PART I

THE TAO OF SCIENTIFIC OOP

1 Development Costs and Complexity

"Premature optimization is the root of all evil."
Donald Knuth

1.1 Introduction

The past several decades have witnessed impressive successes in the ability of scientists and engineers to accurately simulate physical phenomena on computers. In engineering, it would now be unimaginable to design complex devices such as aircraft engines or skyscrapers without detailed numerical modeling playing an integral role. In science, computation is now recognized as a third mode of inquiry, complementing theory and experiment. As the steady march of progress in individual spheres of interest continues, the focus naturally turns toward leveraging efforts in previously separate domains to advance one's own domain or in combining old disciplines into new ones. Such work falls under the umbrella of *multiphysics modeling*.

Overcoming the physical, mathematical, and computational challenges of multiphysics modeling comprises one of the central challenges of 21st-century science and engineering. In one of its three major findings, the National Science Foundation Blue Ribbon Panel on Simulation-Based Engineering Science (SBES) cited "open problems associated with multiscale and multi-physics modeling" among a group of "formidable challenges [that] stand in the way of progress in SBES research." As the juxtaposition of "multiphysics" and "multiscale" in the panel's report implies, multi-physics problems often involve dynamics across a broad range of lengths and times.

At the level of the physics and mathematics, integrating the disparate dynamics of multiple fields poses significant challenges in simulation accuracy, consistency, and stability. Modeling thermal radiation, combustion, and turbulence in compartment fires, for example, requires capturing phenomena with length scales separated by several orders of magnitude. The room-scale dynamics determine the issue of paramount importance: safe paths for human egress. Determining such gross parameters while resolving neither the small length scales that control the chemical kinetics nor the short time scales on which light propagates forms an active area of research.

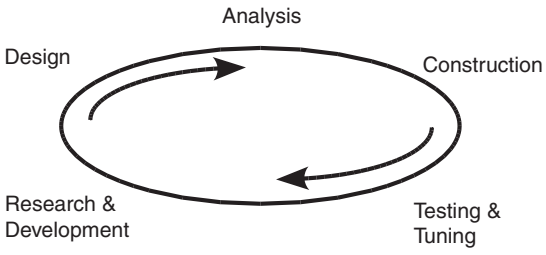

Figure 1.1. Scientific software life cycle.

At the level of the computation, the sheer size of multiscale, multiphysics simulations poses significant challenges in resource utilization. Whereas the largest length and time scales determine the spatial and temporal extent of the problem domain, the shortest determine the required resolution. The ratio of the domain size to the smallest resolvable features determines the memory requirements, whereas the ratio of the time window to the shortest resolvable interval determines the computing time. In modeling-controlled nuclear fusion, for example, the time scales that must be considered span more orders of magnitude than can be tracked simultaneously on any platform for the foreseeable future. Hence, much of the effort in writing related programs goes into squeezing every bit of performance out of existing hardware.

In between the physical/mathematical model development and the computation lies the software development process. One can imagine the software progressing through a life cycle much like any other product of human effort: starting with research and development and ending with the testing and fine-tuning of the final product (see Figure 1.1). However, as soon as one searches for evidence of such a life cycle in scientific computing, gaping holes appear. Research on the scientific software development process is rare. Numerous journals are devoted to novel numerical models and the scientific insights obtained from such models, but only a few journals focus on the software itself. Notable examples include the refereed journals *Scientific Programming* and *ACM Transactions on Mathematical Software*.

Further along in the life cycle, the terrain becomes even more barren. Discussions of scientific software design rarely extend beyond two categories: (1) explanations of a particular programming paradigm, which fall into the implementation stage of the life cycle, and (2) strategies for improving run-time efficiency, which arguably impacts all stages of the life cycle but comprises an isolated activity only once a prototype has been constructed in the testing and tuning phase. Articles and texts on the first topic include presentations of structured and object-oriented programming, whereas those on the second include discussions of parallel programming and high-performance computing.

Implementation-independent design issues such as the chosen modular decomposition and its developmental complexity have received scant attention in the scientific computing community. Not surprisingly then, quantitative analyses of software designs have been limited to run-time metrics such as speed and parallel efficiency. In presenting attempts to analyze static source code organization, the author often encounters a perception that program structure is purely stylistic – of

the "You say 'toe-may-toe'; I say 'toe-mah-toe' " variety. Even to the extent it is recognized that program structure matters, no consensus is emerging on what structural precepts prove best.

