The Trouble with Real Numbers (Invited Paper)

Walid Taha and Robert Cartwright

Walid.Taha@hh.se and Corky.Cartwright@gmail.com

Abstract: Comprehensive analytical modeling and simulation of cyber-physical systems is an integral part of the process that brings to life novel designs and products. 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 outside programming languages, all these observations and questions pertain to how we need to reason and compute with real numbers.

1 Introduction

It is widely anticipated that much of tomorrow's innovations will be in the form of cyber-physical systems, that is, systems that include computating, 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 communicate about, reflect on, and reason about designs, modeling is an integral part of conceiving and developing new products in such domains. Because many important problems defined in terms of mathematical models do not have closed solutions, simulation also becomes an integral part of the same process. Unfortunately, the formidable time and effort needed to convert analytical models to running simulation code can impede or even derail the innovation 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 – 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 [Zhu et al 2010]. 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 what can be addressed with a handful of research papers, and success in demonstrating this thesis can be of far reaching consequences that maybe be of interest to other researchers. At the same time, entering the domain of cyber physical systems can be challenging, primarily because of the vast diversity of technical disciplines that touch on this domain. This diversity of sources is an

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

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 outside programming languages, all of these observations and questions pertain to how we need to reason and compute with real numbers.

Contributions and Organization: While it can be intuitively obvious from the outset that modeling systems that involve real-valued and time-varying quantities is somehow at odds with traditional approaches to computing, it is less 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:

- Analytical modeling and differential equations,
- Traditional numerical methods and floating-point arithmetic,
- Basic properties of real numbers,
- Interval arithmetic,
- Exact real arithmetic,
- A closer look at functions and initial value problems, and
- The educational challenge.

Our hope is that this review can serve as a quick introduction for language researchers into aspects of numerical computing that could play an important role in the future of modeling and simulation of cyber-physical systems.

2 Analytical Modeling and Differential Equations

The first real challenge in understanding how engineers and scientists need to reason and compute with real numbers is that it requires understanding the type of physical modeling problems that give rise to this need in the first place. This challenge actually consists of two parts, one which is domain-specific and pertains to understanding how mathematics can be used to model any given physical system, and the second is concerned with understanding the nature of the mathematical problems that arise from modeling. To enable engineers to design novel cyber-physical systems, we need to train them in modeling. Fortunately, excellent courses exist that introduce students to this art (see for example [Cellier and Greifeneder 1991]). Understanding the nature of the mathematical problems that arise from different types of models — at least from the programming languages point of view — is a bit more challenging.

The analytical models used for studying physical systems are remarkably diverse. For example, even in the absence of a time dimension (in which case the systems are called static), we have two large classes: linear systems and non-linear systems. Linear systems problems are solved by techniques of linear algebra, which require a fair degree of technical sophistication, but non-linear systems are a much more advanced topic. But even in the context of static systems we notice the familiar yet highly effective mathematical pattern of reducing hard problems to problems that we know how to solve. In this case, a non-linear system can often be handled through linearization, a process that essentially consists of computing the symbolic derivative of the system around a certain point (called working point), and pretending that the system is "almost linear" around this working point. Two ingredients of this technique will also recur frequently in the treatment of (dynamic) problems that have a time-dimension: differentiation, and approximation.

Dynamic systems generally give rise to problems where the solution is a function rather than a simple numeric value. A simple example is the trajectory of a flying ball, which is a function from time to a three (or, if we want to model spin, six) dimensional point in space. In principle, all one needs to model dynamic behaviors is to view variables as ranging over functions of time. In practice, however, it is most often the case that many aspects of such models are captured as rates of change, or derivatives, rather than direct constraints on values. For example, in the case of the flying ball, the main constraint (in a Newtonian model of a point mass flying in a vacuum) is the effect of gravity, which is a constraint on the second derivative of the position of the ball. Thus, models of dynamic systems are often differential equations. However, the differential operator is a tremendously express language feature. For certain domains, it is also useful to use the dual operator, integration, and in some cases there is no natural way to "differentiate both side of the equation" to get something that still captures the original problem but makes use only of differentiation. Such *integro-differential equations* arise in many different domains, but they are in some sense the most expressive class of equations, and we can explain and address many different problems in cyber-physical systems before we need to consider a class of equations that is big enough to include this class.

