Some Challenges for Model-Based Simulation * †

Walid Taha[1,2] Robert Cartwright[2,1]

[1]IDE, Halmstad University, Sweden, Walid.Taha@hh.se
[2]Computer Science, Rice University, USA, corky@rice.edu

Abstract

Comprehensive analytical modeling and simulation of cyber-physical systems is an integral part of the process that brings novel designs and products to life. But the effort needed to go from analytical models to running simulation code can impede or derail this process. Our thesis is that this process is amenable to automation, and that automating it will accelerate the pace of innovation. This paper reviews some basic concepts that we found interesting or thought-provoking, and articulates some questions that may help prove or disprove this thesis. While based on ideas drawn from different disciplines, we observe that all these questions pertain in a profound way to how we can reason and compute with real numbers.

1. Introduction

It is widely anticipated that much of tomorrow's innovation will be in the form of cyber-physical systems, that is, systems that include computing, communicating, and physically dynamic components. A vivid example of such a system is a team of robots playing soccer, or a fleet of vehicles functioning collectively as an intelligent transportation system. Because we need to reflect on, reason and communicate about designs, modeling is an integral part of conceiving and developing new products in such domains. Many important problems defined in terms of mathematical models do not have closed solutions. As a result, simulation must be an integral part of the innovation process. Unfortunately, the formidable time and effort needed to convert analytical models to running simulation code can impede or even derail the process. Our thesis is that this transformation process can be more reliably and predictably automated, and that such automation can play a crucial role in training the cadre of future innovators and making them more productive.

Although a vast range of modeling and simulation tools already exists, automating the mapping from models to simulation codes remains a challenging and elusive goal. This is the case even for seemingly elementary domains such as rigid-body dynamics, which is a fairly simplified type of mechanical models that can be used to develop basic models of robot dynamics [7]. While this first work succeeds in identifying some basic problems and showing how programming language techniques such as partial evaluation can play a role in addressing them, the automation problem is larger than this. It cannot be addressed with a handful of research papers, and success in demonstrating this thesis may have far reaching consequences on the CPS domain. At the same time, entering the domain of cyber physical systems can be challenging, primarily because of the vast diversity of technical disciplines that relate to different aspects of this domain. This diversity of sources is an obstacle not just for researchers but also for education.

In an attempt to reduce the effort needed to overcome this problem, this paper reviews basic concepts from several related areas that we found interesting or insightful, and articulates some questions that may assist in investigating this thesis. While drawn from different disciplines, all of these observations and questions pertain to how we can reason and compute with reals.

Contributions: Modeling systems that involve real-valued and time-varying quantities is somehow at odds with traditional approaches to computing. Furthermore, it is not obvious precisely what needs to be done to integrate real numbers with these approaches. With the aim of shedding some light on what is missing, this paper reviews basic concepts and puts forth some questions about

- Traditional floating-point methods, and
- Basic properties of real numbers.
- Interval arithmetic.

Software engineering in general, and in particular programming languages semantics, design, implementation, and engineering have a lot to contribute to the study of cyber-physical systems. We hope that the observations and ques-

* This manuscript is a reduced and edited revision of an invited paper entitled "The Trouble with Real Numbers" and presented at the WS4C workshop of INFORMATIK 2011 held in Berlin.

† This research was supported by Halmstad University, the Swedish Knowledge Foundation (KK) Centre CERES, the Swedish Knowledge Foundation (KK) Environment at Halmstad University, and US NSF CPS awards number 1136099 and 1136104.

The 4th Analytic Virtual Integration of Cyber-Physical Systems Workshop December 3, 2013, Vancouver, Canada.

 Proceedings are available at: http://dx.doi.org/10.3384/ecp13090

AVICPS website: http://www.analyticintegration.org/

tions presented here encourage the reader to share this view.

2. Traditional Floating-Point Methods

Traditional numerical methods and simulation technologies have carried us a long way, allowing us to build airplanes, space ships, and many other advanced, as well as mundane, innovations. However, part of the difficulty in going from analytical models to simulation codes lies in the nature of the codes, which are built using traditional numerical methods techniques. Numerical methods for solving virtually any type of problem are highly varied and yield qualitatively different results when solving the same problem. Yet the user of these methods still has to bear the responsibility of choosing the right method for dealing with each different kind of model that they formulate and wish to simulate. This is a huge distraction for a user whose concern is to study the system that is being modeled, rather than how to implement the solver for different components of the system being modeled. Because it requires making a choice between more than two options for each component, it is a problem that has work-hour cost with an order of complexity exponential in the number of components. This is an optimistic estimate, given that testing whether a certain method works well for a certain type of component is usually done using a certain data set, and this result does not necessarily imply that it will work equally well for other data sets. Even with this simplifying assumption, it is a serious impediment to productivity.

