SimLab: Automatically Creating Physical Systems Simulators

Richard S. Palmer
Computer Science Department
Cornell University
Email: rick@cs.cornell.edu

James F. Cremer
Computer Science Department
University of Iowa
Email: cremer@cs.uiowa.edu

Abstract

SimLab, a software environment for creating simulators directly from computer-readable physics models, is based on the following concept: *creating physical systems simulators should be as simple as describing the underlying physics to a colleague.*

Rather than programming in a conventional programming language, a SimLab user expresses physics models (and thus simulators) directly in terms of the concepts, quantities, and equations familiar to a scientist or engineer.

The benefits of the SimLab approach include: 1) reducing the time and effort required to create simulators, 2) providing more understandable and reliable simulators, and 3) support for more sophisticated simulators, e.g., for multiple-domain problems, which have proved intractable in the past.

1 Introduction

Since their introduction almost fifty years ago, electronic computers have been used to simulate and analyze physical systems. ENIAC, widely regarded as the first digital electronic computer, was originally commissioned to generate ballistics tables[15], i.e., simulating the dynamics of a rigid body in a fluid (a projectile moving through the air) in a gravitational field (that of the earth). Since that beginning, the use of computers for simulating and analyzing physical phenomena has grown to include nearly every domain of engineering and science: fluid dynamics, rigid-body dynamics, visco-elasticity, electrical circuits, etc.

Even though many physical systems simulators are conceptually very similar, and could share algorithms, software components, and user interfaces, they do not. As an example, most simulators use vector operations. But rather than using a general purpose *VectorSpace* package, each implementer defines his or her own data structures and functions. This has some obvious disadvantages: software having a given functionality is repeatedly implemented, tested, and debugged; the result is a great loss in productivity. Also, such software must remain isolated: it does not work well with other programs because it has its own formats and conventions.

There are at least three reasons no shared software base has developed. First, simulator implementers have often worked in isolation, mainly because they have tended to be practitioners in their respective fields. Second, it is only recently that the software base capable of providing the required functionality has matured. Functionally, software components such as geometric modelers[3, 22, 17], ODE integrators[7], linear algebra packages[6], and symbolic algebra systems[20, 9, 21, 23] are well developed. However, these components are not constructed using a software architecture that supports closely coupled integration. Numerical packages such as Linpack are notable exceptions. Linpack provides linear algebra functionality at a variety of levels, using a variety of data structures. The third reason that no common software base has developed is that there is currently no unified *computational* model for physical objects and phenomena. While mathematical models of physics have been steadily advancing since before the time of Newton, computational models of physics have not been fully incorporated into simulation in any fundamental sense. For instance, there is currently no accepted way of specifying, for the purpose of computing, physical domains, their behavior, and interactions among them. In contrast, geometric modeling has various computational models of objects such as point sets(e.g., Boundary Representations (Brep) and Constructive Solid Geometry (CSG), etc.) and operations on these objects (e.g., boolean operations, Minkowski sums, and Euler operations). (For an introduction to geometric modeling and references see Hoffmann [8].) While much work remains in the process of unifying and expanding the computational theory of geometric modeling, the beginnings of a computational theory exist. This is not the case for physical systems simulation.

1.1 Constructing simulators by writing physics

Large physical systems simulators are currently specified at too low a level — they are specified by writing a computer program in a conventional programming language. As a result, creating a such a simulator can be a daunting task taking years to complete. This causes a shortage of physical systems simulation and analysis software. In particular, there is a significant time lag between development of hardware technology (such as new parallel architectures) and its use for physical systems simulation and analysis.

Existing simulation languages[12, 1, 14] are typically a thin veneer over FORTRAN: they are often, in fact, preprocessors for FORTRAN. In addition to providing a high-level/FORTRAN syntax, such systems typically provide subroutines for performing a variety of simulation operations, such as ODE integration. While these software systems are an important advance over unenhanced conventional programming languages, we envisage a more ambitious system: a simulator programming environment in which simulators are specified more or less directly as they are understood by a scientist or engineer. Our prototype system, SimLab, is a first step towards this goal. The input to SimLab looks much like the models of physics described in textbooks. Section 2 develops a SimLab model of particle dynamics by comparing the SimLab syntax with a typical textbook derivation.

Existing special purpose simulation tools such as AUTOSIM[11] and SD/FAST[19] generate simulation code for a particular physics model (rigid body dynamics in this case) using a particular formulation method (Kane's method). In contrast, the SimLab environment allows a *user* to define the physics model and formulation method. Thus, SimLab can be used to create special purpose simulators for a wide range of problems.

Besides the simulation languages described above, the *bond graph* formalism, developed by Paynter in the late 1950's[13], has engendered a number of simulation systems[16, 2]. Bond graphs are essentially a generalization of electrical circuits. Analogs to electrical components, such as inductors, resistors, and capacitors are defined for various physical domains. As an example, in the mechanical domain, the spring is analogous to the capacitor, while the dashpot (shock absorber) is analogous to the resistor. Physical interaction between components is modeled by "power bonds," which model energy transfer between the components, and guarantee conservation of energy.

Transducers may be used to relate phenomena in different physical domains. As an example, a bond graph model of a hydraulic cylinder might include a transducer that transforms the hydraulic pressure into mechanical pressure on the cylinder clevis.

Bond graph based simulation systems are closer in spirit to the current SimLab prototype than the simulation languages described above. Both SimLab and bond graph systems support the notion of primitive objects with associated behavior (primitives and *components*, respectively), and interactions between them (connections and *power bonds*). They differ mainly in that bond graph systems use a predefined class of equations and solution methods instead of allowing the user to define them. One might say that a given bond graph modeler corresponds to a single SimLab model and formulation. The system designer has chosen these; the user has no choice.

To see the value and implications of creating simulators directly from models of physics, suppose a scientist has spent the last two years constructing a simulator for a diesel engine. The resulting program is a combined rigid-body/fluid-dynamics/combustion simulator. It simulates the pistons, connecting rods, and crankshaft as rigid bodies, and uses PDE models of turbulent fluid flow and combustion.

The scientist has defined models of the interaction between these various models of physical phenomena. For example, the force that the expanding gasses exert on the piston due to combustion must be represented. Weeks have been spent defining and implementing data structures, choosing algorithms, and incorporating pieces of various numerical packages such as Linpack and ODEPACK. The simulator has been carefully designed, coded, tested, and debugged.

The result is a large body of intricate and carefully crafted simulation software. This is used to tune the design parameters of a new diesel engine, and proves to be of great value. Although the author is convinced of its correctness, it is difficult to convince others that the results may be relied on, because the simulator implementation itself is difficult for others to understand.