It may seem natural from the programming-languages point of view to define a simple context free grammar formed of arithmetic operators and differential and integral operators and study their semantics. Unfortunately, integro-differential equations are a sufficiently complex subject that it could render such an effort futile. After several rounds of asking domain experts about the possible existence of any universal numerical methods that could approach solving problems in this class, we gave up. We then repeatedly convinced ourselves to settle for smaller problems, posed the same question, and gave up; until we finally reached a class small enough to be tractable, which is, roughly, Ordinary Differential Equations (ODEs), and in particular the non-linear variety. More broadly, what we learned at this point is that, at a coarse granularity, differential equations can be classified in order of decreasing generality as follows:

- Integro-differential equations
- Partial differential equations
- Differential algebraic equations (generally non-linear)

- Ordinary differential equations (linear and non-linear)

This classification is not strict in the sense that the constraints on each class are often orthogonal. Furthermore, it is not always obvious (and maybe not always possible) to make sure that an equation that appears to be in one class is not a member of a lower class. We choose to focus on non-linear ODEs as a starting point rather than linear ones because linear ones have closed form solutions. This is one of many choices that may deserve revisiting, but the rationale for making this choice is that exploring the semantics of a language that can model dynamic systems in general will require understanding what numerical methods achieve. In particular, we were initially naive about what could be achieved with linear systems and about the existence of closed-form solutions for differential equations. It is possible that many electrical engineers may fall in this trap as we did, because much of the courses in undergraduate curricula make extensive use of linear ODEs in modeling circuits. We were therefore surprised that the model for the dynamics of a mechanical system as simple as a two-dimensional pendulum is non-linear. In fact, mechanical systems, which arguably constitute an essential element of cyber-physical systems, are generally non-linear when describing systems in more than one dimension. Thus, dealing with non-linear ODEs as a bare minimum seems necessary.

Glancing upwards from non-linear ODEs is not unreasonable, in particular because occasionally some equations from higher classes can be relatively easily mapped down to this class. For example, we discovered that a slightly modified partial evaluator (consisting of binding-time analysis followed by specialization and symbolic differentiation) can be used to convert some problems that syntactically appear to be partial differential equations — because they use partial derivatives — to equations that do not use partial derivatives at all. This conversion turns out to be highly convenient for expressing an important equation for rigid-body dynamics, namely, the Euler-Lagrange equation. Glancing upwards again from non-linear ODEs we find the DAEs, which are basically the same non-linear ODEs, except that they do not come in the convenient form of $X' = E[X]$, where X is a vector, X' is the derivative of that vector, and $E[X]$ is an expression with a hole. The particular form of ODEs makes a relative-straight forward attach using numerical integration possible, as integrating both sides leaves us with simply X on the left hand-side, which is very helpful when the variable X is exactly what we are trying to solve for. Solving for X is precisely what is needed when we solve what is called an initial value problem (IVP). Simulating what happens when we actuate a rigid body system is an IVP.

Our modified partial evaluator generated DAEs from the Euler-Lagrange equation. As it turns out, for the problems that we used for testing the system, an ad hoc solver was able to convert all the generated DAEs into the ODE form, which could be simulated using integration. In general, however, highly sophisticated methods are needed to convert general DAEs into ODEs, and extensive use of these technologies is made in implementations of the Modelica modeling language (see for example [Broman 2010]). However, in general, it seems that these techniques generally produce ODEs, and so the issue of using DAEs rather than ODEs in models seems to be fairly orthogonal to all other concerns relating to numerical methods (such as floating point issues an integration in the context of IVP problems).

A big challenge in solving numerical problems is whether it is possible to give the user minimal expectations for what kinds of problems a certain numerical library can reasonably solve. Right now the state of the art seems to be either 1) to use numerical methods which are susceptible to all sorts of accuracy errors, 2) to use symbolic methods, in which case we should expect that for most problems no solution will exist, and worse, that for most systems the solution will be very expensive to compute. There is a desperate need for a semantics of real-valued computations that would allow much more moderate expectations to be set and met.

