(2)

SIMLAB: Automatically Creating Physical Systems Simulators

Richard S. Palmer
James F. Cremer

TR 91-1246
November 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

SIMLAB: Automatically Creating Physical Systems Simulators

Richard S. Palmer
James F. Cremer

Cornell University

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 intractible 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[16], 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, 18], ODE integrators[8], linear algebra packages[6], and symbolic algebra systems[20, 11, 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 [9].) 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[13, 1, 15] 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.

Besides the simulation languages described above, the *bond graph* formalism, developed by Paynter in the late 1950's[14], has engendered a number of simulation systems[17, 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. Another use of bond graphs, which is of less interest in terms of simulation, is as a schematic formalism for specifying engineering systems. Bond graph 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 (PRIMITIVESand *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 method, emphasize building a model (instance, that is) using predefined components rather than defining the model of physics, the objects, and their interactions. 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 gases 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 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 the 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, the appendices show how SIMLAB was used to create simultors for simple particle dymanics, rigid body dynamics, and electrical 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 Newtonian 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[12]. 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 "resistor" is defined to be a kind of "2-terminal" in the AC circuit in Appendix C.

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 Newtonian particle mechanics

We begin by describing Newtonian 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 gravitation. The gravitational force between particles p_1 and p_2 is defined by

$$F_{p_1p_2} = \frac{Gm_1m_2}{r^2}, \tag{1}$$

where G is the gravitational constant, m_1 is the mass of p_1, m_2 is the mass of p_2, and r is the distance between p_1 and p_2.

Note that this expression defines only the magnitude of the force between p_1 and p_2. The gravitational force is attractive, which means that the force of p_1 on p_2 is directed from p_2 to p_1.

2.4 The SIMLAB model

Although the current implementation of SIMLAB is written in COMMON LISP for reasons of presentation we use a C-like syntax in this section. Appendix A contains the actual syntax for this model, together with a scene containing two particles, and the result of running the particle simulator on that scene.

In addition to formally expressing Newton's laws as described above, a SIMLAB model of Newtonian particle mechanics adds definitions of what a particle is, what quantities are associated with a particle, and the direction of the gravitational 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 Newtonian-particle-mechanics
```

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
    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 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 Gravity: the *connection* between particles

We now proceed to define the model's *connections*, which define the modes of interaction between the primitives. For particle mechanics, the only interaction between particles is the gravitational force between a pair of particles.

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

```
CONNECTION gravitation(p1, p2 : particle)
  QUANTITIES force in R^3
  CONSTRAINTS
    force =
      (6.672e-11 * mass(p1) * mass(p2) * (position(p2) - position(p1)))
      / DISTANCE(position(p1), position(p2))^3
```

This connection specifies the gravitational 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 assemle the system of equations that model a given instance. In the case of particle dynamics, each particle exerts a gravitational 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 gravitation-set(p)
             IF (p == p1-of(c))  /* force directed towards p1 */
                force-of(c)
             ELSE  /* force directed towards p2 */
                - force-of(c)
```

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

This completes the model for Newtonian 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 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 /* name is optional for any SimLab entity */
        mass 1.0
      position <0.0, 0.0, 0.0>
        velocity <1.0, 0.0, 0.0>

PARTICLE /* at time = 0 */
        name particle-2
        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
```

```
gravitation(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 gravitational interaction occurs between all distinct pairs of particles in the scene. Although the fact that gravitation occurs between all pairs of particles could be specified in the model, we have chosen in this case to specify the existence of gravitation a *per*-scene basis, which allows creating scenes where gravity 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 constraint 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) = 6.672e-11 * mass(p1) * mass(p2)
            * NORMALIZE(position(p2) - position(p1))
            / DISTANCE(position(p1), position(p2))^2)
force(p2) = 6.672e-11 * mass(p2) * mass(p1)
            * NORMALIZE(position(p1) - position(p2))
            / DISTANCE(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 The current implementation of SIMLAB

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 3.3 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)[19]: 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 based on X-WINDOWS and CLM[7]. 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 How SIMLAB creates simulators

SIMLAB, given a model expression, performs the following sequence of operations to create a simulator:

1. CLOS classes and methods are defined for each PRIMITIVE and CONNECTION. In addition, a class is defined for the model itself. The scene generator and simulator created from the model will use these classes to create and interpret instances of the problem.

