{-1}N)^k(x_o - x)|| = 0) \iff \lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0 \tag{3.20}$$

Proof. (**Necessity**): We need to prove that

$$\lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0 \implies (\forall x_o, \lim_{k\to\infty} ||(-M^{-1}N)^k(x_o - x)|| = 0)$$

Given x_o,

$$\begin{aligned}
\lim_{k\to\infty} ||(-M^{-1}N)^k(x_o - x)|| &\leq \lim_{k\to\infty} ||(-M^{-1}N)^k||\,||x_o - x||; \quad [\ ||Ax|| \leq ||A||\ ||x||\ ] \\
&= \left(\lim_{k\to\infty} ||(-M^{-1}N)^k||\right)\left(\lim_{k\to\infty} ||x_o - x||\right) \\
&= 0; \quad [\text{ since, } \lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0\ ]
\end{aligned}$$

(**Sufficiency**): We need to prove that

$$(\forall x_o, \lim_{k\to\infty} ||(-M^{-1}N)^k(x_o - x)|| = 0) \implies \lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0$$

We start by unfolding the definition of the norm of the iterative matrix

$$||(-M^{-1}N)^k|| = \sup_{(x_o - x)\neq 0} \frac{||(-M^{-1}N)^k(x_o - x)||}{||x_o - x||} \tag{3.21}$$

Therefore, we need to prove that

$$\lim_{k\to\infty} \left(\sup_{(x_o - x)\neq 0} \frac{||(-M^{-1}N)^k(x_o - x)||}{||x_o - x||} \right) = 0 \tag{3.22}$$

We can prove (3.22) by choosing an upper bound for the matrix norm in (3.21), proving that the limit of this upper bound converges to zero and then applying the sandwich theorem [125] for limits. We choose this upper bound as $\sum_{j<n} ||(-M^{-1}N)^k e_j||$, i.e.,

$$\sup_{(x_o - x)\neq 0} \frac{||(-M^{-1}N)^k(x_o - x)||}{||x_o - x||} \leq \sum_{j<n} ||(-M^{-1}N)^k e_j|| \tag{3.23}$$

where e_j is the unit vector corresponding to a principal direction in the cartesian coordinate system, i.e., $e_j = \mathbf{1}_j$. The vector $\mathbf{1}_j$ is a unit vector with the entry 1 in the j^{th} place and

other entries in the vector being 0. Therefore,

$$\lim_{k\to\infty}\left(\sup_{(x_o-x)\neq 0}\frac{||(-M^{-1}N)^k(x_o-x)||}{||x_o-x||}\right)\leq\lim_{k\to\infty}\sum_{j<n}||(-M^{-1}N)^k e_j||$$
$$\implies\lim_{k\to\infty}\left(\sup_{(x_o-x)\neq 0}\frac{||(-M^{-1}N)^k(x_o-x)||}{||x_o-x||}\right)\leq\sum_{j<n}\left(\lim_{k\to\infty}||(-M^{-1}N)^k e_j||\right)$$

We can quantify x_o in the hypothesis with $x+e_j, \forall j, j<n$. Therefore, $\forall j, j<n$, we have from the hypothesis,

$$\lim_{k\to\infty}||(-M^{-1}N)e_j||=0$$

Thus,

$$\sum_{j<n}\left(\lim_{k\to\infty}||(-M^{-1}N)^k e_j||\right)=0$$

We can then apply the sandwich theorem [125] for limits to prove that

$$\lim_{k\to\infty}\left(\sup_{(x_o-x)\neq 0}\frac{||(-M^{-1}N)^k(x_o-x)||}{||x_o-x||}\right)=0 \tag{3.24}$$

□

We justify the choice of the upper bound in (3.23) in the following proof.

Proof. We can decompose the vector x_o-x into its components along the principal axes in the cartesian coordinate system as

$$x_o-x=\sum_{j<n}(x_o-x)_j e_j$$

Therefore,

$$(-M^{-1}N)^k(x_o-x)=\sum_{j<n}(-M^{-1}N)^k(x_o-x)_j e_j \tag{3.25}$$

By taking a vector norm on both sides of (3.25),

$$||(-M^{-1}N)^k(x_o-x)||=||\sum_{j<n}(-M^{-1}N)^k(x_o-x)_j e_j||$$

Using the triangle inequality property of the vector norm, we get

$$||(-M^{-1}N)^k(x_o-x)||\leq\sum_{j<n}||(-M^{-1}N)^k(x_o-x)_j e_j|| \tag{3.26}$$

Since $(x_o - x) \neq 0$, $||x_o - x|| \neq 0$. Hence, dividing by $||x_o - x||$ on both sides of (3.26),

$$\begin{aligned}\frac{||(-M^{-1}N)^k(x_o - x)||}{||x_o - x||} &\leq \sum_{j<n} \frac{||(-M^{-1}N)^k(x_o - x)_j e_j||}{||x_o - x||} \\ &\leq \sum_{j<n} \frac{|(x_o - x)_j|}{||x_o - x||} ||(-M^{-1}N)^k e_j|| \\ &\leq \sum_{j<n} ||(-M^{-1}N)^k e_j||\end{aligned}$$

□

Here, we first use the absolute homogenity property ($||ax|| = |a|||x||$, for any scalar a and vector x) of the vector norm. Then we use the fact that

$$|(x_o - x)_j| \leq ||x_o - x||$$

Lemma 17. *For given matrices $M \in \mathbb{R}^{n\times n}$ and $N \in \mathbb{R}^{n\times n}$ respecting the regular splitting, i.e., $A = M + N$, the l^2 matrix norm of the iterated product of the iteration matrix, $(-M^{-1}N)^k$ approaches zero as $k \to \infty$ if and only if the spectral radius of the iteration matrix is less than* 1*, i.e.,*

$$\lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0 \iff \rho(-M^{-1}N) < 1 \tag{3.27}$$

The quantity $\rho(-M^{-1}N)$ is the spectral radius of the iteration matrix, $S = (-M^{-1}N)$ and is defined as

$$\rho(S) = \max_i\{ \mid \lambda_i(S) \mid \}, \forall i = 0, \ldots, (n-1)$$

where $\lambda_i(S)$ is the i^{th} eigenvalue of S. Therefore,

$$\rho(S) < 1 \iff (\forall i,\ i < n \implies |\lambda_i(S)|)$$

Proof. (**Sufficiency**): We need to prove that

$$\lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0 \implies \rho(-M^{-1}N) < 1$$

Since

$$\rho(-M^{-1}N) = \max_{0\leq i<n} |\lambda_i(-M^{-1}N)|,$$

$$\lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0 \implies (\forall i, 0 \leq i < n,\ |\lambda_i| < 1)$$

Applying,

$$\lim_{k\to\infty} |\lambda_i|^k = 0 \implies |\lambda_i| < 1,\ \forall i, 0 \leq i < n$$

to the goal statement, we need to prove:

$$\lim_{k\to\infty} |\lambda_i|^k = 0,\ \forall i,\ 0 \leq i < n$$

under the hypothesis $\lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0$. We now use the definition of an eigensystem, i.e.,

$$(-M^{-1}N)v_i = \lambda_i v_i$$

where v_i is an eigenvector corresponding to the eigenvalue λ_i. Therefore,

$$\begin{aligned}
\lim_{k\to\infty} |\lambda_i|^k = 0 &\implies \lim_{k\to\infty} |\lambda_i|^k ||v_i|| = 0 \\
&\implies \lim_{k\to\infty} ||\lambda_i^k v_i|| = 0; \quad [|\lambda_i|^k = |\lambda_i^k| \ \wedge\ |\lambda_i^k|\ ||v_i|| = ||\lambda_i^k v_i||] \\
&\implies \lim_{k\to\infty} ||(-M^{-1}N)^k v_i|| = 0; \quad [(-M^{-1}N)^k v_i = \lambda_i^k v_i]
\end{aligned} \tag{3.28}$$

But the compatibility relation for vector and matrix norms dictates,

$$0 \leq ||(-M^{-1}N)^k v_i|| \leq ||(-M^{-1}N)^k||\ ||v_i|| \tag{3.29}$$

Since, $||v_i|| \neq 0$ by definition of an eigensystem,

$$\lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0 \implies \lim_{k\to\infty} ||(-M^{-1}N)^k||\ ||v_i|| = 0 \tag{3.30}$$

We can then apply the sandwich theorem [125] to prove that

$$\lim_{k\to\infty} ||(-M^{-1}N)^k v_i|| = 0$$

(**Necessity**): We need to prove that

$$\rho(-M^{-1}N) < 1 \implies \lim_{k\to\infty} ||(-M^{-1}N)^k|| = 0.$$

To prove necessity, we obtain the Jordan decomposition of the iterative matrix, $S = (-M^{-1}N)$. From the Jordan normal form theorem [119], we know that there exists $V, J \in \mathbb{C}^{n\times n}$, V non-singular and J block diagonal such that:

$$S = VJV^{-1} \tag{3.31}$$

with

$$J = \begin{bmatrix} J_{m_1}(\lambda_1) & 0 & 0 & \dots & 0 \\ 0 & J_{m_2}(\lambda_2) & 0 & \dots & 0 \\ \vdots & \dots & \ddots & \dots & \vdots \\ 0 & \dots & 0 & J_{m_{s-1}}(\lambda_{s-1}) & 0 \\ 0 & \dots & \dots & 0 & J_{m_s}(\lambda_s) \end{bmatrix}; \; J_{m_i}(\lambda_i) = \begin{bmatrix} \lambda_i & 1 & 0 & \dots & 0 \\ 0 & \lambda_i & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_i & 1 \\ 0 & 0 & \dots & 0 & \lambda_i \end{bmatrix}$$

where $J_{m_i}(\lambda_i)$ is the Jordan block corresponding to the eigenvalue λ_i. Thus, $S^k = V J^k V^{-1}$. Since J is block diagonal,

$$J^m = \begin{bmatrix} J^k_{m_1}(\lambda_1) & 0 & 0 & \dots & 0 \\ 0 & J^k_{m_2}(\lambda_2) & 0 & \dots & 0 \\ \vdots & \dots & \ddots & \dots & \vdots \\ 0 & \dots & 0 & J^k_{m_{s-1}}(\lambda_{s-1}) & 0 \\ 0 & \dots & \dots & 0 & J^k_{m_s}(\lambda_s) \end{bmatrix}$$

Now, a standard result on the k^{th} power of a $m_i \times m_i$ Jordan block states that, for $k \geq m_i - 1$

$$J^k_{m_i}(\lambda_i) = \begin{bmatrix} \lambda_i^k & \binom{k}{1}\lambda_i^{k-1} & \binom{k}{2}\lambda_i^{k-2} & \dots & \binom{k}{m_i-1}\lambda_i^{k-m_i+1} \\ 0 & \lambda_i^k & \binom{k}{1}\lambda_i^{k-1} & \dots & \binom{k}{m_i-2}\lambda_i^{k-m_i+2} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_i^k & \binom{k}{1}\lambda_i^{k-1} \\ 0 & 0 & \dots & 0 & \lambda_i^k \end{bmatrix}$$

To prove that $\lim_{k\to\infty} ||S^k|| = 0$, we need to prove that $\lim_{k\to\infty} ||J^k|| = 0$, since V is non-singular, i.e. $||V|| \neq 0$. Since the 2-norm of a matrix is bounded above by the Frobenius matrix norm as: $||J^k||_2 \leq ||J^k||_F$, we can prove $\lim_{k\to\infty} ||J^k||_2 = 0$ by proving $\lim_{k\to\infty} ||J^k||_F = 0$. The Frobenius norm of matrix J^k is defined as:

$$||J^k||_F \triangleq \sqrt{\sum_i \sum_j |(J^k)_{(i,j)}|^2}$$

Thus, we need to prove that

$$\forall i \; j, \lim_{k\to\infty} |(J^k)_{(i,j)}|^2 = 0 \tag{3.32}$$

The zero entries of the Jordan block diagonal matrix tend to zero trivially. Since, $|\lambda_i| < 1$, the diagonal elements of the Jordan block, $J^k_{m_i}(\lambda_i)$ tend to zero. Proving that the off diagonal

elements tend to zero is more difficult.

We have to prove that $\lim_{k\to\infty} \binom{k}{m}|\lambda_i|^{k-m} = 0,\ \forall m \leq m_i - 1$. The function $f(k) = \binom{k}{m}$ increases as k increases while the function $g(k) = |\lambda_i|^{k-m}$ decreases as k increases since $0 < |\lambda_i| < 1$. Informally, it can be argued that $g(k)$ decreases at a faster rate than the function $f(k)$. Hence, the limit should tend to zero. But proving it formally in Coq was challenging. We need to first bound the function $f(k)$ with a function $h(k) = \frac{k^m}{m!}$ to obtain a sequence, $\frac{k^m}{m!}|\lambda_i|^{k-m}$, for which it would be easy to prove the limit. Therefore, we split the proof into proofs of two facts:

- $$\binom{k}{m} \leq \frac{k^m}{m!} \tag{3.33}$$
- $$\lim_{k\to\infty} \frac{k^m}{m!}|\lambda_i|^{k-m} = 0 \tag{3.34}$$

The inequality 3.33 follows the following proof

Proof.

$$\begin{aligned}\binom{m}{k} &= \frac{m!}{(m-k)!k!} \\ &= \frac{(m-k+1)(m-k+2)...(m-1)m}{k!}\end{aligned} \tag{3.35}$$

Since $(m-k+1) \leq m$, $(m-k+2) \leq m$, and so on, the product of terms in the numerator of (3.35) is bounded by m^k. Hence,

$$\binom{m}{k} \leq \frac{m^k}{k!}$$

□

In order to prove $\lim_{k\to\infty} \frac{k^m}{m!}|\lambda_i|^{k-m} = 0$, we use the ratio test for convergence of sequences. The formalization of ratio test has not yet been done in Coq to our knowledge. So, we formalized the ratio test since it provides an easier test for proving convergence of sequences as compared to first bounding the function with an easier function for which the convergence could be proved with the existing Coq libraries. The process of bounding the function $\Gamma(k) = f(k)g(k)$ with an easier function to prove convergence was challenging for us due to the behavior of $\Gamma(k)$. Using plotting tools like Wolfram plot or MATLAB plot, we observed that for $|\lambda_i| \leq 0.5$, the function $\Gamma(k)$ was monotonously decreasing, while for $0.5 < |\lambda_i| < 1$, $\Gamma(k)$ increases first and then decreases. Moreover, the location of the maxima of $\Gamma(k)$ in the

interval $0.5 < |\lambda_i| < 1$ depends of the number of iterations. Hence, bounding $\Gamma(k)$ with a monotonically decreasing function is challenging for $0.5 < |\lambda_i| < 1$. But we know that $\Gamma(k)$ decays eventually. For such scenarios, just comparing the terms $\Gamma(k+1)$ and $\Gamma(k)$ provides a much simpler test for the convergence of the sequence $\Gamma(k)$. This is where ratio test for the convergence of sequence comes to rescue. The ratio test for the convergence of sequences is stated as follows.

Theorem 18. *If (a_n) is a sequence of positive real numbers such that $\lim_{n\to\infty} \frac{a_{n+1}}{a_n} = L$ and $L < 1$, then (a_n) converges and $\lim_{n\to\infty} a_n = 0$.*

Proof. Let (a_n) be a sequence of real numbers such that $\lim_{n\to\infty} \frac{a_{n+1}}{a_n} = L$. Since (a_n) is a sequence of positive numbers, $0 \leq L$. By the Density of Real Numbers theorem, there exists a real r, such that $L < r < 1$.

Unfolding the definition of $\lim_{n\to\infty} \frac{a_{n+1}}{a_n} = L$, for $\epsilon = r - L > 0$, $\exists N \in \mathbb{N}$, such that forall $n \geq N$, $\left|\frac{a_{n+1}}{a_n} - L\right| < \epsilon = r - L$, which implies that,

$$-\epsilon < \frac{a_{n+1}}{a_n} - L < \epsilon \implies L - \epsilon < \frac{a_{n+1}}{a_n} < L + \epsilon \implies \frac{a_{n+1}}{a_n} < L + (r - L) = r$$

Therefore, $\forall n,\ n \geq N,\ a_{n+1} < a_n r$. This implies, $a_n r < a_{n-1} r^2$, and $a_{n-2} r^2 < a_{n-3} r^3, \ldots, a_{N+1} r^{n-N} < a_N r^{n-N+1}$. We thus obtain the following inequality:

$$a_{n+1} < a_n r < a_{n-1} r^2 < \ldots < a_{N+1} r^{n-N} < a_N r^{n-N+1}$$

Let $K = \frac{a_N}{r^N}$. Therefore $a_N r^{n-N+1} = K r^{n+1}$. Thus for $n \geq N$, we have that $a_n r < K r^{n+1}$, or rather $a_n < K r^n$. Since $0 < r < 1$, we have that $\lim_{n\to\infty} r^n = 0$. Then by the squeeze lemma, $\lim_{n\to\infty} a_n = 0$. □

In our case, the sequence $a_n = n^m |\lambda_i|^n$. Therefore, the ratio $\frac{a_{n+1}}{a_n} = \frac{(n+1)^m |\lambda_i|^{n+1}}{n^m |\lambda_i|^n}$. There-

fore,

$$\begin{aligned}\lim_{n\to\infty}\frac{a_{n+1}}{a_n} &= \frac{(n+1)^m|\lambda_i|^{n+1}}{n^m|\lambda_i|^n}\\ &= \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^m|\lambda_i|\\ &= |\lambda_i|\left[\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^m\right]\\ &= |\lambda_i|\left[\lim_{n\to\infty}\left(1+\frac{1}{n}\right)\right]^m\\ &= |\lambda_i|\end{aligned}$$

Therefore, $L = |\lambda_i|$, which we know is less than 1. Thus, we can apply Theorem 18 to prove that $\lim_{n\to\infty} n^m|\lambda_i|^n = 0$. Since $m!$ and $|\lambda_i|^m$ are non zero constants, $lim_{k\to\infty}\frac{k^m}{m!}|\lambda_i|^{k-m} = 0$. □

3.4.2.2 Formalization in Coq

We state Theorem 15 in Coq as follows:

```
Theorem iter_convergence:
 ∀(n:nat) (A: 'M[R]_n.+1) (b: 'cV[R]_n.+1) (M N : 'M[R]_n.+1),
  A \in unitmx →
  M \in unitmx →
  A = M + N →
  let x := (A^-1) *m b in
  (let S_mat:= RtoC_mat (- ( M^-1 *m N)) in
    (∀ (i: 'I_n.+1), (C_mod (lambda S_mat i) < 1)%Re)) ↔
   (∀ x0: 'cV[R]_n.+1,
       is_lim_seq (fun k:nat ⇒ vec_norm ((X_m k.+1 x0 b M N) − x)) 0%Re).
```

Since we deal with a generic case where a real matrix is allowed to have complex eigenvalues and eigenvectors, we also work in the complex field. RtoC_mat transforms a real matrix to a complex matrix so as to be consistent with types. In mathcomp , a complex number is defined on top of a real closed field [37, 38]. Thus, given a real matrix A, RtoC_mat transforms a real entry $A_{ij} : \mathbb{R}$ to a complex number $\tilde{A}_{ij} : \mathbb{C} := (A_{ij} + i * 0)$. In Coq, we formally define RtoC_mat as follows:

```
Definition RtoC_mat (n:nat) (A: 'M[R]_n):'M[complex R]_n :=
    \matrix_(i<n, j<n) ((A i j) +i* 0)%C.
```

Let us now discuss the hypothesis of the theorem statement. The first hypothesis states that the matrix A is invertible. Hence, there exists a unique solution to the linear system $Ax = b$. We define this solution vector x in Coq using the let binding

```
let x := (A^-1) *m b
```

The hypothesis A = M + N corresponds to the definition (3.3). As discussed earlier, we want the matrix M to be easily invertible so as to construct the iterative matrix $S = -(M^{-1}N)$. The invertibility condition is stated by the hypothesis

```
M \in unitmx
```

We define the iterative solution after k steps, x_k from the iterative system (3.2) using the **Fixpoint** operator in Coq

```
Fixpoint X_m (k n:nat) (x0 b: 'cV[R]_n.+1) (M N: 'M[R]_n.+1) : 'cV[R]_n.+1:=
  match k with
  | O ⇒ x0
  | S p ⇒ ((- ((M^-1) *m N)) *m (X_m p x0 b M N)) + ((M^-1) *m b)
  end.
```

The **Fixpoint** operator in Coq lets us define a recurrence relation, which in this case is given by the equation 3.4. We define the iterative matrix S in the let binding of iter_convergence in Coq.

In our formalization of the iterative convergence, we rely on the existing formalization of the Jordan canonical forms by Guillaume Cano and Maxime Dénès [32]. We use their definition of eigenvalues of a matrix derived from the roots of the characteristic polynomials of the Smith Normal form of a matrix A. We first define a sequence of eigenvalues as the diagonal entries of Jordan matrix.

```
Definition lambda_seq (n: nat) (A: 'M[complex R]_n.+1):=
  let sizes:= size_sum [seq x.2.-1 | x <- root_seq_poly (invariant_factors A)] in
   [seq (Jordan_form A) i i | i <- enum 'I_sizes.+1].
```

`root_seq_poly p` returns a sequence of pair of roots and its multiplicity, of the polynomial `p`. The `invariant_factors` are the polynomials in the diagonal of the Smith Normal form of a matrix. In this case, the sequence contains the pair of eigenvalues of matrix A and its multiplicity. The Jordan form matrix contains these eigenvalues in its diagonal which we extract using `lambda_seq`.

The i^{th} eigenvalue of a matrix A is then defined as the i^{th} component of the sequence of eigenvalues `lambda_seq`.

```
Definition lambda (n: nat) (A: 'M[complex R]_n.+1) (i: 'I_n.+1) :=
```

```
(@nth _ 0%C (lambda_seq A) i).
```

To take full advantage of the lemmas describing eigenvalues and eigenvectors as defined in `mathcomp`, we had to relate the definition of eigenvalue, `lambda`, and the one defined in `mathcomp`. In `mathcomp`, an eigenvalue a, of a matrix A is defined as a predicate

```
Definition eigenvalue : pred F := fun a ⇒ eigenspace a != 0.
```

stating that the eigenspace corresponding to this eigenvalue is non-zero.

```
Lemma Jordan_ii_is_eigen:
 ∀(n: nat) (A: 'M[complex R]_n.+1),
  ∀(i: 'I_n.+1), @eigenvalue (complex_fieldType _) n.+1 A (@nth _ 0%C (lambda_seq A) i).
```

The lemma Jordan_ii_is_eigen asserts that lambda satisfies the predicate eigenvalue, and is indeed an eigenvalue of a matrix A.

It should be noted that a square matrix A is assumed to be of size at least 1. This design choice is justified by Cano and Dénès in [32]. A square matrix only forms a ring when its size is at least one [32]. The ring property of a square matrix was really helpful in the formalization of diagonal block matrices by Cano and Dénès. Therefore, to use their formalization of canonical forms, we also stick with denoting the type of a complex matrix as `'M[complex R]_n.+1`.

As discussed earlier, we split the proof of the Theorem 15 into two lemmas 16 and 17. The statement of these lemmas are formalized in Coq as:
(Lemma 16):

```
(∀ x0: 'cV[R]_n.+1,
    is_lim_seq (fun m : nat ⇒ vec_norm ((X_m m.+1 x0 b A1 A2) − x)) 0%Re) ↔
    is_lim_seq (fun m:nat ⇒
          (matrix_norm (RtoC_mat ((− (A1^−1 *m A2))^+m.+1) ))) 0%Re
```

(Lemma 17):

```
is_lim_seq (fun m:nat ⇒ matrix_norm
        (RtoC_mat ((− ((A1^−1 *m A2)))^+m.+1 ))) 0%Re ↔
(let S_mat := RtoC_mat (− (A1^−1 *m A2)) in
    ∀i : 'I_n.+1, (C_mod (lambda S_mat i) < 1)%Re)
```

We have used the `is_lim_seq` predicate from the `Coquelicot` [28] library to define the limits for the sequence of solution vectors $\{x_k\}$ converging to x.

For the proof of the fact in the forward direction in the Lemma 16, we prove the following lemma in Coq

```
Lemma lim_max: ∀(n:nat) (A: 'M[R]_n.+1) (x: 'cV[R]_n.+1),
```

```
(∀ x0: 'cV[R]_n.+1,
 let v:= x0 − x in
 let vc:= RtoC_vec v in
 is_lim_seq (fun m: nat ⇒ vec_norm_C ((RtoC_mat (A^+m.+1)) ∗m vc)) 0%Re) →
 is_lim_seq (fun m:nat ⇒ matrix_norm (RtoC_mat (A^+m.+1))) 0%Re.
```

The lemma `lim_max` corresponds to the proof of (3.24). As discussed earlier, to prove `lim_max`, an important step was to decompose the vector $x_o - x$ into its components along the principal axis of the Cartesian coordinate system. We prove this fact in Coq using the following lemma statement

```
Lemma base_exp: ∀(n:nat) (x: 'cV[complex R]_n.+1),
  x = \big[+%R/0]_(i < n.+1) (x i 0 ∗: e_i i).
```

where we defined the principal unit vector `e_i i` in Coq as

```
Definition e_i {n:nat} (i: 'I_n.+1): 'cV[complex R]_n.+1 :=
  \col_(j < n.+1) (if (i==j :> nat) then (1 +i∗ 0)%C else (0 +i∗ 0)%C).
```

To prove the sufficiency in the Lemma 17, we had to prove the following lemma:

```
Lemma is_lim_seq_geom_nec (q:R):
    is_lim_seq (fun n ⇒ (q ^ n.+1)%Re) 0%Re → Rabs q <1.
```

While a lemma exists for the other direction in the Coquelicot [28] formalization of limits, lemma `is_lim_seq_geom_nec` was missing.