Before moving to issues specific to numerical methods and floating-point numbers, three points seem noteworthy. The first is that the problem of simulating the behavior of mechanical systems seems to be primarily an IVP problem. When a system of equation has been cast (possibly after some transformations) as an ODE, this type of problem becomes primarily a matter of performing integration. There are other problems where the initialization constraints are not provided about one starting point from which we can sweep forward in one direction, but rather, as a collection of constraints that somehow must simultaneously be satisfied by the solution, even though this collection does not fall on one point in time and space. These problems, known as boundary-value problems (BVPs), are different from IVPs. And although they can in some case be recast as IVPs or optimization problems on IVPs, we have so far considered them to be beyond our initial investigation, and can only hope that the techniques we develop for IVPs maybe also used for BVPs.

The second point is that differential equation textbooks tend to classify problems by whether they can be in their entirety cast in a particular form, which is generally a isolated, rather concisely described equation with some parameters that can be instantiated differently for different problems. Then, for each type of equation there is a different solution technique. Familiar names such as Maxwell's equations, the wave equation, the Navier-Stokes equations, and the like, are examples of such families. We have not yet encountered a systematic way of dealing with equations that are composites of different types by isolating the distinct subparts.

The third and final point is whether there are principles for effective modeling and simulation, and whether we can articulate them sufficiently clearly so that we can teach them effectively to students? Today we find that simulation codes sometimes demand the full power of super computers that effectively need a small power station to operate, a reasonable question to ask is whether the right phenomena is being modeled in the first place. Answering this question is hampered by the fact that there is a distinction between modeling and codes that run simulations to solve problems relating to these models. When it takes several man months to manually go from the mathematical models to the actual implementations, parties on both sides can easily view the two as being completely distinct entities. It becomes easy to argue that the models are just about capturing the essence of the phenomena and have nothing to do with simulation concerns, and that the codes are just about "the computation" and not the physical phenomena. If the gap between models and simulations is replaced by an automatic mapping, it would disappear, making it much more feasible to experiment with radically different approaches to modeling while keeping an eye on the computation cost of different models. Of course, this does not automatically solve the problem of articulating and explaining clearly to students how cost-efficient models can be constructed, but it is necessary to offer a starting point for the investigation of this possibility.

3 Traditional Numerical Methods and Floating Point Arithmetic (FPA)

Traditional numerical methods and simulation technologies have carried us a long way, allowing us to build airplanes, space ships, and many other advances as well as mundane innovations. However, part of the difficulty in going from analytical models to simulation codes is due to the nature of the codes that 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 who's 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 components 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 golden 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 usually not 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 fraction of the analysis that must be carried out and 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 successful and highly useful numerical codes that have enabled numerous real-world innovations. However, it is reasonable to consider alternatives. In particular, while FPA is highly well suited for hardware implementation, it remains in essence a static data structure that can represent only a finite set of values. As a result, rounding must be done with almost every arithmetic operation. Rounding can introduce enough error that it can be extremely difficult to evaluate simple polynomial expressions (See for example [Tucker 2011]).

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 to a given number of significant digits. If this is achieved, it maybe possible to consider more ambitious questions, such as formally proving these theorems, or generating 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. In particular, 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 variable precision numbers may simultaneously achieve less than what we might expect at first, and they may 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.

4 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 hard to imagine the world without it. For example, the distance between two points on two-dimensional space is a real number. Similarly, the ratio between features of simple geometric forms such as the circle is a real number. The integral of $1/x$ is the natural logarithmic function. The non-zero function that is equal to its derivative is the natural exponential function. Thus, much of the analytical component of 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 [Beals 2004]):

- 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 is uncountably infinite.

It may seem that because rational numbers closer in cardinality than floating-point numbers to that of real numbers (“At least rational numbers are infinite”) that they could make a more natural approximation of real-numbers. But closer inspection suggests that they accentuate the fact that it is 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 [Cantor 1874]). 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 too 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-in 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, and 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 when we try to build implementations by a need for making ad hoc choices. It is useful here also to be aware of the cardinality of various types of irrational real numbers, such as algebraic and transcendental numbers.

5 Interval Arithmetic

Interval arithmetic [Moore et al 2009] is a powerful method for addressing one of the most basic problems with 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 to make the result of any use. 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 float-point way of doing business may in fact not be the way we will ultimately want to compute with real numbers. For example, the rather simple 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 of such a chance, 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 [Goubault and Putot 2007] and [Chapoutot SAS 2010]). Interval arithmetic gives rise to a beautiful theory that can teach us a lot about how 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 immediately recognize this property as a notion of continuity that 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 continuity 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 continuity, 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].