2. The LAWS and DEFINITIONS associated with each PRIMITIVE and CONNECTION are interpreted and code fragments are created for each. These code fragments, which will later be assembled to create a simulator, create the necessary data structures and equations for a given primitive or connection.

3. The FORMULATION is interpreted, and code is created to assemble the global set of equations. This code incorporates the code segments described in 2.

4. The classes and other software components are assembled to create a stand-alone simulator, which takes as input a scene satisfying the model definition. It also generates a scene editor for creating instances of the problem.

The FORMULATION is encoded into the stand-alone simulator, which is responsible for interpreting a *given scene*, *generating* a *system of* ODE's, *and performing* the *simulation*. This simulator, given a scene instance, performs the following operations:

1. For each primitive and connection, equations are added to the ODE system. These equations are generated by applying the model's laws to the primitives and connections.

2. The set of equations is transformed into normal form: $y' = f(y,t))$, 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.

3. Since the symbolic algebraic manipulations are performed using the computer algebra system WEYL[23], and the simulator itself is in FORTRAN[8], the resulting set of equations (from 2) is mapped into a single vector equation, and translated into a FORTRAN subroutine for computing the time derivative of the state vector.

4. The subroutine constructed in the previous step is loaded together with standard numerical codes for integrating ODEs[8], as well as visualization code and user interface components.

5. The resulting code is executed to perform the simulation, and the results are displayed using the user interface tools.

3.2 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 code fragments and subroutines and packages, which will allow algorithmic components to be automatically assembled.

We have implemented a prototype of the first, but have yet to begin implementation of the second and third levels.

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" code fragments. These issues are described in Section 4.1.

3.2.1 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.

3.3 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 struc-

ture. 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[11] 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. [1]

SIMLAB uses, among others, the following WEYL domains: the rational integers, vector spaces, and polynomial rings. It also uses morphisms: maps between domains. For instance, the rigid body dynamics simulator defined in Appendix B 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 step 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.

3.4 Example models of physics

To date we have constructed models (and hence simulators) for Newtonian particle dynamics, rigid body dynamics, and electrical circuits composed of AC generators, capacitors, resistors, etc. All of the examples described in the appendices are actual input files, and the resulting simulators and scene generators have been run and used to analyze simple engineering problems in the their domain. The model specification, runs, and analyses are included together in the respective appendices.

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

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.

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 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. Define a series of configurations having the property that each is very much like its predecessor to be *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[8] 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 DERIVATIVE(y, t) = f(y, t) ON INTERVAL [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[17], 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 report, 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 semantics 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 apendices. 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 con-

structed 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. In particular, it will aid in creating multiple-domain simulators, which are currently very difficult to create, and hence scarce.

A Particle-dynamics

A.1 The model

This appendix contains the actual syntax used by SIMLAB. While parts of the syntax are reasonably acceptable, the *formulation* syntax is currently rather sad, being unadulterated COMMON LISP that accesses lower level SIMLAB functionality directly. Once we have developed the intermediate level language (the algorithmic substrate) sufficiently, the users of SIMLAB will not need to understand or access the low level implementation.

```
;;;   -*- Syntax: Common-Lisp; Package: SIMLAB; Base: 10; Mode: LISP -*-

(in-package "SIMLAB")

(define-model particle-dynamics

    (primitive particle
       (quantities
          (mass :domain R :constant)
          (momentum :domain R3)
          (force :domain R3)
          (position :domain R3)
          (velocity :domain R3))
       (constraints
          (= force (deriv momentum 'time))
          (= velocity (deriv position 'time))
          (= momentum (* mass velocity))))

    (connection gravitation
       (objects
          (p1 particle)
          (p2 particle))
       (quantities
          (force :domain R3))
       (constraints
          (= force
             (/ (*  6.672e-11 (mass p1) (mass p2)
                    (- (position p2) (position p1)))
                (distance-cubed (position p1) (position p2))))))

    (laws
       (resultant_force
          (for p in particle
               (= (force p)
                  (+ (sum g in gravitation such that (= p (p1 g))
```

```
                    (force g))
                (sum g in gravitation such that (= p (p2 g))
                    (- (force g)))))))))

(formulation-scheme particle-dynamics-formulation
   (let force-equations =
        (union
          (mapover (particle (objects-of-type 'particle))
                   (apply-law 'resultant_force particle))
          (mapover (connection (connections-of-type 'gravitation))
                   (constraint-equations-of-connection connection))))
   (let constraints =
        (mapover (particle (objects-of-type 'particle))
                 (constraint-equations-of-object particle)))))
```