As discussed earlier, to prove sufficiency condition for iterative convergence, we had to formalize the ratio test for convergence of sequences which was missing in the existing Coq libraries. In Coq, we state Theorem 18 as:

```
Lemma ratio_test: ∀(a: nat → R) (L:R),
(0 < L ∧ L < 1) →
(∀ n:nat, (0 < a n)%Re) →
(is_lim_seq ( fun n:nat ⇒ ((a (n.+1))/(a n))%Re) L) →
is_lim_seq (fun n: nat ⇒ a n) 0%Re.
```

We then use the lemma `ratio_test` to formally prove

$$\lim_{m\to\infty} (m+1)^k x^{m+1} = 0$$

which we state in Coq as:

```
Lemma lim_npowk_mul_to_zero: ∀(x:R) (k:nat),
(0 < x)%Re → Rabs x < 1 →
is_lim_seq (fun m:nat ⇒ ((m.+1)%:R^k ∗ x^ m.+1)%Re) 0%Re.
```

To bound the binomial coefficient $\binom{m}{k}$ with $\frac{m^k}{k!}$, we formally state the inequality (3.33) as follows

```
Lemma choice_to_ring_le: ∀(n k:nat), (0<n)%N → (k<=n)%N →
    ('C(n,k)%:R <= (n%:R^+k / (k`!)%:R) :> complex R) .
```

Since the inequality holds in a complex field, we have to first convert the complex inequality to a real inequality. This was possible using the existing formalization of complex fields in `mathcomp`. Since the choice function and the factorials are naturals in Coq, we then define a lemma which proves that if the inequality holds for naturals, the inequality also holds in real field after a coercion is defined from naturals to reals. In Coq, we formalize the lemma as:

```
Lemma nat_ring_mn_le: ∀(m n:nat), (m<= n)%N → (m%:R <= n%:R)%Re.
```

Applying the lemma `nat_ring_mn_le`, we obtain the following inequality

$$\left(\binom{n}{k} \leq \frac{n^k}{k!}\right)\%N \tag{3.36}$$

It should be noted that the division in (3.36) is an integer division. The inequality (3.36) can easily be proved as discussed in the previous section. However, to use the inequality (3.36) in the proof of lemma `choice_to_ring_le`, one needs to be careful with converting an integer division to a real division.

To prove that each element of the Jordan block matrix converges to zero as in equation (3.32), we prove the following lemma in Coq

```
Lemma each_entry_zero_lim:
  ∀(n:nat) (A: 'M[complex R]_n.+1),
  let sp := root_seq_poly (invariant_factors A) in
  let sizes := [seq x0.2.-1 | x0 <- sp] in
  ∀i j: 'I_(size_sum sizes).+1,
  (∀ i: 'I_(size_sum sizes).+1 , (C_mod (nth 0%C (lambda_seq A) i) < 1)%Re ) →
  is_lim_seq (fun m: nat ⇒
    let block :=
    (fun n0 i1 : nat ⇒
        let lambda := (nth (0, 0%N) (root_seq_poly (invariant_factors A)) i1).1 in
        \matrix_(i2, j0)
        (('C(m.+1, j0 - i2))%:R * (lambda ^ (m.+1 - (j0 - i2))) *+ (i2 <= j0))) in
    (C_mod ((diag_block_mx sizes block) i j))^2) 0.
```

The lemma `each_entry_zero_lim` states that if the magnitude of each eigenvalue of a matrix A is less than 1, i.e., $|\lambda_i(A)| < 1,\ \forall i,\ 0 \leq i < N$, then the limit of each term in the expanded

Jordan matrix is zero as k approaches ∞. Here, the block diagonal matrix `diag_block_mx` takes an expanded Jordan block $J^k_{m_i}(\lambda_i), \forall i, 0 \leq i < N$ and constructs the Jordan matrix J^k. We then take a modulus of each entry of J^k and prove that its limit is zero as $k \to \infty$.

A key challenge we faced when proving the lemma each_entry_zero_lim was extracting each Jordan block of the diagonal block matrix. The diagonal block matrix is defined recursively over a function which takes a block matrix of size μ_i denoting the algebraic multiplicity of each eigenvalues λ_i. We had to carefully destruct this definition of diagonal block matrix and extract the Jordan block and the zeros on the off-diagonal entries. We can then prove the limit on this Jordan block by exploiting its upper triangular structure.

We state the lemma to extract a Jordan block from the block diagonal matrix as follows:

```
Lemma diag_destruct (R: ringType)
  (s : seq nat) (F : (∀ n, nat → 'M[R]_n.+1)):
  ∀ i j: 'I_(size_sum s).+1,
  (∃ k l m n,
    (k <= size_sum s)%N ∧ (l <= size_sum s)%N ∧
    (∀ p:nat, (diag_block_mx s (fun k l:nat ⇒ (F k l)^+ p.+1)) i j =
                  ((F k l)^+ p.+1) m n) ∧
    (diag_block_mx s F i i = (F k l) m m)) ∨
    (∀ p:nat, (diag_block_mx s (fun k l:nat ⇒ (F k l)^+ p.+1)) i j = 0).
```

where `size_sum` is the sum of the algebraic multiplicities of the eigenvalues and equals the total size of the matrix n. We prove this fact using the following lemma statement in Coq

```
Lemma total_eigen_val: ∀(n:nat) (A: 'M[complex R]_n.+1),
(size_sum [seq x.2.−1 | x <− root_seq_poly (invariant_factors A)]).+1 = n.+1.
```

The lemma `total_eigen_val` helps us get around the dimension constraint imposed by the design of the Jordan form of a matrix A. Limits of the off-diagonal elements can then be trivially proven to zero. This completes the proof of sufficiency condition for convergence of iterative convergence error.

3.4.3 Gauss–Seidel method

To prove the convergence of a Gauss–Seidel method, we need to prove that the spectral radius of S_G is less than 1. But computing the eigenvalues of S_G explicitly is almost impossible for a generic matrix. Therefore, we need an easier check to assert that the spectral radius of S_G is indeed less than 1. The *Reich theorem* [115] provides a sufficient condition for the spectral radius of S_G to be less than 1, for a real and symmetric coefficient matrix A, with all of the elements in its main diagonal positive. This condition provides a much easier check, which

is linear in time, and when layered with the Theorem 15, provides a sufficient condition for convergence of the Gauss–Seidel iteration for a real and symmetric coefficient matrix A, with all of the elements in its main diagonal positive.

We next discuss the formalization of the *Reich theorem*, followed by the main convergence theorem for the Gauss–Seidel iterate.

3.4.3.1 Reich theorem

Theorem 19. *[115] If A is real, symmetric nth-order matrix with all terms on its main diagonal positive, then a sufficient condition for all the n characteristic roots of $(-A_1^{-1}A_2)$ to be smaller than unity in magnitude is that A is positive definite.*

From an application point of view, only the sufficiency condition is important. This is because to apply Theorem (15), we only need to know that the magnitude of the eigenvalues are less than 1. Thus, to prove the convergence of Gauss–Seidel iteration, we first apply Theorem 15 to get the eigenvalue condition in the goal and then apply Theorem 19 to complete the proof. Since computing eigenvalues are not very trivial in most cases, the positive definite property of the matrix A provides an easy test for $|\lambda| < 1$ for Gauss–Seidel iteration matrix. The proof of necessity uses an informal topological argument that would be difficult to formalize. Therefore, it is enough to just formalize the sufficiency condition.

Next we present an informal proof [115] followed by its formalization in the Coq proof assistant.

Proof. Let z_i be the i^{th} characteristic vector of $-(A_1^{-1}A_2)$ corresponding to the characteristic root μ_i. Then

$$-(A_1^{-1}A_2)z_i = \mu_i z_i \tag{3.37}$$

Multiplying by $-(\bar{z_i}'A_1)$ on both sides,

$$(-\bar{z_i}'A_1)(-A_1^{-1}A_2)z_i = -\mu_i \bar{z_i}'A_1 z_i \tag{3.38}$$

where $\bar{z_i}'$ is the conjugate transpose of z_i obtained by taking the conjugate of each element of z_i followed by transpose of the vector. Equation (3.38) then simplifies to:

$$\bar{z_i}'A_2 z_i = -\mu_i \bar{z_i}'A_1 z_i; \quad [A_1 A_1^{-1} = I] \tag{3.39}$$

Consider the bi-linear form, $\bar{z_i}'Az_i$,

$$\bar{z_i}'Az_i = \bar{z_i}'(A_1 + A_2)z_i = \bar{z_i}'A_1 z_i + \bar{z_i}'A_2 z_i = \bar{z_i}'A_1 z_i - \mu_i \bar{z_i}'A_1 z_i = (1 - \mu_i)\bar{z_i}'A_1 z_i \tag{3.40}$$

Taking conjugate transpose of equation (3.40) on both sides,

$$\bar{z_i}' A z_i = (1 - \bar{\mu_i}) \bar{z_i}' A_1' z_i \tag{3.41}$$

Let D be the diagonal matrix defined as:

$$D_{ij} = \begin{cases} A_{ij} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{3.42}$$

It can be shown that

$$A_1' = D + A_2 \tag{3.43}$$

Substituting equation (3.43) in equation (3.41),

$$\begin{aligned} \bar{z_i}' A z_i &= (1 - \bar{\mu_i}) \bar{z_i}' (D + A_2) z_i \\ &= (1 - \bar{\mu_i}) \bar{z_i}' D z_i + (1 - \bar{\mu_i}) \bar{z_i}' A_2 z_i = (1 - \bar{\mu_i}) \bar{z_i}' D z_i + \frac{(1 - \bar{\mu_i})}{1 - \mu_i} \bar{z_i}' A z_i \end{aligned} \tag{3.44}$$

Simplifying equation (3.44),

$$(1 - \bar{\mu_i} \mu_i) \bar{z_i}' A z_i = (1 - \bar{\mu_i})(1 - \mu_i) \bar{z_i}' D z_i \tag{3.45}$$

But, $\bar{\mu_i} \mu_i = |\mu_i|^2$ and $(1 - \bar{\mu_i})(1 - \mu_i) = |1 - \mu_i|^2$ Hence, equation (3.45) simplifies to,

$$(1 - |\mu_i|^2) \bar{z_i}' A z_i = (|1 - \mu_i|^2) \bar{z_i}' D z_i \tag{3.46}$$

Since, the diagonal elements of A is positive, i.e., $A_{ii} > 0$, $\bar{z_i}' D z_i$ is positive definite, i.e., $\bar{z_i}' D z_i > 0$. Since $\bar{z_i}' D z_i > 0$ and $\bar{z_i}' A z_i > 0$, $|\mu_i| < 1$. □

Formalization in Coq: The Theorem 19 is formalized in Coq as follows:

```
Theorem Reich_sufficiency: ∀ (n:nat) (A: 'M[R]_n.+1),
  (∀ i:'I_n.+1, A i i > 0) →
  (∀ i j:'I_n.+1, A i j = A j i) →
  is_positive_definite A →
  (let S_G:= − ( (RtoC_mat (M_G A)^−1) ∗m (RtoC_mat (N_G A))) in
      (∀ i: 'I_n.+1, C_mod (lambda S_G i) < 1)).
```

where positive definiteness of a complex matrix A is defined as:

$$\forall x \in \mathbb{C}^{n \times 1}, Re\,[x^* A x] > 0$$

x^* is the complex conjugate transpose of vector x and $Re\ [x^*Ax]$ is the real part of the complex scalar x^*Ax. We defined `is_positive_definite` in Coq as:

```
Definition is_positive_definite (n:nat) (A: 'M[R]_n.+1):=
  ∀(x: 'cV[complex R]_n.+1), x != 0 →
    Re (mulmx (conjugate_transpose x) (mulmx (RtoC_mat A) x) 0 0) >0.
```

We compared our definition of a positive definite matrix with a related work from Pierre Roux [117]. While their work define positive definiteness for a real matrix, ours define it for a complex matrix. The definitions however are similar. The hypothesis

```
∀i:'I_n.+1, A i i > 0
```

states that all terms in the main diagonal of A are positive. The hypothesis

```
∀i j:'I_n.+1, A i j = A j i
```

states that the matrix A is symmetric.

3.4.3.2 Using the main theorem to prove convergence

We can then apply the theorem iter_convergence with Reich_sufficiency to prove convergence of the Gauss–Seidel iteration method. We formalize the convergence of the Gauss–Seidel iteration method in Coq as

```
Theorem Gauss_Seidel_converges:
  ∀(n:nat) (A: 'M[R]_n.+1) (b: 'cV[R]_n.+1),
  let x:= (A^−1) *m b in
   A \in unitmx →
   (∀ i : 'I_n.+1, 0 < A i i) →
   (∀ i j : 'I_n.+1, A i j = A j i) →
   is_positive_definite A →
    (∀ x0: 'cV[R]_n.+1,
        is_lim_seq (fun k:nat ⇒ vec_norm ((X_m k.+1 x0 b (M_G A) (N_G A)) − x)) 0%Re).
```

We next demonstrate the convergence of the Gauss–Seidel iteration on the example (3.16). We choose $N = 1$. Thus, we have a symmetric tri-diagonal coefficient matrix of size 3×3, which we will denote as A_{GS}. To show that iterative system for the system $A_{GS}x = b$ converges, we need to show that A_{GS} is positive definite. In Coq, we prove that A_{GS} is positive definite as the following lemma statement

```
Lemma Ah_pd: ∀(h:R), (0<h)%Re → is_positive_definite (Ah 2%N h).
```

Proving that A_{GS} is positive definite by definition for a generic N is tedious and does not add much to our line of argument. Hence, we chose to do it for A_{GS} of size 3×3. One

can perform the exercise for any choice of N and get the same result. The statement of convergence of Gauss–Seidel iteration method for the 3×3 matrix is stated in Coq as

```
Theorem Gauss_seidel_Ah_converges:
  ∀(b: 'cV[R]_3) (h:R),
  (0 < h)%Re →
  let A := (Ah 2%N h) in
  let x:= (A^−1) ∗m b in
    ∀x0: 'cV[R]_3,
        is_lim_seq (fun k:nat ⇒ vec_norm ((X_m k.+1 x0 b (A1 A) (A2 A)) − x)) 0%Re.
```

This closes the proof of the convergence of the Gauss–Seidel iteration for the model problem.

3.4.4 Jacobi method on the model problem

We next apply the Theorem 15 to prove convergence of the Jacobi iteration on the model problem 3.16. As discussed earlier, the iteration matrix for a Jacobi iteration method is $S_J = I - D^{-1}A_J$. We choose $P = 1$, thereby obtaining a 3×3 matrix system, like we did for the Gauss–Seidel iteration. However, the Jacobi matrix, A_J, that we will consider will be a negation of the coefficient matrix, A in (3.16). We essentially shift the negation to the right hand side of (3.16). The reason we do this is that, to define the explicit form of eigenvalues of the Jacobi matrix for the example problem (3.16), we need to take the square root of the off-diagonal elements of A_J. In Coq, the square root definition of a real number has a precondition that the real number must be non-negative. Thus, we require the off-diagonal elements of A_J to be non-negative. This does not change the problem in consideration, but transforms it into a form that is easy to formalize in Coq.

The eigenvalues of $I - D^{-1}A_J$ would then be $1 + \frac{h^2}{2}\lambda(A_J)$. For a tri-diagonal matrix with a on the lower diagonal, b on the diagonal and c on the upper diagonal entries, $\lambda_i(A_J)$ is defined as

$$\lambda_i(A_J) = b + 2\sqrt{ac}\left[\cos\left(\frac{m\pi}{P+1}\right)\right], \quad 0 \leq i < P,\ 1 \leq M \leq P \tag{3.47}$$

For the matrix A_J, $b = -\frac{2}{h^2}$, $a = c = \frac{1}{h^2}$.

$$\lambda_i(A) = -\frac{2}{h^2} + 2\frac{1}{h^2}\left[\cos\left(\frac{m\pi}{P+1}\right)\right] = \frac{-2}{h^2}\left[1 - \cos\left(\frac{m\pi}{P+1}\right)\right] \tag{3.48}$$

Hence,

$$|\lambda_i(S_J)| = \left|1 + \frac{h^2}{2}\lambda_i(A_J)\right| = \left|\cos\left(\frac{m\pi}{P+1}\right)\right| \tag{3.49}$$

Thus, to prove the convergence of the Jacobi iteration for this model problem, we need to

prove that

$$\left|\cos\left(\frac{m\pi}{P+1}\right)\right| < 1 \tag{3.50}$$

Formalization in Coq: In Coq, we formalized the condition (3.50) as:

```
Theorem eig_less_than_1:
  ∀(n:nat) (i: 'I_n.+1) (h:R),
  (0 < h)%Re → (0 < n)%N → (C_mod (lambda_J i n h) < 1)%Re.
```

where the eigenvalues of the iterative matrix S_J is defined as below:

```
Definition lambda_J (m P:nat) (h:R) :=
    (b P h) + 2* sqrtc(p P h)* RtoC (cos((m.+1%:R * PI)*/ P.+2%:R)).
```

In Coq the natural numbers start from 0, hence we increment m and P by one as compared to the formula (3.47).

Since the above definition of eigenvalues hold for a tri-diagonal matrix, we formally prove that $S_J = I - D^{-1}A_J$, thereby preserving the tri-diagonal structure of A_J. This is stated in Coq as:

```
Lemma S_mat_simp: ∀(n:nat) (h:R), (0<h)%Re →
S_mat_J n h = RtoC_mat (addmx 1%:M (oppmx (mulmx (invmx (M_J n h)) (Ah n h)))).
```

where `RtoC_mat` is a coercion from a real to a complex matrix as discussed earlier. This lemma holds for $0 < h$. This condition is required since h denotes the discretization step size i.e. $h = x_{i+1} - x_i$.

Since $\lambda_i(S_J)$ is in general complex, to prove (3.50), we need to prove that the real part of $\lambda_i(S_J)$ is equal to the the eigenvalue of A_J. We prove this formally in Coq as the following lemma:

```
Lemma lambda_simp: ∀(m P:nat) (h:R), (0<h) → (0< P) →
  Re (lambda_J m P h) = (1 + ((h^2)/2) *Lambda_Ah m P h)).
```

where Lambda_Ah m P h is the m^{th} eigenvalue of the matrix A_J. The hypothesis $0 < P$ is a constraint on the number of internal points in the domain. The equation (3.15) dictates that there must be at least one point in the interior.

One of the requirements of defining an iterative system is that the matrix M be invertible. We prove this formally in Coq for this example as the following lemma statement:

```
Lemma M_invertible: ∀(n:nat) (h:R), (0<h) → (M_J n h) \in unitmx.
```

To prove the above lemma, we have to directly prove that $MM^{-1} = I$. This is stated in Coq as the following lemma statement:

```
Lemma invmx_A1_mul_A1_1:∀ (n:nat) (h:R),
  (0<h) → (M_J n h) *m (M_J n h)^−1 = 1%:M.
```

Since M is a diagonal matrix, we can obtain a direct formulation of the inverse of M. This is stated in Coq as the following lemma statement:

```
Lemma invmx_A1_J: ∀(n:nat) (h:R), (0<h) → (M_J n h)^−1 = ((−(h^+2)/2) *: 1%:M).
```

The lemma `invmx_A1_J` was the last piece in the puzzle to complete the proof of invertibility of M. We apply the Theorem 15 to the 3×3 example to prove convergence of Jacobi iteration matrix on this example.

```
Theorem Jacobi_converges: ∀(b: 'cV[R]_3) (h:R), (0 < h) →
    let A := (Ah 2h) in
    let x := (A^−1) *m b in
    (∀ x0: 'cV[R]_3, is_lim_seq (fun k:nat ⇒ (X_m k.+1 x0 b (M_J 2 h) (N_J 2 h)) − x).
```

Here, we instantiate the eigenvalue `lambda` with the `lambda_J` using

```
Hypothesis Lambda_eq: ∀(n:nat) (h:R) (i: 'I_n.+1),
  let S_mat := RtoC_mat (− ( (M_J n h)^−1 *m (N_J n h) )) in
  lambda S_mat i = lambda_J i n h.
```

We further formally prove that `lambda_J` is an eigenvalue of the Jacobi matrix, A_J, using the `eigenvalue` predicate in `mathcomp`.

```
Lemma lambda_J_is_an_eigenvalue:
  ∀(h:R), (0 < h) →
  let S_mat := RtoC_mat (oppmx (invmx (A1_J 2 h) *m A2_J 2 h)) in
  (∀ i: 'I_3, @eigenvalue (complex_fieldType _) 3 S_mat (lambda_J i 2 h)).
```

This closes the proof of iterative convergence for Jacobi iteration on the model problem.

3.5 Conclusion

In this work we formalized a *generalized theorem* about convergence of the solutions of an iterative algorithm to the *true numerical solution (direct solution)*. We formalized the sufficient and necessary conditions for asymptotic convergence of a sequence of iterative solutions $\{x_k\}$ to the direct solution x of the original system $Ax = b$, for generic iterative algorithms, which respect the *regular splitting* [119]. To prove convergence, one needs to construct an iterative matrix $S \triangleq -M^{-1}N$, and prove that its eigenvalues are less than 1 in magnitude. We leverage the existing development of the Jordan canonical forms in Coq [32], to develop our *generalized* framework for verifying asymptotic iterative convergence

in the field of reals. In this process, we clarify various details in the proof of convergence, which are missing in the classical numerical analysis literature. We then instantiate this *generalized* theorem to two classical iterative methods – the Gauss–Seidel method, and the Jacobi method. Since it is cumbersome to compute the eigenvalues of a generic matrix system, and verify that its magnitude is less than 1, we provide a much easier check for convergence, especially for the Gauss–Seidel method, which relies on the matrix structure. We formally prove that if the *coefficient matrix* A is *real, symmetric with all of the terms in its main diagonal positive*, then a sufficient condition for the spectral radius of S to be less than 1 is that A is *positive definite*. This theorem is called the *Reich theorem* in classical numerical analysis literature [115]. Positive definiteness of a matrix is a much easier check as compared to computing the eigenvalues. We then show that by composing the Reich theorem with the generalized iterative convergence theorem, we can prove convergence for the Gauss–Seidel iteration for a real, symmetric matrix A with all its main diagonal elements positive. We then instantiate this theorem with our model problem to prove convergence of the Gauss–Seidel method on it. We also proved convergence of the Jacobi iterative process for our model problem. During our fomalization, we develop a library in Coq to deal with complex vectors and matrices. We defined absolute values of complex numbers, common properties of complex conjugates and operations on conjugate matrices and vectors. This development leverages the existing formalization [37, 38] of complex numbers and matrices in `mathcomp`. The overall length of the Coq code and proofs is about 8.5k lines of code. It took us about 8 person-months of full time work for the entire formalization.

An important point to note here is that the analysis done in this chapter is done in the field of reals. However, the actual implementation of an iterative algorithm is done in *finite precision*. We therefore, study the effect of *floating point* errors on iterative convergence. Since we cannot prove asymptotic convergence in floating-point arithmetic, we prove a bound on the *iterative convergence error* in presence of *rounding errors*. This requires us to compute a bound on the *real true numerical solution* x. We therefore prove a corollary to the Theorem 15 which states that $||x|| = \lim_{k\to\infty} ||x_k||$ when A is invertible and $\rho(S) < 1$.

Corollary 20. *Given a coefficient matrix A, a right hand side vector b, and matrix splittings M and N, such that A and M are invertible and M and N respect the regular splitting, $A := M + N$, if all the eigenvalues of the iteration matrix, $S \triangleq -M^{-1}N$, are less than 1 in magnitude, then $||x||_2 = \lim_{k\to\infty} ||x_k||_2$ for any arbitrary initial vector x_0.*

We state and proof the Corollary 20 in Coq as

```
Theorem x_limit_eq:
  ∀(n:nat) (A: 'M[R]_n.+1) (b: 'cV[R]_n.+1) (M N : 'M[R]_n.+1),
```

```
A \in unitmx →
M \in unitmx →
A = M + N →
let x := (A^-1) *m b in
(let S_mat:= RtoC_mat (- ( M^-1 *m N)) in
      (∀ (i: 'I_n.+1), (C_mod (lambda S_mat i) < 1)%Re)) →
(∀ x0: 'cV[R]_n.+1,
  Lim_seq (fun m:nat ⇒ vec_norm (X_m m.+1 x0 b M N)) = vec_norm x).
```

This Corollary 20 comes handy when we bound $||x||$ using the inputs: the *coefficient matrix* A, the *right hand side vector* b, the *tolerance* τ, and the *iteration count* k, as will be discussed in section 4.5.2.2. The proof of lemma 27 uses an intermediate lemma which proves that $||x||_\infty = \lim_{k\to\infty} ||x_k||_\infty$ for a Jacobi iterative algorithm when $x_o = \mathbf{0}$, and $||D^{-1}N||_\infty < 1$. Since $\rho(S) = \rho(D^{-1}N) \leq ||D^{-1}N||_\infty$ (not proved in Coq) and using the relation between the $||.||_2$ norm and the $||.||_\infty$ norm, the corollary 20 can be specialized to the Jacobi iteration to prove this intermediate lemma. *Therefore, the corollary 20 can be used in the proof for bound of* $||x||$ *for a general iteration scheme, which is one of the future directions of our work on iterative covergence in the presence of rounding errors.*

This work could also be extended to verifying solutions of non-linear systems. Most physical systems behave non-linearly, and the analysis of these non-linear systems is usually done by linearlizing it around an optimal trajectory. Once we have a linearized system, we can then plug that in our framework and analyze the convergence error when iterative solvers are used to obtain their solution.

CHAPTER 4

Iterative Convergence with Rounding Error

4.1 Introduction

So far, our analysis on stationary iterative methods have been done in the field of reals. But the actual implementation of an iterative algorithm is done in a finite precision machine. This introduces another source of error called the *floating point error*. Besides, the implementation of an algorithm is prone to *programming errors* like null pointer de-referencing and data-type handling. Because these iterative methods are often used as subroutines deep within larger computational libraries and solvers, it is important that we study the effect of *floating point errors* on the convergence of solutions from iterative algorithms, while guaranteeing that we rule out *programming errors*. More importantly, it is useful to be able to prove theorems of the form: "Given inputs A and b with certain properties, the algorithm will converge to a tolerance τ within k iterations". This allows the user to figure out reasons as to why a subroutine using these methods failed to converge, and also make the development of computational libraries more robust.

Most of the floating-point analysis on stationary iterative methods use a simplified floating-point model that omits subnormal numbers [68], or the analysis is for a model of an algorithm [86] but not the actual software. And when one reaches correctness and accuracy proofs of actual software, it is useful to have machine-checked proofs that connect in a machine-checkable way to the actual program that is executed, for programs can we complex and as programs evolve one must ensure that their correctness theorems evolve with them.

In this chapter, we discuss the development of an end-to-end verification framework [129] using the state-of-the art tools like VCFloat2 [87] for semi-automated analysis of floating-point errors, Verifiable Software Toolchain (VST) [33] analyzing programming errors, and the MathComp library [100] for matrix analysis. The problem that we address here is of the form "Given inputs A and b with certain properties, the algorithm will converge to a

tolerance τ within k iterations". We focus on Jacobi iteration algorithm applied to *strictly diagonally dominant matrices*, i.e., in which in each row the magnitudes of the diagonal element exceeds the sum of the magnitude of the off-diagonals. Strict diagonal dominance is a simple test for invertibility and guarantees convergence of Jacobi iteration in exact arithmetic. Strictly diagonally dominant matrices arise in cubic spline interpolation [4], analysis of Katz centrality in social networks [85], Markov chains associated with PageRank and related network analysis methods [55], and market equilibria in economic theory [104], among other domains.