It is interesting that the phenomena known as the dependence *problem*, from the semantics point of view, is more of a feature than a bug. In particular, it is well matched to the idea of compositionality of interpretations in denotational semantics, which is often an advantage both from the point of view of defining and implementing a language.

A curious fact about several recent treatments of interval arithmetic is that they seem to often resort to opening up the interval, computing with both sides, and putting them back in again. This is an example of breaking abstraction boundaries, and can easily lead to breaking the continuity property mentioned earlier. We may have encountered a related problem, which is not finding simple definitions of transcendental values and trigonometric functions that do not involved separately computing an expected value and an error term and then adding them together. It also seems that algorithms such as 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 to exist, but do not quite know how they work, or whether or not they are defined extensionally.

It is instructive to note that interval arithmetic still uses floating point numbers, and in particular, it does not use rational bounds. This is a rather simple fact that is sometimes easily overlooked, but is significant both because it is unchangeable and informative about the intrinsic nature of interval arithmetic. It is also a fact that is easily omitted either because the term "interval arithmetic" does not explicitly contain the term "floating" point or because it is generally described as a way to dealing with the limitations of floating points. The intrinsic nature of this observation can be illustrated by considering computing $sin([0.9, 1.1])$. 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 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 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?

The second question is what would constitute a well engineered and conceptually clear way to build an interval arithmetic library. In particular, real-analysis is usually not effective. Constructive analysis is in a sense more effective, but not aimed at computing with real numbers in the same way. 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 infra-structure. 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 on current architectures. 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.

6 Exact Real Arithmetic

From a software engineering point of view, a curious fact about interval arithmetic is that it does not have a built-in mechanism for deciding what to do when the precision of the computed answer is not satisfactory. As mentioned earlier, one way of dealing with this problem is to repeat the entire computation with higher precision everywhere. While this approach could work for some applications, intuitively it seems unlikely that same precision is needed everywhere in a large computation. It is also plausible that there will be certain computations where some parts are needed to a much higher precision than others. A natural question to ask is therefore weather there is a systematic approach to performing computations to a higher-precision on an as-needed basis. Exact Real Arithmetic (ERA) is an approach to representing real numbers that provides this type of automation [Boehm 1986].

The theory of ERA reveals a structure that is at least as beautiful and as insightful as that of interval arithmetic. Mathematically, we can view the basic idea behind ERA is representing real numbers by a potentially infinite sequence of shrinking intervals. One way to define a computational representation of this idea is simply to represent a number by a potentially infinite sequence (or “stream”, for short) of digits. Streams provide a natural implementation of a demand-driven way to implement the iteration that we perform over interval arithmetic computation until we reach a satisfactory answer. The stream of digits approach is intuitively appealing because it is similar to the way we learn about decimal numbers from an early age. Interestingly, it does reveal some of the more peculiar issues that can arise when computing with real numbers. For example, in general, real numbers will not have unique representations [Brouwer 1921]. This discovery is intuitively illustrated by considering the fact that the two streams 1.000... and 0.999... represent the same number when the last digit repeats forever. Possibly more interesting is the fact that operations as simple as addition become undecidable if we work with the traditional (non-overlapping) interpretation of digits. We can intuitively see this as a problem of carry propagation. Deeper explanations in terms of the expressivity of the representation could offer more insight. For example, with non-overlapping interpretations of digits reveal that the set of intervals expressible without overlap is tree-structured and that the boundary between any two adjacent intervals of the same size is a point that cannot be contained in an interval of the same size. Fortunately, this problem of undecidability of addition with non-overlapping digit representations is easily solved either by relaxing the interpretation of the digits or by adding negative digits to achieve essentially the same effect.

Other ERA representations exists besides streams of digits, including representing a real number (computation) as a function from a tolerance to an interval that has size that is no greater than that tolerance. In fact, it has long been viewed that this type of representation can be more efficient than the stream of digit representation [Boehm 1986]. Precisely why this appears to be the case is debatable. However, we expect that the renewed interested in this area will enable more extensive investigation of the large space of possible ERA representations, including the impressively efficient iRRAM model [Müller 2001].