Depending on the implementer's level of effort and foresight, it may be possible to change a few parameters to modify the simulation to correspond to design changes such as larger pistons, stroke, or improved models of phenomena. It may even be possible to accommodate a change in the number of cylinders. The difficulty arises when the simulation must incorporate a heat flow analysis of the piston and block. Because this requirement did not exist in the original specification, it is very difficult to patch this additional phenomenon into the simulator: software must be modified and augmented in a multitude of places distributed throughout the program. In addition to requiring months of effort, the resulting program is less elegant, and because it is now more difficult to understand, the level of confidence in the results is reduced considerably.

Suppose, in contrast, that an implementer creates a simulator using the SimLab system. The initial stage is identical: one determines a mathematical model of the physical phenomena. In this case, however, the initial stage is much closer to the final stage: The input to SimLab is the mathematical model, together with high-level instructions for solving the resulting systems of equations. From these, the system creates a simulator as well as an editor (called the *scene generator*) for creating instances of the problem. Rather than requiring the user to define data structures and algorithms, combine various numerical packages, and write analysis and visualization routines, SimLab performs these steps automatically. This means that changing the model, for instance adding heat transfer to a simulation, requires only defining a *heat transfer* model, and defining the interactions between *heat transfer* and the various other phenomena. Again the simulator and related modules are created directly from the model. While the resulting simulator may be as intricate as a hand crafted one, the time, effort, and cost required to create it is reduced by orders of magnitude. In addition, the results are more credible, because the author has expressed the underlying mathematical model explicitly. The simulator has been automatically constructed from that model, which aids in convincingly defending the simulation results.

The process of creating a simulator is not the only aspect of simulation that is time consuming. In fact, the design process itself, i.e., creating the artifacts to be simulated, is the major purpose of most engineering and design departments. In the previous example, the design of the engine itself could easily require more effort than the design of the simulator that analyses it. The ability to quickly create and modify simulators is of limited value if it means that extant artifact designs must be discarded or laboriously re-created when a simulator is modified. Fortunately, the explicit representation of the physical model allows the simulator designer to specify translations directly in terms of the physical model itself.

1.1.1 A "simulation language" to *think* in

The goal in creating a computer language or environment is to provide the scientist with the means to express the problem in a manner as close as possible to the way he or she *thinks* about the problem. Specifying a simulator should be, as nearly as possible, like describing the solution method to a colleague. Beyond providing semantics corresponding to the user's current mode of understanding the phenomena, we believe that our environment will provide an enhanced *understanding* of the engineering problems themselves. By way of comparison, the `goto` statement encourages a low-level difficult-to-understand style of programming, while the `while` loop construct encourages a structured style, which is more easily verified, understood, and explained. In a analogous manner, we believe that the SimLab environment for specifying simulators will make explicit the meaning and assumptions typically hidden in simulation software. This in turn can provide engineers and scientists with an improved model for *understanding* the process of creating software for physical systems modeling and simulation.

Ultimately, we believe the benefits of the SimLab approach will be:

1. Development of a high-level means of expressing simulation problems.
2. Improved understanding of physical phenomena and interactions.
3. Quickly constructed, reliable, and understandable simulators.
4. Simulators that can be quickly and easily modified.
5. Retargeting simulators to take advantage of new machine architectures.

The remainder of the paper investigates the means for realizing the vision of high-level specification and automatic generation of simulators. Section 2 presents a language for expressing mathematical models for physical systems simulators. Section 3 details the current implementation of SimLab — the overall architecture of the system, the structure of the packages comprising the system's mathematical substrate, and so forth. Clearly, an enormous amount of research remains to be done before automatic generation of high-quality simulators for complex physical phenomena becomes practical. Section 4 identifies and discusses a number of these challenging problems. Finally, Appendix A presents examples of the use of

SimLab to create simulators for particle dynamics, rigid-body dynamics and electric circuits.

2 Representing physics in SimLab

In this section we describe the means for specifying physical systems simulators in SimLab. We introduce the concepts with a simple example: a model of electrostatic particle physics.

As we have stated, SimLab creates simulators directly from models of physics. In developing this example we, in addition to introducing the constructs of the model specification language, compare the SimLab formulation of a mathematical model with the development in a standard textbook[10]. The differences are not significant, and arise mainly in making certain assumptions explicit. For instance, SimLab models specify mathematical domains of quantities. Unlike human readers, SimLab does not infer the mathematical domains of quantities. Also, SimLab constructs data structures for the primitive objects in a model (such as the *particle* in this one). For this reason, the primitives and their quantities are explicitly defined. Thus, a model contains the necessary information for automatically constructing a simulator directly from the model, which is our goal.

2.1 Defining models of physics

In the modeling and simulation literature, "model" is generally used to describe a model of a particular instance. In contrast, a SimLab *model* is a computer-readable representation of the model of physics itself – a notion that has not until now been needed in the simulation literature, since models of physics have not been explicitly represented. Particular *instances* of a problem are referred to as *scenes*.

A SimLab model contains definitions of *primitives*, the basic entities in the model, together with *connections*, which are used to specify interactions between primitives. Primitives may have *quantities*, which represent the state of the primitive. *Constraints* may also be associated with primitives to constrain the values of their quantities. For example, the *particle* primitive in particle-model contains the constraints

$$momentum = mass * velocity$$

and

$$force = deriv(momentum, time).$$

Simple inheritance is provided: a primitive can be defined to be a "kind-of" some other primitive, as when a "ball-and-socket" hinge is defined to be a kind of "kinematic constraint" in the rigid-body dynamics model in Appendix A.2.

2.2 Operating on physics models

The *formulation* provides a global interpretation for the model: it specifies how to assemble a set of equations from the primitives, connections, constraints, and laws. Once the model and formulation have been defined, SimLab creates data structures (i.e., classes) and associated methods for creating and operating on the primitives, connections, etc. From these it constructs a simulator and a scene generator for creating instances of the problem. A user may then create an initial value problem (a scene), and invoke the simulator. The simulator uses the model definitions and formulation to construct a set of equations and solution methods that model that scene, performs the simulation, and displays the results of analysis.

2.3 Electrostatic particle mechanics

We begin by describing electrostatic particle mechanics as stated in textbooks. Newton formulated mechanics in terms of *particles*, together with the following three *laws*:

1. If there are no forces acting on a particle, its velocity is constant.

2. The momentum of a particle is defined to be the product of its mass and velocity. ($p = mv$)

3. If A exerts a force on B, then B exerts a force of equal magnitude and opposite direction on A. ($F_{AB} = -F_{BA}$)

In addition to these laws, we note that velocity is the time rate of change of position ($v = \frac{dx}{dt}$), and force is the time rate of change of momentum ($f = \frac{dp}{dt}$).

The mass of a particle is a scalar constant (for Newtonian, i.e., non–relativistic, mechanics) that defines the relationship between the momentum and the velocity.

