Applications of Lightweight Formal Methods in Engineering and Scientific Software Design

Tristan Dyer

May 31, 2019

1 Introduction

Scientists increasingly rely on computational models to explore and understand the world around us, with diverse applications throughout the physical, chemical, and biological sciences. In 2005, The President's Information Technology Advisory Committee (PITAC) issued a report referring to computation as a "third pillar" of science, placing it alongside theory and experimentation. The report underscores qualitative transformations and discoveries throughout the natural sciences, as well as in engineering, manufacturing, and economic processes, which often rely on scientific models to predict the effects of decisions. Coastal engineers, as just one example, evaluate alternative flood protection measures by subjecting them to large-scale, simulated storm events. The performance and fidelity of scientific models are therefore key determinants affecting the quality of those designs.

Despite its importance, the field of scientific computation faces a number of problems. Meeting quality and reproducibility standards is a growing concern [Wilson, 2006], as is productivity [Faulk et al., 2009]. Not merely anecdotes, numerous empirical studies of software "thwarting attempts at repetition or reproduction of scientific results" have been cataloged in a recent article by Storer [2017]. In one [Hatton and Roberts, 1994] and in a more recent follow-up [Hatton, 2007], over a dozen independently developed commercial codes for seismic data processing were compared, showing a rate of numerical disagreement between them of about 1% per 4,000 lines of implemented code, and that "even worse, the nature of the disagreement is nonrandom." In addition to cataloging the quality concerns reported in those studies, Storer [2017] further cites their effects, including a widespread inability to reproduce results and subsequent retractions of papers in scientific journals. Productivity problems are also reported, which Faulk et al. [2009] refer to as a *productivity crisis* because of "frustratingly long and troubled software development times" and difficulty achieving portability requirements and other goals.

Sources of difficulty may be the result of fundamental characteristics of the problem domain, as well as cultural and development practices within it. To examine these and related issues, researchers from the scientific computation and software engineering communities came together for a workshop whose outcomes are summarized by Carver [2009]. In an observation about development practices, Carver notes that programmers in scientific areas are often domain experts with a Ph.D. in their respective fields, and that a single scientist

may take the lead on a project and then rely on self-education to pick up whatever software development skills are needed. Such practices are long-standing, and represent a disconnect from the software engineering community that has been variously referred to as a "chasm" [Kelly, 2007; Sanders and Kelly, 2008] and a "communication gap" [Faulk et al., 2009].

Other observations, however, point to the unique challenges facing the developers of scientific software. For instance, projects are often undertaken, as one might imagine, for the purpose of advancing scientific goals, so results may represent novel findings that are difficult to validate. In the absence of test oracles, developers may have to settle for plausibility checks based on, say, conservation laws or other principles that are expected to hold [Storer, 2017]. Then, if the software is successful, its lifetime may span a 20 or 30 year period, starting with development and then moving through hardware upgrades and evolving requirements that are intended to keep up with ongoing scientific advancements. Development priorities are such that traditional software engineering concerns, like time to market and producing highly maintainable code, may receive relatively less attention compared to performance and hardware utilization [Faulk et al., 2009].

The overall goal of the proposed research is to address fundamental design and quality assurance challenges that are intrinsic to scientific computation and related types of numerical software. Addressing them may also lead to accompanying gains in productivity, particularly if better designs improve maintainability and extensibility, given the lifespans typical of scientific software products.

In the following sections I propose a state-based approach for reasoning about scientific software, using abstraction and refinement principles to separate numerical concerns from other sources of complexity, such as those introduced to meet performance goals. Elements of the approach include declarative modeling and automatic, push-button analysis using bounded model checking, as embodied in lightweight tools like the Alloy Analyzer [Jackson, 2012] which, while increasingly promoted in areas like communication protocols and control systems, may also be exploited in scientific domains.

The remainder of the proposal is outlined as follows: ...

2 Literature Review

3 Methodology

The objective of this research is to develop modeling approaches using formal methods that allow scientists to represent and reason about the essential structure and behavior of the programs they create. By *structure* I mean static relationships between elements of program state, that is, an assignment of values to variables. By *behavior* I mean dynamic relationships, that is, a sequence of states or steps in a computation, which may include interleaving. Structure and behavior may each find expression in different ways in different programming languages, and so demonstrating the relationship to code is important and also within the scope of the research.