That scientific program design warrants greater attention has been noted at high levels. The 1999 Presidential Information Technology Advisory Committee (PITAC) summarized the situation for developers of multidisciplinary scientific software:

> Today it is altogether too difficult to develop computational science software and applications. Environments and toolkits are inadequate to meet the needs of software developers in addressing increasingly complex interdisciplinary problems... In addition, since there is no consistency in software engineering best practices, many of the new applications are not robust and cannot easily be ported to new hardware.

Fortunately, the situation with regards to environments and toolkits has improved since the PITAC assessments. Using tools such as Common Component Architecture (CCA), for example, one can now link applications written by different developers in different languages using different programming paradigms into a seamless, scalable package without incurring the language interoperability and communication latency issues associated with non-scientific toolkits that facilitate similar linkages. Several groups have constructed development frameworks on top of CCA to facilitate distributed computing (Zhang et al. 2004), metacomputing (Malawski et al. 2005), and other computing models.

However, the situation with regard to software engineering best practices has seen far less progress. Most authors addressing scientific programming offer brief opinions to aid robustness before retreating to the comfortable territory of run-time efficiency or numerical accuracy. Without a healthy debate in the literature, it is unsurprising that no consensus has emerged. By contrast, in the broader software engineering community, consensus has been building for over a decade on the best practices at least within one development paradigm: object-oriented design (OOD). In this context, the best practices have been codified as *design patterns*, which are cataloged solutions to problems that recur across many projects.

Experience indicates many scientific programmers find the term "design pattern" vague or awkward on first hearing. However, its application to software development conforms with the first four definitions of "pattern" in the Merriam Webster dictionary:

1. a form or model proposed for imitation,
2. something designed or used as a model for making things,
3. an artistic, musical, literary, or mechanical design or form,
4. a natural or chance configuration.

Having proved useful in the past, design patterns are offered as models to be followed in future work. The models have artistic value in the sense of being elegant and in the sense that software engineering, like other engineering fields, is part science and part art. And much like mechanical design patterns, some software design patterns mimic patterns that have evolved by chance in nature.

Gamma et al. (1995) first collected and articulated object-oriented software design patterns in 1995 in their seminal text *Design Patterns: Elements of Reusable*

Object-Oriented Software. They drew inspiration from the 1977 text, *A Pattern Language*, by architects Alexander et al., who in turn found inspiration in the beauty medieval architects achieved by conforming to local regulations that required specific building features without mandating specific implementations of those features. This freedom to compose variations on a theme facilitated individual expression within the confines of proven forms.

Gamma et al. did not present any domain-specific patterns, although their book's introduction suggests it would be worthwhile for someone to catalog such patterns. Into this gap we thrust the current text. Whereas Part I of this text lays the object-oriented foundation for discussing design patterns and Part III discusses several related advanced topics, Part II aims to:

1. Catalog general and domain-specific patterns for multiphysics modeling,
2. Quantify how these patterns reduce complexity,
3. Present Fortran 2003 and C++ implementations of each pattern discussed.

Each of these objectives makes a unique contribution to the scientific computing field. The authors know of only a handful of publications on object-oriented software design patterns for scientific software (Blilie 2002; Decyk and Gardner 2006; Decyk and Gardner 2007; Gardner and Manduchi 2007; Markus 2006; Rouson et al. 2010) and even fewer quantitative analyses of how patterns reduce scientific programming complexity (Allan et al. 2008; Rouson 2008). Finally, published design pattern examples that take full advantage of the new object-oriented constructs of Fortran 2003 have only just begun to appear (Markus 2008; Rouson et al. 2010).

That Fortran implementations of object-oriented design patterns have lagged those in other languages so greatly is both ironic and understandable. It is ironic because object-oriented programming started its life nearly four decades ago with Simula 67, a language designed for physical system simulation, which is the primary use of Fortran. It is understandable because the Fortran standards committee, under heavy influence by compiler vendors, moved at glacial speeds to provide explicit support for object-orientation. (See Metcalf et al. 2004 for some history on this decision.) In the absence of object-oriented language constructs, a small cadre of researchers began developing techniques for emulating object-orientation in Fortran 90/95 in the 1990s (Decyk et al. 1997a, 1997b, 1998; Machiels and Deville 1997), culminating in the publication of the first text on OOP in Fortran 90/95 by Akin (2003).

The contemporaneous publication of the Fortran 2003 standard, which provides object-oriented language constructs, makes the time ripe for moving beyond the basic mechanics to higher-level discussions of objected-oriented design patterns for scientific computing. Because the availability of Fortran 2003 motivates this book, Fortran serves as the primary language for Part I of this text. Because most C++ programmers already know OOP, there is less need to provide C++ in this part of the text. Some of the complexity arguments in Part I, however, are language-independent, so we provide C++ translations to make this section accessible to C++ programmers. In Part II, greater effort is made to write C++ examples that stand on their own as opposed to being translations of Fortran.