Contributions: We present both a Coq *functional model* of floating-point Jacobi iteration (at any desired floating-point precision) and a C program (in double-precision floating-point), with Coq proofs that:

- the C program (which uses efficient sparse-matrix algorithms) correctly implements the functional model (which uses a simpler matrix representation);
- for any inputs A, b and desired accuracy τ that satisfy the *Jacobi preconditions* for a given natural number k, the functional model (and the C program) will converge within k iterations to a vector x_k such that $||Ax_k - b||_2 < \tau$;
- this computation will not overflow into floating-point "infinity" values;
- and the *Jacobi preconditions* are natural properties of A, b, τ, k that (1) are easily tested and (2) for many natural engineering problems of interest, are guaranteed to be satisfied.

We thus provide a first convergence proof of Jacobi that takes into account floating-point underflow and overflow, and a first machine-checked connection to a real program. Software packages not written in C can still be related to our functional model and make use of our floating-point convergence and accuracy theorems. And even for inputs that do not satisfy the Jacobi preconditions, we have proved that our C program correctly and robustly detects overflow.

Specific contributions from co-authors: This work [1] has been accepted for publication at the 16^{th} *Conference on Intelligent Computer Mathematics, 2023*, and is a joint work with my collaborators: Prof. Andrew Appel from Princeton University, Ariel Kellison from Cornell University and Prof. David Bindel from Cornell University. Prof. Andrew

[1] Our Coq formalization is available at https://github.com/VeriNum/iterative_methods/tree/v0.1.0

Appel worked on the implementation of the Jacobi iteration algorithm in C using the sparse matrix-vector operations and the proof of correctness of this implementation with respect to a functional model, using the Verifiable Software Toolchain [33]. Ariel Kellison worked on defining the dot-product operation using *fused-mutiply add (FMA)*, and formalized a bound on the forward error for dot-product operation and conditions for absence of overflow for this operation. Prof. David Bindel provided us with notes on error analysis for this work.

We will first provide an overview of the floating-point arithmetic, and then discuss about our error analysis framework.

4.2 Introduction to floating-point arithmetic

Floating point arithmetic is ubiquitous in computer systems. Numerical quantities are handled by computer using floating-point arithmetic. Implementation of many safety critical software systems like control software, weather forecasts, trading algorithms, embedded systems depend on floating-point computations. In the past, errors in rounding have led to various disastrous consequenes, some of which are enumerated below:

- **Trading companies:** An infamous case study of the effect of rounding errors in trading is the Vancouver Stock Exchange [74]. This stock exchange set up a new index in January 1982, with a value set to 1,000. The index was updated with each trade but the index was truncated to three decimal places. Since this rounding operation was done about 3,000 times a day, the error accumulated, resulting in the exchange to close at 524.811 points on Friday, November 25th, 1983. When, the rounding error was fixed, and the exchange reopened, the index was 1098.892. *This difference was solely due to the rounding error bug.*

- **Defense:**

    i Another infamous disaster due to floating point errors has been the failure of the Patriot missile system to intercept the Iraqi "Scud" missile during the Gulf war on February 25, 1991 [60]. The Scud missile hit the barracks, killing 28 people. This failure was attributed to an error in differencing of floating point numbers obtained by converting and scaling an integer timing register. This error was traced to the software powering the clock of the system, which recorded time in one tenth of a second but stored the data as an integer. It converted the time into a 24-bit floating point number to do this. However, this system experienced gradually decreasing accuracy due to the errors accumulated from rounding the

time. As a result, the Patriot missile system was not able to intercept missiles after 20 hours of continuous use.

ii A misplaced decimal point in the calculation led to a flawed design of the first submarine built under the Spain's S-80 submarine program [98] in 2003. After the submarine was designed, it was discovered that this submarine was about 70 tons heavier than expected. This mistake is expected to have incurred a total loss of about €14 million.

iii The Ariane 5 rocket exploded 36 seconds after the lift-off on June 4, 1996 which led to a loss of about $ 370 million to the European Space Agency. The problem was in the Inertial Reference System, which produced an operation exception trying to convert a 64-bit floating-point number to a 16-bit integer [15].

- **Parliamentary election:** The effect of rounding error was also felt in the Parliamentary elections for the state of Schleswig– Holstein in Germany [74]. According to the rules of the German election, no party with less than 5% of votes was allowed to have a candidate seated in the parliament. There were two parties in close race – the *Green party* and the *Social Democrats (SPD) party*. The Green party secured 4.97% of votes, but program rounded it to 5%, which put the Green party to an advantage. This rounding error was discovered later in the midnight, and after re-calculation, the Green party failed to satisfy the 5% clause, and SPD party got a one seat majority in the parliament.

- **Embedded systems:** The infamous "Y2K bug" or the *Millenium bug* affected many devices containing computer chips, ranging from elevators to temperature-control systems in commercial buildings to medical equipment [42]. The problem was that up until 1990s, many computer programs were designed to abbreviate four digits of year into two digits to save memory space. For example, the year 1998 was saved as 98. This issue first surfaced when the clock struck midnight on January 1, 2000. The year 2000 was now saved as 00, which could also represent the year 1900. This anomaly created a lot of havoc, and as a result about $ 300 billion was spent on upgrading the computer systems to fix this anomaly.

These are a few of the many practical examples that demonstrate how critical the issue of floating-point error is. Therefore, it is of utmost importance that we study this error rigorously, and formalize the conditions under which such critical errors could be caught earlier in the design cycle.

4.2.1 Theory of floating-points

We will be providing an overview of the IEEE-754 standard [75], which defines floating-point formats, attributes and rounding, exceptional values, and exceptional handling. This discussion is mostly adopted from the book on Computer Arithmetic and formal proofs [31].

4.2.1.1 Overview of floating point arithmetic

FP formats Let us consider an integer β called the *radix*. In the binary number system, $\beta = 2$, while in the decimal system, $\beta = 10$. In a simplified model, an FP number is just a real value $m\beta^e$ satisfying constraints on m and e. The integer m is called the *significand*, and the integer e is called the *exponent*. The exponent e is bounded as: $e_{\min} \leq e \leq e_{\max}$. The significand, m satisfies the bound $|m| < \beta^p$, where p is called the *precision*. Hence, a floating-point number is a real value representable in the considered floating-point format:

$$m\beta^e; \quad \text{such that } |m| < \beta^p \ \wedge \ e_{\min} \leq e \leq e_{\max}$$

The standard floating-point formats described by the IEEE-754 standards [75] are illustrated in the Table 4.1.

	Single	Single-extended	Double	Double-extended
p	24	≥ 32	53	≥ 64
$e_{\max}$	$+127$	≥ 1023	$+1023$	≥ 16383
$e_{\min}$	-126	≤ -1022	-1022	≤ -16382

Table 4.1: IEEE-754 Format Parameters [56]

The floating point finite numbers are divided into two classes:

- **normal numbers:** The normal numbers satisfy the following constraints on m and e:

  $$\beta^{p-1} \leq |m| < \beta^p \ \wedge \ e_{\min} \leq e \leq e_{\max}$$

  Normal numbers are floating-point numbers greater than or equal to $\beta^{p-1+e_{\min}}$.

- **sub-normal numbers:** The subnormal numbers are represented as $m\beta^{e_{\min}}$ with $|m| < \beta^{p-1}$. Equivalently, subnormal numbers are floating point numbers that are smaller than $\beta^{p-1+e_{\min}}$.

The numbers in an FP format form a discrete finite set that may be represented on the real axis, as illustrated in the Figure 4.1.

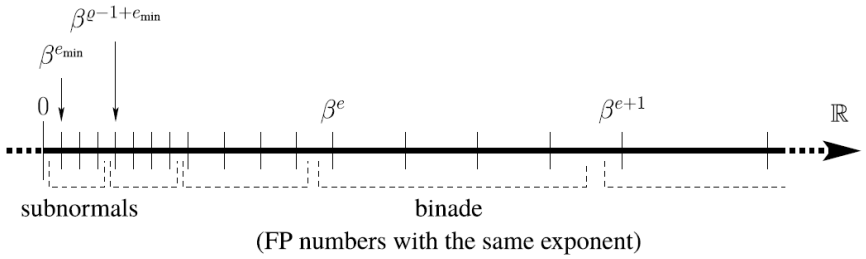

Figure 4.1: Distribution of floating-point numbers over the real axis. Figure from [31]

The maximum representable floating-point number is therefore given by $F_{\max} = (\beta^p - 1)\beta^{e_{\max}}$. If the exponent of a floating point number is larger than $e_{\max}$, i.e. $e > e_{\max}$, we have an *overflow*. When a floating-point number has a value smaller than the minimum normal floating-point number, i.e., $< \beta^{p-1+e_{\min}}$, we have an *underflow*, and may get a subnormal number.

Unit in the last place: As illustrated in the Figure 4.1, when the exponent increases, the distance between the floating-point numbers also increases. In fact, the distribution of the floating-point numbers is logarithmic, i.e., denser closer to zero, and scarcer away from zero. For a positive floating-point number x, *ulp*, or the "unit in the last place" is defined as the δ_x such that there is no floating-point number between x and $x + \delta_x$. More precisely, if $x = m\beta^e$, then

$$ulp(x) = \beta^e$$

Rounding: Most floating-point computations are not exact in general. When a result of such FP computations are between two floating-point numbers, one of the bracketing floating-point numbers must be returned instead. The choice of this floating-point number depends on the *rounding mode*. The IEEE-754 standard recommends five rounding modes for a floating-point number x:

- Rounding of x to $+\infty$
- Rounding of x to $-\infty$
- Rounding of x to 0
- Rounding *to nearest*, i.e., the result is expected to be the floating-point number nearest to x. When the real to be rounded is exactly at the mid-point of two floating-point numbers, tie-breaking is done using the following rules

a Tie breaking to even: the chosen floating-point number is the one with even significand

b Tie breaking away from zero: the chosen floating-point number is the one with larger magnitude

For binary formats, the default tie-breaking rule, is *rounding to nearest, tie breaking to even.*

Floating-point operations: The IEEE-754 mandates *correct rounding* for basic operations: addition, subtraction, multiplication, division, and square root. *Correct rounding* means that the operations are performed as if they are done over reals, and then rounded to a floating-point number according to one of the rounding rules discussed earlier. For other functions like exp, sin, and pow, IEEE-754 standard *only recommends* correct rounding, since correct rounding is much harder for these functions due to the *table maker's dilemma* [95], which is the problem of unknown cost of rounding transcendental functions. In these cases, one needs to be very accurate during intermediate computations to be able to decide which FP number is the closest.

Error analysis: Although *correct rounding* is the best we might expect from a floating-point computation, it is not exact, and the results of floating-point computations are often truncated, depending on the precision. This introduces an error called the *round-off error.* We might have to bound this round-off error to assess the quality of truncated result. There are two ways of representing this error – *absolute error*, and *relative error.* For a real number x and its approximation $\tilde{x}$, the absolute error is defined as $|\tilde{x} - x|$, and the relative error is defined as $(|\tilde{x} - x|)/x$, provided $x \neq 0$. *In this chapter, we will be discussing only the absolute error analysis for the Jacobi iteration.*

4.2.1.2 Issues and anomalies with floating points

Floating-point computations are infamous for anomalies, some of which are enumerated below:

- Floating-point numbers do not follow the rules of real arithmetic. For instance, the addition is not associative, i.e, $(x + (y + z))$ is different from $(x + y) + z$.

```
x=0.1+(0.2+0.3);
y=(0.1+0.2)+0.3;
sprintf('%.17f',x)
% ans = '0.59999999999999998'
```

```
sprintf('%.17f',y)
% ans = '0.60000000000000009'
```

The above MATLAB code illustrates that the floating-point addition is non-associative.

- Adding very big values to very small values results in inaccuracies. The small numbers get lost. For example, for 32-bit floats, $262144.0 + 0.01 = 262144.0$. This is because there are no 32-bit floating-point numbers between 262144.0 and 262144.03125. So, $262144.0 + 0.01$ is rounded to the nearest, 262144.0.

  Similarly, subtracting two floating-point numbers of similar magnitude results in inaccuracies. Such cancellations are called *catastrophic cancellations* [106], and have to dealt with carefully. One way to deal with catastrophic cancellation is to re-arrange terms, or do some re-writings, as in the tool Herbie [111].

  Example illustrating catastrophic cancellation: Suppose that we are computing $b^2 - 4ac$ for $b = 3.34,\ a = 1.22$, and $c = 2.28$.

$$b^2 = (3.34)^2 = 11.1556 \approx 1.12 \times 10^1$$

$$4ac = 4 \times 1.22 \times 2.28 = 4.88 \times 2.28 = 11.1264 \approx 1.11 \times 10^1$$

$$b^2 - 4ac \approx 1.12 \times 10^1 - 1.11 \times 10^1 = 0.01 \times 10^1 = 1.00 \times 10^{-1}$$

  But the exact value is $0.0292 = 2.92 \times 10^{-2}$. This has a relative error of

$$\frac{|0.0292 - 0.1|}{0.0292} = \frac{0.0708}{0.0292} = 2.242\ldots > 240\%$$

  Thus, subtracting two floating-point numbers that are very close to each other leads to loss of significant digits.

- There are two zeros $+0$ and -0, and they are not represented in the same way. Therefore, when we multiply a finite floating-point number x with $+0$, we need to consider that the result could be either $+0$ or -0, depending on the sign of x. *In our proofs for this project, we had to write lemmas that would consider the cases of* $+0$ *and* -0 *separately.*

- NaN/infinity values can propagate and cause chaos. For instance, in floating-point it is not the case that $\forall y,\ 0 \cdot y + s = s$, when y is ∞ or NaN.

Therefore, floating-point computations must be dealt with carefully, and the subtleties be treated in a *formal* and rigorous way.

4.2.1.3 Floating points in Coq

The floating point formats and operations have been formalized in the *Flocq* [30] library in Coq. This library follows an approach similar to the IEEE-754 standard [75] in providing several levels of abstraction to characterize the various ways to represent and manipulate floating-point numbers. Although, there is usually not a one-to-one correspondence between the various ways to represent FP formats in Coq and the IEEE-754 standard. An example of this difference is in the way Flocq treats infinity and the way IEEE-754 standard treats infinity. The level 1 in the IEEE-754 standard includes infinities as possible values of floating-point numbers, while Flocq supports only real numbers as an abstract representation of floating-point numbers and treats infinities separately.

Discussion about generic floating-point format in Flocq: Formats are formalized in Flocq as predicates over real numbers. They mostly deal with a subset of rational numbers that can be represented as $m\beta^e$, where β is a fixed radix and m and e are two integers representing the *significand* and *exponent*, respectively.

A *radix* is defined in Flocq using a record data structure containing an integer radix_val and a proof that this integer is at least 2, so as to avoid the degenerate cases $\beta = 0;\ \beta = 1$.

```
Record radix := {
    radix_val : Z;
    radix_prop : Zle_bool 2 radix_val = true
}
```

The bpow function performs an exponentiation by a signed integer constant e, and returns a real number, β^e. In Flocq, this function is defined as

```
Definition bpow : radix → Z → R.
```

A float type is defined as a record type in Flocq, which contains two fields: an integer significand and an exponent, and is parameterized by the radix of the format.

```
Record float (beta : radix) := Float {Fnum : Z; Fexp : Z }.
```

Flocq also defines an F2R function which converts an element of the float type to a real number.

```
Definition F2R {beta : radix} (f : float beta) := Z2R (Fnum f) * bpow beta (Fexp f).
```

Mathematically, F2R function takes $\beta,\ m,\ e$ and returns the real $m \cdot \beta^e$.

A generic format is then defined in Flocq as

```
Definition generic_format (x:R) :=
  x = F2R (Float beta (Ztrunc (scaled_mantissa x)) (cexp x)).
```

Mathematically, this is equivalent to

$$\exists m \in \mathbb{Z},\ x = m \cdot \beta^{cexp(x)}.$$

The integer m is called the *scaled mantissa* of x. An important observation here is that the scaled mantissa is a real number and not an integer in general. Thus Ztrunc (scaled_mantissa x) returns an integer by either invoking a ceiling function if $x < 0$ or a floor function if $x \geq 0$. cexp is called the *canonical exponent* of a real number, and returns an integer exponent e such that the modulus of a real number x, $|x|$ lies in the range $[\beta^{e-1};\ \beta^e)$.

Thus, x is said to be part of a format if it is equal to the floating-point number that has a significand equal to the truncated scaled mantissa of x and an exponent equal to the canonical exponent of x [31]. This is how the predicate generic_format is defined in Flocq. Thus, generic_format gives an explicit representation of x as a number of type float beta. The basic properties of this generic format can be referred to in the book [31]. There are two main specializations of this generic format in floating-points that is worth discussing. The Flocq library defines these specializations as the *FLT* format, which supports underflow, and the *FLX* format, which does not support underflow. Both these formats do not support overflow. Therefore, while in theory, an arbitrary large floating-point numbers fit in both the formats, in practice, one need to prove the absence of overflow separately. Similarly, if one uses the FLX format, one needs to prove the absence of underflow separately. These levels of abstraction are defined such that it makes the proofs of error analysis less complicated and eases the verification effort. Details on their formal definitions and properties can be referred to in the book [31].

IEEE-754 binary formats: Our discussion so far has ignored exceptional values like infinities and signed zeros, and defined a generic floating-point format. Flocq also provides support for binary IEEE-754-compliant formats. Flocq defines an IEEE-754 binary format as an inductive type binary_float, and is implicitly parameterized by p and $e_{\max}$.

```
Inductive binary_float :=
  | B754_zero (s: bool)
  | B754_infinity (s : bool)
  | B754_nan (s: bool) (p1 : positive) : nan_pl pl = true → binary_float
```

```
| B754_finite (s: bool) (m : positive) (e : Z),
    bounded m e = true → binary_float.
```

Thus the inductive type binary_float has four cases: B754_zero corresponding to signed zero, B754_infinity corresponding to signed infinities, B754_nan corresponding to signed NaN, and B754_finite correspoding to a finite non-zero number. In all of these cases, the boolean value s encodes the sign of the floating-point number, and is true when the number is negative. The cases for NaNs and for finite numbers store proofs that their arguments are not out of range. Flocq then uses the inductive definition binary_float to define the three IEEE-754 binary formats *binary32*, *binary64*, and *binary128* by instantiating the implicit parameters p and $e_{\max}$ with their corresponding precision and maximum exponent values characterized by Table 4.1.

Discussion about floating-point operations Flocq defines basic arithmetic operations like $+$, $\div$, $\times$, $\sqrt{\cdot}$ on floats compliant with IEEE-754 specifications by accepting a binary_float number and a rounding mode, and returning a binary_float number. These operators have to be structured such that they handle exceptional inputs such as zeros, infinities, and NaNs carefully. For instance, consider the following formalization of $\times$ operator in Flocq

```
Definition Bmult (mult_nan: binary_float → binary_float → {nan | is_nan nan = true})
(m: mode) (x y : binary_float):=
  match x, y with
  | B754_nan, _ | _, B754_nan ⇒ B754_nan
  | B754_infinity sx, B754_infinity sy ⇒ B754_infinity (xorb sx sy)
  | B754_infinity sx, B754_finite sy _ _ _ ⇒ B754_infinity (xorb sx sy)
  | B754_finite sx _ _ _, B754_infinity sy ⇒ B754_infinity (xorb sx sy)
  | B754_infinity _, B754_zero _ ⇒ B754_nan
  | B754_zero _, B754_infinity _ ⇒ B754_nan
  | B754_finite sx _ _ _, B754_zero sy ⇒ B754_zero (xorb sx sy)
  | B754_zero sx, B754_finite sy _ _ _ ⇒ B754_zero (xorb sx sy)
  | B754_zero sx, B754_zero sy ⇒ B754_zero (xorb sx sy)
  | B754_finite sx mx ex Hx, B754_finite sy my ey Hy ⇒
  SF2B _ (proj1 (Bmult_correct_aux m sx mx ex Hx sy my ey Hy))
end.
```

Since, the binary_float format requires us to provide a payload for NaNs, Bmult function needs to provide them, which is why Flocq parameterizes Bmult with mult_nan. The Bmult operator handles the exceptional values as follows:

The result of an infinity with sign sx and another infinity with sign sy is an infinity with

sign that is xor of sx and sy. The product of infinity and a finite number is infinity with sign computed as xor of each of their signs. Similary, the product of zero and a finite number is zero. However, the product of zero and infinity is NaN.

The product two finite numbers is finite when no overflow occurs and the result depends on the rounding mode. The rounding modes that we discussed earlier are formalized using the inductive data-type in Flocq as

Inductive mode := mode_NE | mode_ZR | mode_DN | mode_UP | mode_NA

Thus, we also need to pass the rounding mode m as a parameter to the Bmult function.

Other arithmetic operators are also defined similarly in Flocq. An important observation to make in the case of finite inputs for these operators is the presence of the *correctness* specification in the definition of the operator itself. For instance, the Bmult_correct_aux predicate in the Bmult definition specifies what the correct result of the product of two floating-point finite numbers mean. This predicate is a conjunction of three clauses: (i) the result of Bmult is a correctly rounded result, (ii) the result is finite, and (iii) the result respects the sign convention of Bmult, assuming the result is not a NaN. All of these three clauses hold if the product does not overflow. Flocq treats the case of overflow separately. Thus, for the finite cases, proj1 (first clause of the conjunction) will return the correct result for an arithmetic operation. Details about the implementation of other arithmetic operations in Flocq can be referred to in the book [31].

4.2.2 Specialization to our problem

We have proved accuracy bounds for any floating-point precision. That is, our floating-point functional models, and the proofs about them, are parameterized by a *floating point type*, expressed in Coq as type:Type, with operations: [7]

fprec: type → ℤ *(* number of mantissa bits *)*
femax: type → ℤ *(* maximum binary exponent *)*
ftype: type → Type *(* floating—point numbers *)*

So for t:type, we have x:ftype(t) meaning that x is a floating-point number in format t. In fact, ftype is defined in VCFloat [114] using the inductive definition binary_float from Flocq as

Definition ftype (ty :type) := binary_float (fprec ty) (femax ty).

We will write p for fprec(t) and $e_{\max}$ for femax(t). The maximum representable finite value is $F_{\max} = 2^{e_{\max}}(1 - 2^{-p})$. If $|r| \leq F_{\max}$ then rounding r to the nearest float yields a number f such that $f = r(1 + \delta_f) + \epsilon_f$, where $|\delta_f| \leq \delta = \frac{1}{2}2^{1-p}$ and $|\epsilon_f| \leq \epsilon = \frac{1}{2}2^{3-e_{\max}-p}$. We follow the IEEE-754 binary format formalized in Flocq, and leverage the automation provided by

semi-automated floating-point reasoning tools: VCFloat [114] and its recent improvement, VCFloat2 [7].

4.3 Overview of the verification framework

The main objective of this work has been the development of a modular verification framework for the Jacobi stationary iterative method. As discussed in § 3.2, for the Jacobi iterative method, the easily invertible matrix $M = D$. Therefore, the unknown x_k is obtained from the iterative system 3.2 as

$$x_k = D^{-1}(b - Nx_{k-1}) \tag{4.1}$$

We typically start with $x_o = \mathbf{0}$, and equation 4.1 iterates until x_k satisfies $\|Ax_k - b\|_2 < \tau$, or until the program detects *failure:* overflow in computing x_k, or maximum number of iterations exceeded. Throughout this chapter, we let $\|\cdot\|$ denote the infinity vector norm and its induced matrix norm, and we let $\|\cdot\|_2$ denote the ℓ^2 norm on vectors.

This approach is modular in the sense that we separate the proof of correctness from the proof of convergence and accuracy, as will be discussed in detail in Section 4.3.2. These two layers of proof interact with each other via a *functional model*, which we will discuss in Section 4.3.1. Furthermore, we formalize lemmas about norm-wise error bounds for matrix-vector operations like addition, subtraction and multiplication, each of which compose well together to provide a forward error-bound for the solution at each iteration of the Jacobi iteration, as will be discussed in the Section 4.4. This forward error-bound is then used in the proof of convergence to guarantee that the residual of Jacobi computation converges within a user-specified tolerance within a finite number of iterations. In this way, each layer interacts modularly with one another to achieve an end-to-end verification of the Jacobi iteration algorithm. This modularity in the approach is illustrated in the Figure 4.2.

For our model problem, the steps for an end-to-end verification of the implementation of the Jacobi iteration 4.1 are as follows.

1. Write a C program that implements 4.1 by Jacobi iterations (and also implements an appropriate stopping condition).

2. Write a *floating-point functional model* in Coq (a recursive functional program that operates on floating-point values) that models Jacobi iterations of the form 4.1. This model must perform almost exactly the same floating-point operations as the C program.

3. Prove that the program written in Step 1 implements the floating-point functional

model of Step 2, using a program logic for C.

4. Prove a relation between x_k (the k-th iteration of the floating point model) and the real solution x of $Ax - b$: the *Jacobi forward error bound*. If one could run the Jacobi method purely in the reals, this is obviously contractive: $\|x_{k+1} - x\| < \rho\|x_k - x\|$, where $\rho < 1$ is the spectral radius of $D^{-1}N$. But in the floats, there is an extra term caused by roundoff error.

5. Prove *floating-point convergence:* under certain conditions, this extra term does not blow up, and within a specified k iterations the residual $\|Ax_k - b\|_2$ is less than tolerance τ.

6. Compose these to prove the main theorem: the C program converges to an accurate result.

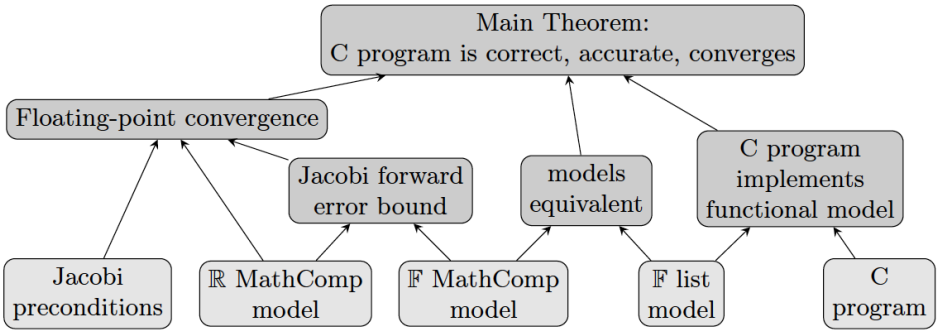

Figure 4.2: Theorem dependency. Bottom row: models and definitions; middle row: theorems relating models.

4.3.1 Discussion about functional models

A functional model describes how an algorithm functions. It is often chosen as a functional program that implements an algorithm and rules out the intricate semantics of a programming language. These models are amenable to verification. We will use a functional model for the floating-point implementation of the Jacobi iteration algorithm, and another functional model for the computation of real solution.