An important question from the software engineering point of view is whether there is a way to hide these implementation choices from the user in such a way that the right one is always chosen (if it exists). A key technical challenge in approaching this question is to determine whether or not there exists a single method (or semantics) that can be viewed as a gold standard, and which can be used for both statically proving or dynamically checking the correctness of any given method. Since dynamic checking will always allow us to validate more methods than static techniques (due to the reduction in the number of quantifiers in the problem), the natural and important question to ask becomes whether there are also *universal numerical methods* that can enable the automatic selection of the right method for the given problem dynamically. The existence of such methods could be most insightful if they end up being simpler than many of the other prevailing methods, as it is likely that they would be revealing of some deeper ideas that are today only implicit in such methods.

Thus, traditional numerical codes only work correctly if certain assumptions about their inputs are satisfied, and these assumptions are not usually checked by the code. Furthermore, these codes generally do not raise any errors to indicate that the output that they produce is meaningless, nor do they generally confirm the level of accuracy in the answer that they produce. Traditional methods do involve careful analysis of how the values in the model are represented in the computation. But this analysis is generally done at the meta-level and not in the code. In essence, this approach allows the programmer to bake into the implementation ad hoc assumptions about how they expect the code to be used. We believe that it is these restrictions that result in significant loss in usability and reusability of numerical codes developed using traditional methods.

A large part of the analysis that must be carried out, and of the problems that arise when trying to use such codes, relate directly to the question of how real numbers are represented. Today, the vast majority of numerical codes are based on floating-point arithmetic (FPA). Again, floating-point technology has been very successful in that it enabled the development of numerous highly useful numerical codes that have enabled many CPS innovations. However, it is reasonable to consider alternatives. In particular, while FPA is well suited for hardware implementation, it remains in essence a static data structure that can represent only a finite set of values. So, rounding must be done with almost every arithmetic operation, which can introduce enough error that it can be extremely difficult to evaluate simple polynomial expressions (An example in [6]).

The most noteworthy feature of floating-point numbers is that they do not explicitly tell us how many digits (or bits) of the result are truly valid or *representative* of the real answer that such a computation should produce, if we were computing with some idealized form of real numbers. When we consider this feature together with the fact that numerical computations usually involve an extremely large number of operations, putting aside the problem that this feature creates for users of such codes, it is really impressive that any large numeric codes can be built correctly. Clearly correctness can only be achieved with significant meta-level reasoning. However, it remains an important question to determine how we can express the precise conditions needed to guarantee that a particular result of numeric computation is truly valid up to a given number of significant digits. If this is achieved, it may be possible to consider more ambitious questions, such as how to formally prove these theorems, or how to generate code guaranteed to be correct. In contrast to the overarching goal of mapping models to simulation codes, the last question is focused specifically on the issue of producing code that implements a real arithmetic computation using floating points correctly.

Stepping back from the questions of understanding traditional numerical techniques, it is important to note that, from a software engineering point of view, floating-point numbers are a somewhat curious choice for implementing real numbers. Specifically, they are a completely static data structure being used to represent values that, in general, may need an unbounded number of digits to represents. Interestingly, much care is needed if we wish to address this problem. For example, seemingly plausible alternatives such as variable precision numbers may simultaneously achieve less than what we might expect, and also bring more complexity to the problem than we started off with. Before we consider promising alternatives to floating-point numbers, it will be useful to recall some basic mathematical properties of numbers.

3. Basic Properties of Real Numbers

Real numbers are a concept so widely used in science and engineering to model both abstract and concrete systems that it is hard to imagine the world without them. Examples of real numbers include: the distance between two points in two- or three-dimensional space, the ratio between features of simple geometric forms such as the circle, and the integral of $1/x$ between two positive values of x. Much of the analytical component of the engineering and science disciplines rely heavily on this concept.

Real numbers derive both their power and their troublesomeness from their ability to transcend simpler forms of numbers, such as natural numbers, integers, and rational (fractional) numbers. For example, none of these sets are big enough to include numbers like π or e. It is therefore remarkable that scientific computing has been able to achieve so much while using a finite representation for real numbers, namely the floating-point representation.