A Pressing Need for 21st-Century Science

The 20th-century successes in simulating individual physical phenomena can be leveraged to simulate multiscale, multiphysics phenomena much more simply if we begin distilling, analyzing, and codifying the best practices in scientific programming.

1.2 Conventional Scientific Programming Costs

The word "costs" plays two roles in this section's title. As a noun, it indicates that the section discusses the costs associated with writing and running conventional scientific programs. As a verb, it indicates that the conventional approach to writing these programs costs projects in ways that can be alleviated by other programming paradigms. Whereas this section estimates those costs, section (1.4) presents alternative paradigms.

The software of interest is primarily written for research and development in the sciences and engineering sciences. Although many such codes later form the core of commercial packages used in routine design and analysis, all are grouped under the heading "scientific" in this text. During the early phases of the development, many scientific programming projects tackle problems that are sufficiently unique or demanding to preclude the use of commercial off-the-shelf software for the entire project. Although some commercial library routines might be called, these serve supporting roles such as solving linear and nonlinear systems or performing data analysis. Viewed from a client/server perspective, the libraries provide general mathematical services for more application-specific client code that molds the science of interest into a form amenable to analysis with the chosen library.

Some mathematical libraries are portable and general-purpose, such as those by the Numerical Algorithms Group (NAG) or the Numerical Recipes series. Others are tuned to exploit the features of specific hardware. The Intel Math Kernel Library (MKL) targets Intel processors and scales up to large numbers of processors on distributed-memory clusters, whereas the Silicon Graphics, Inc. (SGI) math library scales on distributed shared-memory supercomputers with SGI's proprietary high-bandwidth memory architecture.

Mathematical libraries attempt to embody the best available numerical algorithms without imposing any particular program organization. Responsibility for program organization, or *architecture*, thus rests with the developers. Although the definition of software architecture remains a subject of active debate, there is long-standing agreement on the attractiveness of modular design, so for present purposes, "architecture" denotes a specific modular decomposition. This decomposition would include both the roles of individual *modules* and the relationships between them. To avoid language- and paradigm-specificity, "module" here denotes any isolated piece of code whether said isolation is achieved by separation into its own file or by some other construct.

Most scientific programmers are trained in engineering or science. Typical curricula in these fields include only an introductory programming course that teaches the use of a particular programming language and paradigm. Teaching only the

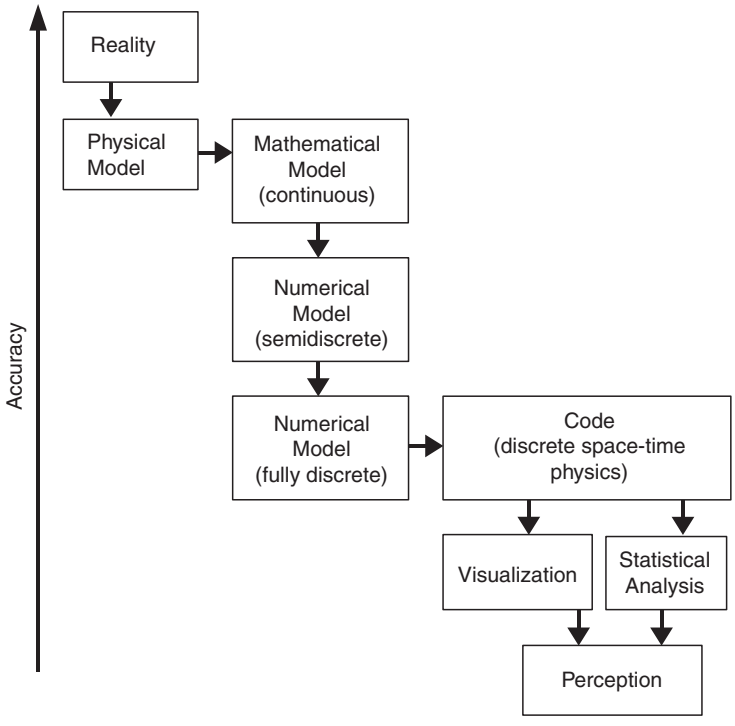

Figure 1.2. Sequence of models in the conventional scientific code development process.

implementation phase differs considerably from the rest of the curriculum, wherein students are encouraged to think abstractly about systems and to quantitatively evaluate competing models for describing those systems. In the automatic controls courses taken by many engineering students, for example, model elements are delineated precisely and their roles and relationships are specified mathematically, so the system behavior can be analyzed rigorously.