4.3.1.1 Floating-point functional model

Our floating-point functional model for the implementation of the Jacobi algorithm is parametric in terms of the floating point type `t: type`. This model takes as input the matrix A

represented by the matrix type `matrix t`, the right hand side vector b which is repesented by the vector type `vector t`, the vector from previous iteration x repesented by the vector type `vector t`, and the number of iterations repesented by the variable n of type `nat`. The resuling vector from this model is also a vector of type `vector t`.

```
Definition jacobi_n {t: type} (A: matrix t) (b: vector t) (x: vector t) (n: nat) : vector t :=
  let A1 := diag_of_matrix A in
  let A2 := remove_diag A in
  Nat.iter n (jacobi_iter A1 A2 b) x.
```

The model jacobi_n implements the recurrence relation repesented mathematically as

$$x_k = \tilde{D}^{-1} \otimes (b \ominus (N \otimes x_{k-1})). \tag{4.2}$$

The operators $\otimes, \ominus$ represent the floating-point matrix-vector multiplication, and the floating-point vector subtraction, respectively. The diagonal matrix D is repesented in the definition `jacobi_n` by `diag_of_matrix`, which is defined in Coq as

```
Definition diag_of_matrix {t: type} (A: matrix t) : diagmatrix t :=
  map (fun i ⇒ matrix_index m i i) (seq 0 (matrix_rows_nat m)).
```

The `diag_matrix A` maps a diagonal function `(fun i => matrix index m i i)` to a sequence of row numbers in the matrix A and extracts the diagonal elements in each row, and builds a vector of type `diagmatrix t`, which is essentially a wrapper around the vector type `vector t`. The matrix N in the relation 4.2 is repesented in Coq by `remove_diag A`, which is defined in Coq as

```
Definition remove_diag {t} (A: matrix t) : matrix t :=
 matrix_by_index (matrix_rows_nat m) (matrix_rows_nat m)
  (fun i j ⇒ if Nat.eq_dec i j then Zconst t 0 else matrix_index m i j).
```

The `remove_diag` maps a non-diagonal function `(fun i j => if Nat.eq dec i j then Zconst t 0 else matrix index m i j)` to the matrix A, and builds a matrix by extracting the non-diagonal elements in A. The `jacobi_iter` function builds the recurrence relation in 4.2 and is defined in Coq as

```
Definition jacobi_iter {t: type} (A1: diagmatrix t) (A2: matrix t)
(b: vector t) (x: vector t) : vector t :=
   diagmatrix_vector_mult (invert_diagmatrix A1) (vector_sub b (matrix_vector_mult A2 x)).
```

The `jacobi_iter` function takes the diagonal matrix D, and inverts it using the `invert_diagmatrix` function, which is defined in Coq as

```
Definition invert_diagmatrix {t} (v: diagmatrix t) : diagmatrix t :=
  map (BDIV (Zconst t 1)) v.
```

The `invert_digmatrix` extracts an element from the vector `diagmatrix t`, and inverts it using the division operation defined in Flocq library of Coq as `BDIV`. `BDIV` function performs a floating point division and therefore adds a source of the division error. This builds the inverse matrix $\tilde{D}^{-1}$. The `jacobi_iter` function then multiplies $\tilde{D}^{-1}$ with the vector $(b \ominus (N \otimes x_{k-1}))$. The vector $(b \ominus (N \otimes x_{k-1}))$ is built by subtracting the vector b with the vector $N \otimes x_{k-1}$. This vector subtraction function is defined in Coq by `vector_sub` as

```
Definition vector_sub {NAN: Nans}{t:type} (v1 v2 : vector t) :=
  map2 (@BMINUS _ t) v1 v2.
```

The `vector_sub` function maps the function `(@BMINUS t)` with the vectors v_1 and v_2 to performs a floating point subtraction using `BMINUS`. The operation `BMINUS` adds another source of floating point error. The function `matrix_vector_mult` perform a matrix-vector multiplication in floating point by performing a dot-product between a row of the matrix and the vector. Therefore, this matrix-vector multiplication inherits the floating-point error coming from such dot-product operations.

```
Definition matrix_vector_mult {NAN: Nans}{t: type} (m: matrix t) (v: vector t) : vector t :=
      map (fun row ⇒ dotprod row v) m.
```

The `jacobi_iter` is now the vector x_k, which is further fed into the recurrence relation defined by Coq's `iter` function as

```
Nat.iter n (jacobi iter A1 A2 b) x
```

to build the next vector x_k by instantiating `x` with x_{k-1}.

4.3.1.2 Real functional model

The functional model for a real solution x of the matrix system $Ax = b$ is defined as

```
Definition x_fix {n:nat} x b (A: 'M[R]_n.+1) : 'cV[R]_n.+1 :=
  let r := b − ((A2_J_real A) ∗m x) in
  diag_matrix_vec_mult_R (A1_diag A) r.
```

and implements the following mathematical model

$$x = D^{-1}(b - Nx) \tag{4.3}$$

Thus, x is the fixed point solution of the Jacobi iteration method, and can be obtained by re-arranging the original system $Ax = b$ by splitting the matrix $A = D + N$. Here, a fixed

point solution refers to the solution from a fixed-point iteration, i.e. $f(x_{fix}) = x_{fix}$. The Coq definition `x_fix` takes the fixed point vector x, the right hand side vector b, and the A matrix and returns the fixed point vector x, whose type is `'cV[R]_n.+1`. `'cV[R]_n.+1` is a mathcomp notation for a column vector of size $n+1$, whose elements are of real type, denoted in Coq as a set `R`. `x_fix` builds a vector of dimension $n+1$ instead of n, because natural numbers in Coq are defined using the Peano arithmetic, i.e., by successively building natural numbers from 0. Therefore, the natural 1 will be built in Coq as `S 0`, i.e., successor of 0. Similarly 2 will be built as `S (S 0)`. Therefore, there is a possibility of getting a vector of dimension 0, operations on which are unsound and do not type check in Coq. To type-check the vector operations, we need to either introduce an additional constraint $0 < n$ in the definition `x_fix`, or add that constraint instrinsically into the type of matrix and vector by replacing n by $n+1$. Now, we are able to define operations on a matrix A and a vector b, which will be accepted by the type system in Coq, since the minimum dimension of the matrix A will be 1×1, and that of the vector will be 1. Therefore, in this case, the matrix A will be a scalar, and the vector b will be a scalar, and operations on them will be sound.

The function `x_fix` then first builds a diagonal matrix D defined out of A using the definition `A1_diag A`. It then builds another vector r by subtracting the vector b with Nx, which is defined by the mathcomp matrix vector multiplication denoted by the operator `*m`, as `((A2 J real A) *m x)`. `x_fix` then performs a diagonal matrix D multiplication with r, and returns the fix-point vector x in 4.3.

4.3.1.3 Defining forward error using the functional models

Once we have a functional model for the iterative solution x_k, and the fixed-point solution x, we can then define the forward error for the Jacobi iteration as

```
Definition f_error {ty} {n:nat} m b x0 x (A: 'M[ftype ty]_n.+1):=
  let x_k := X_m_jacobi m x0 b A in
  let A_real := FT2R_mat A in
  let b_real := FT2R_mat b in
  let x := x_fix x b_real A_real in
  vec_inf_norm (FT2R_mat x_k − x).
```

The function `f_error` builds the forward error $||x_k - x||_\infty$ by taking the inputs: m for the iteration count, b; the right hand vector of floats $b \in \mathbb{F}^{(n+1)\times 1}$; initial guess vector $x_o \in \mathbb{F}^{(n+1)\times 1}$; the fixed-point real vector $x \in \mathbb{R}^{(n+1)\times 1}$; and the matrix $A \in \mathbb{F}^{(n+1)\times(n+1)}$ of floats `'M[type t]_n.+1`. The function `f_error` first builds the ierative solution x_k using another floating point functional model `X_m_jacobi`, which is defined in Coq as

```
Definition X_m_jacobi {ty} {n:nat} m x0 b (A: 'M[ftype ty]_n.+1) : 'cV[ftype ty]_n.+1 :=
  Nat.iter m (fun x0 ⇒ jacobi_iter x0 b A) x0.
```

where `jacobi_iter` is a mathcomp style implementation of 4.2. We need two different floating point models `X_m_jacobi`, which is implemented in the mathcomp style, and `jacobi_n`, which is implemented in the standard Coq lists of lists definition, because the mathcomp style is easier to perform mathematical analysis upon, while the lists of lists definition is more amenable to reason about the matrix and vectors data structures in the C program itself. We then prove an equivalence between these two styles of defining the functional model using the lemma

```
Lemma func_model_equiv {ty} (A: matrix ty) (b: vector ty) (x: vector ty) (n: nat) :
  let size := (length A).−1 in
  let x_v := vector_inj x size.+1 in
  let b_v := vector_inj b size.+1 in
  let A_v := matrix_inj A size.+1 size.+1 in
  (0 < length A)%nat →
  length b = length A →
  length x = length A →
  vector_inj (jacobi_n A b x n) size.+1 = @X_m_jacobi _ ty size n x_v b_v A_v.
```

where `vector_inj` is an injection of a vector defined using the lists definition to the mathcomp vector, and `matrix_inj` is an injection of a matrix defined using the lists of lists definition to the mathcomp matrix. Using the lemma `func_model_equiv` we bridge the gap between the world of numerical analysis and the world of program verification. `A_real`, and `b_real` are real-projections of the floating point matrix A and the floating point vector b, respectively, using our Coq definition

```
Definition FT2R_mat {m n: nat} {ty} (A : 'M[ftype ty]_(m.+1, n.+1)) :
  'M[R]_(m.+1, n.+1):= \matrix_(i, j) FT2R (A i j).
```

`FT2R` is a real projection of the floating point number A_{ij}. The function `f_error` then takes an infinity norm of the difference between x_k and x. We defined the infinity norm of a vector in Coq as

```
Definition vec_inf_norm {n:nat} (v : 'cV[R]_n) :=
 bigmaxr 0%Re [seq (Rabs (v i 0)) | i <− enum 'I_n].
```

The mathcomp definition `bigmaxr` takes a maximum of the sequence $\{|v_o|, |v_1|, \ldots, |v_{n-1}|\}$.

Now that we have defined the forward error `f_error` for the Jacobi iteration, we would like to get a bound on this error, which we discuss in the Section 4.4.

4.3.2 Discussion about the end-to-end verification framework

We follow the modular approach for an end-to-end verification of programs, first discussed by [6], and is illustrated in Figure 4.3

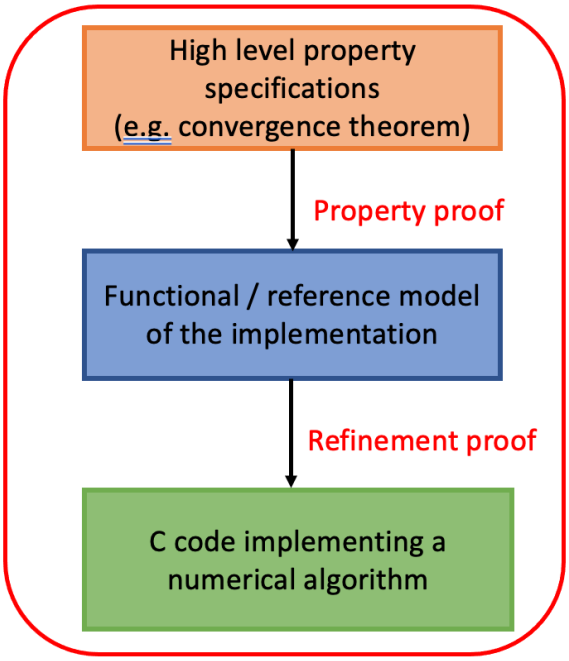

Figure 4.3: Illustration of the end-to-end framework

We write a high level property specification using higher order logic in the Coq proof assistant. This high level property specification will be a statement on iterative convergence in the presence of floating point errors. The actual implementation of the algorithm – Jacobi iteration algorithm in our case, is done in the C programming language. We cannot directly prove that this C program respects the high level property specification, since the standard Coq library does not contain theory for reasoning about programs. Reasoning about C programs is handled by the Verifiable Software Toolchain (VST) [33]. VST provides a foundational framework for reasoning about C programs in the Coq theorem prover. We write an intermediate representation of an algorithm we want to verify, which is called the *functional model*. This functional model is a functional program implementing the algorithm and is easy to verify. The proof is then divided into two layers:

- *Property proof:* In this layer, we prove that the functional model respects the high level property specification. For the Jacobi algorithm, this layer would constitute proving that the functional model jacobi_n satisfies the high level theorem

jacobi_iteration_bound_lowlevel, stated formally in Section 4.5.

- *Refinement proof:* In this layer, we prove that the C program is a refinement of the functional model jacobi_n, i.e., the C program faithfully implements jacobi_n. This layer requires us to deal with the intricate semantics of C programming language, and data structures in C. Proofs in this layer show the absence any programming errors.

Due to the modularity of this approach, the *property proof* layer can be handled separately by numerical analysts, while the *refinement proof* layer can be handled separately by experts in programming languages. These two communities interact at the *functional model* layer. By composing these two layers, we get an end-to-end proof of the program with respect to a high level specification.

In this work, the *property proof* layer was handled by me, while the *refinement proof* layer was handled by Prof. Andrew Appel.

4.4 Error analysis for Jacobi iteration method

In this section, we discuss the building blocks of the first layer of our framework, *property proof*. These building blocks include a formal dot-product analysis, its adaptation to norm-wise error bounds for common matrix-vector operations, and finally a forward error analysis for the Jacobi iteration method.

4.4.1 Dot-product forward error analysis

A dot-product $\langle u, v\rangle$ between two real vectors u and v is defined as $\sum_{0\leq i<n} u_i v_i$. A matrix-vector multiplication Av can be seen as the dot-product of each row of A with vector v. Forward error bounds for a matrix-vector multiplication are therefore based on forward error bounds for dot-product.

Our implementation and functional model of the dot-product use *fused multiply-add* (FMA), which computes a floating-point multiplication and addition (i.e., $a \otimes b \oplus c$) with a single rounding error rather than two.

```
Definition dotprod {t: type} (u v: list (ftype t)) : ftype t :=
  fold_left (fun z a ⇒ FMA (fst a) (snd a) z) (List.combine u v) (Zconst t 0).
```

The parameters to the dotprod functional model are the floating-point format t and two lists of floating point numbers. The algorithm zips the two lists into a list of pairs (using List.combine) and then adds them from left to right, starting with a floating-point 0 in format t.

We denote floating-point summation by $\bigoplus$, so the floating-point dot product is $\bigoplus_{0\le i<n} u_i v_i$; real-valued summation is denoted as $\sum_{0\le i<n} u_i v_i$. The notation finite(z) signifies that the floating-point number z is within the range of the floating-point format (not an infinity or NaN).

Theorem 21 (forward error + no overflow)**.** *Let u, v be lists of length n of floats in format $t = (p, e_{\max})$, in which every element is $\le v_{\max}$, and no more than s elements of u are nonzero. The absolute forward error of the resulting dot product is*

$$\left| \bigoplus_{0\le i<n} u_i v_i - \sum_{0\le i<n} u_i v_i \right| \;\le\; g_\delta(s) \sum_{0\le i<n} |u_i v_i| \;+\; g_\epsilon(s). \tag{4.4}$$

$g_\delta(s) = (1+\delta)^s - 1;\; g_\epsilon(s) = s\epsilon(1 + g_\delta(s-1))$

Proof. See Kellison *et al.* [89]. □

[89] prove (in Coq) the correctness and accuracy of floating-point dot-product and sparse matrix-vector multiply, as Coq functional models and as C programs.

Subnormal numbers. When some of the vector elements are order-of-magnitude 1, the term $g_\epsilon(s)$ is negligible. But if the u and v vectors are composed of subnormal numbers, then neglecting the underflow-error term would be unsound. Most previous proofs about dot-product error (and about Jacobi iteration), and all previous machine-checked proofs to our knowledge, omit reasoning about underflow.

4.4.2 Discussion about the norm-wise error bounds

The norm-wise forward error analysis for the Jacobi algorithm is based on the following auxiliary error bounds on matrix-vector operations.

- Matrix-vector multiplication:

$$\|N \otimes x - Nx\| \le \|N\|\|x\| g_\delta(n) + g_\epsilon(n). \tag{4.5}$$

  This bound is derived from the dot product error bound that we stated in Section 4.4.1,. In the $\|\cdot\|$ norm, the dot product errors directly give the error bound for matrix-vector multiplication. In Coq, we state this error relation as the following lemma statement

```
Lemma matrix_vec_mult_bound_corollary {n:nat} {ty}:
∀(A: 'M[ftype ty]_n.+1) (v : 'cV[ftype ty]_n.+1),
let l1 := vec_to_list_float n.+1 (\row_j A i j)^T in
```

let l2 := vec_to_list_float n.+1 v **in**
(∀ (xy : ftype ty * ftype ty) (i : 'I_n.+1),
In xy (combine l1 l2) → finite xy.1 ∧ finite xy.2) →
(∀ (i : 'I_n.+1), finite (dotprod_r l1 l2)) →
vec_inf_norm (FT2R_mat (A *f v) − (FT2R_mat A) *m (FT2R_mat v)) <=
(matrix_inf_norm (FT2R_mat A) * vec_inf_norm (FT2R_mat v)) * g ty n.+1 +
g1 ty n.+1 (n.+1 − 1).

The lemma `matrix_vec_mult_bound_corollary` states that for a given floating-point matrix A, and vector v, the error relation 4.5 holds if each element in the row vector constructed from A - `(row_j A i j)`, and the vector v is finite, and the dot-product of the row vector and v is also finite.

The `vec_to_list_float` is an injection from a mathcomp vector to a list of floats. We need this injection, since we use the membership reasoning for lists, and our dot-product functional model is defined over a list of pairs. `xy.1` is the first projection of a pair `xy`, and `xy.2` is the second projection of a pair `xy`, i.e. `xy = (xy.1, xy.2)`.

- Vector subtraction:

$$||(b \ominus (N \otimes x_k) - b(N \otimes x_k)|| \leq (||b|| + ||N \otimes x_k||)\delta. \tag{4.6}$$

Here, we use the fact that $|x \ominus y - (x - y)| \leq (|x| + |y|)\delta$. Since we are considering infinity vector norm, the above element wise error bound composes well to provide the resulting norm-wise error bound for vector subtraction. We state the error relation 4.6 in Coq as the following lemma statement

Lemma vec_float_sub {ty} {n:nat} (v1 v2 : 'cV[ftype ty]_n.+1):
(∀ (xy : ftype ty * ftype ty),
In xy (combine (vec_to_list_float n.+1 v1) (vec_to_list_float n.+1 v2)) →
finite xy.1 ∧ finite xy.2 ∧ finite (BPLUS xy.1 (BOPP xy.2))) →
vec_inf_norm (FT2R_mat (v1 −f v2) − (FT2R_mat v1 − FT2R_mat v2)) <=
(vec_inf_norm (FT2R_mat v1) + vec_inf_norm (FT2R_mat v2)) * (default_rel ty) .

The lemma `vec_float_sub` states that for a given vectors v_1 and v_2, the error relation 4.6 holds if each element of the vectors v_1 and v_2 is finite, and the floating point subtraction between each element of v_1 and v_2 is also finite.

- Vector inverse:

$$||\tilde{D}^{-1} - D^{-1}|| \leq ||D^{-1}||\delta + \epsilon \tag{4.7}$$

Here, we make use of the fact that inverting each element of the vector D satisfies, $|(1 \oslash D_i) - (1/D_i)| \leq |(1/D_i)|\,\delta + \epsilon$. We state the error relation 4.7 in Coq as the following lemma statement

```
Lemma inverse_mat_norm_bound {ty} {n:nat} (A: 'M[ftype ty]_n.+1):
  (∀ i, finite (BDIV (Zconst ty 1) (A i i ))) →
  (∀ i, finite (A i i)) →
  let A_real := FT2R_mat A in
  (vec_inf_norm (FT2R_mat (A1_inv_J A) − A1_diag A_real) <=
    vec_inf_norm (A1_diag A_real) ∗ (default_rel ty) + (default_abs ty))%Re.
```

The lemma `inverse_mat_norm_bound` states that for a given matrix A, the error relation 4.7 holds if each element of A, and its inverse is finite.

An important observation to make in the Coq statements of each of these error relations is that, the error relations 4.5, 4.6, and 4.7 hold if each element in the matrix and the vector is finite, and the corresponding operations do not result in an overflow. This is important because we will compose these atomic error relations to derive a forward error bound for the Jacobi iteration in Section 4.4.3. It will then become important that we choose bounds on the inputs, defined in `jacobi_preconditions` 24 such that each of these atomic error operations do not overflow, and the resulting composition do not overflow as well.

4.4.3 Jacobi-error-bound

We prove an explicit bound on the distance between the approximate solution x_k at step k and the exact solution x of the problem. Such bounds are commonly studied in computational science, but rarely take into account the details of floating-point arithmetic (including underflow and overflow). They also usually have paper proofs while we provide a machine-checked proof.

Theorem 22 (jacobi_forward_error_bound)**.** *After k Jacobi iterations, if no iteration resulted in an overflow, then the distance between the approximate (floating-point) solution x_k and the exact (real) solution x is bounded:*

$$\|x - x_k\| \leq \rho^k \|x - x_0\| + \frac{1-\rho^k}{1-\rho} d_{\text{mag}} \tag{4.8}$$

where ρ is a bound on the spectral radius (largest eigenvalue) of $D^{-1}N$, adjusted for floating-point roundoff errors that arise in one iteration of the Jacobi method. The (small) value d_{mag} is the floating-point roundoff error in computing the residual $\|Ax_k - b\|$. In Coq,

Theorem jacobi_forward_error_bound {ty} {n:nat}
(A: 'M[ftype ty]_n.+1) (b: 'cV[ftype ty]_n.+1): ∀ x0: 'cV[ftype ty]_n.+1,
forward_error_cond A x0 b →
(∀ k i, finite (x_{k, i})) ∧
(f_error k ≤ rho^k ∗ (f_error 0) + ((1 − rho^k) / (1−rho))∗ d_mag.

where f_error(k) is $\|x - x_k\|$, and the conditions on $A \in \mathbb{F}^{(n+1)\times(n+1)}, x_0 \in \mathbb{F}^{(n+1)\times 1}, b \in \mathbb{F}^{(n+1)\times 1}$ for $n \geq 0$, are characterized as follows:

Definition forward_error_cond {ty} {n:nat}
(A: 'M[ftype ty]_n.+1) (x0 b: 'cV[ftype ty]_n.+1) :=
(∀ i, finite (A _{i,i}) ∧ (rho < 1) ∧ invertible A ∧ (∀ i, finite (1 / A_{i,i})) ∧
(∀ i, finite (x_{o,i}) ∧ (∀ i j, finite(N_{i,j})) ∧ (∀ i, finite (b_i) ∧
size_constraint n ∧ input_bound A x0 b.

`size_constraint n` is a constraint on the dimension n of the matrix A (in double-precision, about $6 \cdot 10^9$). The predicate `input_bound` provides conditions on the bounds for the inputs A, b, x_0; it is implied by the *Jacobi preconditions* (Theorem 24) defined in the next section.

Proof. Assuming no overflow, the floating point iteration satisfies

$$x_{k+1} = \tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))$$

where the operators $\otimes$, $\ominus$ represent the floating point multiplication and subtraction operations respectively. $\tilde{D}^{-1}$ is the floating-point (not exact) inverse of the diagonal elements. The forward error at step $k+1$ is defined as

$$f_{k+1} = \|x_{k+1} - x\| = \|(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (D^{-1}(b - Nx))\|$$

Here, we write the true solution as $x = D^{-1}(b - Nx)$ which can be derived from $Ax = b$. We use these norm-wise errors defined in Section 4.4.2 to expand the error definition f_{k+1}:

$$\begin{aligned} f_{k+1} \leq & \left(\left(||\tilde{D}^{-1}|| \; (\; ||b|| + ||N|| \; ||x_k|| \; (1 + g_\delta) + g_\epsilon \;) \right) (1 + \delta) g_\delta + g_\epsilon \right) + \\ & ||\tilde{D}^{-1}|| \; (\; ||b|| + ||N|| \; ||x_k|| (1 + g_\delta) + g_\epsilon \;) \, \delta + \\ & ||\tilde{D}^{-1}|| \; (\; ||N|| \; ||x_k|| g_\delta + g_\epsilon \;) + \\ & (\; ||D^{-1}|| \delta + \epsilon \;) \; (\; ||b|| + ||N|| \; ||x_k|| \;) + ||D^{-1}|| \; ||N|| f_k \end{aligned}$$

Details of the derivation is presented in Appendix A. We then collect the coefficients of $||x_k||$ and expand using the error relation for f_k as $||x_k|| \leq f_k + ||x||$ to get the error recurrence

relation:

$$f_{k+1} \leq \rho f_k + d_{\text{mag}}$$

where d_{mag} is a constant independent of k depending only on $||x||$, δ, ϵ, g_δ, g_ϵ. We can then expand the above error recurrence relation to get

$$f_{k+1} \leq \rho^{k+1} f_o + (1 + \rho + \rho^2 + \ldots + \rho^k) d_{\text{mag}}$$

This geometric series converges if $\rho < 1$ with closed form

$$f_{k+1} \leq \rho^{k+1} f_o + \frac{1 - \rho^{k+1}}{1 - \rho} d_{\text{mag}}$$

Note that if the above iterative process were done in reals, then we would only require $\rho_r := ||D^{-1}N||$ to be less than 1. Thus, the presence of rounding errors forces us to choose a more conservative convergence radius ρ. □

4.5 Proof of convergence

In this section, we discuss the high level theorem, which constitutes the *property proof*. We use the building blocks from Section 4.4 to prove the high level convergence theorem.

4.5.1 Theorem statement

Theorem 22 had the premise, "if no iteration resulted in an overflow." Most previous convergence theorems for Jacobi iteration [121] have been in the real numbers, where overflow is not an issue: multiplying two finite numbers cannot "overflow." Higham and Knight [69] proved convergence (but not absence of overflow), on paper, in a simplified model of floating-point (without underflows). Let us now state a theorem, for an accurate model of floating point, which accounts for the exceptional floating-point behaviors like overflow and underflow.

Theorem 23. *If the inputs A, b, desired tolerance τ, and projected number of iterations k satisfy the (efficiently testable) Jacobi preconditions, then the floating-point functional model of Jacobi iteration converges, within $j \leq k$ iterations, to a solution x_j such that $\|Ax_j - b\|_2 < \tau$, without overflowing.*

Proof. The proof for this theorem follows by the following cases:

i If A is diagonal then N is a zero matrix. Therefore, the solution vector at each iteration is given by the constant vector $x_k = \tilde{D}^{-1} \otimes b$. Hence, the solution of Jacobi iteration

has already converged after the first step, assuming certain bounds on b and A implied by *Jacobi preconditions*.

ii If A is not diagonal but the vector b is small enough, Jacobi iteration has already converged, without even running the iteration.

iii Suppose A is not diagonal and the vector b is not too small. Then

(a) The residual does not overflow for every iteration $\leq j$. This follows from the *Jacobi preconditions* and Theorem 22.

(b) We can calculate $k_{\min}$ such that the residual $< \tau$ within $k_{\min}$ iterations.