An interesting but somewhat technical feature of ERA representations is that they are often based on continued fractions, and this is typically viewed as important for efficiency. From the software engineering point of view, they are rather mysterious contraptions, as they are typically implemented with the use of a large number of "magic numbers" that are actually the result of a wide range of important results discovered by different mathematicians in the past. This feels a bit like bringing in high-voltage cables to power up a pilot lamp. A more serious concern is that the introduction of such special constants can get in the way of understanding the real computational foundations underlying computing with real numbers. It is therefore a compelling question to determine precisely whether or not continued fractions are essential for the efficient implementation of ERA, or if there lower-level primitives that are more natural.

Finally, given the performance demands of much of numerical computation, possibly the most important question to consider is the presence of effective techniques for strictness analysis that can reduce the need for unnecessary iterations. Another — possibly equally important — question is the feasibility of defining primitive and composite arithmetic operators in such a way that they can naturally extract the most amount of information about the result using the least amount of work by their sub-computations. We are not currently aware of any works that have addressed this issue.

7 A Closer Look at Functions and Initial Value Problems

As soon as we consider the performance of an abstract data type, understanding the context in which it is used becomes imperative. As noted earlier, for many applications simulations will consist of solving initial value problems on differential equations. Mathematically, the solution to such equations is a function from real numbers to real numbers.

Before looking more closely at the nature of continuous functions, it is useful to note that in the context of cyber-physical systems solutions are generally not continuous. For example, if we extend the basic model of a bouncing ball that we mentioned earlier to support the possibility of instantaneous bouncing when the ball hits the ground, we suddenly have a system where the first derivative (speed) is not continuous, as it undergoes an instantaneous change in direction when it hits the ground. Such *hybrid systems* which can exhibit both continuous and discrete behaivor can introduce a wide range of additional complexities to the solution techniques. For example, a bouncing ball can exhibit *Zeno behavior*, which is a real problem for simulation. In particular, if we model a system where the impact is imperfect and energy is lost, then the ball will stop bouncing after a finite amount of time but after an infinite number of changes in direction of its speed. Realizing the changes in direction would lead any simulator that does not have special support for this kind of behavior to diverge. Engineers often solve this problem by modifying the problem to make the ball stop bouncing completely after a certain threshold. From a software engineering point of view, it is clear that we have just introduced a magic number that makes this otherwise reasonable model much less scalable than it would have been in its pure form.

It is somewhat surprising that a system as simple as a bouncing ball can reveal that discrete changes can lead to such complexities. The bouncing ball is not wholly artificial, as it is infact representative of almost any type of impact that occurs in the context of a rigid-body model of a robotic system. Thus, even in the context of a domain that might seem elementary at first we are forced to consider whether it is really reasonable to assume that there is such a thing as a discrete event in the first place. Discrete events are convenient computationally, because they are easily and naturally handled by a computer. But then it seems that we run into a myriad of other problems in virtually all aspects of computation to deal with such issues. An example is the problem of structural dynamism, which generates a lot of complexities for current tools. Other problems include the treatment of impacts in mechanical system, and the need to introduce notions such as super-dense time. Is it better to start off from a foundation where everything is continuous but things can happen at higher speeds than others, or is it better to start off with a hybrid foundation?

Hybrid behaviors are not completely orthogonal to how we use real numbers. In particular, a discrete event such as bouncing requires solving what is sometimes called the *zero-crossing* problem, but which seems indistinguishable from the possibly more familiar problem of *root-finding* from high-school mathematics. The basic idea is that we need to determine (up to some precision) when the bouncing should occur. This is typically an iterative process which can be achieved using bisection or Newton's method. With both interval arithmetic or ERA, there is a challenge in translating the uncertainty in the time dimension about when the event occurred to the precise uncertainty in the speed of the ball after that time.