Below I provide background on lightweight formal methods and some of the characteristics of scientific programs that inform the approach, which is based on a separation of concerns that accounts for numerical computations. I also describe what is meant by soft-

ware abstractions, and provide a simple example that illustrates the rationale for viewing scientific programs as I do.

3.1 Lightweight Formal Methods

The fundamental idea is to characterize structure, behavior, and properties of interest in a way that allows one to frame questions and get answers to them using tools like Alloy Analyzer [Jackson, 2012] and TLA+/TLC [Lamport, 2002], which support automatic, push-button analysis. What I propose, then, is a *lightweight* approach to formal methods [Jackson and Wing, 1996], in which there is both *partiality in modeling*, since we do not commit to modeling entire systems, and *partiality in analysis*, since the analyses being performed are bounded. An additional aspect of lightweight formal methods is an incremental style of modeling, which tools like Alloy support by offering immediate feedback while models are being constructed: we start with a minimal set of constraints and "grow" them via conjunction.

From their declarative underpinnings comes expressive power, along with an effective means of reducing complexity and probing designs. As Jackson [2012] writes, code is a poor medium for exploring abstractions, and lightweight tools offer modeling environments that support an iterative design scenario akin to what might be performed by, say, a civil engineer designing a bridge. As with an algebraic approach to specification, as has been found [Baugh, 1992], not all aspects of a software system warrant the same level of attention from a modeling perspective.

With respect to a bounded or partial analysis, the merit of performing one springs from an appeal to the *small scope hypothesis:* if an assertion is invalid, it probably has a small counterexample [Jackson and Damon, 1996], as has been observed in a prior case study [Baugh and Altuntas, 2017]. In addition, in some cases, *domain-specific* arguments can be used to bolster the conclusions of such analyses, e.g., by calling on the so-called Courant-Friedrichs-Lewy (CFL) condition when solving hyperbolic partial differential equations [Baugh and Altuntas, 2017; Altuntas and Baugh, 2017].

Alloy Analyzer. For background on the primary tool I expect to use, Alloy is a declarative modeling language combining first-order logic with relational calculus and associated quantifiers and operators, along with transitive closure. It offers rich data modeling features based on class-like structures and an automatic form of analysis that is performed within a bounded scope using a SAT solver. For *simulation*, the analyzer can be directed to look for instances satisfying a property of interest. For *checking*, it looks for an instance violating an assertion: a counterexample. The approach is *scope complete* in the sense that all cases are checked within user-specified bounds. Alloy's logic supports three distinct styles of expression, that of predicate calculus, navigation expressions, and relational calculus. The language used for modeling is also used for specifying properties of interest and assertions. Alloy supports expressions with integer values and basic arithmetic operations.

3.2 Scientific Programs

The proposed work is being undertaken in a relatively uncharted domain, scientific computation, for which there is little community experience in applying formal methods. We might ask about essential complexities, what they are, and where formal methods might help. In contrast, when computer engineers model systems, for instance, they already have some experience in getting at these questions. So when specifying a two-phase handshake protocol they know whether they can ignore what's going through the pipe: they generally have some sense of how and what to specify, and what to ignore. There is far less of this kind of experience with programs in scientific areas, so it is helpful to pause and characterize what they are like.

The subject matter of scientific programs includes the physical and natural processes that surround us, where space and time are traditionally viewed as being continuous. Circulation of currents within the atmosphere and oceans, for instance, involves state that is continuously varying and where, indeed, continuity arguments are used to "fill in" gaps that may be associated with sampled data. The computational apparatus underlying ocean circulation models, however, looks less like purely analytic functions and more like an amalgam of discrete and continuous constructions that allow, as one example, the representation of irregular land and seafloor geometries as piecewise polynomial surfaces. The types of discretizations that may be employed in both time and space are varied, and each has its own performance, accuracy, and ease-of-development implications.

That problems such as these are discretized implies there is a level beneath which certain physical features and processes are not resolved—a modeling concern to be managed. But more than that, discretizing time and space means that contingent processes, like the continuous movement of the shoreline due to tides and storms, require special handling to account for them in physically meaningful and often nontrivial ways. As a result, it may be helpful to separate concerns and avoid mixing the types of reasoning best suited to the respective types of processes, whether discrete or continuous.