In this model, the only interaction between particles is that of electrostatic force. The magnitude of the electrostatic force between particles p_1 and p_2 is defined by

$$F_{p_1 p_2} = \frac{E q_1 q_2}{r^2}, \tag{1}$$

where E is the electrostatic force constant, q_1 is the charge of p_1, q_2 is the charge of p_2, and r is the distance between p_1 and p_2.

2.4 The SimLab model

Although the current implementation of SimLab uses a LISP-like model syntax, for reasons of presentation we use a C-like model syntax in this paper.

In addition to formally expressing Newton's laws as described above, a SimLab model of electrostatic particle mechanics adds definitions of what a particle is, what quantities are associated with a particle, and the direction of the electrostatic force. The process of defining this model is carried out below:

The first step in defining a SimLab model is to name it:

```
DEFINE-MODEL Electrostatic-particles
```

2.4.1 The primitive: the *particle*

Next, we define the only primitive in this model, the particle. In general, the components of a primitive are its *quantities*, which characterize its state, and its *constraints*, which constrain the values of these quantities. For the case at hand we have the following definition:

```
PRIMITIVE particle
  QUANTITIES
    charge in R
    mass in R
    momentum in R^3
    force in R^3
    position in R^3
    velocity in R^3
  CONSTRAINTS
    momentum = mass * velocity
    force = deriv(momentum, time)
    velocity = deriv(position, time)
```

A *particle* in this model is defined to have charge, mass, momentum, force, velocity, and position. The particle's constraints enforce Newton's first and second laws, as well as the relationship between force and momentum, and that between velocity and position.

2.4.2 Electrostatic force: the *connection* between particles

We now proceed to define the model's *connections*, which define the modes of interaction between the primitives. In this case, the only interaction between particles is the electrostatic force between a pair of particles.

Thus, we define the *connection* for electrostatic force between two particles:

```
CONNECTION electrostatic-force
               (p1,p2: particle)
  QUANTITIES force in R^3
  CONSTRAINTS
    force =
     (E * charge(p1) * charge(p2)
      * (position(p2) - position(p1)))
     / DIST(position(p1), position(p2))^3
```

This connection specifies the electrostatic interaction between a pair of particles. It defines the force that p1 exerts on p2.

2.4.3 The *formulation* of the mathematical model

The final step in defining a model is to specify how the connections are to be applied to the primitives to assemble the system of equations that model a given instance. In the case of particle dynamics, each particle exerts an electrostatic force on every other. Therefore, the *formulation* for this model is as follows:

```
FORMULATION
  FOREACH p in particle-set(scene)
    force(p) =
       SUM c in electrostatic-force-set(p)
          IF (p == p1-of(c))
             force-of(c)
          ELSE
             - force-of(c)
```

The formulation specifies that the total electrostatic force on p is the sum of all forces in *electrostatic-force-set(p)* (the set of *electrostatic-force* constraints for which p is a primitive). The awkward *if-then-else* form is currently used because electrostatic force is represented as a directed force.

This completes the model for electrostatic particle mechanics. From this model, SimLab automatically constructs a simulator, as well as a scene generator for creating and editing instances of the problem.

2.5 The scene generator

Given a mathematical model, SimLab creates data structures, mathematical domains, and associated access functions for each of the primitives, quantities, constraints, and connections. These are used to create the scene generator and the simulator. In the present case, the scene generator allows one to define a set of particles, and to then specify the charge and mass of each, as well as initial conditions on the position and velocity. As an example, we used the scene generator to create the following scene, represented textually below:

```
time[0, 10]
PARTICLE /* at time = 0 */
        name particle-1
        charge .02
        mass 1.0
        position <0.0, 0.0, 0.0>
        velocity <1.0, 0.0, 0.0>

PARTICLE /* at time = 0 */
        name particle-2
        charge .03
        mass 1.0
        position <0.0, 0.0, 10.0>
        velocity <0.0, -10.0, 0.0>

FORALL UNORDERED-PAIRS
    (p1:particle, p2:particle) IN scene
       electrostatic-force(p1,p2)
```

This scene specifies that a system of two particles should be simulated in the time interval $[0, 10]$. The final two lines in the above scene specification declare that electrostatic interaction occurs between all distinct pairs of particles in the scene. Although the fact that electrostatic force occurs between all pairs of particles could be specified in the model, we have chosen in this case to specify the existence of electrostatic force on a *per*-scene basis, which allows creating scenes where such force does not occur between all pairs of particles.

2.6 The simulator

The particle-mechanics simulator, when given such a scene as input, constructs a system of ordinary differential equations that represents the behavior of the scene as defined by the model from which the simulator was created. For each primitive in the scene, it creates equations corresponding to the constraints applied to that primitive. In the case of the instance described in the previous section, the constraints associated with primitives result in the following set of equations:

```
force(p1) = deriv(momentum(p1), t)
velocity(p1) = deriv(position(p1), t)
momentum(p1) =  mass(p1) * velocity(p1)
force(p2) = deriv(momentum(p2), t)
velocity(p2) = deriv(position(p2), t)
momentum(p2) = mass(p2) * velocity(p2)
```

To this, equations resulting from the *formulation* are added.

```
force(p1) = E * charge(p1) * charge(p2)
  * NORMALIZE(position(p2) - position(p1))
  / DIST(position(p1), position(p2))^2)
force(p2) = E * charge(p2) * charge(p1)
  * NORMALIZE(position(p1) - position(p2))
  / DIST(position(p2), position(p1))^2)
```

Thus it is seen that a scene serves two purposes: defining an initial value problem (that is, the set of primitives and the relationships between them), as well as defining initial conditions for that problem. Given a scene, the automatically generated simulator assembles the set of equations, manipulates these equations into a "normal form" (either explicit form: $y' = f(y,t)$, or implicit form: $A(y,t)y' = f(y,t)$), and simulates the motion of the particles by integrating the resulting set of equations. These actions are described in the next section.

3 Implementation issues

This section describes the current SimLab prototype. We first outline the process it uses to create simulators. We then describe the software aspects of the implementation itself. Section 3.4 describes the symbolic algebra component, which is implemented using Weyl. We conclude by describing some of the implementation issues concerning the example applications.

The prototype implementation of SimLab is written using the Common Lisp Object System (CLOS)[18]: the standard object-oriented version of Common Lisp. SimLab generates simulators in conventional programming languages (currently Common Lisp, C, and FORTRAN). These simulators incorporate user interface and visualization tools using X-windows. Once SimLab has created a simulator and scene generator for a given model, the user interface tools are used to create *scenes*, display a scene's evolution, and graph the results of analyses.

3.1 The process of simulator generation

The current SimLab prototype can create simulators in two different ways. In the first method, called *model interpretation/scene compilation*, a model and scene are input to SimLab, which produces a standalone FORTRAN program that simulates the particular scene in accordance with the model. In the second process, called *model compilation*, the only input to SimLab is a model. The output is a program that takes scenes as input and simulates them using the compiled model.