A.2 A particle scene

This scene contains three particles. The simulator generated, together with a graph of the trajectories of the particles for the first 300 seconds follows.

```
(define-scene
    (primitive  particle (name p1)
      (quantities
        (mass 1.0e6)
        (position 0.0 0.0 0.0)
        (momentum 0.0 0.0 0.0)))
    (primitive particle (name p2)
      (quantities
        (mass 1.0e6)
        (position 1.0 0.0 0.0)
        (momentum 0.0 5000.0 0.0)))
    (primitive particle (name p3)
      (quantities
        (mass 1.0e6)
        (position 0.0 1.0 0.0)
        (momentum 0.0 0.0 0.0)))
    (connection gravitation
      (objects (p1 p1) (p2 p2)))
    (connection gravitation
      (objects (p1 p2) (p2 p3)))
    (connection gravitation
      (objects (p1 p3) (p2 p1))))
```

A.3 The generated simulator

```
C **   Automatically generated fortran main program

C **  first the declarations
      external ydot
      integer NEQ
      double precision Y(0:20)
      double precision TIME
      double precision TOUT
      integer ITOL
      double precision RTOL
      double precision ATOL
      integer ITASK
      integer ISTATE
      integer IOPT
      double precision RWORK(0:507)
      integer LRW
      integer IWORK(0:37)
      integer LIW
      integer JT

C **  now perform the initial assignments
      NEQ = 18
      Y(0) = 0.0
      Y(1) = 0.0
      Y(2) = 0.0
      Y(3) = 0.0
      Y(4) = 0.0
      Y(5) = 0.0
      Y(6) = 0.0
      Y(7) = 5000.0
      Y(8) = 0.0
      Y(9) = 0.0
      Y(10) = 0.0
      Y(11) = 1.0
      Y(12) = 0.0
      Y(13) = 0.0
      Y(14) = 0.0
      Y(15) = 0.0
      Y(16) = 1.0
      Y(17) = 0.0
      Y(18) = 1000000.0
      Y(19) = 1000000.0
      Y(20) = 1000000.0
      TIME = 0.0
      TOUT = 0.3
```

```
      ITOL = 1
      RTOL = 1.0E-6
      ATOL = 1.0E-6
      ITASK = 1
      ISTATE = 1
      IOPT = 0
      LRW = 508
      LIW = 38
      JT = 2
      print*, NEQ, Y, TIME, TOUT, ITOL, RTOL, ATOL, ITASK, ISTATE, IOPT,
     c LRW, LIW, JT

C **  now execute the code
      do 10 i=1,1000
      call lsoda(YDOT,neq,y,time,tout,itol,rtol,atol,itask,istate,iopt,r
     cwork,lrw,iwork,liw,YDOT,jt)
      call prntsv(time, neq, y)
      tout=tout+ 0.3
   10 continue

      stop
      end


C **   Automatically generated subroutine

      subroutine ydot(neq,time,y,yp)
      double precision time,y(0:20),yp(0:17)
      double precision dcubed

C **  D{momentum-4<2>, time} = ((6.672E-11 dcubed{position-6<2>, positio
C     n-6<1>, position-6<0>, position-15<2>, position-15<1>, position-15
C     <0>} mass-3 position-24<2> + -6.672E-11 dcubed{position-6<2>, posi
C     tion-6<1>, position-6<0>, position-15<2>, position-15<1>, position
C     -15<0>} position-6<2> mass-3) mass-21 + ((6.672E-11 dcubed{positio
C     n-24<2>, position-24<1>, position-24<0>, position-6<2>, position-6
C     <1>, position-6<0>} mass-3 position-15<2> + -6.672E-11 dcubed{posi
C     tion-24<2>, position-24<1>, position-24<0>, position-6<2>, positio
C     n-6<1>, position-6<0>} position-6<2> mass-3) mass-12))/(dcubed{pos
C     ition-24<2>, position-24<1>, position-24<0>, position-6<2>, positi
C     on-6<1>, position-6<0>} dcubed{position-6<2>, position-6<1>, posit
C     ion-6<0>, position-15<2>, position-15<1>, position-15<0>})
      yp(0)=(((( 6.672E-11  * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(1
     c1)))) * Y(18)) * Y(15) + (( -6.672E-11  * (dcubed(Y(3), Y(4), Y(5)
