Contents

Chapter 1

Software Process for Multiphysics Multicomponent Codes

1.1 Introduction

Computational science and engineering (CSE) communities develop complex applications to solve scientific and engineering challenges, but these communities have a have a mixed record of using software engineering best practices. Many codes developed by CSE communities adopt standard software practices when the size and complexity of an application becomes too unwieldy to continue without them [**?**]. The driving force behind adoption is usually the realization that without using software engineering practices, the development, verification and maintenance of applications can become intractable. As more codes cross the threshold into increasing complexity, software engineering processes are being adopted from practices derived outside of the scientific and engineering domain. State-of-the-art for software engineering practices in CSE codes often lags behind that in the commercial software space. **KA NOTE: Do we have any reference for this claim? AD: I don't know of one, though this seems to be the consensus. Let's just soften a little then. I added "often"** There are many reasons for it, first, is that not all software engineering practices can actually be adopted by the CSE code developers. Secondly, there is often little funding for code development and maintenance. And finally, there is very little research in the software engineering community targeted to specific needs of CSE codes.

Many software best practices are not well-suited for CSE codes without modification and/or customization. In particular, multiphysics and multicomponents codes that run on the large High Performance Computing (HPC) platforms have specific requirements. In some cases, the inherent physics of scientific applications require different software methodologies. In other cases, a large premium is placed on performance, rather than code architecture, making it necessary to sacrifice some known software engineering best practices. Still others are more sociological because codes may be developed by domain scientists and their graduate students who have different priorities and scientific goals. The challenges in developing scientific and engineering applications range from data dependencies, to architectural trade-offs, to the

process for their maintenance and growth. The standard practices adopted by the CSE codes include repositories for code version control, modular code design, licensing process, regular testing, documentation, release and distribution policies and contribution policies. Less frequently used practices include code-review and code-deprecation. The degree of adoption and sophistication in using software engineering practices varies among teams. Many of the reasons for lower penetration of more formal software engineering practices are discussed in section **??**. Even among the widely adopted practices, most are modified and customized by the developers for their own needs. The next two sections outline the challenges that are either unique to, or are more dominant in this domain than elsewhere.

1.2 Lifecycle

Scientific software is designed to model some phenomena in the physical world. The phenomena may be at the microscopic level, for example protein folding, or at extremely large scales, for example galaxy cluster mergers. In some applications multiple scales are modeled. (The term 'physical' used here includes chemical and biological systems since physical processes are underlying building blocks for those systems too.) The physical characteristics of the systems being modeled are translated into mathematical models that are said to describe the essential features of the behavior of the system being studied. These equations are then discretized, and numerical algorithms are used to solve them. This process requires diverse expertise and adds many stages in the development and lifecycle of scientific software that may not be encountered elsewhere.

1.2.1 Development Cycle

For scientific simulations, modeling begins with equations that describe the general class of behavior to be studied, for example the Navier-Stokes equations describe the flow of compressible and incompressible fluids, and Van-der-vaal equations describe force interactions among molecules in a material. There may be more than one set of equations if there are behaviors that are not adequately captured by one set. In translating the model from mathematical representation to computational representation two processes go on simultaneously, discretization and approximation. One can argue that discretization is, by definition, an approximation because it is in effect sampling continuous behavior where information is lost between sampling intervals. This loss manifests itself as error terms in the discretized equations, but error terms are not the only approximations. Depending upon the level of understanding of specific sub-phenomena, and available compute resources,