There are important differences between the two processes, and advantages and disadvantages to both. In simulators resulting form the first process, model information is lost. At the time of simulator creation, the model information required for simulating the particular scene is extracted and used to create code for simulating that scene. The model information itself is not explicitly incorporated into the resulting simulator. The main disadvantage of this approach is that the simulators produced are useful only for simulations in which events do not occur, i.e. for simulations without discontinuities. [1]

Using the model interpretation/scene compilation method events must be handled by executing the whole process over again: when the simulation reaches the event point (the point beyond which the simulator is no longer valid), a scene corresponding to a new initial state is constructed and passed, along with the original model, to SimLab, resulting in a new simulator that can be used to continue the simulation.

An advantage of the model interpretation/scene compilation approach is that it can be used to generate very efficient simulators. Since the scene to be simulated is available at simulator generation time, the corresponding state equations can be explicitly instantiated and manipulated, and optimized code can be generated specifically for evaluation and solution of those equations.

In the model compilation approach to simulator generation, the information contained in a model is compiled into the simulator code produced by SimLab. The first advantage of this approach is that it enables the generated simulators to handle events. In this kind of simulator, continuing a simulation from the point at which an event occurs involves (logically, at least) calling the simulator again with a new scene corresponding to the beginning of the continuous region on the "other side" of the event or discontinuity.

Of course, using the second approach, it can be more difficult to generate efficient simulators. Simulators generated with this approach don't contain code optimized for a specific scene. Instead they contain more generic code, designed to be used on a class of scenes. We believe, however, that it is possible to develop model compilation techniques that will yield very efficient code. Such techniques include incremental compilation of program components, and incremental updates to state equations and the corresponding code. Our previous work on the Newton system[5], for example, demonstrated the value of incremental equation updates and code compilation for handling formation and breakage of contacts in rigid-body dynamics simulation.

The two currently implemented simulation generation processes represent opposite ends of a continuum of possible processes; at one end, as much time as necessary is taken to generate code optimized for one particular set of equations, while at the other, the resulting simulator may be less efficient, but works for a class of problem instances. For simulations with events, then, there are tradeoffs between the time required to generate a simulator and the performance of the resulting simulator. If events are infrequent, more time and effort can be spent generating efficient code for the particular continuous regions between events. When events are frequent, less efficient code that can be quickly modified is superior.

The two processes are depicted in Figure 1.

3.2 Implementation of the two simulator creation processes

Here we describe in a bit more detail how the current prototype implements the two simulator generation processes.

3.2.1 Model interpretation/scene compilation

Using the model interpretation/scene compilation approach, SimLab performs the following sequence of operations to create a simulator:

1. Given an expression defining a model, CLOS classes and methods are defined for each type of primitive and connection. In addition, a class is defined for the model itself.

2. The laws and definitions associated with each primitive and connection are interpreted and program components or equation schemata are created for each. These create the necessary data structures for primitives and connections.

3. Given a scene, then, the model's formulation is interpreted. This involves "applying" laws to objects, i.e., evaluating code fragments and instantiating equation schemata described above. In the current prototype, the result is a set of ODEs [2] representing the behavior of scene.

[1] This is not strictly true. Such simulators can handle a limited class of events - those that represent a discontinuity in the simulation but that do not require a change in the governing equations. For example, an event that causes a simple jump discontinuity in the state vector just requires reinitializing the integration routines.

[2] Actually, the class of equations is somewhat more general — differential-algebraic equations, or DAEs.

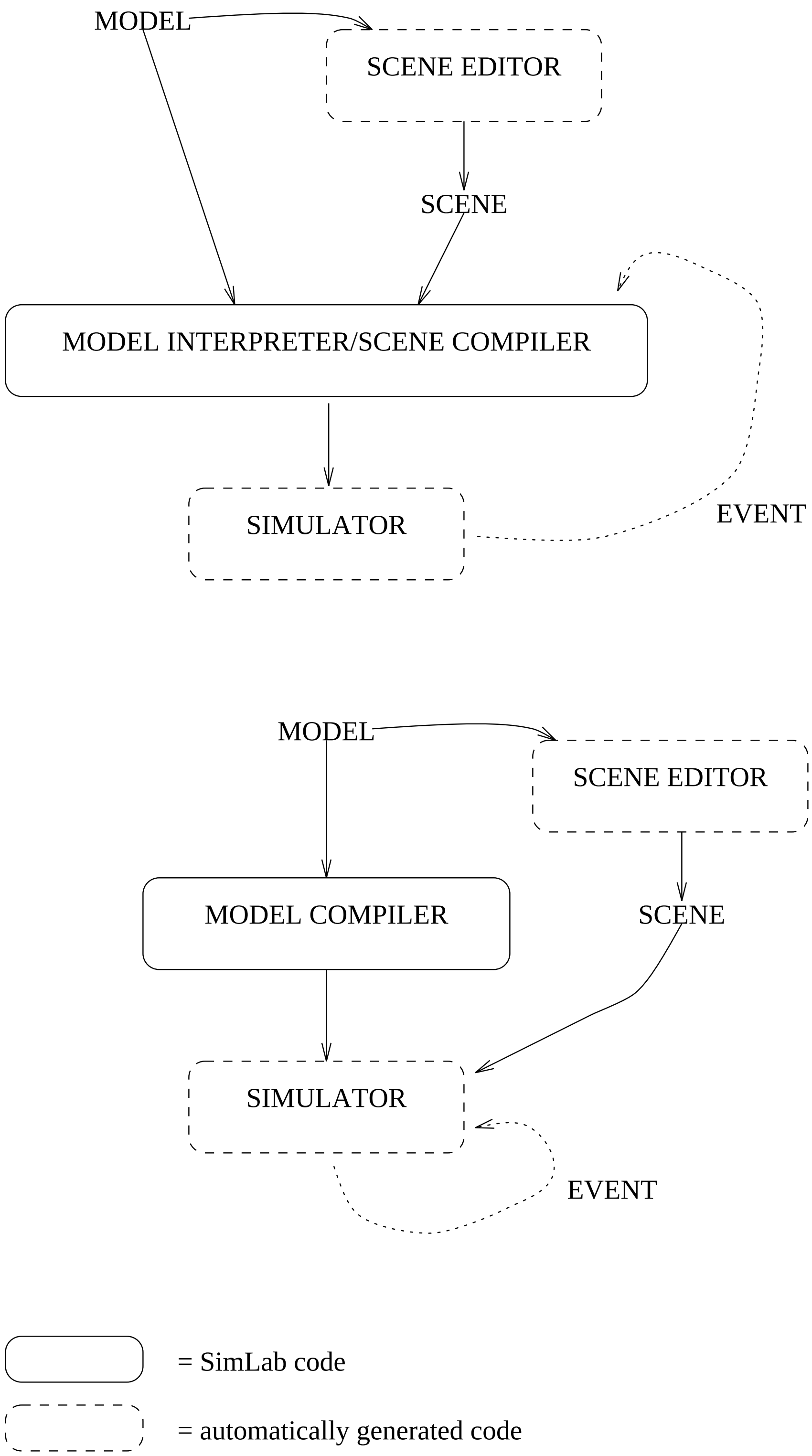

Figure 1: Simulator Generation Processes

4. The set of equations is transformed into a normal form, e.g., $y' = f(y,t))$, where y is the state vector, f is a vector valued function, and t is time. Thus the state variables are identified, and all time derivatives of state variables are moved to the left hand side of the equations.