Figure 1.2 situates programming within the conventional modeling process. One starts off wishing to model physical reality. Based on observations of that reality, one constructs a physical model, which for present purposes comprises a conceptual abstraction of reality. Both "model" and "abstraction" here connote useful simplifications that retain only those elements necessary for a given context. For example, one might abstract a cooling fin on a microprocessor chip as a thin solid with constant properties transporting thermal energy in the direction orthogonal to the chip. One might further assume the energy transfer occurs across a distance that is large relative to the material's intermolecular spacing, so it can be considered a continuum.

Although physical models are typically stated informally, Collins (2004) recently proposed the development of a physics markup language (PML) based on the extensible markup language (XML). Formal specifications in such a markup language would complement the programming techniques presented in this book. Specifically, automatic code generation from a markup document might produce software counterparts of algebraic and differential operators specified in PML. Chapter 3 discusses the implementation of such operators.

The first formal step in conventional scientific work involves expressing the physical model in a mathematical model with continuous variation of the dependent variables in space and time. For the fin problem, this leads to the one-dimensional (1D), unsteady heat equation:

$$\begin{aligned} \frac{\partial T}{\partial t} &= \alpha \frac{\partial^2 T}{\partial x^2}, \quad \Omega \times T = (0, L_{fin}) \times (0, t_{final}) \\ T(0,t) &= T_{chip} \\ T(L_{fin},t) &= T_{air} \\ T(x,0) &= T_{air} \end{aligned} \tag{1.1}$$

where $T(x,t)$ is the temperature at time t a distance x from the chip, α is the fin's thermal diffusivity, $\Omega \times T$ is the space-time domain, L_{fin} is the fin length, t_{final} is the final time of the simulation, and where T_{chip} and T_{air} are boundary conditions.

Solving the mathematical model on a computer requires rendering equations (1.1) discrete. Most numerical schemes employ the *semidiscrete method*, which involves discretizing space first and time second. Most spatial discretizations require laying a grid over the spatial domain. Given a uniformly spaced grid overlaid on Ω, applying the central difference formula to the right-hand side of (1.1) leads to a set of coupled ordinary differential equations for the nodal temperatures:

$$\frac{d\vec{T}}{dt} \equiv \frac{\alpha}{\Delta x^2} \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & & \\ & & & 1 & -2 \end{bmatrix} \vec{T} + \frac{\alpha}{\Delta x^2} \begin{Bmatrix} T_{chip} \\ 0 \\ \vdots \\ T_{air} \end{Bmatrix} \tag{1.2}$$

where $\Delta x \equiv L_{fin}/N$ is the uniform spacing in between the $N+1$ grid points covering Ω. One can derive the same discrete system of equations via a finite volume scheme or a Galerkin finite element method with linear-basis functions. We refer to the intervals between nodes generically as *elements* and assume the reader is familiar with one or another path that starts from equations (1.1) and leads to equations (1.2).

The semidiscrete model is rendered fully discrete by applying a time integration algorithm to equations (1.2). The simplest case is explicit Euler, which corresponds to a two-term Taylor series expansion:

$$\vec{T}^{n+1} = \vec{T}^n + \Delta t \frac{d\vec{T}}{dt}\Big|_{t_n} = \vec{T}^n + \Delta t \left(A\vec{T}^n + \vec{b} \right) \tag{1.3}$$

where $\vec{T}^n$ and $\vec{T}^{n+1}$ are vectors of $N-1$ internal grid point temperatures at time steps$t_n \equiv n\Delta t$ and $t_{n+1} \equiv t_n + \Delta t$, respectively.

Most scientific computing publications focus on the previously described modeling steps, treating their implementation in code as an afterthought, if at all. This has numerous adverse consequences. First, several very different concepts are hardwired into equation (1.3) in ways that can be hard to separate should one stage in the modeling process require reconsideration. Equation (1.3) represents a specific numerical integration of a specific discrete algorithm for solving a specific partial differential equation (PDE). Without considerable forethought, the straightforward

expression of this algorithm in software could force major code revisions if one later decides to change the discretization scheme, for example, the basis functions or the PDE itself. The next section explains why the conventional approach to separating these issues scales poorly as the code size grows.

A side effect of conflating logically separate modeling steps is that it becomes difficult to assign responsibility for erroneous results. For example, instabilities that exist in the fully discrete numerical model could conceivably have existed in continuous mathematical model. Rouson *et al.* (2008b) provided a more subtle example in which information that appeared to have been lost during the discretization process was actually missing from the original PDE.