□

We formalized the Theorem 23 in Coq as

```
Lemma jacobi_iteration_bound_lowlevel {t: type} :
 ∀(A: matrix t) (b: vector t) (acc: ftype t) (k: nat),
   jacobi_preconditions A b acc k →
   let acc2 := BMULT acc acc in
   let x0 := (repeat (Zconst t 0) (length b)) in
   let resid := jacobi_residual (diag_of_matrix A) (remove_diag A) b in
   finite acc2 ∧
   ∃j, Nat.le j k ∧
    let y := jacobi_n A b x0 j in
    let r2 := norm2 (resid y) in
    (∀ i, Nat.le i j → finite (norm2 (resid (jacobi_n A b x0 i)))) ∧
    BCMP Lt false (norm2 (resid (jacobi_n A b x0 j))) acc2 = false.
```

The lemmas jacobi_iteration_bound_lowlevel takes as input the matrix A, the right hand side vector b, the user-specified tolerance acc (τ), and the maximum iteration count k, and proves that there exists some iteration count $j \leq k$, such that the residual, resid is finite for all iteration iterations i through j, and the l^2 norm of the residual is less than than acc (τ). BCMP is the floating point comparison defined in the Flocq library of Coq. BCMP Lt false x y = false means that the float x is less than y in floating points, and if x and y are finite, then $x < y$ in reals. We instantiate j with 1 in Case (i), j with 0 in Case (ii), and j with $k_{\min}$ in Case (iii), in the proof of Theorem 23.

4.5.2 Discussion about the `jacobi_preconditions`

The efficiently computable (a straightforward arithmetic computation with computational complexity linear in the number of nonzero matrix elements) *Jacobi preconditions* are defined as follows

Definition 24 (jacobi_preconditions_Rcompute)**.** A, b, τ, k satisfy the *Jacobi preconditions* when:

- All elements of A, b, and $\tilde{D}^{-1}$ are finite (representable in floating point);
- A is strictly diagonally row-dominant, that is, $\forall i.\ D_{ii} > \sum_j |N_{ij}|$;
- $\tilde{\tau}^2$ is finite;
- $\tilde{\tau}^2 > g_\epsilon + n(1+g_\delta)(g_\epsilon) + 2(1+g_\delta)(1+\delta)\|D\|(\hat{d}_{\text{mag}}/(1-\hat{\rho}))^2$;
  $\tilde{\tau}^2 > n(1+g_\delta)(\|D\|(\|\tilde{D}^{-1}\|(\|A\|\hat{d}_{\text{mag}}/(1-\hat{\rho}) + g_\epsilon)(1+\delta)(1+g_\delta) + g_\epsilon)(1+\delta)(1+g_\delta) + g_\epsilon)^2 + g_\epsilon$;
- $k_{\min} \leq k$;
- $n < ((F_{\max} - \epsilon)/(1+\delta) - g_\epsilon - 1)/(g(n-1)+1)$; $n < F_{\max}/((1+g(n+1))\delta) - 1$
- $\forall i.\ |A_{ii}|(1+\hat{\rho})x_{\text{bound}} + 2\hat{d}_{\text{mag}}/(1-\hat{\rho}) + 2x_{\text{bound}} < v_{\max} - \epsilon)/(1+\delta)^2$;
- $\forall i, j.\ |N_{ij}| < v_{\max}$;
- $\forall i.\ |b_i| + (1+g_\delta)((2x_{\text{bound}} + \hat{d}_{\text{mag}}/(1-\hat{\rho}))\sum_j |N_{ij}|) + g_\epsilon < F_{\max}/(1+\delta)$;
- $\forall i.\ |\tilde{D}_{ii}^{-1}|(|b_i| + (1+g_\delta)(2x_{\text{bound}} + \hat{d}_{\text{mag}}/(1-\hat{\rho}))\sum_j |N_{ij}|) + g_\epsilon < F_{\max}/(1+\delta)$;
- $(1+\hat{\rho})x_{\text{bound}} + 2\hat{d}_{\text{mag}}/(1-\hat{\rho}) + 2x_{\text{bound}} < F_{\max}/(1+\delta)$;
- $\forall i.\ |b_i| < (F_{\max} - \epsilon)/(1+\delta)$;
- $\forall i.\ |\tilde{D}_{ii}^{-1}||b_i| < (F_{\max} - \epsilon)/(1+\delta)$;
- $\forall i.\ |\tilde{D}_{ii}^{-1}||b_i|(1+\delta) + \epsilon < (F_{\max} - \epsilon)/(1+\delta)$;
- $\forall i.\ |A_{ii}|(|\tilde{D}_{ii}^{-1}||b_i|(1+\delta) + \epsilon) < (v_{\max} - \epsilon)/(1+\delta)$.

where $\hat{d} = (\|\tilde{D}^{-1}\| + \epsilon)/(1-\delta)$ is a bound on $\|D^{-1}\|$. Defining $R = \hat{d}\,\|N\|$, we define an upper bound on the norm of the solution x to $Ax = b$ as $x_{\text{bound}} = \hat{d}\|b\|/(1-R)$. $\hat{\rho}$ is the adjusted spectral radius ($\rho_r = \|D^{-1}N\|$) of the iteration matrix, obtained by accounting for the floating point errors in its computation. For the iteration process to converge in

presence of rounding, we want $\hat{\rho} < 1$. $\hat{d}_{\text{mag}}$ is a bound on the additive error in computing the residual $\|Ax_k - b\|$, the difference between computing the residual in the reals versus in floating point. $\tilde{\tau}^2 = \tau \otimes \tau$ is the floating-point square of τ. The minimum k for which we guarantee convergence is calculated as

$$k_{\min} = 1 + \left\lceil \frac{\ln\left(\frac{x_{\text{bound}}(1+\delta)}{((\sqrt{(\tilde{\tau}^2 - g_\epsilon)/(n(1+g_\delta))} - g_\epsilon)/((1+g_\delta)+\|D\|(1+\delta))) - 2\hat{d}_{\text{mag}}/(1-\hat{\rho})}\right)}{\ln(1/\hat{\rho})} \right\rceil \tag{4.9}$$

Indeed these conditions are quite tedious – one might have difficulty trusting them without a machine-checked proof. But they are all easy to compute in linear time. And, although we state them here (mostly) in terms of operations on the reals, they are all straightforwardly boundable by floating-point operations.

Remark 25 (not proved in Coq). The Jacobi preconditions can be computed in time linear in the number of nonzero entries of A.

Proof. Let S be the number of nonzeros. Then $n < S$ since the diagonal elements are nonzero. The inverse diagonal $\tilde{D}^{-1}$ is computed in linear time. The infinity norm ($\|N\|, \|D\|, \hat{d}, \|b\|$) is simply the largest absolute value of any row-sum (for matrix) or element (for vector), which can be found in $O(S)$ time. Then the values $x_{\text{bound}}, \hat{\rho}, \hat{d}_{\text{mag}}, v_{\max}, k_{\min}$ can all be computed in constant time. Then each of the tests in Definition 24 can be done in $O(S)$ time. □

The proof for Theorem 23 relies on a couple of intermediate, but important results.

4.5.2.1 Proof that strict diagonal row dominance implies inveribility

The Theorem `jacobi_forward_error_bound`, which is used the proof of Theorem 23 requires that the matrix A is invertible. This clause is defined as a conjuction in the `forward_error_cond`. But computing an inverse of A, and checking for invertibility, is computationally expensive. Therefore, we need a more efficient and direct approach to check for invertibility of A, without actually computing A^{-1}. A sufficient condition to guarantee that A is invertible is the diagonal row dominance of A. Mathematically, a matrix A is said to be strictly diagonally row dominant if

$$\forall\, i \in \mathbb{N}, |A_{ii}| > \sum_{j \neq i} |A_{ij}| \tag{4.10}$$

In Coq, we define the strict diagonal row dominance as

Definition strict_diagonal_dominance {t} {n:nat} (A: 'M[ftype t]_n.+1):=

∀ i, Rabs (FT2R_mat A i i) > \sum_(j < n.+1 | j != i) Rabs (FT2R_mat A i j).

We then formally prove that strict diagonal row dominance implies that A is invertible.

Lemma 26. *If A is strictly row diagonally dominant, then A is inverible.*

Proof. The matrix A is invertible if and only if its rows are linearly independent. By definition of linear row independence, forall column vectors v, if $Av = 0$, then $v = 0$. Therefore, to prove that A is invertible, we need to prove that $v = 0$, under the assumption that $Av = 0$, $\forall\ v$. We will prove this by introducing an axiom on the decidability of reals for the norm of vector v, i.e., $||v||_\infty = 0\ \vee\ ||v||_\infty \neq 0$, and then reason by the following cases:

- Case 1: $||v||_\infty = 0$. This trivially implies that $v = 0$.
- Case 2: $||v||_\infty \neq 0$. We will prove $v = 0$ by contradiction.

  Since $||v||_\infty \neq 0$, $v \neq 0$. We will then show that $Av \neq 0$.

$$Av \neq 0 \implies \exists\ j,\ (Av)_j \neq 0 \tag{4.11}$$

  Since $||v||_\infty \neq 0$, there exists a k such that $|v_k| = ||v||_\infty\ \wedge\ v_k \neq 0$. We then instantiate j in 4.11 with k to get $(Av)_k \neq 0$. Therefore,

$$\begin{aligned}
&\sum_i A_{ki} v_i \neq 0 \\
&\Leftarrow |A_{kk} v_k + \sum_{i \neq k} A_{ki} v_i| \neq 0 \\
&\Leftarrow 0 < |A_{kk} v_k + \sum_{i \neq k} A_{ki} v_i| \\
&\Leftarrow 0 < |A_{kk} v_k| - |\sum_{i \neq k} A_{ki} v_i| \\
&\Leftarrow 0 < |A_{kk}|\ |v_k| - \sum_{i \neq k} |A_{ki}|\ |v_i| \\
&\Leftarrow |A_{kk}|\ ||v||_\infty - \left(\sum_{i \neq k} |A_{ki}|\right)\ ||v||_\infty \\
&\text{Since, } |v_k| = ||v||_\infty\ \wedge\ \forall\ i,\ |v_i| \leq ||v||_\infty \\
&\Leftarrow 0 < ||v||_\infty \left(|A_{kk}| - \sum_{i \neq k} |A_{ki}|\right) \\
&\Leftarrow 0 < ||v||_\infty\ \wedge\ 0 < \left(|A_{kk}| - \sum_{i \neq k} |A_{ki}|\right)
\end{aligned}$$

Since $||v||_\infty \neq 0$, $0 < ||v||_\infty$ holds. $0 < \left(|A_{kk}| - \sum_{i \neq k} |A_{ki}|\right)$ holds by definition of strict row diagonal dominance of the matrix A. This proves that $Av \neq 0$. But, we started with $Av = 0$. Hence, by contradiction $v = 0$.

□

We state the Lemma 26 in Coq as

```
Lemma diagonal_dominance_implies_invertibility {t} {n:nat} (A: 'M[ftype t]_n.+1):
  strict_diagonal_dominance A → (FT2R_mat A) \in unitmx.
```

where `(FT2R_mat A) in unitmx` is MathComp's style of stating that `(FT2R_mat A)` is invertible.

4.5.2.2 Proof for bound on $||x||_\infty$

The residual r_k is defined in the C implementation of the Jacobi iteration as $||\tilde{D}^{-1} \otimes (x_k \ominus x_{k-1})||_{2,f}$. $||\cdot||_{2,f}$ denotes l^2 norm of a vector in floating points. The square of the residual r_k is related to the foward error `f_error` as

$$
\begin{aligned}
r_k \otimes r_k &\leq n||\tilde{D}^{-1} \otimes (x_k \ominus x_{k-1})||_\infty^2 \ (1+g_\delta) + g_\epsilon \\
&\leq n \left(||\tilde{D}^{-1}||_\infty \ ||x_k \ominus x_{k-1}||_\infty \ (1+g_\delta) + g_\epsilon\right)^2 (1+g_\delta) + g_\epsilon \\
&\leq n \left(||\tilde{D}^{-1}||_\infty \ ||x_k - x_{k-1}||_\infty \ (1+\delta) \ (1+g_\delta) + g_\epsilon\right)^2 (1+g_\delta) + g_\epsilon \\
&\leq n \left(||\tilde{D}^{-1}||_\infty \ (||x_k - x||_\infty + ||x_{k-1} - x||_\infty) \ (1+\delta) \ (1+g_\delta) + g_\epsilon)\right)^2 (1+g_\delta) + g_\epsilon \\
&\leq \left(||\tilde{D}^{-1}||_\infty \ \left(\rho^k \ f_o + \frac{1-\rho^k}{1-\rho} d_{\text{mag}} + \rho^{k-1} f_o + \frac{1-\rho^{k-1}}{1-\rho} d_{\text{mag}}\right) (1+\delta) \ (1+g_\delta) + g_\epsilon\right)^2 \\
&\quad n(1+g_\delta) + g_\epsilon
\end{aligned} \tag{4.12}
$$

Here, we first use the fact that $||.||_{2,f} \leq ||.||_2(1+g_\delta) + g_\epsilon$, and then use the compatibility relation between $||.||_2$ and $||.||_\infty$, i.e., $||.||_2 \leq \sqrt{n} \ ||.||_\infty$.

Since $x_o = \mathbf{0}$, $f_o = ||x_o - x||_\infty = ||x||_\infty$. Therefore, a bound on $r_k \otimes r_k$ depends on the $||x||_\infty$. But, we do not know $||x||_\infty$ a priori, since the entire purpose of Jacobi iteration algorithm is to compute x approximately. Fortunately, we can derive a bound on $||x||_\infty$, and obtain a bound on the computation of residual r_k. The following lemma shows that there exists a bound on $||x||_\infty$.

Lemma 27. *Let A be a $n \times n$ square matrix of floats in format $t = (p, e_{\max})$, and b be an n-dimensional column vector of floats in format $t = (p, e_{\max})$. Let A_{real} be the real projection*

of A, and b_{real} be the real projection of b. Suppose each element of the matrix A and its inverse is finite, and A_{real} is invertible. Let $\hat{d} = (||\tilde{D}^{-1}||_\infty + \epsilon)/(1-\delta)$, $\tilde{R} = \hat{d}||N||_\infty$, and $x_{\text{bound}} = (\hat{d}||b_{\text{real}}||_\infty/(1-\tilde{R})$. Assuming that we start with $x_o = \mathbf{0}$, the infinity norm of real solution $x \triangleq A_{\text{real}}^{-1}\, b_{\text{real}}$ is bounded by x_{bound} if $\tilde{R} < 1$.

Proof. The real-iterative solution vector at k^{th} iteration step for a Jacobi iteration is given by

$$x_k = D^{-1}(b_{\text{real}} - Nx_{k-1}) = -D^{-1}Nx_{k-1} + D^{-1}b_{\text{real}} \tag{4.13}$$

Taking the norm on both sides,

$$\begin{aligned}
||x_k||_\infty &= || - D^{-1}Nx_{k-1} + D^{-1}b_{\text{real}}||_\infty \\
&\leq || - D^{-1}N||_\infty \; ||x_{k-1}||_\infty + ||D^{-1}b_{\text{real}}||_\infty \\
&\leq R \; ||x_{k-1}||_\infty + f; \quad [R := ||D^{-1}||_\infty \; ||N||_\infty;\; f := ||D^{-1}||_\infty \; ||b_{\text{real}}||_\infty] \\
&\leq R \; (R \; ||x_{k-2}||_\infty + f) + f \\
&= R^2 \; ||x_{k-2}||_\infty + Rf + f \\
&\leq R^2 \; (R \; ||x_{k-3}||_\infty + f) + Rf + f \\
&\vdots \\
&\leq R^k \; ||x_o||_\infty + f(1 + R + R^2 + \ldots + R^{k-1}) \\
&= f(1 + R + R^2 + \ldots + R^{k-1}); \quad [\; ||x_o||_\infty = 0 \;]
\end{aligned} \tag{4.14}$$

Since $R < 1$ (can be proved from the fact that $\tilde{R} < 1$), the geometric series in (4.14) converges and we get a bound for $||x_k||_\infty$ as

$$||x_k||_\infty \leq \frac{f(1-R^k)}{1-R} \tag{4.15}$$

Since $||x||_\infty = \lim_{k\to\infty} ||x_k||_\infty$ (we proved this using the lemma lim_of_x_minus_xk_is_zero in

Coq),

$$
\begin{aligned}
||x||_\infty &\leq \lim_{k\to\infty}\left(\frac{f(1-R^k)}{1-R}\right) \\
&= \lim_{k\to\infty}\left(\frac{f}{1-R}\right) - \lim_{k\to\infty}\left(\frac{f}{1-R}R^k\right) \\
&= \frac{f}{1-R} - \left(\frac{f}{1-R}\right)\lim_{k\to\infty} R^k \\
&= \frac{f}{1-R} - \left(\frac{f}{1-R}\right)0; \quad \text{Since } R<1 \\
&= \frac{f}{1-R}
\end{aligned} \tag{4.16}
$$

Thus,

$$
||x||_\infty \leq \frac{||D^{-1}||_\infty \; ||b_{\text{real}}||_\infty}{1-||D^{-1}||_\infty \; ||N||_\infty} \tag{4.17}
$$

Since ϵ, δ are non-negative, we can show that

$$
\frac{||D^{-1}||_\infty \; ||b_{\text{real}}||_\infty}{1-||D^{-1}||_\infty \; ||N||_\infty} \leq x_{\text{bound}}
$$

Hence,

$$
||x||_\infty \leq x_{\text{bound}} \tag{4.18}
$$

□

We state the lemma 27 in Coq as

```
Lemma x_bound_exist {t} {n:nat} (A : 'M[ftype t]_n.+1) (b : 'cV[ftype t]_n.+1)
 (Hinv: ∀ i, finite (BDIV (Zconst t 1) (A i i)))
 (Ha : ∀ i j, finite (A i j)):
 let A_real := FT2R_mat A in
 let b_real := FT2R_mat b in
 let x := A_real^−1 *m b_real in
 let x1 := x_fix x b_real A_real in
 let A2_real := FT2R_mat (A2_J A) in
 let R_def :=
 (((vec_inf_norm (FT2R_mat (A1_inv_J A)) + default_abs t) / (1 − default_rel t)) *
                    matrix_inf_norm (A2_real)) in
 (R_def < 1) →
 A_real \in unitmx →
  (vec_inf_norm x1 <=
```

```
(((vec_inf_norm (FT2R_mat (A1_inv_J A)) + default_abs t) / (1 − default_rel t)) *
  vec_inf_norm (\brealeal)) / (1 − R_def)).
```

The lemma `x_bound_exist` holds if each element of A and its inverse is invertible, A_{real} is invertible, and $\tilde{R} < 1$.

4.6 Proof of correctness

In this section, we discuss the next layer of the proof structure, *refinement proof*. In this proof layer, the C program is proven correct with respect to a floating-point functional model.

4.6.1 Refinement proof in Coq

Here's the C program for a single iteration of Jacobi iteration,

```
double jacobi2_oneiter(double *A1, struct crs_matrix *A2, double *b, double *x, double *y)
{
  unsigned i, n=crs_matrix_rows(A2); double s = 0.0;
  for (i=0; i<n; i++) {
    double u = b[i] − crs_row_vector_multiply(A2,x,i);
    double a1 = A1[i], new = (1/a1)*u, r = a1*(new − x[i]);
    s = fma(r,r,s);
    y[i] = new;
    }
  return s;
}
```

It loops over rows i of the matrix, which is also elements i of the vectors b and x. For each i it computes a new element y_i of the result vector, as well as an element r_i of residual vector. It returns s, the sum of the squares of the r_i. By carefully computing r_i from y_i, and not vice versa, we can prove (in Coq, of course) that all overflows are detected: if s is finite, then all the y_i must be finite.

Here's the program for iteration until convergence ($s < \tau^2$), giving up early if there's overflow (tested by `s*0=0.0`, since if s overflows then `s*0` is NaN) or if `maxiter` iterations is reached.

```
double jacobi2(double *A1, struct crs_matrix *A2, double *b, double *x, double τ²,
                                                          unsigned maxiter) {
  unsigned i, n=crs_matrix_rows(A2);
  double s, *t, *z=x, *y = (double *)surely_malloc(n*sizeof(double));
```

```
  do { s = jacobi2_oneiter(A1,A2,b,z,y);
       t=z; z=y; y=t;
       maxiter--;
  } while (s*0==0.0 && s ≥ τ^2 && maxiter);
  if (y==x) y=z; else { for (i=0; i<n; i++) x[i]=y[i]; }
  free(y);
  return s;
}
```

This program starts with $x^{(0)}$ in x, computes $x^{(1)}$ into y, then $x^{(2)}$ back into x, and so on. It mallocs y for that purpose and frees it at the end. Depending on whether the number of iterations is odd or even, it may need to copy from y to x at the end.

The matrix A2 in the C program jacobi2() is a sparse matrix, and has been represented in the standard sparse representation called the *Compressed Row Storage (CRS)* form. The function jacobi2_oneiter() thus performs a sparse matrix vector multiply – crs_row_vector_multiply on the matrix A2 and a vector x. The CRS implementation has been discussed in detail by [89].

The C program jacobi2() has been proven correct with respect to a functional model

```
Definition jacobi {t: type} (A: matrix t) (b: vector t) (x: vector t) (acc: ftype t) (n: nat) :
        ftype t * vector t :=
  let A1 := diag_of_matrix A in
  let A2 := remove_diag A in
  iter_stop norm2 (jacobi_residual A1 A2 b) (jacobi_iter A1 A2 b) (Nat.pred n) acc x.
```

The jacobi function returns the solution vector x_k after k iterations, and implements the stopping criteria (s*0==0.0 && s ≥ τ^2 && maxiter). This functional model was proven equivalent to jacobi_n, i.e., both jacobi_n and jacobi return the same vector x_k once the program terminates well within maxiter.

Correctness theorem The jacobi2 function is specified and proved with a VST function-spec that we will not show here, but in English it says,

Theorem 28 (body_jacobi2)**.** *Let A be a matrix, let b and $x^{(0)}$ be vectors, let A1p be the address of a 1-dimensional array holding the diagonal of A, let A2p be the address of a CRS sparse matrix representation of A without its diagonal, let bp and xp be the addresses of arrays holding b and $x^{(0)}$, let τ be desired residual accuracy, and let maxiter be an integer. Suppose these* ***preconditions*** *hold: the dimension of A is $n \times n$, b and $x^{(0)}$ have length n, $0 < n < 2^{32}$, $0 <$ maxiter $< 2^{32}$, all the elements of A, b, x, acc^2 (as well as*

the inverses *of A's diagonal) are finite double-precision floating-point numbers; the data structures A1p,A2p,bp have read permission and xp has read/write permission. Suppose one calls jacobi2(A1p,A2p,bp,xp,acc,maxiter); then afterward it will satisfy this* ***postcondition:*** *the function will return some s and the array at xp will contain some* $x^{(k)}$*, such that* $(s, x^{(k)}) \simeq \mathsf{jacobi}\, A\, b\, x\, \tau^2, \mathsf{maxiter}$*, where* $\simeq$ *is the floating-point equivalence relation and jacobi is our functional model in Coq of Jacobi iteration; and the data structures at A1p,A2p,bp will still contain their original values.*

The proof of Theorem 28 is a refinement proof – the C program is a refinement of, or correctly implements the functional model jacobi. This proof was done using the Verified Software toolchain (VST), and was implemented by Prof. Andrew Appel.

4.6.2 Main theorem

The C program jacobi2() (and its functional model jacobi) satisfies either of two different specifications:

Theorem 28 (above): if A, b, x satisfy the *basic preconditions*[2] then perhaps Jacobi iteration will return after maxiter iterations – having failed to converge – or might overflow to floating-point infinities and stop early. But even so, the result (s, y) will be such that the (squared) residual $s = |Ay - b|_2^2$ accurately characterizes the result-vector y: if y contains an ∞ then $s = \infty$, but if $\sqrt{s} < \tau$ then y is indeed a "correct" answer. That's because the *functional model* preserves infinities in this way, and the C program correctly implements the model.

Theorem 29: if A, b, x,maxiter satisfy the *Jacobi preconditions* then the result (s, y) will be such that $s = |Ay - b|_2^2$, and $\sqrt{s} < \tau$ and indeed y is a "correct" finite answer. In fact this is our main result:

Theorem 29 (`main_jacobi`)**.** *If the inputs satisfy the Jacobi preconditions, then the C program will converge within k iterations to an accurate result.*

Proof. Using Theorems 23 and 28, with some additional reasoning about the stopping condition in the functional model of the C program. □

The theorem main_jacobi integrates the proof of correctness (body_jacobi2) and the proof of convergence (jacobi_iteration_bound_lowlevel).

[2] A an $n \times n$ matrix; b and x dimension n; $0 < n < 2^{32}$; A, b, x all finite; A, b, x stored in memory in the right places—but nothing else about the *values* of A, b, x.

Jacobi iteration on inputs not known to satisfy the Jacobi preconditions Theorem 28 is useful on its own, since there are many useful applications of stationary iterative methods where one has not proved in advance the convergence conditions (e.g., Jacobi preconditions)—one just runs the program and tests the residual. For such inputs we must take care to correctly stop on overflow.

The induction hypothesis, for $0 < k \leq \mathsf{maxiter}$ iterations, requires that x_k has not yet overflowed, otherwise our sparse-matrix reasoning cannot be proved. Therefore the program must check for floating-point overflow in x_k after each iteration. In order to do that efficiently, the program tests $s \otimes 0 = 0$ (which is a portable and efficient way of testing that s is finite); and if so, then x_{k+1} is all finite.

Thus, the Theorem 29 provides an end-to-end proof for the Jacobi algorithm, i.e., our proofs are faithful to the details of the implementation, including C program semantics and floating-point arithmetic.

4.7 Empirical discussion

While we have formalized some theoretical bounds on forward error and the minimum number of iterations required to converge, it is useful to compare them with the actual computed values, so that their feasibility can be accessed in practical adoption by numerical analysts. We therefore perform some numerical experiments and compare the results. We compare the theoretical value for minimum number of iterations $k_{\min}$ (theoretical) (eq. 4.9), required for the Jacobi iteration to converge, against the actual number of iterations required by the C/C++ program to converge, $k_{\min}$ (computed). We use the BigFloat library from Julia to compute $k_{\min}$ (theoretical) in arbitrary floating-point precision. The BigFloat library in Julia is a wrapper around the *GNU MPFR* library [53]. The BigFloat(x) method takes a floating-point number x, in our case Float64 and generates an arbitrary precision floating-point number, with the default precision set to 256 and the default rounding mode being *Rounding to nearest with ties being rounded to the nearest even number.* We write a C++ program to compute $k_{\min}$ (computed) in double-precision floating-point, which uses dense matrix and vector operations in the Jacobi iteration implementation. This implementation is similar to the implementation jacobi2_oneiter and jacobi2, except that we use a dense matrix-vector multiplication. We analyze a linear system $Ax = b$, with $A(1.0, -3.0, 1.0)$ and $b = -\mathbf{1.0}$. $A(1.0, -3.0, 1.0)$ represents a tri-diagonal matrix with the entry -3.0 in its main-diagonal, and 1.0 in its lower diagonal and upper diagonal entries. b is a constant vector of -1.0. We compare $k_{\min}$ (theoretical) and $k_{\min}$ (computed) for two parameters: *dimension of the linear system* N and *user-specified tolerance* τ. Table 4.2 and Figure 4.4 illustrate the

Dimension (N)	$k_{\min}$ (theoretical)	$k_{\min}$ (computed)
10	39	19
50	41	23
100	42	24
500	44	26
1000	45	27
5000	47	29

Table 4.2: Comparison of $k_{\min}$ with the dimension of the matrix A.