Keeping in mind some basic mathematical theoretic properties of real numbers can help us navigate the complex space of alternative representations for real numbers. A basic example of these types of properties is cardinality, or the size of a set (See for example [1]):

- The set of different values that a floating-point number can take is finite,
- The set of rational numbers is countably infinite, and
- The set of real numbers is uncountably infinite.

It may seem that because rational numbers closer in cardinality to real numbers than floating-point numbers ("at least rational numbers are infinite") that they could make a more natural approximation of real numbers. But closer inspection suggests that they make it too easy to introduce unnecessary ad hoc decisions when trying to devise representations of real numbers. Indeed, rational numbers can be easily represented exactly on a computer through the use of dynamic data-structures. The basic arithmetic operators of addition, multiplication, and division for rational numbers can all be computed exactly on a computer. However, rational numbers are still insufficient for representing real numbers (a fact long known in mathematics [2]). When programming, this mathematical fact is reflected by the absence of an obvious way to compute with rational numbers in place of real numbers. This difficulty arises when we try to carefully explain why we cannot extend our set of operators to trigonometric functions, if we limit ourselves to rational numbers as a representation. Mathematically, the problem can be seen as the absence of a best rational number approximation for the result of the trigonometric function. It is tempting to take a pragmatic approach and make an ad hoc choice and choose *some* rational number to return when the result is not really rational, but then we begin to fall in the same traps that make it easy to abuse FPA to produce completely incorrect results. From the software engineering point of view, such choices are baking magic numbers into our code, which will eventually make it brittle and hard to maintain. Thus, rational numbers do not seem very well suited as a direct representation for real numbers, without building significant additional machinery around them. What is interesting here is that the consideration of the cardinality of the sets seems to provide the clearest indication of this mismatch, which is then echoed by a need for making ad hoc choices, when we try to build implementations. It is useful here also to be aware of the cardinality of various types of irrational real numbers, such as algebraic and transcendental numbers.

4. Interval Arithmetic

Interval arithmetic [5] is a powerful method for addressing one of the most basic problems in working with plain floating-point numbers: We can get information about the actual precision of the answer. Interval arithmetic uses two floating-point numbers to bound the exact answer of a real-valued computation from above and from below. Thus, interval arithmetic provides us with an immediate warning if rounding error has grown too large, rendering the result useless. Qualitatively, this additional information about precision is a huge improvement over allowing results to be silently corrupted by rounding. With this kind of information, the programmer or user can redo the full computation with a higher precision, and hope that this produces an answer with an acceptable level of precision.

Interval arithmetic is not a plug-and-play replacement for floating-point arithmetic. In fact, conceptually, interval arithmetic provides us with great concrete examples of why the floating-point way of doing business may in fact not be the way we will ultimately want to compute with real numbers. For example, the elementary operation of comparison on float-point numbers will have to behave differently when we move to intervals. How do we answer the question of whether the interval [1, 2] is less than [1.5, 2.5]? The answer cannot be yes or no, but rather, that the two are incomparable. As a result, it may not always be possible to expect that numerical algorithms can work without modification, using intervals rather than arithmetic.

It is interesting to note that a kind of abstract interpretation is almost built into interval arithmetic. It is not clear that this view has been fully developed (exceptions include [4] and [3]). Interval arithmetic gives rise to a beautiful theory that can teach us a lot about how we can compute effectively with real numbers.

A basic result of interval analysis is that performing a computation (that consists of the basic arithmetic operators) with more information about its inputs will always lead to a result that has no less information. Denotational semantics experts and domain-theorists will recognize this property as a notion of monotonicity. This property provides an elegant way to characterize well-behaved operators that we may or may not want to introduce into the language being interpreted using the primitives of interval arithmetic. An elegant observation from interval analysis is that monotonicity can occasionally provide a nice method for producing an answer with higher-precision without necessarily increasing the precision of the intermediate results. For example, because of monotonicity, we can always split

the input interval into two overlapping parts, compute two results for the two parts, and merge them together. This can often produce an improved result, especially in cases where the computation suffers from what is often referred to as the dependence problem. Evaluating an expression such as $x - x$ with x equal to [1,3] does not produce [0,0] but rather [-2,2]. Operators such as addition have no way of knowing that there is a special relation between their two inputs. Splitting the input into smaller parts, however, helps us get closer to the most precise answer. For example, if we compute the expression with x equal to [1,2] we get [-1,1], and with x equal to [2,3] we get [-1,1], which is clearly more precise than [-2,2].