As previously mentioned, the code writing is the first step with no notion of abstraction. Except for flow charts – which, experience indicates, very few programmers construct in practice – most discussions of scientific programs offer no formal description of program architecture other than the code itself. Not surprisingly then, scientific program architecture is largely ad hoc and individualistic. Were the problems being solved equally individualistic, no difficulty would arise. However, significant commonalities exist in the physical, continuous mathematical, and discrete numerical models employed across a broad range of applications. Without a language for expressing common architectural design patterns, there can be little dialogue on their merits. Chapter 2 employs just such a language: the Unified Modeling Language (UML). Part II presents design patterns that exploit commonalities among a broad range of problems to generate flexible, reusable program modules. Lacking such capabilities, conventional development involves the reinvention of architectural forms from scratch.

How does redevelopment influence costs? Let's think first in the time domain:

$$\text{Total solution time} = \text{development time} + \text{computing time} \tag{1.4}$$

Until computers develop the clairvoyance to run code before it is written, programming will always precede execution, irrespective of the fact that earlier versions of the program might be running while newer versions are under development. Since the development time often exceeds the computing time, the fraction of the total solution time that can be reduced by tuning the code is limited. This is a special case of Amdahl's law, which we revisit in Section 1.5 and Chapter 12.

Time is money, so equation (1.4) can be readily transformed into monetary form by assuming the development and computing costs are proportional to the development and computing times as follows:

$$\begin{aligned} \$_{solution} &= \$_{development} + \$_{computing} \\ &= N_{dev} p_{avg} t_{dev} + \frac{\$_{computer}}{N_{users}} \frac{t_{run}}{t_{useful}} \end{aligned} \tag{1.5}$$

where the \$ values are costs; N_{dev}, p_{avg}, and t_{dev} are the number of developers, their average pay rate, and the development time, respectively; and N_{users}, t_{run}, and t_{useful} are the number of computer users, the computing time, and the computer's useful life, respectively. In the last term of equation (1.5), the first factor estimates the fraction of the computer's resources available for a given user's project. The second

factor estimates the fraction of the computer's useful life dedicated to that user's runs.

There are of course many ways equation (1.5) could be refined, but it suffices for current purposes. For a conservative estimate, consider the case of a graduate student researcher receiving a total compensation including tuition, fees, stipend, and benefits of $50,000/yr for a one-year project. Even if this student is the sole developer of her code and has access to a $150,000 computer cluster with a useful life of three years, she typically shares the computer with, say, four other users. Therefore, even if her simulations run for half a year on the wall clock, the solution cost breakdown is

$$\$_{solution} = (1 \cdot \text{developer})(\$50{,}000/\text{year})(1 \cdot \text{year}) \tag{1.6}$$

$$+\frac{\$150{,}000}{5 \cdot \text{users}}\frac{0.5 \cdot \text{user} \cdot \text{years}}{3 \cdot \text{years}} \tag{1.7}$$

$$= \$50{,}000 + \$5{,}000 \tag{1.8}$$

so the development costs greatly exceed the computing costs. The fraction of costs attributable to computing decreases even further if the developer receives higher pay or works for a longer period – likewise if the computer costs less or has a longer useful life. By contrast, if the number of users decreases, the run-time on the wall clock decreases proportionately, so the cost breakdown is relatively insensitive to the resource allocation. Similar, the breakdown is insensitive to adding developers if the development time is inversely proportional to the team size (up to some point of diminishing returns).

For commercial codes, the situation is a bit different but the bottom line is arguably the same. The use and development of commercial codes occur in parallel, so the process speedup argument breaks down. Nonetheless, development time savings lead to the faster release of new features, providing customer benefit and a competitive advantage for the company.

This forms the central thesis on which this book rests:

Your Time Is Worth More Than Your Computer's Time

Total solution time and cost can be reduced in greater proportion by reducing development time and costs than by reducing computing time and costs.

1.3 Conventional Programming Complexity

Part of the reason scientific computing has traditionally focused on the earlier stages of Figure 1.2 is that the accuracy losses at these stages are well known and rigorously quantifiable. As indicated by the figure's vertical axis, each transition in the discretization process has an associated error. For example, the truncation error, e , of the spatial discretization inherent in equation (1.2) is bounded by a term proportional to the grid spacing:

$$e \leq C\Delta x^3 \tag{1.9}$$

where C typically depends on properties of the exact solution and its derivatives.