5. Since the symbolic algebraic manipulations are performed using the computer algebra system Weyl[23], and the simulator itself is in FORTRAN, the set of equations is mapped into a single vector equation and translated into FORTRAN subroutines, which are used by ODEPACK integrators to compute the time derivative of the state vector. The routines generated depend on the integration method being used. For LSODA or LSODE, a derivative routine, YDOT, is generated. For LSODI, two other routines, RES and ADDA, are generated.

6. A main program is generated. It sets up the simulation loop, initializes data structures, calls the integrator, outputs results, and so on.

3.2.2 Model compilation

The first two steps of creating a simulator using the model compilation approach are the same as described above. However, where the previous method can construct explicit instructions that correspond to a given instance of the problem, this method must create programs that simulate the entire class of problems. Thus, for instance, the resulting YDOT function must be independent of the number of primitives and connections.

Our current implementation of model compilation uses the same ODEPACK interface and routines as the previous method, but instead of generating FORTRAN routines, it generates optimized CLOS routines that are called from FORTRAN. An initial implementation using CLOS instances and Weyl arithmetic proved too slow for moderately large problems (e.g., more than about 40 degrees of freedom).

For this reason we defined "cache storage" data structures and operations to map between them and CLOS objects. The CLOS objects are useful for setting up, changing, and viewing scenes, while the cache storage allows SimLab to generate LISP routines (e.g., YDOT) that run at near FORTRAN speed. Specifically, the operations map the CLOS classes that SimLab defines for primitives and connections into floating point (and fixnum) arrays. The routines and arrays are created from the model specification, and include declarations that allow the LISP compiler to generate fast program fragments. The subset of LISP used is limited to simple arithmetic and iteration, so generation of other code, such a FORTRAN or C would be straightforward. (Note that all compiled code is scene-independent, and does not need to be recompiled when events occur.) This scheme provides an order of magnitude performance increase, allowing reasonable simulation times for particle problems with about 400 degrees of freedom.

Steps 3 through 6 of first process are thus replaced by a single code generation step:

3. The code fragments are combined into a simulator. The laws and constraints are compiled into fast LISP routines that are scene independent.

   The resulting simulator, when given a scene, creates CLOS instances and fast storage for the primitives and connections. The simulator contains methods for copying between the fast storage and the instances. It may also include an event handling mechanism, which allows the continuous model to be changed at discontinuities, such as collisions.

3.3 Where we stand: the current state of SimLab

Our first steps toward automatic simulation generation have been to define the system architecture of SimLab and to implement the first prototype. Our architecture comprises three levels:

1. A high-level language for expressing and representing models of physics.

2. An intermediate level *algorithmic substrate* which implements computational forms of algebra, combinatorial topology, and control specification.

3. A meta-language for representing facts about program components, subroutines, and packages, which will allow algorithmic components to be automatically assembled.

We have implemented a prototype of the first, and have begun to investigate the second. We have yet to begin implementation of third level.

The "semantic gap" between SimLab's model language and the conventional languages it uses to create simulators suggests the need for an an intermediate language and associated software, which we refer to as the *algorithmic substrate*: a language in which to program mathematics and algorithms. Because we currently lack such a in intermediate level, all the simulators created so far contain some "hardwired" program components. These issues are described in Section 4.1.

3.3.1 Lumped and distributed parameter systems

The current implementation supports "lumped parameter" models of physical phenomena. That is, although we are modeling physical phenomena distributed continuously in space and time, we are able to abstract the infinite number of parameter values into a single vector or scalar parameter.

Because the combinatorial structure of lumped parameter systems is naturally represented by a graph, we have defined PRIMITIVES and CONNECTIONS (which correspond to the nodes and edges of the graph) as the basic model entities for our first prototype. The next SIMLAB prototype will support distributed parameter models in addition to lumped models. Since distributed parameter systems have a richer combinatorial structure, this will require additional basic model entities. Extension of SIMLAB to handle non-lumped systems is discussed in Section 4.3.

3.4 Algebraic manipulation

In SIMLAB, the quantities associated with CONNECTIONS and PRIMITIVES contain, in addition to information for generating data structures and methods, a rich mathematical structure. In particular, the mathematical domain of a QUANTITY is represented using the WEYL computer algebra substrate.

WEYL offers significant advantages over the traditional symbolic mathematics systems such as Macsyma[20], Maple[9] or Mathematica[21]. While these are large stand-alone programs, WEYL is an extensible toolbox for computational algebra, whose functionality is accessible at a variety of levels. Furthermore, in contrast to previous systems, WEYL explicitly represents mathematical *domains*, *elements* of those domains, and *morphisms* between domains. [3]

SIMLAB uses, among others, the following WEYL domains: the rationals, vector spaces, and polynomial rings. It also uses morphisms: maps between domains. For instance, the rigid-body dynamics simulator described in the appendix uses the *morphism* operator to define a homomorphism between unit quaternions and SO(3). This allows SIMLAB to automatically generate code that transforms elements of one representation to the other. For more details on the use of WEYL, see Zippel[23].

WEYL provides an important first step in raising the semantic level of simulators: it allows the mathematics to be specified using well-understood mathematical concepts and operations rather than *ad hoc* packages in conventional programming languages.

3.5 Example models of physics

To date we have constructed models (and hence simulators) for particle dynamics, rigid-body dynamics, and electrical circuits composed of AC generators, capacitors, resistors, etc. Our particle simulators have simulated both electrostatic and gravitational interactions, including a simulator with charged particles, wires and plates that models the electrostatic interactions of a copier.

The appendix contains the output of some of these simulators, as well as a syntactically changed (i.e., C-like syntax instead of LISP) rigid-body dynamics model.

4 The challenges

This section describes challenges that must be met to extend SIMLAB. These fall into two classes: 1) extension of the model language to include distributed parameter physical models, and 2) development of a software architecture that supports high performance simulation.