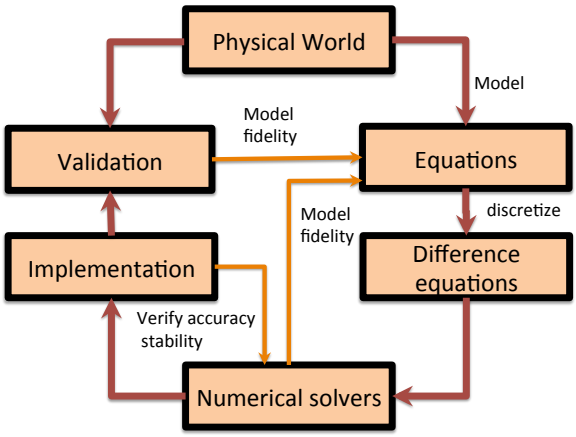

FIGURE 1.1: Development cycle of modeling with partial differential equations

scientists also use their judgement to make other approximations. Sometimes, to focus on a particular behavior, a term in an equation may be simplified or may be even completely dropped. At other times some physical details may be dropped from the model because they are not understood well enough by the scientists. Or the model itself may be an approximation.

The next stage in developing the code is finding the appropriate numerical methods for each of the models. Sometimes good methods exists that can be used "as-is". Other times, they may need to be customized, or new methods may need to be developed. There may need to be validation of the method's applicability to the model if the method is new or significantly modified. Unless an implementation of the method is readily available as a third party software (stand-alone or in a library), it has to be implemented and verified. It is at this stage that the development of a CSE code begins to resemble that of general software. The numerical algorithms are specified, the semantics are understood, and they need to be translated into executable code. Figure 1.1 gives an example of the development cycle of a multiphysics application modeled using partial differential equations.

1.2.2 Verification and Validation

The terms verification and validation are often used interchangeably, but to some communities have specific definitions. In one narrow definition, validation, ensures that the mathematical model correctly defines the physical phenomena, while verification makes sure that the implementation of the model

is correct. In other words, a model is validated against observations or experiments from the physical world, whereas a model is verified by other forms of testing. Other definitions give broader scope to validation. For example, validation of a numerical method may be constructed through code-to-code comparisons, and its order can be validated through convergence studies. Similarly, the implementation of a solver can be validated against an analytically obtained solution for some model if the same solver can be applied and the analytical solution is also known, though this is not always possible. Irrespective of specific definitions, what is true is that correctness must be assured at all the stages from model to implementation.

There are many degrees of freedom in the process of deriving a model as discussed in the previous section, therefore, the validation of the model must be carefully calibrated by scientific experts. Similarly, verification of a numerical method requires applied math expertise because the method needs verification of its stability, accuracy and order of convergence, in addition to correctness. Numerical methods have their own error analysis because of approximations and many of these methods are themselves objects of ongoing research, so their implementation may need modifications from time to time. Whenever this happens, the entire gamut of verification and validation needs to be applied again. This is an instance of a particular challenge in the CSE software where no amount of specification is enough to hand the implementation over to software engineers or developers who do not have domain or method knowledge. A close collaboration with applied mathematicians and method developers is necessary because the process has to be iterative with scientific judgement applied at every iteration.

One other unique verification challenge in CSE software is the consequences of finite machine precision of floating point numbers. Any change in compilers, optimization levels, and even order of operations can cause numerical drift in the solutions. Especially in applications that have a large range of scales, it can be extremely difficult to differentiate between a legitimate bug and a numerical drift. Therefore, relying upon bitwise reproducibility of the solution is rarely a sufficient method for verifying the continued correctness of an application behavior. Robust diagnostics (such as statistics or conservation of physical quantities) need to be built into the verification process. This issue is discussed in greater detail in chapter **??**.

1.2.3 Maintenance and Extensions

In a simplified view of software lifecycle, there is a design and development phase, followed by production and maintenance phase. Even in well engineered codes this simplified view typically applies only to the infrastructure and API's which have a distinct development phase which has limited spill into the remainder of the lifecycle. The numerical algorithms and solvers can be in a continually evolving state reflecting the advances in their respective fields. The development of CSE software is usually responding to an immediate sci-

entific need, so the codes get employed in production as soon as a minimal set of computational modules necessary for even one scientific project are built. Similarly, the development of computational modules almost never stops all through the code lifecycle because new findings in science and math almost continuously place new demands on them. The additions are mostly incremental when they incorporate new findings into an existing feature, they can also be substantial when new capabilities are added. The need for new capabilities may arise from greater model fidelity, or from trying to simulate a more complex model. Sometimes a code designed for one scientific field may have enough in common with another field that capabilities may be added to enable it for the new field.