Turning to purely continuous segments, possibly the most striking fact is possibly the extensive use that is made of logical quantifiers in the definition of the most basic properties of functions. Looking at things from this point of view one sees that there is even a pervasive, repeating pattern of quantification (and quantification alternation) that earns the name of a *limit*. While for many this term may conjure images of magic tricks with numbers, its significance in the context of computing with real numbers is that it gives rise to the question of whether we are really computing the right kinds of things when we perform simulation. Quantification is used in the definition of key notions about functions as follows:

- A limit of a sequence or a function involves quantifiers (the alternation is for all, there exists, and then for all again)
- Continuity at a point simply says that it is equal to the limit approaching that point
- The derivative is a limit (of a function that is in general not continuous)
- Differentiability is simply whether the derivative exists
- The definition of integration is a limit as the mesh (step size) goes to zero
- An interesting exception is the property of being Lipschitz continuous (bounded gradient), which does not rely on the notion of a limit (but uses quantifiers, of course), and plays an important role in characterizing the conditions under which solutions to some differential equations exist and are unique

Writing numerical algorithms involves a lot of reasoning about these properties. The question that intrigues us here is whether we should really be computing with logical statements directly. A useful step in investigating this possibility could be to determine the feasibility of having a compositional interpretation of equations or even expressions over continuous functions of time that can satisfy the needs of these quantifiers in a local, efficient manner.

8 The Educational Challenge

Computer science has long focused on systems built up of discrete components. Examples of such discrete systems include digital circuits, automata, Turing machines, the lambda calculus, and various forms of concurrent calculi. But with the increasing interest in cyber-physical systems (CPS), we can no longer isolate ourselves from the worldly concerns of the physical environment. In particular, keeping up with the demands of the rapidly growing embedded and real-time systems labor market dictates that we find effective ways to teach our students the principles of our science in a context where models of computation are tightly coupled with models of physical systems. This is a formidable challenge, because so far key concepts of physical modeling, such as real-number quantities and time-varying functions have generally been viewed as being outside the science. Because we rarely hold real-numbers to the same standards that we do other core concepts to, they currently stand more as an appendage than an integral part of the science. This problem is far from being purely pedagogic. Even the collective knowledge of the research community (in multiple fields, combined) remains sparse on systematic treatments of real numbers that could stand up to the standards to which we hold other types of software. If we do not fix this problem, future graduates will have an Achilles' heal: All the principles they learn for developing correct, reliable, and cost-effective systems will simply fail when they work in physical contexts. We believe that it is essential to rise up to the challenge of rebuilding

Computer Science, Computer Engineering, Electrical Engineering, Mechanical Engineering and several other engineering curricula to take into account CPS. This is a significant challenge simply because there is not enough time in the day for engineering students to learn everything there is to know about engineering in all disciplines. Instead, we believe that it is time to reconsider what are the universal principles underlying these disciplines, and to make sure that tomorrow's graduates know them. As a starting point, we are developing a sequence of courses relating to the development of cyber-physical systems with a focus on robotics applications. However, succeeding in addressing this challenge will only happen as a result of a community-wide effort in search for these principles in the context of a sufficiently wide range of physical phenomena and application domains.

9 Epilogue

Software engineering in general, and in particular programming languages semantics, design, implementation, and engineering have a lot to contribute the study of cyber-physical systems. We hope that the observations and questions presented here encourage the reader to share this view.

References

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

[Boehm et al 1986] Boehm, Cartwright, Riggle and O'Donnell. Exact Real Arithmetic: A Case Study in Higher Order Programming, *Proceedings of the 1986 ACM conference on LISP and functional programming*, 1986.

[Broman 2010] Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. PhD thesis. Dept. of Computer and Information Science, Linköping University, Sweden, 2010.

[Brouwer 1921] Brouwer. Besitzt jede reelle Zahl eine Dezimalbruchentwicklung? *Math. Ann.*, 1921.

[Cellier and Greifeneder 1991] Cellier and Greifeneder. *Continuous System Modeling*, Springer, 1991.

[Chapoutot 2010] Chapoutot. Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables, In *Static Analysis Symposium*, 2010.

[Goubault and Putot 2007] Goubault and Putot. Under-Approximations of Computations in Real Numbers Based on Generalized Affine Arithmetic, In *Static Analysis Symposium*, 2007.

[Müller 2001] Müller. The iRRAM: Exact Arithmetic in C++, In *Computability and Complexity in Analysis*, LNCS, 2001.

[Moore et al 2009] Moore, Kearfott and Cloud. *Introduction to interval analysis*, SIAM, 2009

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

[Zhu et al 2010] Zhu, Westbrook, Inoue, Chapoutot, Salama, Peralta, Martin, Taha, O'Malley, Cartwright, Ames and Bhattacharya. Mathematical Equations as Executable Models of Mechanical Systems, In *the International Conference on Cyber-Physical Systems*, 2010.

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4