The first challenge, that of extending the language of physics models to include distributed parameter systems, requires a *computational* theory of physics and engineering systems that includes distributed phenomena. Such a theory will guide the addition of semantic constructs (the equivalent of primitives and connections for lumped systems) to the physics model language. Section 4.3 discusses these issues. The second class of challenges is described in the following sections.

4.1 The algorithmic substrate

The goal of the algorithmic substrate is to provide mathematical functionality such as algebra, topology, and geometry at an appropriate level: the level at which it is taught and understood. Programming languages that directly implement a computational form of standard mathematics will have obvious benefits: they will be widely accessible and understood, as well as sound.

[3] Domains can be viewed as sets of elements together with additional algebraic structure.

4.1.1 Components of the algorithmic substrate

While some forms of computational mathematics are well developed and available as software (e.g., linear and symbolic algebra, as well as geometry), other forms are not (e.g., point-set topology). Even those well developed technologies are not necessarily suitably packaged for high performance simulation. Thus there are two challenges facing the designer of computational mathematics software: suitable data structures and operations for currently undefined computational mathematics, and an appropriate software architecture that will serve the needs of high performance computing.

A computational form of geometry – geometric modeling – is a well developed computer technology in the following sense: theory for computing with geometric objects (such as data structures for representing geometric objects, and algorithms for performing boolean operations) has been developed and used to create computer codes for performing geometric computation. The following example shows that current geometric modeling software does not satisfy our second criterion: its architecture is not sufficiently general.

4.1.2 Example: Collision detection in a rigid-body dynamics simulation

The rigid-body model of dynamics assumes that bodies do not intersect or deform. In a given simulation, the topology of the contact relationships changes. Each different contact topology gives rise to a (generally different) set of equations that model the dynamics of the system. Thus, when contact topology changes, the continuous model must change as well. The most general way to handle this problem is to use a geometric modeler to test for interference at each time step.[5]

When a new contact relationship arises, the set of equations is reformulated. Thus, any geometric modeler capable of representing the physical objects, and testing for intersection in various configurations can be used. However, several factors combine to make this "naive" interface impractical for all but scenes containing a few primitives having the simplest shapes. Generally speaking, interference checking will take orders of magnitude longer than performing an integration step. Thus testing for intersection becomes the bottleneck. On the other hand, the position of the primitives changes very little during a given time step. Let us call a series of configurations having the property that each is very much like its predecessor *coherent*. It is clear that a significant performance improvement can be gained by recognizing properties such as coherence: it is especially wasteful to perform a series of interference checks that "know" nothing of the results of previous tests. Unfortunately, current single-level interface technology provides little choice. Usually, it is only possible to ask "Do A and B intersect?", without any sense of history.

Suppose a more general interface to geometric modeling functionality were available. How could this be used to improve the performance? Rather than request intersection "from scratch" each time, the interface could provide a means of informing the modeler what results should be kept between calls. For instance, the minimum distance between pairs of objects could be kept, perhaps together with the closest pair of points on the objects. This would allow the modeler to compare the positions with those of the next time step to determine whether there is any possibility of a change in contact topology. If not, the next integration step could be taken while the new distances are computed. (That is, the integration and intersection detection could be performed in parallel.) More ambitiously, the modeler could use the current velocities to try to predict when contact topology changes, backing up the computation when it "guesses" wrong.

4.2 A meta-language for algorithmic programming

A system that combines algorithmic components to create simulators without human intervention must represent knowledge of the components themselves. As an example, the system could "know" that the LSODA[7] routine solves ODEs represented in a particular format, what parameters it requires, etc.

The key is to explicitly represent the software components purpose, data conventions, and interfaces. This allows the system to reason about the code fragments it combines to create a simulator.

Consider the following example:

```
MODULE ODE-SOLVER-1
   DECLARE f: REAL[n] x REAL -> REAL[n];
           y(t:REAL): REAL[n];
           t0, t1: REAL;
   GIVEN f, t0, t1, y(t0);
   OPTIONALLY GIVEN JACOBIAN(f,y)
   SOLVES DERIV(y,t) = f(y,t) ON [t0, t1];
```

This specification defines the module ODE-SOLVER-1, which given function f, an interval $[t_0, t_1]$, and a value for y at t_0, finds an approximate solution to the differential equation $\frac{\partial y}{\partial t} = f(y,t)$.

Once this specification (and the code that it describes) is added to the database, it is possible to automatically generate code to solve ODE problems.

4.3 Physical systems and PDEs

As stated previously, the current SimLab prototype supports only lumped parameter systems. While lumped systems are extensively used to model a variety of engineering design and analysis problems, and in fact form the basis for computer simulation systems based on *bond graphs* such as Enport[16], incorporation of distributed phenomena into SimLab will allow it to fully support problem domains such as electrostatics, fluid flow and heat transfer.

Distributed parameter models are topologically more complex than lumped systems: lumped systems are graphs, topologically, while distributed parameter systems are represented with higher dimensional cell complexes. This added complexity will require additional language entities and operations to support, for instance, the discretization process.

In fact, the process of discretizing a PDE is, in essence, a lumping process. For instance, although not usually described this way, a finite element discretization can be viewed as substituting a system of interacting lumps (the finite elements) for the continuous model.

With traditional lumped systems, the lumping is done *a priori* for a class of objects (e.g., for *all* capacitors), while it must be done on a case-by-case basis for more difficult distributed phenomena, such as fluid flow. The behavior of each element (the "laws" for that element, in SimLab terms) is defined by unique geometric or other considerations. Thus, in contrast to simulation systems that support only lumped systems, an environment that supports distributed parameter must support the process of lumping itself.

5 Conclusion

In this paper, we have argued that it is both possible and practical to create physical systems simulators directly from the underlying models of physics. We have proposed a twofold approach: First, we need a *computational* theory to formalize algorithmic physics. Such a theory will provide the basis for computing with physical systems in much the same way that geometric modelers enable us to compute with geometry. Second, we need to be able to specify the implementation of physics models at a higher level than that provided by conventional programming languages. We propose that this specification language should be, in essence, computational forms of standard mathematics.

We have defined, as a first step towards the goal of creating a computational theory of physics, a language capable of representing models of physics for a large class of engineering systems: the lumped parameter systems. Using this language, we have constructed a prototype simulation generation environment, SimLab. We have used this prototype to create a few simple simulators, which are described in the appendix. Although the current implementation of SimLab is far from a production level software environment, we are encouraged in our belief that scientists and engineers will presently "write" simulators by explicitly defining the underlying physics.

To address the second issue, we propose a two part solution. First, an intermediate level *algorithmic substrate*, a software architecture and environment supporting a variety of computational mathematics functionality, such as symbolic and linear algebra, point-set and combinatorial topology, and geometry. Second, we propose a meta-language for representing knowledge about the software components that comprise a simulator. Such a system will allow automatic assembly of software components.