Whatever may be the cause, co-existence of development and production/maintenance phases is a constant challenge to the code teams. It becomes acute when the code needs to undergo major version changes. The former can be managed with some repository discipline in the team coupled with a solid testing regime. The latter is a much bigger challenge where the plan has to concern itself with questions such as how much backward compatibility is suitable, how much code can go offline, and how to reconcile ongoing development in code sections that are substantially different between versions. FLASH's example in section **??** describes a couple of strategies that met the conflicting needs of developers and production users in both scenarios. Both required co-operation and buy-in from all the stakeholders to be successful.

1.2.4 Testing

KA NOTE: KA will expand this section Testing of CSE software needs to reflect the layered complexity that the codes themselves have. The first line of attack is to develop the unit tests which isolate testing of one component of the code. However, as mentioned in chapter **??**, in CSE codes, often there are dependencies between different components of the code that can not be isolated, making unit testing more difficult. In these cases, testing should be done with a minimal possible combination of components. In effect, these minimally combined tests play the same role in the testing regime that unit tests do because they focus on possible defects in a very narrow section of the code. In addition, multicomponent CSE software should test various permutations and combination of components in different ways. Configuring tests in this manner will help verify that all configurations are within the accuracy and stability constraints.

1.2.5 Performance Portability

KA will expand this section Another aspect of multiphysics CSE software is its need for performance portability. HPC machines are expensive and rare resources and in order to achieve high application performance, codes need to be optimized for the unique HPC architectures. However, typical

lifecycle of a multiphysics application spans many generations HPC systems which have a typical lifespan of about 4-5 years. Depending upon the size of the code, optimization for a specific target platform can take a significant fraction of the platform lifecycle, time when the code may not be available for science runs. Without careful planning and coordination, a large fraction of scientists' time could be lost in porting and optimizing a code for a new system. Developers have the choice of adding machine specific optimizations or creating more general optimization that will work for a broader class of systems. CSE codes must consider the trade-offs and advantages of a highly optimized code for a single platform or a design HPC CSE codes consider the trade-offs and opt to design their software using constructs that perform modestly well across a range of platforms.

1.2.6 Using CSE Software

There is a fundamental requirement from the users of scientific software that rarely comes into play for users of other kinds of software. For obtaining valid scientific results the users must have a basic understanding of the phenomena being modeled and the approximations, and they must know which questions can be validly addressed by these models. For example, Eulerian equations of hydrodynamics do not account for viscosity. If one is modeling a fluid where viscosity is important, one should not use a code that uses Eulerian equations to model the fluid. Similarly, they must know and understand the valid regimes for applicability of numerical algorithms, and also the accuracy and stability behavior of the algorithms. For example a numerical method that can resolve a smooth fluid flow only will fail if there are discontinuities. Similarly Fast Fourier Transform (FFT) methods work really well for periodic boundaries. For all other boundaries they either need some additional processing, or are not applicable. Also, in order to use the FFT method the user must know what is the highest mode they are resolving. If enough terms are not used in the Fourier series approximation the computed result may filter out important information and lead to wrong results.

Similarly, sometimes equations have mathematically valid but physically invalid solution. A badly applied numerical scheme may converge to such a non-physical solution. At best any of the above situations may lead to a waste of computing resources if the defect in the solution is detected. At worst they may lead to wrong scientific conclusions being drawn. Some in the scientific community even argue that those who have not written at least some basic version of their own code for their own problem should not be using other's public or community code. Though that argument goes too far, it embodies a general belief in the scientific community that users of scientific codes have a great responsibility to know their tools and understand their capabilities and limitations. These are some of the reasons that also play a role in tendency of scientific codes to do strict gatekeeping for contributions, and mostly operate in the cathedral mode.