     c, Y(9), Y(10), Y(11)))) * Y(3)) * Y(18)) * Y(19) + ((( 6.672E-11
     c* (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5)))) * Y(18)) * Y(9)
     c + (( -6.672E-11  * (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5))
     c)) * Y(3)) * Y(18)) * Y(20)) * ((dcubed(Y(15), Y(16), Y(17), Y(3),
```

```
     c Y(4), Y(5))) * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(11)))) **
     c  -1

C **  D{momentum-4<1>, time} = ((6.672E-11 dcubed{position-6<2>, positio
C     n-6<1>, position-6<0>, position-15<2>, position-15<1>, position-15
C     <0>} mass-3 position-24<1> + -6.672E-11 dcubed{position-6<2>, posi
C     tion-6<1>, position-6<0>, position-15<2>, position-15<1>, position
C     -15<0>} position-6<1> mass-3) mass-21 + ((6.672E-11 dcubed{positio
C     n-24<2>, position-24<1>, position-24<0>, position-6<2>, position-6
C     <1>, position-6<0>} mass-3 position-15<1> + -6.672E-11 dcubed{posi
C     tion-24<2>, position-24<1>, position-24<0>, position-6<2>, positio
C     n-6<1>, position-6<0>} position-6<1> mass-3) mass-12))/(dcubed{pos
C     ition-24<2>, position-24<1>, position-24<0>, position-6<2>, positi
C     on-6<1>, position-6<0>} dcubed{position-6<2>, position-6<1>, posit
C     ion-6<0>, position-15<2>, position-15<1>, position-15<0>})
      yp(1)=(((( 6.672E-11  * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(1
     c1)))) * Y(18)) * Y(16) + (( -6.672E-11  * (dcubed(Y(3), Y(4), Y(5)
     c, Y(9), Y(10), Y(11)))) * Y(4)) * Y(18)) * Y(19) + ((( 6.672E-11
     c* (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5)))) * Y(18)) * Y(10
     c) + (( -6.672E-11  * (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5)
     c))) * Y(4)) * Y(18)) * Y(20)) * ((dcubed(Y(15), Y(16), Y(17), Y(3)
     c, Y(4), Y(5))) * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(11)))) *
     c*  -1

C **  D{momentum-4<0>, time} = ((6.672E-11 dcubed{position-6<2>, positio
C     n-6<1>, position-6<0>, position-15<2>, position-15<1>, position-15
C     <0>} mass-3 position-24<0> + -6.672E-11 dcubed{position-6<2>, posi
C     tion-6<1>, position-6<0>, position-15<2>, position-15<1>, position
C     -15<0>} position-6<0> mass-3) mass-21 + ((6.672E-11 dcubed{positio
C     n-24<2>, position-24<1>, position-24<0>, position-6<2>, position-6
C     <1>, position-6<0>} mass-3 position-15<0> + -6.672E-11 dcubed{posi
C     tion-24<2>, position-24<1>, position-24<0>, position-6<2>, positio
C     n-6<1>, position-6<0>} position-6<0> mass-3) mass-12))/(dcubed{pos
C     ition-24<2>, position-24<1>, position-24<0>, position-6<2>, positi
C     on-6<1>, position-6<0>} dcubed{position-6<2>, position-6<1>, posit
C     ion-6<0>, position-15<2>, position-15<1>, position-15<0>})
      yp(2)=(((( 6.672E-11  * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(1
     c1)))) * Y(18)) * Y(17) + (( -6.672E-11  * (dcubed(Y(3), Y(4), Y(5)
     c, Y(9), Y(10), Y(11)))) * Y(5)) * Y(18)) * Y(19) + ((( 6.672E-11
     c* (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5)))) * Y(18)) * Y(11
     c) + (( -6.672E-11  * (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5)
     c))) * Y(5)) * Y(18)) * Y(20)) * ((dcubed(Y(15), Y(16), Y(17), Y(3)
     c, Y(4), Y(5))) * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(11)))) *
     c*  -1

C **  D{position-6<2>, time} = (momentum-4<2>)/(mass-3)
      yp(3)=Y(0) * Y(18) **  -1
```