Finally, we have examined some of the research issues that must be addressed to fully realize the goal of creating simulators directly from physics models. These challenges include language support for distributed parameter systems, developing the software architecture for the algorithmic substrate, a meta-language for reasoning about algorithmic components, and support for high performance parallel architectures.

Ultimately, we envisage an environment in which state-of-the-art simulators are constructed and modified by defining the physical properties and interactions of physical objects. In addition to making it easier to create simulators, making simulators easier to understand and trust, and providing for automatic retargeting of simulators to rapidly advancing parallel machine architectures, we believe that such an environment will enable designers to analyze a wider variety of problems, because they will be able to quickly create special purpose simulators for problems that would otherwise be too costly and time consuming to simulate.

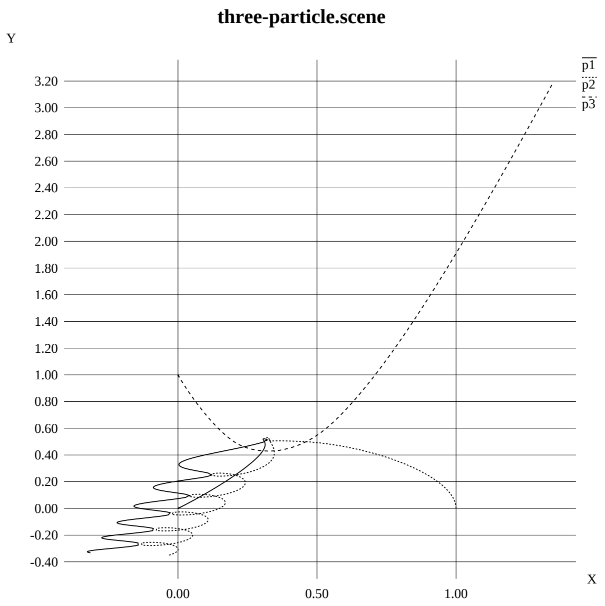

Figure 2: Trajectories of three particles. Gravitational attraction is the only force.

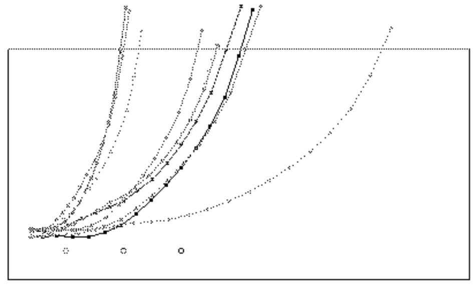

Figure 3: Electrostatic particle dynamics example.

A Examples

Here we present three example SimLab models and simulations performed using simulators produced from these models by the prototype SimLab system.

A.1 Particle dynamics

Figure 2 shows results from the the first kind of simulation done using SimLab. A simulator was produced from a model defining particles and a single type of connection — interparticle gravitational attraction.

More recently, we've used SimLab to generate simulators for electrostatic particle dynamics. These were the first SimLab generated simulators to support events. In particular, particles can be added and deleted during the course of simulations. Figure 3 shows a simulation involving a number of particles introduced at random time intervals at the left end of the image. The small circles represent wires having charges that vary in accordance with an alternating current. In addition, the particles are attracted upwards by a charged plate. This example was inspired by simulations used to design xerographic copiers.

A.2 Rigid-body dynamics

One of our goals is to be able to use SimLab to produce the equivalent of the Newton rigid-body dynamics simulator from a concise, high-level specification. The Newton system is thousands of lines of LISP and FORTRAN code; it should be possible to generate most of that from a few-page-long SimLab model.

SimLab requires further development before that goal can be fully realized. However, our current progress is promising. We have generated one rigid-body dynamics simulator from a two-page model; its coverage, however, is only a subset of Newton's. The model defines the basic motion laws of rigid-body dynamics and allows for various kinds of hinges (in this model, holonomic constraints only) between bodies. It does not provide for collision detection and resolution, or contact formation and release. Part of the model is shown in Figure 4. The figure uses a more readable syntax than the LISP-flavored one used by the existing system.

Figure 5 shows the simplest rigid-body dynamics simulation — a block falling under the influence of gravity. The rotation seen is due an initial angular velocity, as specified in the scene. The rotation axis then changes due to the $\omega \times J\omega$ term in the dynamics equations.

Figure 6 shows a more complicated simulation. In this example two parallelepipeds are hinged together at the corners. In addition, one is "hinged" to the "world," i.e., a point on it is fixed in space. The parallelepipeds were hinged at their corners to induce the interesting spinning motion. The only external force is gravity.

A.3 Electric circuits

We have also used SimLab to generate very simple circuit simulators. From a model defining capacitors, resistors, inductors, ac-generators and current laws, SimLab produced a circuit simulator. Figure 7 graphs the sample charges and current in components of a simple circuit containing a resistor, inductor, ac-generator, and two capacitors.

```
model RigidBodyDynamics
    objects
        RigidBody
            properties
                m: real                 /* mass */
                V: real                 /* volume */
                J: real^{3*3}           /* moment of inertial */
                r: real^3               /* position */
                p: UnitQuaternion       /* orientation */
                ...
                F_c: real^3             /* constraint force*/
                F_e: real^3             /* external force */
    connections
        HolonomicConstraint (object1, object2: RigidBody)
            φ                           /*the constraint equations*/
            ...
        BallAndSocketHinge (c1, c2: real^3) inherits from HolonomicConstraint
            φ = r(object1) + c1 - r(object2) - c2
            ...
        AppliedForce(object:RigidBody)
            force: real^3
    laws
        Resultant: for o:RigidBody,
```

$$F_i = \sum_{h \in HolonomicConstraint\ s.t.\ h(o,*)} \text{force1(h)} - \sum_{h \in HolonomicConstraint\ s.t.\ h(*,o)} \text{force2(h)}$$

$$F_e = \sum_{f \in AppliedForce\ s.t.\ f(o)} \text{force(f)}$$

Newton: for o: RigidBody, $m(o) \times \ddot{r}(\mathrm{o}) = F_c(o) + F_e(o)$

Euler: ...

VirtualWork: for h:Hinge, o1=object1(h), o2=object2(h),

$$\text{force1} = \frac{\partial \phi}{\partial r(o1)}^T \lambda$$

torque1 = ...