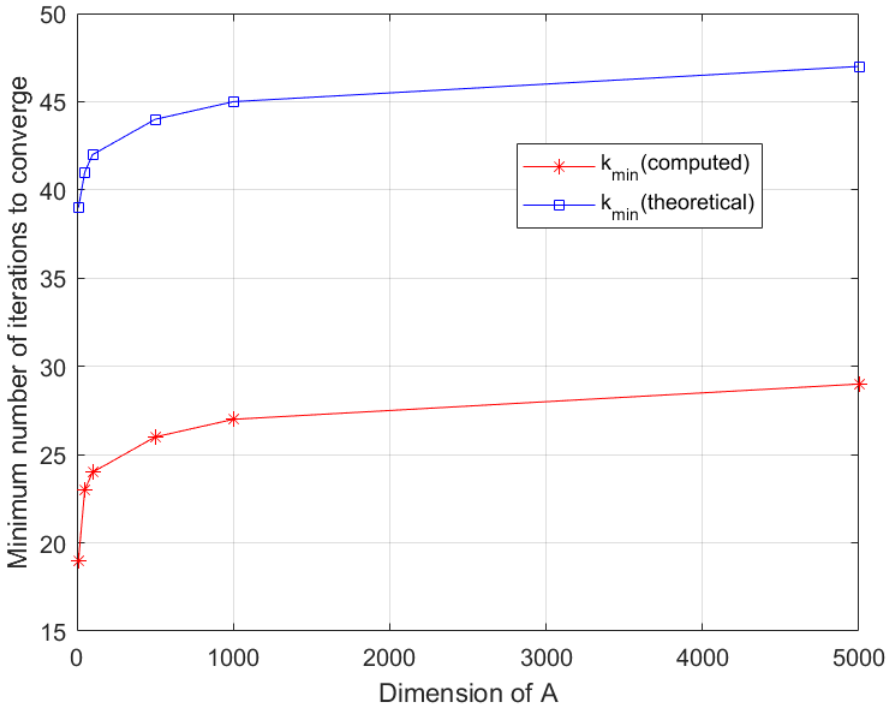

Figure 4.4: Comparison of $k_{\min}$ with the dimension of the matrix A.

comparison of $k_{\min}$ with respect to the dimension of the linear system N. It can be observed that the value of $k_{\min}$ (theoretical) is greater than $k_{\min}$ (computed), and each of their values increase as we increase N. This is expected because $k_{\min}$ (theoretical) uses the maximum error bound from each floating-point operation and error bounds from dense matrix-vector operation. We propagate the maximum forward error bound for each iteration of the Jacobi method, for the computation of $k_{\min}$ (theoretical) in our error-analysis framework. The actual Jacobi method might not necessarily propagate the maximum forward error for each iteration, which explains lower values of $k_{\min}$ (computed). Both $k_{\min}$ (theoretical) and $k_{\min}$ (computed) increase with N, because of increase in accumulation of floating-point error due to increase in floating-point operations.

Tolerance (τ)	$k_{\min}$ (theoretical)	$k_{\min}$ (computed)
10^{-3}	25	16
10^{-4}	31	18
10^{-6}	42	24
10^{-9}	59	33
10^{-10}	65	35
10^{-11}	73	38

Table 4.3: Comparison of $k_{\min}$ with the user-specified tolerance τ.

Table 4.3 and Figure 4.5 illustrate the comparison of $k_{\min}$ with respect to the user-specified tolerance τ. It can be observed the minimum number of iterations required to converge to a solution such that the residual is less than the user-specified tolerance τ, increases as we decrease τ. This is consistent with the equation 4.9, which establishes an inverse relation between $k_{\min}$ and τ. The value of $k_{\min}$ (theoretical) is greater than $k_{\min}$ (computed) at each data point for the same reason as explained earlier.

Comparing the theoretical bound for the Jacobi forward error with the actual error: We compare the theoretical bound for forward error from equation 4.8 for the Jacobi method, computed after k iterations with the computed error from implementation of the algorithm in C++. We compute the theoretical error bounds using the arbitrary floating-point precision provided by the `BigFloat` library in Julia. We compute the actual or the computed error in double precision in C++. The error bounds are obtained for the implementation of the Jacobi iteration algorithm to compute approximate solution for the matrix system $Ax = b$ corresponding to the differential equation $\frac{d^2u}{dx^2} = -1$. This differential equation is discretized using a centered finite difference scheme to obtain the linear system $Ax = b$. As observed from the Table 4.4 and the Figure 4.6, the theoretical error bound is

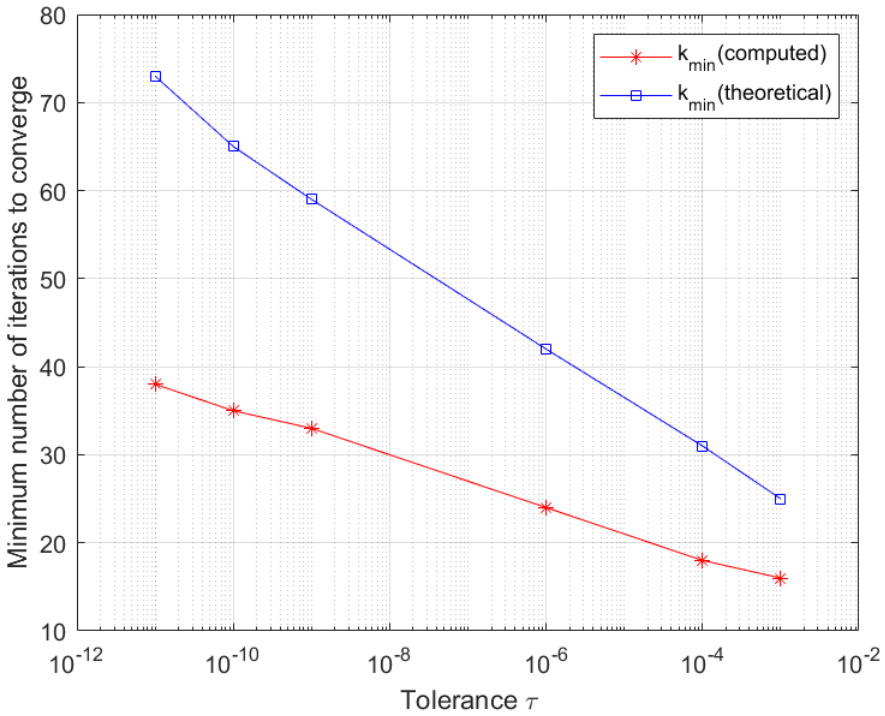

Figure 4.5: Comparison of $k_{\min}$ with the user-specified tolerance τ. The plot is in semi-log in $x-$ axis.

about *twice* the computed error in magnitude.

4.8 Conclusion

In this work, we have presented a formal proof in Coq of the correctness, accuracy, and convergence of Jacobi iteration in floating-point arithmetic. We derived *generic* maximum error bounds for the matrix-vector operations, i.e., matrix-vector multiplication, vector addition and subtraction, matrix inversion, etc., and used these bounds to derive a forward error bound for the Jacobi iteration. We then used this forward error bound to prove the convergence of Jacobi iteration by proving that the residual after $k_{\min}$ iterations is less than the user-specified tolerance. This convergence proof takes into account exceptional floating-point behaviors including overflow and underflow, and we derive conditions on the inputs A, b, x_o to prove the absence of overflow, formally. We also derive a closed form expression for $k_{\min}$, and compare this bound with the actual number of iterations required to converge, as computed using a C++ implementation of the Jacobi algorithm. From an empirical comparison of this theoretical iteration bound $k_{\min}$ (theoretical) and the actual number of iterations

k	Computed error	Theoretical error bound
10	0.06433	0.12346
20	0.03798	0.12346
50	0.05486	0.12346
100	0.06086	0.12346
500	0.06173	0.12346
1000	0.06173	0.12346
5000	0.06173	0.12346
10000	0.06173	0.12346
100000	0.06173	0.12346

Table 4.4: Comparison of the theoretical forward error bound 4.8 and the computed forward error for the Jacobi iteration.

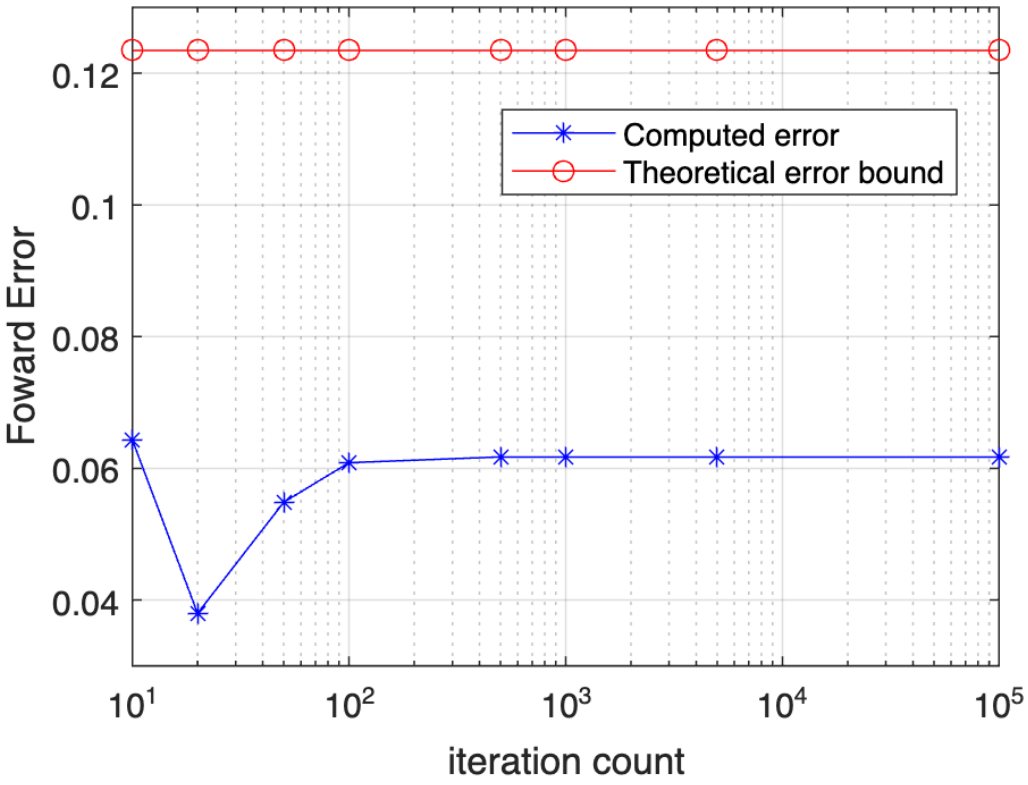

Figure 4.6: Comparison of the theoretical forward error bound 4.8 and the computed forward error for the Jacobi iteration. The plot is semi-log in x-axis.

required to converge $k_{\min}$ (computed) , we observe that $k_{\min}$ (theoretical) is greater than the $k_{\min}$ (computed) and both $k_{\min}$ (theoretical) and $k_{\min}$ (computed) increase as we increase the dimension of the coefficient matrix A. We further compare the theoretical forward error bound with the actual forward error for the Jacobi iteration method, and perform numerical experiments on the error bounds for the naive-dot product, which is discussed in the Appendix B. We also connect the proof of accuracy and convergence to an actual C implementation of the Jacobi iteration algorithm which uses the sparse matrix-vector operations. This C program is proven correct against a floating-point functional model, using the Verified Software Toolchain. In this way, we provide an end-to-end correctness of the Jacobi iteration algorithm in floating-point arithmetic. Finally, we present empirical results comparing our theoretical bounds with respect to the actual/computed bounds.

CHAPTER 5

Related Work

Formalization of Numerical analysis: There have been significant contributions by the interactive theorem proving community on formalization of important results in numerical analysis, in the past few years. This has been motivated by the development of libraries on real analysis, linear algebra, functional analysis, etc. in interactive theorem provers like Coq [34], HOL [58], PVS [110], and these libraries have been used in many applications pertaining to the analysis of differential equations.

In terms of development of libraries in theorem provers, Sylvie Boldo and her colleagues have made immense contributions in the formalization of results from functional analysis in Coq. They [19] formalized the *Lax–Milgram theorem*, which is an important result in finite element methods. This formalization led to the development of a general purpose library on linear mappings, continuous linear mappings, operator norms, Hilbert spaces, etc. Besides functional analysis, they Boldo et al have also made important contributions to the real analysis in their development of the Coquelicot [28] library. The Coquelicot library provides necessary infrastructure for defining derivatives, limits, integrals, algebraic structures like rings, fields, abelian groups, etc., and topology like metric spaces, normed modules, complete normed modules, etc. *Both of these formalizations of functional analysis and real analysis have been really helpful in our formalization of the convergence of finite difference schemes* – Lax–equivalence theorem*, and formalization of asymptotic convergence of iterative convergence error.* Boldo et al also formalized the Lebesgue integration of non-negative functions [18] and Tonelli's theorem in Coq [24], with the aim of developing necessary infrastructure like sigma algebra, product measures and their uniqueness, and construction of iterated integrals, with application to probability theory, real analysis and numerical mathematics.

The mathematical components library [100] formalizes concepts from linear algebra, finite sequences, finite sets, and algebraic structures in Coq. This library was a result of a six year long work on formalization of the Feit-Thompson odd order theorem [57]. The big operators formalization by [16] provides necessary infrastructure for iterated summation and product,

as well as for computing the maximum and minimum in a list of real and natural numbers. Besides the formalization of basic linear algebra, the mathematical components library has been extended to real analysis through the development of mathcomp–analysis [1]. This extension has been facilitated by the recent work on design of hierarchical algebraic structures in dependent type theory by [2]. In [3], the authors formalize a set of techniques and notations to carry asymptotic analysis in Coq. This work was integrated into the mathcomp–analysis library and provides a rigorous approach to delay existential quantification in the classical $\epsilon - \delta$ reasoning to improve proof stability and readability, and develops a small theory for the *Bachmann–Landau* notations or little-o and big-$\mathcal{O}$ notations. Asymptotic analysis forms the core of reasoning about numerical approximations, and this development is an important contribution towards a more generalized classical reasoning. *We have used the rich formalization of linear algebra in the mathematical components library in our formalization of iterative convergence error, both in reals and floats.*

The development of these libraries have facilitated its use in various applications of practical interest. Boldo et al. formalized the consistency, stability and convergence properties of a centered scheme for 1-D wave equation [21]. They used the Coquelicot real analysis library extensively in their formalization. *However, their formalization of convergence was done for a specific finite-difference method. In our work, we formalized the convergence of a general class of finite-difference schemes, and later instantiated it with a centered finite-difference scheme.*

Evgeny Makarov and Bas Spitters formalized a constructive proof of the Picard–Lindelöf theorem in Coq [101] based on the constructive reals from CoRN library and the MathClasses library. Ioana Pasca formalized the Kantorovich theorem in Coq to prove convergence of Newton method [112]. During this formalization, they also formalized concepts of multivariate analysis, and formally verified criteria for regularity of interval matrices. Micaela Mayero formalized a proof of correctness of the algorithm used by an automated differentiation tool called Odyssée, which deals with FORTRAN programs. Martin-Dorel et al formalized Taylor models in Coq [103]. Taylor models consist of a pair (P, Δ), where P is a polynomial in a given basis, and Δ is an interval error bound. They also formalized the Taylor–Lagrange theorem, which we use for our formalization of the Lax–equivalence theorem in Coq. Mahboubi et al. [99] provided an efficient method for computing and proving bounds on definite intergrals in Coq.

Besides Coq, numerical analysis of ODEs have also been done in Isabelle/HOL [78]. They formalize the *initial value problem* of ODEs and formally prove the existence of its solution using the Picard–Lindelöf theorem. They also formalize a generic one-step method for obtaining numerical approximations of the solution, and formalize local and global error

analysis for this method. Immler et al. [77, 80, 81] formalized flows, Poincaré map of dynamical systems, and verified rigorous bounds on numerical algorithms in Isabelle/HOL. In [76], Immler formalized a functional algorithm that computes enclosures of solutions of ODEs in Isabelle/HOL, and proved safety of these enclosures. Fabian Immler and Yong Kiam Tan formalized the Poincaré-Bendixon theorem [79] in Isabelle/HOL to prove the existence of (limiting) periodic behavior in planar dynamical systems.

Formalization of floating-point error analysis: A formal analysis of floating-point error has been facilitated by the recent developments in interactive theorem provers and development of automated static analysis tools like MetiTarski [5], Daisy [44], Fluctuat [59], FPTaylor [126], PRECiSA [134] and Gappa [46].

In terms of developing libraries for a formalized floating-point error analysis, Daumas et al provided a first generic formalization of floating-point computations [47] in Coq. This formalization is generic in terms of the base of the floating-point representation, and the mantissa and exponent of the floating-point format. Boldo et al later provided a comprehensive formalization of a generic floating-point format and the IEEE-754 standard [75] in Coq as a general purpose library *Flocq* [30]. They then extended [17] the CompCert verified compiler [97] to compiling floating-point programs by augmenting the platform with the formalization of IEEE-754 floating-point standard in Flocq. Boldo et al also implemented a mechanism for calling *Gappa* [46] within Coq for verifying floating-point programs. Gappa [46] is a tool to automatically discharge proof obligations involving floating-point computations. Gappa uses interval arithmetic and forward error analysis to generate bounds on mathematical expressions that involve rounded and exact operators. The proofs generated by Gappa can be automatically checked with proof assistants like Coq or HOL-Light. Martin-Dorel et al provide a tactic [102] in Coq to compute tight bounds on univariate functions, which is based on the formalization of floating-point arithmetic and interval arithmetic, and relies on on-the-fly computation of Taylor polynomials. In [48], the authors use the Gappa tool to verify tight error bounds for elementary functions.

John Harrison made important contributions in the development of a generic floating-point library in the HOL theorem prover [63]. This library develops a theory for floating-point computations, which is generic in terms of the floating-point format, and was also specialized to prove correctness of the implementation of various pieces of the mathematical software used in *Merced*, the first implementation of Intel's IA-64 computer architecture. He then applied this formalization to formally verify [64] an algorithm that evaluates transcendental functions like *sine* and *cosine* in double-extended floating-point precision. In [62], John Harrison formalized an algorithm given by Tang [128] to compute the value of *exponential*

function in floating-point arithmetic. Following Tang, he formalized three sources of error – error in range reduction for large input intervals, error in polynomial approximation of the exponential function, and the rounding error, and also treats exceptional behavior of floating-points like overflow and underflow, formally. In [65], John Harrison formalized in HOL-Light, the division algorithm in floating-point arithmetic for the IA-64 computer architecture. John Harrison also formalizes [66] in HOL-Light, square root algorithms based on fused-multiply add (FMA), developed for Intel Itanium architecture.

Becker et al formalized a new language *Icing* [11], which provides formal semantics for fast-math style optimizations in a verified compiler. They then extended this formal semantics in the verified CakeML compiler (*RealCake* [12]) to generate an end-to-end correctness of the fast-math style optimization of floating-point arithmetic which includes an accuracy bound for the result from this optimized code. Becker et al developed an automated tool called *FloVer* [14] to certify the round-off error bounds generated by static analysis tool, like *Daisy* [44]. FloVer proves correctness of each analyzed expression with respect to concrete bit-level IEEE-754 floating-point semantics [75], and the soundness of this tool was verified in Coq and HOL4.

The formal methods group at NASA have also made immense contributions to the formalization of floating-point arithmetic. They developed a modular static analysis technique [105] for computing provably sound over-approximation of floating-point round-off errors, which has been implemented in their prototype tool *PRECiSA*. Given a floating-point expression, this tool computes a symbolic upper bound automatically using denotational semantics, and generates a proof certificate of soundness in the PVS theorem prover [110]. Titolo et al implemented an abstract analysis framework [134] in PRECiSA, which defines a parameterized semantics that collects an error expression representing provably strong round-off error bounds for each combination of an ideal and floating-point computational path of a functional program. This framework also provides a strong support for typical programming language constructs like conditions, recursions and loops, and defines a widening operator to ensure convergence of recursive functions and loops. One of the issues with floating-point programs is the case of *unstable tests*, which arise due to the divergence between the control flow of a real program and a floating point programs at *conditionals* due to accumulation of round-off error. Titolo et al developed a program transformation techique [135] to transform the original program into another program that conservatively (and soundly) detects and corrects unstable tests. The correctness of this transformation was formally verified in the PVS theorem prover.

There have been some work in automating the floating-point error analysis to some extent, by extending the libraries of SMT solvers to support floating-point arithmetic. Rümmer and

Wahl [118] proposed a theory for floating-point arithmetic in the SMT-LIB 2.0 standard, which defines semantics for floating-point arithmetic following the IEEE-754 standard [75]. They defined a floating-point format and added support for all the five rounding modes standarized by the IEEE-754, along with a formal treatment of the exceptional floating-point values: $+\infty$, $-\infty$ and NaN. In [41], the authors extend SMT solvers with floating-point reasoning. They propose a three-layered approach to reason about floating-point arithmetic in SMT solvers, which extends the SMT-LIB standard of reducing a floating-point expression to reals with explicit rounding and then reasons about rounding using the lemmas from Flocq. This strategy was implemented in the Alt-Ergo SMT solver.

In the context of rigorous numerical approximation of the solutions of ordinary differential equations, Boldo et al formalize [25] tight error bounds on the solutions obtained from integration schemes like the Runge-Kutta (RK) family of methods (RK-2 and RK-4 methods) and Euler's method in floating-point arithmetic. This analysis however does not take into account the effect of overflow and underflow in their formalization. Later [26], they revised the analysis by taking into account overflow and underflow, and refined the error bounds using optimal bounds on relative errors studied in details by Jeannerod and Rump [82].

Work on end-to-end verification: There have been significant work in developing an end-to-end verification framework, which connects an actual implementation in a programming language like C, to the formal analysis developed using an interactive theorem prover or an automated theorem prover.

Some early experiments on verifying floating-point poperties of numerical programs were done by Slyvie Boldo and Claude Marché in [29]. They used the Frama-C [43] tool, its Jesse plugin and the Why platform, and provers like Coq, Z3, Alt-Ergo, Gappa [46], CVC3 to carry an end-to-end verification of problems like Sterbenz subtraction [127], Veltkamp/Dekker Algorithm [50, 136] to compute the exact error of floating-point multiplication, and the Kahan algorithm for accurate discriminant [84]. Boldo and her colleagues [22, 23] also provided an end-to-end verification of a C program implementing a 1-D wave equation. They used the Frama-C [43] tool to annotate the C program with formal specifications of correctness, especially the *method error and floating-point errors*, and generate theorems that guarantee soundness of the code. These theorems were then discharged using the SMT solvers, Gappa [46], and Coq.

The work by Andrew Appel and Yves Bertot [6] provides a modular approach to perform end-to-end verification of a C program implementing Newton's method to compute square root. The modularity in their approach comes from the fact that the proof of correctness of the C program was carried separately using VST, from the proof of numerical accuracy and

numerical correctness, which was done by leveraging the IEEE-754 standard formalization done in the Flocq library in Coq. *This work was inspirational to our work on end-to-end verification of C program implementing the Jacobi iteration algorithm.*

Becker et al developed a tool called *Dandelion* [13], which generates a certified polynomial approximation of transcendental functions, and provided an end-to-end correctness and accuracy guarantees by extracting a verified binary with the formally verified CakeML [93] compiler. This tool takes a Scala program implementing transcendental functions as input, and generates a certificate containing the original transcendental function, its polynomial approximation, the maximum error bound in a given interval, which is proved sound in the HOL4 theorem prover [58].

In the context of an end-to-end verification of solutions of ordinary differential equations, Ariel Kellison and Andrew Appel provided a formalization of a C program implementing the Störmer-Verlet method [88]. They verified that the C program faithfully implements this method assuming only the operational semantics of C and of IEEE-754 floating-point arithmetic. They used the VCFloat tool [114] in their formalization. VCFloat automatically generates an annotated real expression with appropriate error bounds from a floating-point expression. This tool provides an infrastructure to prove numerical properties of C programs with trusted base limited to formal specifications of C in Coq, the IEEE-754 floating-point standard formalization in Coq and the underlying logic of Coq. Appel and Kellison extended VCFloat in their tool VCFloat2 [7] with key improvements to VCFloat – better integration with the Coq-interval package and Gappa to produce useful bounds for cases where VCFloat fails, and support for natural-style functional models which are based on *reification approach*– lifting a formula to an abstract syntax tree for symbolic analysis, thereby making these models independent of C or CompCert. VCFloat on the other hand took input expressions from C programs as parsed by the CompCert's [97, 27] front end. *We used the VCFloat2 tool for its Coq library of definitions such as "type" and "fprec" and "ftype", and the VST tool for proving correctness of the C program implementing this algorithm.*

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

In this work, we proposed an approach for end-to-end verification of numerical programs. We started with a differential equation, a 1-D differential equation in our case. We showed an approach to discretize this differential equation using a centered finite difference scheme, in a formal setting such as the Coq theorem prover. We then proved convergence of this scheme by proving its *consistency* with respect to the differential equation and *convergence* of its solution with respect to the "true" solution of the differential equation. To prove convergence, we first formalized the *Lax–equivalence theorem* and then applied this theorem to this particular scheme. The Lax–equivalence theorem is a statement of convergence for the solution obtained from a general class of finite-difference scheme, and we discussed its formalization in detail in Chapter 2. We focused on the *spatial discretization error*, since the discretization of the 1-D differential equation was done in space with a uniform discretization step, Δx. Using the *Taylor–Lagrange theorem*, whose formalization can be found in the Coq.Interval library, we formally proved that this discretization error can be bounded above by a term of the order of $\mathcal{O}(\Delta x)^2$. Since this bound approaches zero in the limit of $\Delta x \to 0$, the centered finite-difference scheme that we used for discretization is *consistent* with respect to the differential equation. We also proved the *stability* of the scheme, by formalizing spectral properties of the coefficient matrix A of the linear system obtained from the resulting discretization using the centered finite-difference scheme. We showed that the l^2 matrix norm of the inverse of the coefficient matrix A, i.e., $||A^{-1}||_2$, is uniformly bounded. Thus, this linear system is robust with respect to any perturbations in the system, and the numerical errors are bounded. With the discretized equation approaching the differential equation in the limit of $\Delta x \to 0$, and the numerical errors being bounded, the solution obtained from this scheme is expected to converge to the "true" solution of the differential equation, i.e., $\lim_{\Delta x \to 0} ||u_{\Delta x} - u|| = 0$.