3.3 Separating Concerns

Related studies on formal methods, though few, have targeted somewhat different aspects of scientific programs, allowing us to make observations and draw some comparisons. Most have focused on numerics—of which the works of Bientinesi et al. [2005] and Siegel et al. [2008] are representative—by combining model checking and the reals with applications to some of the direct methods of linear algebra.[1] However, the scope of systems that can be so analyzed will remain rather limited until the deep, semantic proofs of numerical analysis [Linz, 1988] can be accommodated as part of the reasoning process.

We take the position that a separation of concerns is in order, and propose something akin to the two-phase handshake protocol analogy, where the data going through the pipe are, in this case, numerical expressions. We cannot ignore them, of course, but we aim to consider them separately using the conventional tools of numerical analysis.

[1] By direct methods, we mean those that produce an exact solution, modulo rounding errors, in a finite number of steps. Iterative methods, on the other hand, successively approximate a solution, gradually improving it until convergence is reached.

Accordingly, we advance the following perspective:

$$\boxed{\begin{array}{c}\text{scientific}\\ \text{programs}\end{array}} = \boxed{\begin{array}{c}\text{numerical}\\ \text{expressions}\end{array}} + \boxed{\begin{array}{c}\text{interstitial}\\ \text{machinery}\end{array}}$$

By *interstitial machinery* we mean the data structures and algorithms throughout which numerical expressions are embedded. In many cases, the interstitial machinery is itself a complex apparatus, as we find in the case of adaptive, multiscale, multiphysics applications, for instance, and these are aspects of a program that warrant increased scrutiny and care. Correctness arguments for this part of scientific programs can be made without simultaneously reasoning about, say, elements of the Gershgorin circle theorem for eigenvalues. Instead, results of numerical analyses may be brought into the modeling process in the form of invariants and other structural properties.

3.4 Software Ecosystem

With respect to numerical analysis, the form of the results needed from the effort may vary, but this type of work may be conducted independently and used to inform the process undertaken when reasoning about the interstitial machinery so that the overall conclusions are sound. Beyond that example, we must further recognize that these artifacts do not exist in a vacuum. While the numerical analyses for scientific computations remain unchanged, their application must evolve throughout a broad range of implementation choices and modifications, changes in libraries and solvers, and diverse hardware upgrades, over the life of the program. Examples include new applications for hardware such as GPUs in general purpose computation, new software platforms such as web-based computation, evolving user interface and data interaction capabilities, and new and evolving languages that offer varying levels of convenience and performance. Approaching the development of scientific software with the knowledge that the software must exist in a modern ecosystem of emerging and evolving technologies requires a deeper understand of how the machinery of that software might also evolve, without adversely effecting the underlying numerics.

4 Contributions

I expect to make significant intellectual contributions to the design and understanding of scientific software systems. The contributions result from the application of the following concepts:

- *Software abstractions:* The distinction between an implementation and its essential structure and behavior is made too infrequently in scientific software development. By separating concerns, we avoid overcommitting to implementation languages and other details, which allows us to develop and communicate transferable design and quality assurance concepts. I plan to address these in the context of a number of case studies that span a broad range of abstractions and computational complexities. In developing and advancing a methodology of designing and working with software abstractions, my contributions have the potential to change the way a rather different class of developers thinks about programming.

- *Lightweight formal methods:* A significant component of the research is on how to do modeling, with a focus on design thinking and constructing arguments that rely on the existing tools of numerical analysis and lightweight formal methods. In taking the perspective of separating concerns in scientific software, I will be exploring the relationship between numerical analysis and tools like Alloy in a way that has not been previously attempted.

- *Analysis tools:* As the application of lightweight formal methods to scientific software development is a new area of research, domain specific tools to aid modelers simply do not exist. Additionally, tools such as the Alloy Analyzer are purposely designed with generality in mind: they make no assumptions about the ecosystems in which they exist, relying solely on the mathematics upon which they are built. Therefore, visualizations generated from tools like Alloy make no assumptions about the models they represent. Certain applications of formal methods, as in representing a planar mesh, may warrant an opinionated visualization in order to aid in the interpretation of model results. As such, I will develop tools for this purpose, advancing the ecosystem of modeling tools as well as lowering the barrier to entry into software modeling for developers of scientific software.