...

formulation schemes

UnreducedNewtonEuler:

MotionEquations = $\forall$o$\in$RigidBody: ApplyLaw(Newton, o) $\cup$ $\forall$o$\in$RigidBody: ApplyLaw(Euler, o)

ConstraintEquations = $\cup_{h \in HolonomicConstraint} \phi(h)$

Figure 4: Rigid-body dynamics model

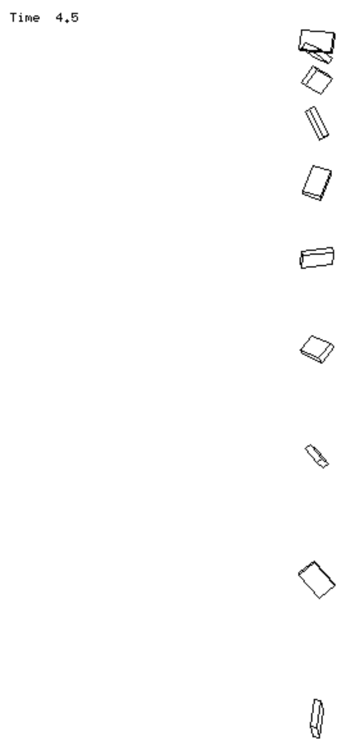

Figure 5: A single falling rigid body. Gravity is the only force.

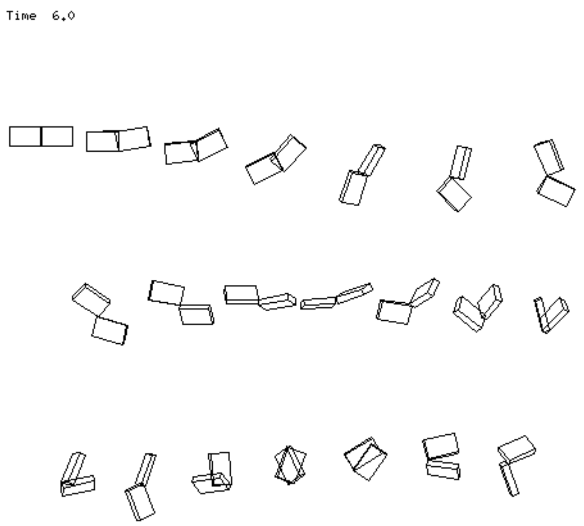

Figure 6: A double pendulum under the influence of gravity. The lower right rear corner of the rightmost block is fixed in space. The image sequence is left to right, top to bottom.

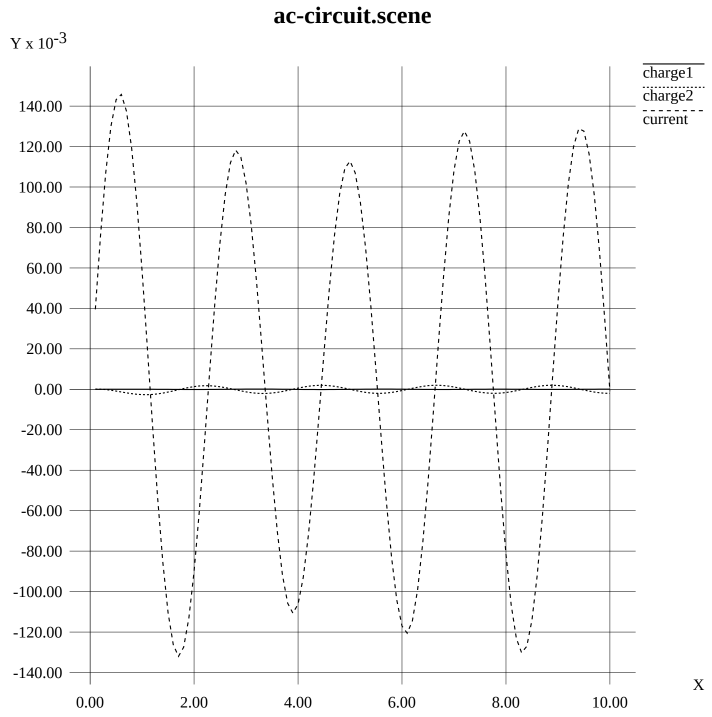

Figure 7: Sample charge, current trajectories for a simple circuit simulation. Although the graph of the current appears flat, it in fact oscillates correctly; it's simply below the resolution of this figure.

References

[1] Donald C. Augustin, Mark S. Fineberg, Bruce C. Johnson, Robert N. Linebarger, F. John Sansom, and Jon C. Strauss. The csi continuous system simulation language (cssl). *Simulation*, 9:281–303, 1967.

[2] Jan F. Broenink. *Computer-Aided Physical-Systems Modeling and Simulation: A Bond-Graph Approach*. PhD thesis, University of Twente, Enschede, The Netherlands, 1990.

[3] C. K. Brown. PADL-2: a technical summary. *IEEE Computer Graphics and Applications*, 2(2):69–84, March 1982.

[4] Franqis E. Cellier. *Continuous System Modeling*. Springer-Verlag, New York, 1991.

[5] James F. Cremer and A. James Stewart. The architecture of *newton*, a general-purpose dynamics simulator. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pages 1806–1811, May 1989.

[6] J. Dongarra, J. R. Bunch, Cleve B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, PA, 1978.

[7] A. C. Hindmarsh. Odepack, a systematized collection of ODE solvers. In R. S. Stepleman and et al., editors, *Scientific Computing*, pages 55–64. North-Holland, Publ., Amsterdam, 1983.

[8] Christoph M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, California, 1989.

[9] Maple Group, Waterloo, Canada. *Maple*, 1987.

[10] Leonard Meirovitch. *Methods of Analytical Dynamics*. McGraw-Hill, New York, 1970.

[11] Mitchell & Gauthier Assoc., Concord, Mass. *AUTOSIM*.

[12] Mitchell & Gauthier Assoc., Concord, Mass. *ACSL: Advanced Continuous Simulation Language — User Guide and Reference Manual*, 1986.

[13] Henry M. Paynter. *Analysis and Design of Engineering Systems*. The M.I.T. Press, Cambridge, Massachusetts, 1961.

[14] A.A.B. Pritsker. *Introduction to Simulation and SLAM II*. Systems Publishing Corporation, 1986.

[15] Howard Reingold. *Tools for Thought*. Simon and Schuster, 1985.

[16] RosenCode Associates, Inc., Lansing, Mich. *The ENPORT Reference Manual*, 1989.

[17] Spatial Technology, Inc., Applied Geometry, Inc., and Three-Space, Ltd., Boulder, Colorado. *ACIS Geometric Modeler Interface Guide*, 1990.

[18] Guy L. Steele, Jr. *Common Lisp, the language*. Digital Press, second edition, 1990.

[19] Symbolic Dynamics, Inc., Mountain View ,CA. *SD/FAST User's Manual*, 1990.

[20] Symbolics, Inc., Burlington, MA. *MACSYMA Reference Manual*, 14th edition, 1989.

[21] Steven Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Redwood City, CA, 1988.

[22] XOX Corporation, Bloomington, MN. *SHAPES Geometry Library Reference Manual*, 1990.

[23] Richard Eliot Zippel. The Weyl computer algebra substrate. Technical Report 90–1077, Department of Computer Science, Cornell University, Ithaca, NY, 1990.