The above formalization proved the convergence of a numerical solution obtained from the scheme to the "true" solution, assuming that this numerical solution exists. We did not discuss a methodology to obtain such a numerical solution from the original linear system though. One can argue that this solution can be obtained simply (naively) by inverting the coefficient matrix A, using the formula $u_{\Delta x} \triangleq A^{-1}b$. But such matrix inversion is computationally expensive (time complexity is the order of $\mathcal{O}(n^3)$, where n is the dimension of A), and often intractable for large and dense matrices. Therefore, low cost methods such as *iterative methods* are used extensively in the scientific computing community. In this work, we focus on a class of iterative methods called the *stationary iterative methods*, which is presented in Chapter 3. This methods build a sequence of solution vectors x_k, which are an approximation of $u_{\Delta x}$. This approximation introduces another layer of error called the *iterative convergence error*. The main goal in using these iterative methods is to carry sufficient number of iterations such that the iterative convergence error approaches zero in the limit of $k \to \infty$, i.e., $\lim_{k\to\infty} ||x_k - u_{\Delta x}|| = 0$. In this work, we formalize the sufficient and necessary conditions for the convergence of iterative solutions to $u_{\Delta x}$. We then instantiate this theorem to two classical stationary iterative methods – the Gauss–Seidel method and the Jacobi method. Since the proof of convergence involves the computation of eigenvalues of the iteration matrix, which is often expensive and intractable for most practical problems, we formalize easily testable conditions to show convergence of the Gauss–Seidel method. These easily testable conditions, which are captured by the *Reich theorem* [115], rely on the positive definiteness of the coefficient matrix A, which has a specific structure: *real, symmetric, with all elements in the main diagonal as positive.* We then layer the Reich theorem with the main theorem for convergence to prove convergence of the solutions obtained from the Gauss–Seidel iteration algorithm to $u_{\Delta x}$. This approach can be followed for other stationary iterative methods as well. One just needs to come up with easily testable conditions that relate to the proof of the eigenvalues of the iteration matrix being less than 1, and then layer this relation with the main iterative convergence theorem, which we proved in Coq. We then demonstrate convergence on the same centered scheme that we use in Chapter 2 for both the Gauss–Seidel method and the Jacobi method, in Coq.

It is important to note that formalizations presented in Chapter 2 and 3 are done in the field of reals. But the actual implementation of an algorithm is implemented in finite precision. This introduces another layer of error called the *floating-point error*. In Chapter 4, we provide an overview of the floating-point arithmetic, formalization of the basic concepts of the floating-point arithmetic and the IEEE-754 standard in the *Flocq* [30] library in Coq. Since we can no longer talk about the asymptotic convergence of the iterative solutions to $u_{\Delta x}$ in the floating-point arithmetic, we need to bound the floating-point errors or the rounding

errors for each floating-point operations. Therefore, in this work, we develop an error-analysis framework which formalizes norm-wise error bounds for the matrix-vector operations like matrix-vector multiplication, vector addition and vector subtraction. These norm-wise error bounds are computed for the l^∞ norms which also allow us to do componentwise analysis of errors and the bounds obtained are tighter as compared to the l^2 norm. Nevertheless, since l^2 norms are also widely used in numerical programs, we define a compatibility relation between the l^∞ and l^2 vector norms. This helps us connect the residual computation in C program to its error analysis in our framework in Coq. We use the above norm-wise error analysis framework to prove convergence of the solutions obtained from the Jacobi algorithm. The convergence here is defined in terms of the norm of the residual $(b - Ax_k)$ being less than the user-defined tolerance τ. We first prove a forward error bound for the Jacobi iteration algorithm, and then use this to prove convergence for this algorithm. Our error analysis takes into account the exceptional floating-point behaviors including overflow and underflow, and we prove the absence of overflow in the solutions for each iteration, by proving concrete bounds on the inputs A, b, x_o. We go a bit further and also provide the minimum number of iterations $k_{\min}$ need to achieve convergence for the Jacobi iteration. An important point to note is that our proofs of accuracy and convergence are connected to the actual implementation of the Jacobi algorithm in C. The proof of correctness of the C program with respect to a *functional model*, which we develop in Coq and against which we prove our error bounds, is done by Prof. Andrew Appel using the Verifiable Software Toolchain (VST), developed by his group [33]. In this way, we develop an end-to-end framework for verifying numerical programs and take into account the *spatial discretization error, iterative convergence error, floating-point error* and *computer programming error*, as illustrated in Figure 1.1. An overview of our verification effort is illustrated in Figure 6.1.

Lessons learnt: A key lesson we learnt during our formalization effort was in relating Chapter 3 and Chapter 4. When we started the work on Chapter 4, we analyzed the iterative convergence error in reals and the rounding error separately, which was published in our *Correctness workshop, 2022* paper [86]. In this initial experiment, we formalized the forward error bound for the Jacobi iteration, which provides the error incurred after k iterations. Our initial impression when we started generalizing this work to an $n \times n$ matrix was that we could compose this forward round-off error with the real iterative convergence error to get the total iterative error. Mathematically, this would mean composing, using the triangle inequality, the round-off error $||\widehat{x_k} - x_k||$ with the real-iterative convergence error, $||x_k - x||$ to get the total error $||\widehat{x_k} - x||$, where $\widehat{x_k}$ is the numerical iterative solution, x_k is the real iterative solution and x is the *true numerical solution*. However, the bounds obtained

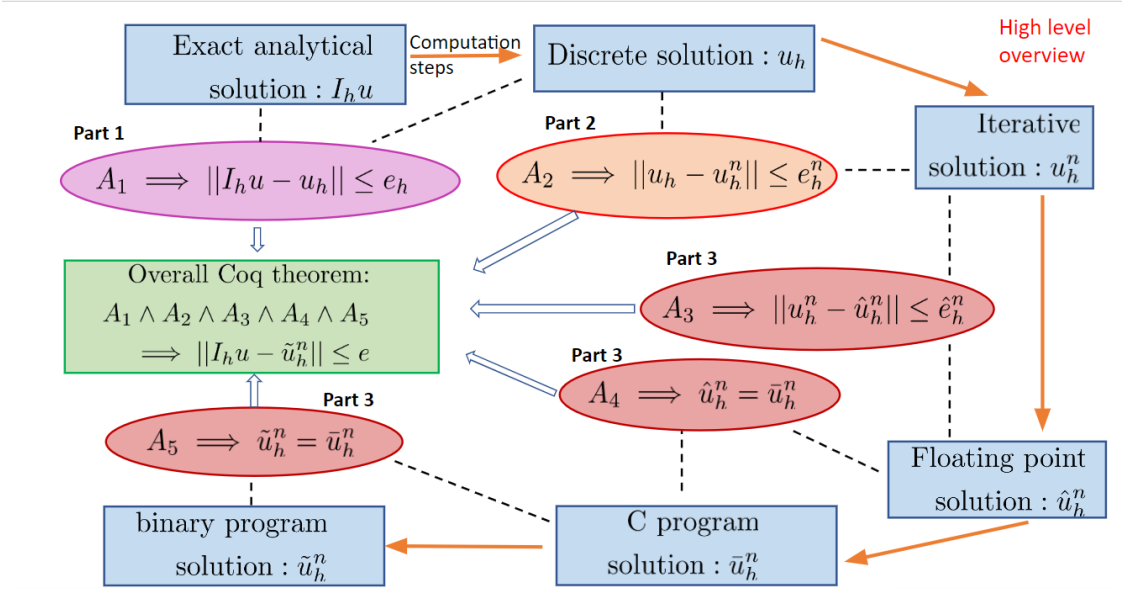

Figure 6.1: High level overview of how our formalization of different errors fit in together. Part 1 refers to our formalization of the *spatial discretization error*, i.e., the Lax–equivalence theorem and the proof of convergence of the centered finite difference scheme. Part 2 refers to our formalization of the *iterative convergence theorem* in the field of reals. Part 3 refers to our formalization of the *floating-point error* for iterative convergence for a concrete implementation of the Jacobi iteration algorithm. Part 3 also tackles the *computer programming error* in the proof of correctness of the algorithm.

through this approach were coarse, and did not leverage the fact that some numerical errors might be cancelled out in the iteration process as we move towards the *true numerical solution*. After discussions with numerical experts, we changed our strategy and used the fixed-point argument to derive a tighter bound for the total iterative error, i.e., directly bounding $||\widehat{x_k} - x||$.

Another lesson that we learnt was during the formalization effort in Chapter 3, with respect to effectively navigating between different libraries in Coq. Continuing our work with the Coquelicot definition of matrices, as discussed in Chapter 2, we ran into issues with formalizing theory of matrix inverse, determinants, eigenvalues etc. When we posted this issue in the *Coq-club* mailing list, the Coq community pointed us to the mathcomp library, which alleviated our issue by providing us with a rich formalization of linear algebra and connecting our formalization to the pre-existing formalization of the Jordan canonical forms. Thus, it is important to make an extensive survey of the existing libraries for any formal analysis, before beginning any new formalization.

Formalization effort: The formalization of the Lax–equivalence theorem and the proof of convergence of the centered difference scheme for solving the 1-D differential equation numerically, took about 15 person-months. The total length of the Coq proof scripts is about 14,000 lines of code, of which less than 1200 lines are specific to the scheme. The rest of the formalization can be reused for a generic symmetric tri-diagonal matrix.

The formalization of the iterative convergence theorem and its application to the classical iterative methods – the Gauss–Seidel method and the Jacobi method, as discussed in the Chapter 3 took about 8 person-months of full time work. The overall length of the Coq proof scripts is about 8.5k lines of code.

The formalization of the accuracy and convergence proofs of the Jacobi iteration algorithm, which includes a generalized error-analysis framework based on norm-wise bounds for matrix-vector operations, spans over 14,000 lines of Coq proof script and took about 5 person-months of full-time work. The proof of correctness of the algorithm, which was carried using the VST tool spans over 2,000 lines of proof script.

6.2 Future work

6.2.1 Direct extension of our work

Our work on formalized error analysis for stationary iterative methods can be extended in the following ways:

- The error analysis framework for floating-points can be extended to a more generic matrix splitting for the stationary iterative methods. The floating-point error analysis that we have done is specialized to the Jacobi iteration method, for which $M = D$ and $N = L+U$. While we have the basic tools like norm-wise error bounds for matrix-vector multiplication, vector addition and subtraction, vector-vector multiplication etc., which can be used for any matrix and vector based error analysis, we can generalize our framework to any stationary iterative methods, which respect *regular splitting*. To do this, the main convergence theorem needs to be parameterized in terms of the matrix splittings, M and N. This theorem can then be instantiated to the Jacobi method and the Gauss–Seidel method, and their implementation in C can be verified correct with respect to this main theorem using the analysis framework that we discussed in Chapter 4.

- Our work on error analysis of stationary iterative methods can also be extended to the Krylov subspace methods. Stationary iterative methods are used as building blocks for

Krylov subspace methods [119]. A Krylov subspace method computes an approximation of the *true numerical solution* $x \triangleq A^{-1}b$ of the linear system $Ax = b$ from the basis of Kyrlov subspace defined as

$$\mathcal{K}_m(A, r_o) = span\{r_o, Ar_o, A^2r_o, \ldots, A^{m-1}r_o\}; \quad r_o = b - A\tilde{x}_o$$

to obtain the approximate solution at m^{th} step defined as

$$\tilde{x}_m = \tilde{x}_o + q_{m-1}(A, r_o)$$

where $\tilde{x}_o$ is an arbitrary initial vector and $q_{m-1}(A, r_o)$ is a polynomial of degree $m-1$ constructed from the basis of the Krylov subspace $\mathcal{K}_m(A, r_o)$.

Since Krylov subspace methods are used to compute approximate solutions for high-dimensional linear algebra problems, they are used for testing the *observability* and *controllability* of systems in control theory. These tests involve checking the rank of Krylov subspaces [67]. The *Arnoldi methods*, which are a class of the Krylov subspace methods, are used for approximating the eigenvalues of large sparse matrices [119]. Krylov subspace methods could also be used in the frequency response calculations in input/output analysis, since one needs to compute $c(A - j\omega I)^{-1}b$ for many values of ω, and the Krylov subspaces are invariant under arbitrary shifts to the matrix A, i.e., $\mathcal{K}_m(A, v) = \mathcal{K}_m(A - sI, v)$ for any s. An approximation of the solution vector for $(A - j\omega I)^{-1}b$ can then be obtained using the formula

$$x_m(\omega) = \beta V_m(H_m - j\omega I)^{-1}e_1$$

as suggested in [45]. In [120], Youcef Saad also discussed about the use of Krylov subspace methods to solve partial pole placement problems in control systems. *Since Krylov subspace methods have several applications in dynamical systems, it is crucial to formalize the error bounds for approximate solution obtained from these methods.*

The error analysis framework that we developed in this work, which includes norm-wise bounds on matrix-vector operations, provides necessary tools for analyzing Krylov subspace methods. Besides, the stationary iterative methods are used as *preconditioners* for Krylov subspace methods, to aid faster convergence. In a preconditioned Krylov subspace method, instead of solving the linear system $Ax = b$, one solves the modified system $M^{-1}Ax = M^{-1}b$, where M^{-1} is chosen as an inexpensive approximation of A^{-1}. Thus, if we start from an initial guess $\tilde{x}_o = 0$, the approximate solution at k^{th} step, $\tilde{x}_k \in \mathcal{K}_k(M^{-1}A, M^{-1}b)$. *The M matrix used in classical stationary iterative methods*

like the Gauss–Seidel method, Jacobi method or the successive over-relaxation (SOR) methods, are used as default preconditioners for the Krylov subspace methods. Thus, our work on error analysis for stationary iterative methods are directly relevant to the Krylov subspace methods in this context.

- The approach for achieving an end-to-end verification by decomposing the errors at each stage of approximation as illustrated in the Figure 1.1 can also be applied to obtain rigorous estimates for the following use cases

  (a) **Reduced order modeling:** We have discussed earlier that a physical system is modeled mathematically using differential equations, which is discretized to be solved numerically in a finite computational domain. These set of discretized equations is often abstracted in terms of a linear system $A_{n\times n}\ x_{n\times 1} = b_{n\times 1}$, and then solved for x using a linear solver. Often the dimension of the system n could be large, especially if such computations are performed real-time in embedded control applications. For those cases, *reduced order modeling* seeks a low dimensional problem $\tilde{A}_{k\times k}\ \tilde{x}_{k\times 1} = \tilde{b}_{k\times 1}$, where $k << n$. This is typically accomplised by finding a trial basis $V_h \in \mathbb{R}^{n\times k}$ that spans a subspace $\mathcal{V} \subset \mathbb{R}^n$ such that $V_h\tilde{x}$ is a *good approximation* to x. *Thus, we have to ensure that this approximation error is small and the rounding error due to its implementation in a finite precision machine is also small. The error analysis framework that we developed is directly applicable for this problem, except that there is an added layer of approximation coming from the dimension reduction, that needs to be formalized.*

  (b) **Uncertainty quantification:** The analysis that we have done in this work assumes that the differential equation is deterministic, i.e., there is no *randomness* is the physical system. However, in most practical systems, there is natural variability in the system. For instance, if one wants to simulate the power generated by a wind turbine, it is not accurate to assume that the wind speed is deterministic. It is more sensible to assume that the wind speed is a *random variable* with a known probability distribution, and seek the probability distribution of the output of the simulation, i.e., power, such that more informed decision can be made. This process of *uncertainty propagation* is relevant in many areas of science and engineering, such as weather forecasting, and the testing of nuclear weapons, as uncertainties abound in those problems. Propagating the uncertainty in random variables through the differential equation amounts to evaluating the solution of the differential equation at pre-specified sampling locations, which are determined by the quadrature points, and using a quadrature rule to estimate

the statistical moments of the quantities of interest. Since the quadrature rule is an approximation of definite integrals, this introduces additional errors. *In the context of this problem, our error analysis framework can be extended by adding another layer of error, which accounts for this approximation error.*

To summarize, the error analysis framework that we have developed is **modular**, and this framework can be extended to problems involving approximation of differential equations by adding an *extra layer of approximation error*.

6.2.2 Automation

A drawback of this work is the lack of automation. Most of the proofs are done interactively and rely heavily on rewriting and the use of Coq tactics to build a proof script. This often leads to large proof scripts, which could be difficult to maintain and lead to robustness issues for formalization of more complicated problems. While we try to alleviate this issue by introducing some abstraction in terms of intermediate lemmas which could be used across theorems, there is still scope for more automation.

One of the ways in which automation can be achieved is the use of Coq's tactic language *Ltac* [51] or *Ltac2* [113]. Coq's tactic language is a small functional core with recursors, and provides powerful pattern matching for Coq terms and proof contexts. This allows us to manipulate the Coq term directly at the desired location, group Coq's tacticals, and perform repeated rewrites by leveraging its powerful pattern matching feature, in a few lines of Coq. For instance consider the proof of the following theorem in Coq

$$a : R, b : R, c : R \ \vdash a + b + c - a = b + c$$

If we were to prove this theorem in Coq using the set of primitive Coq tactics, the Coq script would be as illustrated in the listing 6.1

Listing 6.1: Proof script without tactic language

```
Require Import Reals Psatz.
Local Open Scope R_scope.
Theorem prove_simple:
  ∀a b c : R, a + b + c − a = b + c.
Proof.
intros.
assert (a + b + c − a = a + b + c + (− a)).
{ reflexivity. } rewrite H.
```

```
rewrite Rplus_comm. rewrite Rplus_assoc.
rewrite <−Rplus_assoc. rewrite Rplus_comm.
assert ((− a + a) = a − a).
{ rewrite Rplus_comm. reflexivity. } rewrite H0.
rewrite Rminus_diag_eq; try reflexivity.
rewrite Rplus_comm. rewrite Rplus_0_l.
reflexivity.
Qed.
```

The proof of theorem proof_simple is done by replacing the term $(a+b+c-a)$ with $(a-a)+b+c$. Using the lemmas from the Coq standard library, we need to perform a lot of rewrites, which leads to larger proof script. However, this proof can be discharged succintly using the Coq's tactic for nonlinear arithmetic nra, as illustrated in the listing 6.2

Listing 6.2: Proof script with tactic language

```
Theorem prove_tactic:
  ∀ a b c:R,
  a + b + c − a = b + c.
Proof. intros. nra. Qed.
```

The tactic nra provides an automated decision procedure for non-linear arithmetic and is defined using the *Ltac* language as

```
Ltac nra := first [ Lra.nra | Lqa.nra ].
```

nra performs recursive search in a Coq term using pattern matching, invokes the lemma from the standard real library to perform rewrites, and solves atomic propositions over reals. Thus, Coq's tactic language provides a powerful infrastructure to perform such automation. *In the context of our work, this tactic language could be used to define automated decision procedures for performing norm-wise error bound analysis in the proof context and perform repeated rewrites for similar expressions.*

Another approach for automation is leveraging the existing interface between SMT solvers and Coq, *SMTCoq* [52], to invoke the floating-point theory of SMT-LIB [118]. SMTCoq is an open source tool that allows the user to send a goal to external SAT/SMT solvers, which when proven correct by these solvers, are returned back to Coq as a *proof witness* or a *certificate* and its soundness is verified correct in Coq in a fully automated way. The trusted base consists only of the Coq itself; if something goes wrong like Coq fails to validate the certificate or the SMT solver fails to prove the goal, the tactic will fail. Thus, no unsoundness is introduced in the system. Rümmer and Wahl [118] proposed a theory for floating-point arithmetic in the SMT-LIB 2.0 standard, which defines semantics for floating-point arithmetic

following the IEEE-754 standard [75]. This formalization defines a floating-point format of the form (_ FP <ebits> <sbits>), where <ebits> represents the number of bits in the exponent and <sbits> represents the number of bits in the significand; supports all the five rounding modes standardized by IEEE-754; and also treats exceptional floating-point values : $+\infty, -\infty$ and NaN, formally. This formalization also defines a semantics for the *fused-multiply add* and square root operations, and largely alleviates the issue of lack of reliable and scalable floating-point decision procedures for automated verification. *Thus, we believe that a great degree of automation can be introduced in reasoning about floating-point verification, using a healthy integration of the SMT-LIB theory for floating-point arithmetic and the SMTCoq checker. This will help us prove soundness of the error analysis for numerical methods, which we currently prove interactively using the Flocq formalization of floating-point arithmetic, with a greater degree of automation.*

Following up on the previous discussion on SMTCoq, one could automate the accuracy or convergence analysis for numerical methods to some extent using the approach of *certification*. This approach is a bit different from the full formal verification of a numerical algorithm that we discuss in this thesis, in that instead of verifying the algorithm itself, we would like to certify that the result from this numerical algorithm is accurate within a given bound. This error bound is provided by an external analysis tool, an example of that being the tool *Sollya* [35]. This error bound, along with a functional representation of the algorithm would form a *certificate*, which can be proven sound in an interactive theorem prover like Coq. This approach is inspired by a work that we did in collaboration with Dr. Heiko Becker, Prof. Eva Darulova and Prof. Anastasia Volkova, on the development of a tool called *Dandelion* [13] for certified polynomial approximation of transcendental functions, and other tools like *Flover* [14] for certifying soundness of error bounds for floating-point arithmetic from static analysis tools like *Daisy* [44]. Even though the trusted base includes the semantics of these external tools in addition to the basic kernel of a theorem prover, this approach provides a neat generalization to a class of methods that rely on a similar error analysis. The basic error analysis and formalization of the property of interest, along with a proof of soundness of the certificate checker is formalized in a theorem prover only once, and can be used with any off-the shelf external tools that implements the numerical algorithm of interest. In this way, we can introduce *automation* and *generalization* for formal verification of numerical algorithms.

APPENDIX A

Derivation for the forward error bound for the Jacobi iterate

Here, I present a derivation of the bound on the forward error for the Jacobi iteration f_{k+1} after $k+1$ steps, which is defined as

$$f_{k+1} = ||x_{k+1} - x|| = ||(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (D^{-1}(b - Nx))||$$

in Section 4.4.3. Using the norm-wise error relations in Section 4.4 to expand the error definition f_{k+1}, we get:

$$\begin{aligned}
f_{k+1} &= ||(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (D^{-1}(b - Nx))|| \\
&\leq ||(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (D^{-1}(b - Nx_k))|| + ||(D^{-1}(b - Nx_k)) - (D^{-1}(b - Nx))|| \\
&= ||(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (D^{-1}(b - Nx_k))|| + ||D^{-1}||N||||x_k - x|| \\
&\leq ||(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (\tilde{D}^{-1}(b - Nx_k))|| + ||(\tilde{D}^{-1}(b - Nx_k)) - D^{-1}(b - Nx_k)||+ \\
&\quad ||D^{-1}||\ ||N||f_k \\
&\leq ||(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (\tilde{D}^{-1}(b - Nx_k))|| + ||\tilde{D}^{-1} - D^{-1}||\ ||b - Nx_k||+ \\
&\quad ||D^{-1}||\ ||N||f_k \\
&\leq ||(\tilde{D}^{-1} \otimes (b \ominus (N \otimes x_k))) - (\tilde{D}^{-1}(b \ominus (N \otimes x_k)))||+ \\
&\quad ||(\tilde{D}^{-1}(b \ominus (N \otimes x_k))) - (\tilde{D}^{-1}(b - Nx_k))||+ \\
&\quad (||D^{-1}||\delta + \epsilon)\ (||b|| + ||N||\ ||x_k||) + ||D^{-1}||\ ||N||f_k
\end{aligned}$$

$$
\begin{aligned}
&\leq ((||\tilde{D}^{-1}||\ ||(b \ominus (N \otimes x_k))||)g_\delta + g_\epsilon) + ||\tilde{D}^{-1}||||(b \ominus (N \otimes x_k)) - (b - Nx_k)||+ \\
&\quad (||D^{-1}||\delta + \epsilon)\ (||b|| + ||N||\ ||x_k||) + ||D^{-1}||\ ||N||f_k \\
&\leq ((||\tilde{D}^{-1}||\ (||b|| + ||N||\ ||x_k||(1 + g_\delta + g_\epsilon))(1 + \delta)g_\delta + g_\epsilon)+ \\
&\quad ||\tilde{D}^{-1}||\ ||(b \ominus (N \otimes x_k)) - (b - (N \otimes x_k))|| + ||\tilde{D}^{-1}||||(b - (N \otimes x_k)) - (b - (Nx_k))|| \\
&\quad + (||D^{-1}||\delta + \epsilon)\ (||b|| + ||N||\ ||x_k||) + ||D^{-1}||\ ||N||f_k \\
&\leq ((||\tilde{D}^{-1}||\ (||b|| + ||N||\ ||x_k||(1 + g_\delta + g_\epsilon))(1 + \delta)g_\delta + g_\epsilon)+ \\
&\quad ||\tilde{D}^{-1}||\ (||b|| + ||N \otimes x_k||)\delta + ||\tilde{D}^{-1}||\ ||(N \otimes x_k) - Nx_k||+ \\
&\quad + (||D^{-1}||\delta + \epsilon)\ (||b|| + ||N||\ ||x_k||) + ||D^{-1}||\ ||N||f_k \\
&\leq ((||\tilde{D}^{-1}||\ (||b|| + ||N||\ ||x_k||(1 + g_\delta + g_\epsilon))(1 + \delta)g_\delta + g_\epsilon)+ \\
&\quad ||\tilde{D}^{-1}||\ (||b|| + ||N||\ ||x_k||(1 + g_\delta) + g_\epsilon)\delta + ||\tilde{D}^{-1}||\ (||N||\ ||x_k||g_\delta + g_\epsilon)+ \\
&\quad + (||D^{-1}||\delta + \epsilon)\ (||b|| + ||N||\ ||x_k||) + ||D^{-1}||\ ||N||f_k
\end{aligned}
$$

APPENDIX B

Numerical experiments for error bounds on dot-product

The dot-product of two vectors is a building block for matrix-vector multiplication, since each entry of a matrix-vector multiplication is computed from the dot-product of a row from the matrix and the vector. Thus, the accuracy of a matrix-vector multiplication depends on the accuracy of a dot-product operation. We therefore performed numerical experiments comparing the *theoretical error bound* and the *actual error bound* between a naive floating-point dot-product of two vectors, which is denoted as $\bigoplus_{0\leq i<n} u_i \otimes v_i$, and the real dot-product, which is denoted as $\sum_{0\leq i<n} u_i v_i$. The theoretical error bound comprises of floating-point errors coming from both the *addition* and *product* of two floating-point numbers. For the naive-dot product, we obtain the theoretical bound as

$$\left| \bigoplus_{0\leq i<n} u_i \otimes v_i - \sum_{0\leq i<n} u_i v_i \right| \leq ((1+\delta)^n - 1) \sum_{0\leq i<n} |u_i v_i| + n\epsilon(1+\delta)^{n-1} + \frac{\epsilon}{\delta}((1+\delta)^{n-1} - 1) \quad \text{(B.1)}$$

assuming that no overflow occurs for the dot-product operation.