5 Overview of Chapters

5.1 Moment Distribution

Declarative languages like Alloy are expressive, combining the quantifiers of first-order logic and the operators of relational calculus, but it is not always clear how they can be used to specify a problem. There are no special constructs for parallelism, message-passing, synchronization or other mechanisms that give some insight into what one is "supposed" to do with it. In other words, there are few affordances [Norman, 1999], or "action possibilities" that are readily perceivable.

To make this type of modeling more accessible, we begin with what might be considered a 'toy' problem—the moment distribution method [Cross, 1930]—that despite its apparent simplicity, displays structural and behavioral features that are commonly found in a broad range of scientific software. The method is an iterative technique, well-known among civil engineers, for finding the internal member forces that develop in building structures when external forces are applied to them. The topological relationships between the structural components are similar to those found in, for example, methods requiring a spatial discretization such as finite volume and finite elements. Similarly, the behavioral components of the method exhibit communication patterns often encountered in scientific software. In its most general form, the method is similar to asynchronous, chaotic relaxation algorithms, where portions of a building structure converge numerically at differing rates as the computation unfolds, depending on the process scheduling. The nondeterminism available here is also inherent in methods used to solve elliptic partial differential equations, which may exploit nondeterminism in different ways depending on problem characteristics and hardware features.

The combination of commonly found topological and communicative components exhibited by the moment distribution method presents us with an opportunity to explore the inherent structural and behavioral complexities at numerous levels of abstraction. The models created to represent and reason about these complexities can then be used as a starting point when approaching problems that have structural and behavioral similarities. This template-based approach is described by Schrage [1991], who contrasts it with, as he calls it, a constructive approach for formulating models, where one starts with a blank sheet. About the template approach, he says

> "In this approach, examples of standard applications are illustrated in substantial detail. If you have a problem that closely resembles one of these 'template' models, you may be able to adjust it to your situation by making modest changes to the template model."

5.2 Sparse Matrices

The types of problems found in scientific and engineering software frequently require a discretization over some continuous domain, resulting in a system of equations that can be solved using linear algebraic methods. As a result, it is not uncommon to find powerful linear algebra libraries at the core of some piece of scientific software. These libraries, some of which have existed for decades, provide us with a unique perspective, one that lends itself to software modeling: while the underlying mathematics have remained constant, the structural and behavioral components of the libraries have had to adapt as evolving technologies have created potentially new performance benefits. For example, the advent of massively parallel systems, in concert with the introduction of general purpose computing on GPUs, has resulted in systems that are capable of incredible data throughput. These systems can employ numerous types of processors, each built on different memory models and each capable of leveraging multiple types of parallelism. As a result, many linear algebra libraries have adapted in order to properly leverage the performance benefits made available by these types of systems.

Often, the methods of discretization that are employed in scientific software, such as in the finite element method, will result in a system of equations that can be represented using sparse matrices—matrices in which the majority of the values they contain are zero. As such, most linear algebra libraries provide very fast and efficient computational methods for representing and using sparse matrices. From the structural perspective, there are many different ways of representing a sparse matrix in memory, each with strengths in some applications and weaknesses in others. For example, the DOK (dictionary of keys) format excels at matrix assembly, but is slower compared to other formats when being used in some algebraic operation. Conversely, the CSR (compressed sparse row) format excels when used in certain matrix operations, such as matrix-vector multiplication, but is slower to assemble than others. From a behavioral perspective, the algorithmic implementation used to perform an algebraic operation is highly dependent on the format, i.e. the memory layout, used to represent the sparse matrix.

Nonetheless, all sparse matrix formats and the corresponding algorithms used in algebraic expressions represent exactly the same thing at an abstract, mathematical level. This

presents us with a valuable opportunity to model these formats and operations, the benefits of which are threefold:

1. To provide a clear and useful example of how both data and procedural refinement can be used in scientific and engineering software development.

2. To verify that all formats and operations are accurate representations of the underlying mathematics.

3. To provide a model that can be built upon as new technologies and techniques are developed, in order to ease development and instill confidence in implementations.