An unexpected feature several recent treatments of interval arithmetic is that they often resort to opening up the interval, computing with both endpoints, and putting them back in again. This is an example of breaking abstraction boundaries, and can easily lead to breaking the monotonicity property mentioned earlier. We may have encountered a related problem, which is finding a simple definitions of transcendental values and trigonometric functions that do not involve separately computing an expected value and an error term and then adding them together (that is, without breaking the interval abstraction). It also seems that algorithms such as an interval version of Euler's forward method for integration (which is needed for doing integration in the context of solving an initial value problem, for example) are things that the authors have been told do exist, but do not quite know how they work, or whether or not they are defined without breaking the interval abstraction.

While it may be counter-intuitive, interval arithmetic implementations usually do not use rational bounds. Instead, they typically use floating point numbers. The intrinsic nature of this observation can be illustrated by considering the computation $sin([0.9, 1.1])$. If we want the result to be a pair of rational numbers, then the only "right answer" would be the pair of rationals that are closest to the exact answer for $sin(0.9)$ and $sin(1.1)$ from the outside. But there are not two such best approximations of real numbers in general, and therefore, there is no ideal rational candidate. This observation is important for realizing that floating-point bounds are not just an implementation convenience for interval arithmetic, but rather a necessity. Because floating-point numbers have maximal degree of precision, the result of the above computation is well-defined, because there is always a best floating-point approximation to any real number.

We conclude this section with three questions. The first question is whether demand-driven iteration or incremental evaluation is inherent to the way we use interval arithmetic. For example, the most basic (albeit not the only) method for improving a result that we attain with interval arithmetic is to repeat the computation with a higher-precision floating-point representation for the bounds. What does this really tell us about the idea of interval arithmetic? It could mean that doing interval arithmetic forward is inherently iterative. Would it be useful to do it backwards, that is, in a demand-driven way?

The second question is what would constitute a well engineered and conceptually clear way to build an interval arithmetic library? Real analysis is usually not computationally effective; constructive analysis is, in a sense, effective, but not aimed at computing efficiently. How can we build up such a library in a way that would allow us to explain clearly and easily to students how they are expected to program with interval arithmetic? What is the right way to approach the problem of re-computing with higher precision? It is reasonable to expect that essentially the same question will also need to be answered for more sophisticated representations of real numbers.

The third question is whether there are high-level formulations for the computational problems and solution techniques that arise in this domain. In particular, it seems that important techniques such as interval arithmetic are typically constructed in ways that isolate some underlying floating-point infrastructure. While this appears perfectly sensible from the point of view of efficient implementation, it means that interval arithmetic is typically not built "from the ground up". Instead, it depends critically on the idea that floating-point arithmetic is the most efficient approach to implementing stream-based computation. It seems plausible that this could be the case in current architectures, but not for future ones. In particular, it is less obvious why this approach should be the most appropriate for other emerging architectures such as GPU, FPGAs, many-core systems, or for future microprocessor designs.

Acknowledgement

Michal Konecny and Alexandre Chapoutot provided valuable comments on drafts of this paper.

References

[1] Richard Beals. *Analysis: An Introduction*. Cambridge University Press, 2004.

[2] Georg Cantor. Über eine eigenschaft des inbegriffes aller reellen algebraischen zahlen. *Journal für die Reine und Angewandte Mathematik*, 1874.

[3] Alexandre Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. In *Static Analysis Symposium*, 2010.

[4] Eric Goubault and Sylvie Putot. Under-approximations of computations in real numbers based on generalized affine arithmetic. In *Static Analysis Symposium*, 2007.

[5] Moore, Kearfott, and Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[6] Warwick Tucker. *Validated Numerics: A Short Introduction to Rigorous Computations*. Princton University Press, 2011.

[7] Angela Yun Zhu, Edwin Westbrook, Jun Inoue, Alexandre Chapoutot, Cherif Salama, Marisa Peralta, Travis Martin, Walid Taha, Marcia O'Malley, Robert Cartwright, Aaron Ames, and Raktim Bhattacharya. Mathematical equations as executable models of mechanical systems. In *International Conference on Cyber-Physical Systems*, 2010.