We used the "uniform real distribution" method to generate random floating-point numbers. This generator picks random floating-point numbers between 1.0 and 2.0 with uniform probability. The *theoretical error bound* is defined as the difference between the theoretical bounds, provided by the equation B.1, for a float dot and a long-double dot. Here, we used the long-double data-type in C++ to store the *real dot-product*, and the float data-type to store the *floating-point dot-product*. The *actual error bound* is defined as the difference between a float dot and a long-double dot that the machine computes. We generate 1000 random samples for the dot-product, and plot the error distribution for actual error bound and the theoretical error bound in Figure B.1. The Listing B provides the C++ code for this numerical experiment.

```
#include<bits/stdc++.h>
```

```
#include<vector>
#include<fstream>
#include<iostream>
#include <random>
using namespace std;

int main(){

    int n = 10;
    long double d = 1e-7;
    long double e = 1e-45;
    int m = 1000;
    float random, random1;
    default_random_engine gen;
    uniform_real_distribution <float> distribution(1.0, 2.0);
    float dot_sum;
    long double dot_sumr;

    vector<float> vec_sum1;
    vector<long double> vec_sum2;

    // generate two random vectors and do independent dots
    for(int i = 1; i <= m ; i++){
     vector<float > vec1, vec2;
     vector<long double> vec1r, vec2r;

        // first vector
        for(int j = 0; j < n; j++){
            random = distribution(gen);
            vec1.push_back(random);
            vec1r.push_back(random);
        }
        // second vector
        for(int j = 0; j < n; j++){
            random1 = distribution(gen);
            vec2.push_back(random1);
            vec2r.push_back(random1);
        }
```

```
        //cout << vec1[0] << '\t' << vec2[0] << endl;
        // dot these two vectors
        dot_sum = 0.0;
        dot_sumr = 0.0;
        for (int j = 0; j < n; j++){
            dot_sum = dot_sum + vec1[j] * vec2[j];
            dot_sumr = dot_sumr + vec1r[j] * vec2r[j];
        }

        vec_sum1.push_back(dot_sum);
        vec_sum2.push_back(dot_sumr);
    }

    // compute differences
    vector<long double> diff;
    vector<long double> dot_comp;
    for (int i = 0; i < m; i++){
        diff.push_back(fabs(vec_sum1[i] - vec_sum2[i]));
        dot_comp.push_back( vec_sum2[i]* (pow((1. +d), n) - 1) + n * e * pow(1+d, n-1) +
(e / d * (pow (1. +d, n-1) - 1)));
    }

    ofstream myfile;
    myfile.open("exp5.txt");
    for (int i =0; i< m ;i++){
        myfile << i+1 << '\t' << diff[i] << '\t' << dot_comp[i] << endl;
    }
    myfile.close();

    return 0;
}
```

From Figure B.1, we observe that the theoretical bound for the dot-product error has an even Gaussian distribution, whereas the actual dot-product error has a skewed distribution. We also observe that the dot-product error increases as we increase the dimension of the vector. This is due to increased accumulation of errors due to increase in floating-point addition and product operations. In Figure B.2, we plot the mean and standard deviation of

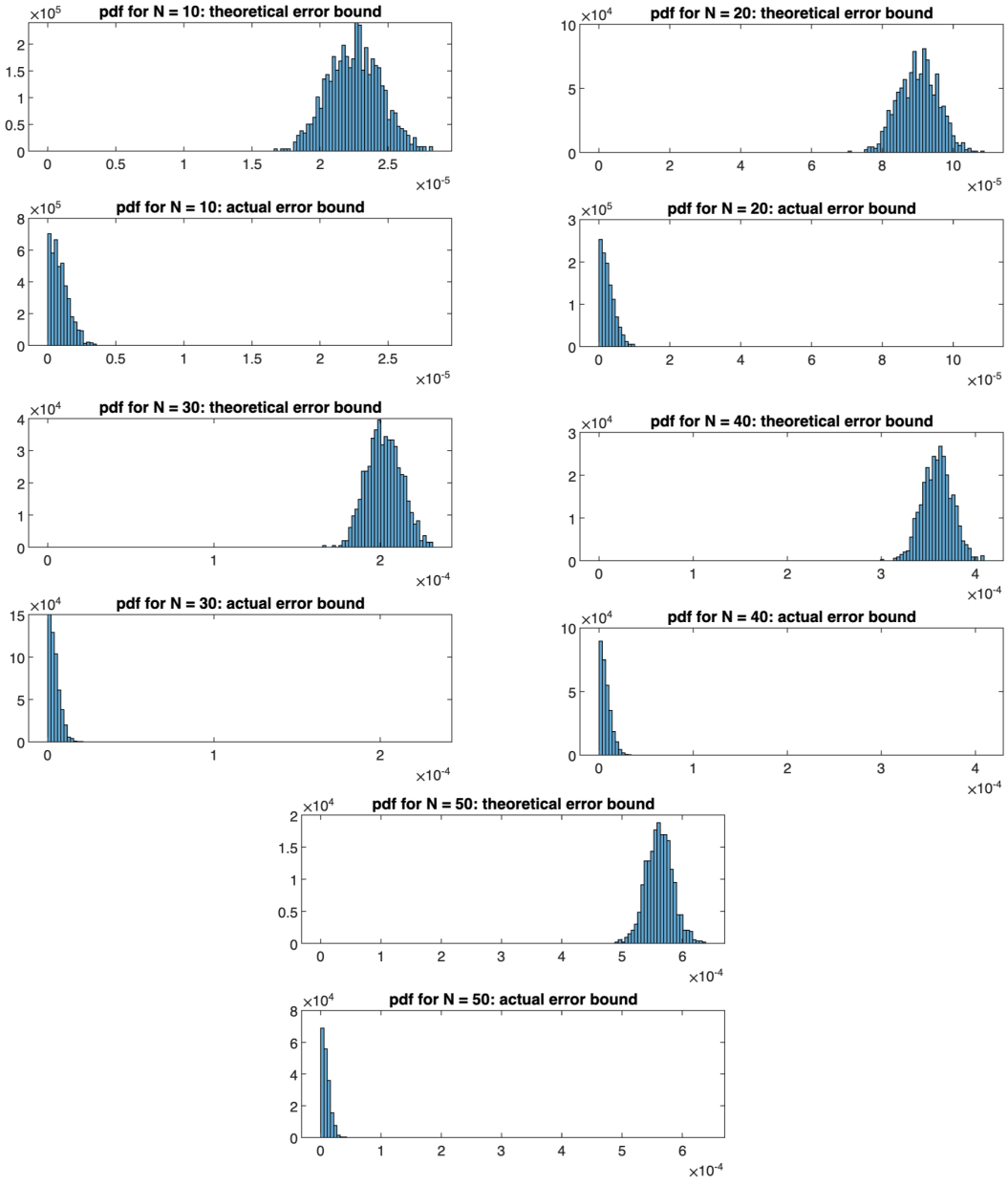

Figure B.1: Probability distribution for the error with respect to the dimension of the matrix.

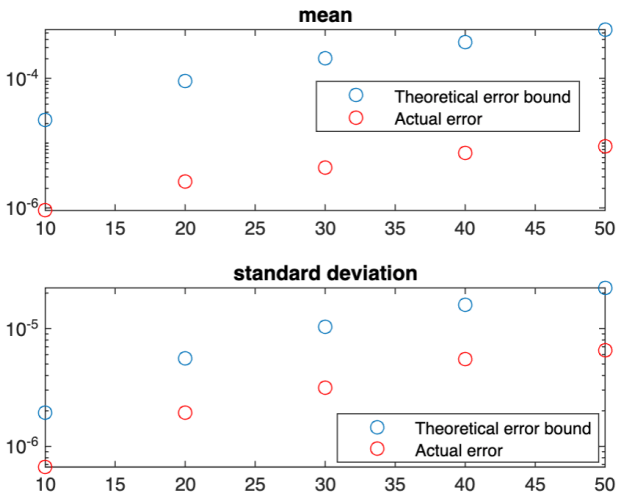

Figure B.2: Mean and standard deviation of the error distribution. The plot compares the error (on Y-axis) with the matrix dimension (on X-axis).

the error distribution with respect to the matrix dimension. We can observe that the mean of the theoretical error bound is roughly *two orders of magnitude* greater than the actual error.

One of the reasons for higher gaps between these two errors is that, our analysis is conservative. We perform the worst case error analysis, while the actual errors might behave differently. For example, there might be cancellation of terms, in which case the actual relative error might be very small (assuming the absence of *catastrophic cancellation*), instead of adding up the relative errors, which is usually done in most traditional error analysis. One of the ways to obtain tighter error bounds is to treat round-off errors as a random variables and make probabilistic assumptions about the behavior of rounding errors. Higham and Mary [70] argue that the probabilistic error bounds are smaller than the traditional error bounds by a factor of $\sqrt{n}$, for a problem size n. Another approach to obtain tighter error bounds is to use better summation algorithms like the *Kahan–Babuška summation* [83], where the relative error (proportional to $2\epsilon + \mathcal{O}(n\epsilon^2)$) is effectively independent of n. Even though, in principle, the relative error grows linearly with n, the term $n\epsilon^2$ is practically zero, unless n is roughly $1/\epsilon$ or larger.

BIBLIOGRAPHY

[1] Reynald Affeldt, Yves Bertot, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, and Laurent Théry. Mathematical components compliant analysis library. https://github.com/math-comp/analysis, 2017. (Accessed on 05/07/2023).

[2] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: a case study in functional analysis. In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II 10*, pages 3–20. Springer, 2020.

[3] Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *Journal of Formalized Reasoning*, 2018.

[4] J. H. Ahlberg and E. N. Nilson. Convergence properties of the spline fit. *J. Soc. Indust. Appl. Math.*, 11:95–104, 1963.

[5] Behzad Akbarpour and Lawrence Charles Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.

[6] Andrew W. Appel and Yves Bertot. C-language floating-point proofs layered with VST and Flocq. *Journal of Formalized Reasoning*, 13(1):1–16, December 2020.

[7] Andrew W. Appel and Ariel E. Kellison. VCFloat2: Floating-point error analysis in Coq. 2022.

[8] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

[9] G. K. Batchelor. *An introduction to fluid dynamics*. Cambridge mathematical library. Cambridge University Press, Cambridge, U.K., New York, NY, 2nd pbk. ed. edition, 1999.

[10] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi-c specification language. *CEA-LIST, Saclay, France, Tech. Rep. v1*, 2, 2008.

[11] Heiko Becker, Eva Darulova, Magnus O. Myreen, and Zachary Tatlock. Icing: Supporting fast-math style optimizations in a verified compiler. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 155–173, Cham, 2019. Springer International Publishing.

[12] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. Verified Compilation and Optimization of Floating-Point Programs in CakeML. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[13] Heiko Becker, Mohit Tekriwal, Eva Darulova, Anastasia Volkova, and Jean-Baptiste Jeannin. Dandelion: Certified Approximations of Elementary Functions. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[14] Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O. Myreen, and Anthony Fox. A verified certificate checker for finite-precision error bounds in coq and hol4. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.

[15] Mordechai Ben-Ari. The bug that destroyed a rocket. *ACM SIGCSE Bulletin*, 33(2):58–59, 2001.

[16] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21*, pages 86–101. Springer, 2008.

[17] S. Boldo, J.-H Jourdan, X. Leroy, and G. Melquiond. A formally-verified c compiler supporting floating-point arithmetic, 2014.

[18] Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A coq formalization of lebesgue integration of nonnegative functions. *J. Autom. Reason.*, 66(2):175–213, may 2022.

[19] Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq formal proof of the Lax-Milgram theorem. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 79–89. ACM, 2017.

[20] Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. https://www.lri.fr/~sboldo/elfic/index.html. https://www.lri.fr/~sboldo/elfic/index.html, 2017. (Accessed on 04/30/2023).

[21] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal proof of a wave equation resolution scheme: the method error. In *International Conference on Interactive Theorem Proving*, pages 147–162. Springer, 2010.

[22] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: a comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4):423–456, 2013.

[23] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Trusting computations: a mechanized proof from partial differential equations to actual program. *Computers & Mathematics with Applications*, 68(3):325–352, 2014.

[24] Sylvie Boldo, François Clément, Vincent Martin, Micaela Mayero, and Houda Mouhcine. A Coq Formalization of Lebesgue Induction Principle and Tonelli's Theorem. In *25th International Symposium on Formal Methods (FM 2023)*, volume 14000 of *Lecture Notes in Computer Science*, pages 39–55, Lübeck, Germany, March 2023.

[25] Sylvie Boldo, Florian Faissole, and Alexandre Chapoutot. Round-off error analysis of explicit one-step numerical integration methods. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 82–89, 2017.

[26] Sylvie Boldo, Florian Faissole, and Alexandre Chapoutot. Round-off error and exceptional behavior analysis of explicit runge-kutta methods. *IEEE Transactions on Computers*, 69(12):1745–1756, 2020.

[27] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified c compiler supporting floating-point arithmetic. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 107–115. IEEE, 2013.

[28] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[29] Sylvie Boldo and Claude Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5:377–393, 2011.

[30] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252. IEEE, 2011.

[31] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. Elsevier, 2017.

[32] Guillaume Cano and Maxime Dénès. Matrices à blocs et en forme canonique. In Damien Pous and Christine Tasson, editors, *JFLA - Journées francophones des langages applicatifs*, Aussois, France, February 2013.

[33] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reason.*, 61(1-4):367–422, June 2018.

[34] Pierre Castéran and Yves Bertot. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions., 2004.

[35] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. Sollya: An environment for the development of numerical codes. In *Mathematical Software–ICMS 2010: Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings 3*, pages 28–31. Springer, 2010.

[36] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

[37] Cyril Cohen. Construction of real algebraic numbers in Coq. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, Princeton, United States, August 2012. Springer.

[38] Cyril Cohen and Assia Mahboubi. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8(1:02):1–40, February 2012.

[39] Cyril Cohen and Damien Rouhling. A formal proof in Coq of LaSalle's invariance principle. In *International Conference on Interactive Theorem Proving*, pages 148–163. Springer, 2017.

[40] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

[41] Sylvain Conchon, Mohamed Iguernlala, Kailiang Ji, Guillaume Melquiond, and Clément Fumex. A three-tier strategy for reasoning about floating-point numbers in smt. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 419–435, Cham, 2017. Springer International Publishing.

[42] Andrew S Crouch. When the millennium bug bites: Business liability in the wake of the y2k problem. *Hamline L. Rev.*, 22:797, 1998.

[43] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Software Engineering and Formal Methods: 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings 10*, pages 233–247. Springer, 2012.

[44] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. Daisy - framework for analysis and optimization of numerical programs (tool paper). In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms*

for the Construction and Analysis of Systems, pages 270–287, Cham, 2018. Springer International Publishing.

[45] BN Datta and Y Saad. Solution of large linear algebra problems in control. In *Presentation at the 1986 SIAM meeting on Linear Algebra and its Applications, Boston Mass*, 1991.

[46] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software (TOMS)*, 37(1):1–20, 2010.

[47] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library for floating-point numbers and its application to exact computing. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, pages 169–184, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[48] Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, page 1318–1322, New York, NY, USA, 2006. Association for Computing Machinery.

[49] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[50] Theodorus Jozef Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[51] David Delahaye. A proof dedicated meta-language. *Electronic Notes in Theoretical Computer Science*, 70(2):96–109, 2002.

[52] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. Smtcoq: A plug-in for integrating smt solvers into coq. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*, pages 126–133. Springer, 2017.

[53] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13–es, jun 2007.

[54] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *International Conference on Theorem Proving in Higher Order Logics*, pages 327–342. Springer, 2009.

[55] David F. Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.

[56] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[57] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings 4*, pages 163–179. Springer, 2013.

[58] Michael JC Gordon and Tom F Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[59] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings 12*, pages 232–247. Springer, 2011.

[60] Michael Grottke and Kishor S Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2):107–109, 2007.

[61] S.J. Hammarling and J. H. Wilkinson. The practical behaviour of linear iterativemethods with particular reference to s.o.r. *Report NAC 69, National Physical Laboratory, Teddington, UK*, page 19 pp, 1976.

[62] John Harrison. Floating point verification in hol light: The exponential function. In Michael Johnson, editor, *Algebraic Methodology and Software Technology*, pages 246–260, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[63] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 113–130, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[64] John Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 254–270, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[65] John Harrison. Formal verification of ia-64 division algorithms, 2000.

[66] John Harrison. Formal verification of square root algorithms, 2003.

[67] João P Hespanha. Linear systems theory, 2017.

[68] Nicholas J Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.

[69] Nicholas J Higham and Philip A Knight. Componentwise error analysis for stationary iterative methods. In *Linear Algebra, Markov Chains, and Queueing Models*, pages 29–46. Springer, 1993.

[70] Nicholas J Higham and Theo Mary. A new approach to probabilistic rounding error analysis. *SIAM journal on scientific computing*, 41(5):A2815–A2835, 2019.

[71] Alan C Hindmarsh. Odepack, a systematized collection of ode solvers. *Scientific computing*, pages 55–64, 1983.

[72] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[73] GY Hu and Robert F O'Connell. Analytical inversion of symmetric tridiagonal matrices. *Journal of Physics A: Mathematical and General*, 29(7):1511, 1996.

[74] Thomas Huckle and Tobias Neckel. *Bits and Bugs: A Scientific and Historical Review on Software Failures in Computational Science*. SIAM, 2019.

[75] IEEE-SA. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[76] Fabian Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, pages 113–127, Cham, 2014. Springer International Publishing.

[77] Fabian Immler. *A Verified ODE Solver and Smale's 14th Problem*. Dissertation, Technische Universität München, München, 2018.

[78] Fabian Immler and Johannes Hölzl. Numerical analysis of ordinary differential equations in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 377–392. Springer, 2012.

[79] Fabian Immler and Yong Kiam Tan. The poincaré-bendixson theorem in isabelle/hol. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 338–352, New York, NY, USA, 2020. Association for Computing Machinery.

[80] Fabian Immler and Christoph Traut. The flow of ODEs. In *International Conference on Interactive Theorem Proving*, pages 184–199. Springer, 2016.

[81] Fabian Immler and Christoph Traut. The flow of ODEs: Formalization of variational equation and Poincaré map. *Journal of Automated Reasoning*, 62(2):215–236, 2019.

[82] Claude-Pierre Jeannerod and Siegfried M Rump. On relative errors of floating-point operations: optimal bounds and applications. *Mathematics of Computation*, 87(310):803–819, 2018.

[83] W Kahan. Further remarks on reducing truncation errors, commun. *Assoc. Comput. Mach*, 8:40, 1965.

[84] William Kahan. On the cost of floating-point computation without extra-precise arithmetic. *World-Wide Web document*, 21, 2004.

[85] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[86] Ariel Kellison, Mohit Tekriwal, Jean-Baptiste Jeannin, and Geoffrey Hulette. Towards verified rounding error analysis for stationary iterative methods. In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 10–17, 2022.

[87] Ariel E. Kellison and Andrew W. Appel. Verified numerical methods for ordinary differential equations. In *15th Int'l Workshop on Numerical Software Verification*, 2022.

[88] Ariel E Kellison and Andrew W Appel. Verified numerical methods for ordinary differential equations. In *Software Verification and Formal Methods for ML-Enabled Autonomous Systems: 5th International Workshop, FoMLAS 2022, and 15th International Workshop, NSV 2022, Haifa, Israel, July 31-August 1, and August 11, 2022, Proceedings*, pages 147–163. Springer, 2022.

[89] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. LAProof: a library of formal accuracy and correctness proofs for sparse linear algebra programs. in preparation, 2023.

[90] David B. Kirk and Wen-Mei W. Hwu. Chapter 10 - parallel patterns: Sparse matrix–vector multiplication: An introduction to compaction and regularization in parallel algorithms. In David B. Kirk and Wen mei W. Hwu, editors, *Programming Massively Parallel Processors (Second Edition)*, pages 217 – 234. Morgan Kaufmann, Boston, second edition edition, 2013.

[91] Michael P Knapp. Sines and cosines of angles in arithmetic progression. *Mathematics magazine*, 82(5):371, 2009.

[92] Erwin Kreyszig. *Introductory functional analysis with applications*, volume 1. wiley New York, 1978.

[93] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

[94] Peter D Lax and Robert D Richtmyer. Survey of the stability of linear finite difference equations. *Communications on pure and applied mathematics*, 9(2):267–293, 1956.

[95] Vincent Lefevre, Jean-Michel Muller, and Arnaud Tisserand. *The Table Maker's Dilemma.* PhD thesis, Laboratoire de l'informatique du parallélisme, 1998.

[96] Catherine Lelay. How to express convergence for analysis in coq. 2015.

[97] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[98] Heather A Lewis. Math mistakes that make the news. *Primus*, 25(2):181–192, 2015.

[99] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. *Journal of Automated Reasoning*, 62:281–300, 2019.

[100] Assia Mahboubi and Enrico Tassi. Mathematical components, 2017.

[101] Evgeny Makarov and Bas Spitters. The picard algorithm for ordinary differential equations in coq. In *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings 4*, pages 463–468. Springer, 2013.

[102] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions in coq. *Journal of Automated Reasoning*, 57, 10 2016.

[103] Érik Martin-Dorel, Laurence Rideau, Laurent Théry, Micaela Mayero, and Ioana Pasca. Certified, efficient and sharp univariate Taylor models in Coq. In *15th Int'l Symp. on Symbolic and Numeric Algorithms for Scientific Computing*, pages 193–200. IEEE, 2013.

[104] L. McKenzie. Matrices with dominant diagonals and economic theory. In K. Arroa, S. Karlin, and S. Puppes, editors, *Mathematical Methods in the Social Sciences*, pages 47–60. Stanford University Press, 1960.

[105] Mariano Moscato, Laura Titolo, Aaron Dutle, and César A Munoz. Automatic estimation of verified floating-point round-off errors via static analysis. In *Computer Safety, Reliability, and Security: 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15, 2017, Proceedings 36*, pages 213–229. Springer, 2017.

[106] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2018.

[107] Peter Naur and Brian Randell. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[108] William L Oberkampf and Christopher J Roy. *Verification and validation in scientific computing*. Cambridge University Press, 2010.

[109] Russell O'Connor. Certified exact transcendental real number computation in Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 246–261. Springer, 2008.

[110] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, pages 748–752, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[111] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices*, 50(6):1–11, 2015.

[112] Ioana Pasca. *Formal Verification for Numerical Methods*. PhD thesis, Université Nice Sophia Antipolis, 2010.

[113] Pierre-Marie Pédrot. Ltac2: tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, pages 13–19, 2019.

[114] Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. A unified coq framework for verifying c programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 15–26, 2016.

[115] Edgar Reich. On the convergence of the classical iterative method of solving linear simultaneous equations. *The Annals of Mathematical Statistics*, 20(3):448–451, 1949.

[116] William T. Reid. *Riccati differential equations*. Mathematics in science and engineering, v. 86. Academic Press, New York, 1972.

[117] Pierre Roux. Formal proofs of rounding error bounds. *Journal of Automated Reasoning*, 57(2):135–156, 2016.

[118] Philipp Rümmer and Thomas Wahl. An smt-lib theory of binary floating-point arithmetic. In *International Workshop on Satisfiability Modulo Theories (SMT)*, page 151, 2010.

[119] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, Pa., 2nd ed. edition, 2003.

[120] Youcef Saad. Overview of krylov subspace methods with applications to control problems. Technical report, 1989.

[121] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[122] Louise Ben Salem-Knapp, Sylvie Boldo, and William Weens. Bounding the round-off error of the upwind scheme for advection. *IEEE Transactions on Emerging Topics in Computing*, 10(3):1253–1262, 2022.

[123] JM Sanz-Serna and C Palencia. A general equivalence theorem in the theory of discretization methods. *Mathematics of computation*, 45(171):143–152, 1985.

[124] John W. Slater. Uncertainty and error in cfd simulations. https://www.grc.nasa.gov/WWW/wind/valid/tutorial/errors.html, 2021.

[125] Houshang H. Sohrab. *Basic real analysis*. Birkhäuser, Boston, 2003.

[126] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Trans. Program. Lang. Syst.*, 41(1), dec 2018.

[127] Pat H Sterbenz. *Floating-point computation*. Prentice Hall, 1973.

[128] Ping-Tak Peter Tang. Table-driven implementation of the exponential function in ieee floating-point arithmetic. *ACM Transactions on Mathematical Software (TOMS)*, 15(2):144–157, 1989.

[129] Mohit Tekriwal, Andrew W. Appel, Ariel E. Kellison, David Bindel, and Jean-Baptiste Jeannin. Verified correctness, accuracy, and convergence of a stationary iterative linear solver: Jacobi method. https://mohittkr.github.io/cicm_paper.pdf, 2023. 16th Conference on Intelligent Computer Mathematics (to appear).

[130] Mohit Tekriwal, Karthik Duraisamy, and Jean-Baptiste Jeannin. A formal proof of the Lax equivalence theorem for finite difference schemes. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NASA Formal Methods*, pages 322–339, Cham, 2021. Springer International Publishing.

[131] Mohit Tekriwal, Joshua Miller, and Jean-Baptiste Jeannin. Formal verification of iterative convergence of numerical algorithms, 2022.

[132] Mohit Tekriwal, Avi Tachna-Fram, Jean-Baptiste Jeannin, Manos Kapritsos, and Dimitra Panagou. Formally verified asymptotic consensus in robust networks, 2022.

[133] René Thiemann. A Perron-Frobenius theorem for deciding matrix growth. *Journal of Logical and Algebraic Methods in Programming*, page 100699, 2021.

[134] Laura Titolo, Marco A. Feliú, Mariano Moscato, and César A. Muñoz. An abstract interpretation framework for the round-off error analysis of floating-point programs. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 516–537, Cham, 2018. Springer International Publishing.

[135] Laura Titolo, César A. Muñoz, Marco A. Feliú, and Mariano M. Moscato. Eliminating unstable tests in floating-point programs. In Fred Mesnard and Peter J. Stuckey, editors, *Logic-Based Program Synthesis and Transformation*, pages 169–183, Cham, 2019. Springer International Publishing.

[136] Gerhard W Veltkamp. Algol procedures voor het berekenen van een inwendig product in dubbele precisie. *RC-Informatie, Technishe Hogeschool Eindhoven, Tech. Rep*, 22, 1968.

[137] Paul Wilmott, Sam Howison, and Jeff Dewynne. *The Mathematics of Financial Derivatives: A Student Introduction*. Cambridge University Press, Cambridge, 1995.