In this research, I will create an abstract model of a sparse matrix as well as the sparse matrix-vector multiplication operation. I will then create models of the DOK and CSR sparse matrix formats and use abstractions functions along with implication to verify that the data refinement is sound. Additionally, for both formats I will model the sparse matrix-vector multiplication operation and similarly use abstraction functions and implications to verify that the procedural refinements are sound. Finally, I will create a concrete implementation of both formats and their corresponding operations in order to show the complete story as a guide to engineers and scientists—beginning with a mathematical problem, how might one go through the process of incrementally stepping through the modeling process, moving closer to an implementation at each step, and finally writing a concrete implementation.

5.3 Ocean Modeling

Coastal flooding from tropical storms is the result of large-scale processes whose simulation is computationally demanding. Used to predict the geographic extent and severity of storm effects, ocean circulation models are essential tools for evacuation planning, vulnerability assessment, and infrastructure design. In order to capture the resulting hydrodynamics as accurately and efficiently as possible, models employ a variety of mechanisms that, while improving performance, can also lead to complex software implementations. Based on numerical techniques such as finite element methods [Zienkiewicz et al., 2013], these models start from a characterization of wind velocities, atmospheric pressure, and land and seafloor surfaces to produce time histories of spatially varying water surface elevations and velocities. Ocean circulation models are designed to perform these large-scale simulations while incorporating both tidal effects and more extreme storm events into the analysis.

As in other areas of science, validation of the models is based on observational comparisons and, as much as anything, review and evaluation by an active community of scientists and engineers. Quantities like wind, wave, and water level are compared with data from the National Data Buoy Center (NDBC), National Ocean Service (NOS) observation stations, high water marks, and other sources [Hanson et al., 2013]. In addition to such studies, since 2008 a regional testbed has been actively comparing the accuracy of ocean circulation models and their relative abilities to hindcast the effects of historical storms [Sheng et al., 2012].

In order to reduce the computational cost of certain types of analyses, an extension called *subdomain modeling* [Baugh et al., 2015] was incorporated in ADCIRC v51.42 in early 2015 and in experimental versions years ahead of that. Subdomain modeling is an

exact reanalysis technique that allows scientists and engineers to analyze the effects of an incremental problem change at an incremental computational cost. To realize a subdomain modeling capability in ADCIRC, boundary conditions must be imposed to accommodate a local change in a problem, and doing so results in a particularly tricky interaction with the program's discrete wet-dry algorithm, which lets floodwaters advance and recede. Using Alloy [Baugh and Altuntas, 2017], relevant parts of ADCIRC were modeled in a way that enabled the exploration and optimization of possible boundary conditions, with insights then used to implement the actual extension in Fortran.

It should be emphasized that ADCIRC itself, as it is implemented, is designed without any notion of data abstraction, but it most certainly has rich state, including geometric and topological relationships, which must be modeled to enable the type of analysis described above. While software abstractions built with Alloy could be used to inform a new implementation of ADCIRC and improve its program structure, doing so is not a prerequisite for making effective use of Alloy models. Still, the work referenced here is preliminary, and additional studies and applications are warranted.

Finite Element Meshes. Fundamental to ocean modeling is the concept of discretizing what in reality are the continuous domains of space and time. A byproduct of the process is that adjustments in the level of refinement employed —both spatially and temporally— can be used to make tradeoffs between the competing objectives of fidelity and computational performance. That these kinds of adjustments can be made demonstrates why gains in computational performance are so significant: they can be used either directly, say, to provide greater lead time in an operational emergency management scenario, or indirectly, to increase fidelity without increasing computational cost.

A spatial discretization commonly found in engineering software is that of the finite element mesh. Properties of a finite element mesh, such as topology, dimensionality, and order are influenced by the physical system they represent. ADCIRC, for example, uses linear two-dimensional triangular finite elements to represent the Earth's surface. Similarly, the abstractions one might choose when developing software that uses a finite element mesh are influenced by the intended use case. In the context of engineering design, finite element meshes are part of an iterative process; one involving mesh editing, simulation, and interpretation of simulation results.

What are finite element meshes?

Identify some essential complexities.

What am I going to model, and why will that be useful?

What does an ENGINEER need? What abstractions help in developing tools for ENGINEERS? For meshes: automatic adaptivity, refinement, interactivity/editing, interpretation of results.

I'm imagining OOP principles will be most useful for refinement, editing. Arrays likely best for performance during simulation, but mix of OOP and arrays likely best for tradeoff with features needed during simulation such as adaptivity. Indexed arrays very clearly best for visualization. Coordinating multiple mesh representations...this is an iterative design process, so likely that you'll need to frequently switch between representations.

References

Altuntas, A. and Baugh, J. (2017). Adaptive subdomain modeling: A multi-analysis technique for ocean circulation models. *Ocean Modelling*, 115:86–104. https://doi.org/10.1016/j.ocemod.2017.05.009.

Baugh, J. and Altuntas, A. (2017). Formal methods and finite element analysis of hurricane storm surge: A case study in software verification. *Science of Computer Programming*. https://doi.org/10.1016/j.scico.2017.08.012.

Baugh, J., Altuntas, A., Dyer, T., and Simon, J. (2015). An exact reanalysis technique for storm surge and tides in a geographic region of interest. *Coastal Engineering*, 97:60–77. https://doi.org/10.1016/j.coastaleng.2014.12.003.

Baugh, J. W. (1992). Using formal methods to specify the functional properties of engineering software. *Computers & Structures*, 45(3):557–570. https://doi.org/10.1016/0045-7949(92)90440-B.

Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and Geijn, R. A. (2005). The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 31(1):1–26.

Carver, J. C. (2009). Report: The second international workshop on software engineering for CSE. *Computing in Science & Engineering*, 11(6):14–19.

Cross, H. (1930). Analysis of continuous frames by distributing fixed-end moments. *Proceedings of the American Society of Civil Engineers*, pages 919–928.

Faulk, S., Loh, E., Van De Vanter, M. L., Squires, S., and Votta, L. G. (2009). Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science & Engineering*, 11(6):30–39.

Graham, L., Butler, T., Walsh, S., Dawson, C., and Westerink, J. J. (2017). A measure-theoretic algorithm for estimating bottom friction in a coastal inlet: Case study of Bay St. Louis during Hurricane Gustav (2008). *Monthly Weather Review*, 145(3):929–954.

Hanson, J. L., Wadman, H., Blanton, B., and Roberts, H. (2013). Coastal Storm Surge Analysis: Modeling System Validation. Report 4: Intermediate Submission No. 2.0 (No. ERDC/CHL-TR-11-1). Technical report, U.S. Army Engineering Research and Development Center, Kitty Hawk, NC, 2013.

Hatton, L. (2007). The chimera of software quality. *Computer*, 40(8).

Hatton, L. and Roberts, A. (1994). How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20(10):785–797.

Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

Jackson, D. and Damon, C. A. (1996). Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495.

Jackson, D. and Wing, J. (1996). Lightweight formal methods. *Computer*, 29(4):21.

Kelly, D. F. (2007). A software chasm: Software engineering and scientific computing. *IEEE Software*, 24(6).

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc.

Linz, P. (1988). A critique of numerical analysis. *Bulletin of the American Mathematical Society*, 19(2):407–416.

Norman, D. A. (1999). Affordance, conventions, and design. *interactions*, 6(3):38–43.

Sanders, R. and Kelly, D. (2008). Dealing with risk in scientific software development. *IEEE Software*, 25(4).

Schrage, L. (1991). *LINDO: An Optimization Modeling System*. The Scientific Press, fourth edition.

Sheng, Y. P., Davis, J. R., Figueiredo, R., Liu, B., Liu, H., Luettich, R., Paramygin, V. A., Weaver, R., Weisberg, R., Xie, L., and Zheng, L. (2012). A regional testbed for storm surge and coastal inundation models—An overview. In *Proceedings of the 12th International Conference on Estuarine and Coastal Modeling*, pages 476–495. ASCE.

Siegel, S. F., Mironova, A., Avrunin, G. S., and Clarke, L. A. (2008). Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):10.

Storer, T. (2017). Bridging the chasm: A survey of software engineering practice in scientific programming. *ACM Computing Surveys (CSUR)*, 50(4):47.

Wilson, G. V. (2006). Where's the real bottleneck in scientific computing? *American Scientist*, 94(1):5.

Zienkiewicz, O. C., Taylor, R. L., and Nithiarasu, P. (2013). *The Finite Element Method for Fluid Dynamics*. Butterworth-Heinemann, Oxford, UK, 7th edition.