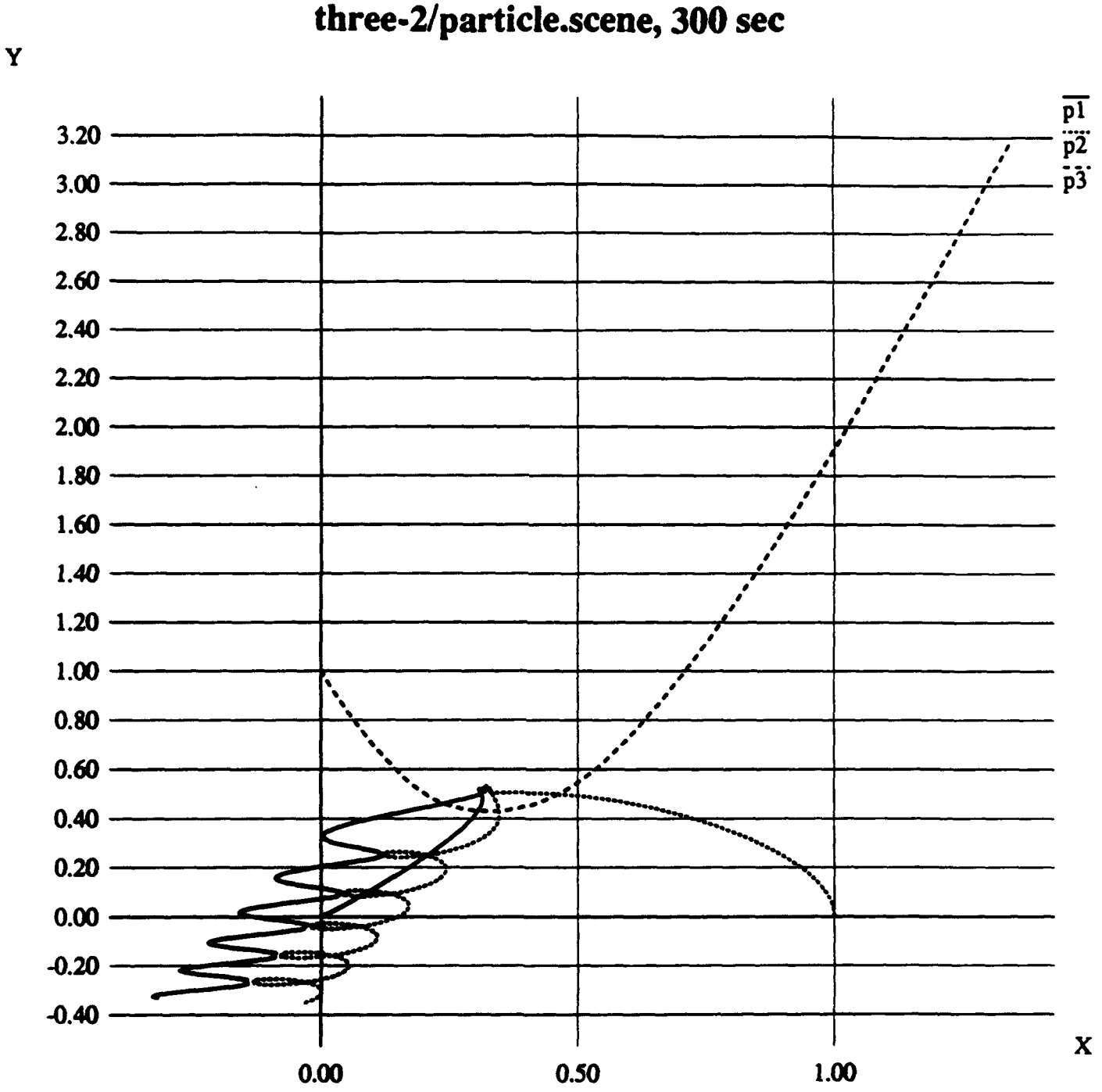
```
C **  D{position-6<1>, time} = (momentum-4<1>)/(mass-3)
      yp(4)=Y(1) * Y(18) **  -1

C **  D{position-6<0>, time} = (momentum-4<0>)/(mass-3)
      yp(5)=Y(2) * Y(18) **  -1

C **  D{momentum-13<2>, time} = ((6.672E-11 dcubed{position-6<2>, positi
C     on-6<1>, position-6<0>, position-15<2>, position-15<1>, position-1
C     5<0>} mass-12 position-24<2> + -6.672E-11 dcubed{position-6<2>, po
C     sition-6<1>, position-6<0>, position-15<2>, position-15<1>, positi
C     on-15<0>} position-15<2> mass-12) mass-21 + ((-6.672E-11 dcubed{po
C     sition-15<2>, position-15<1>, position-15<0>, position-24<2>, posi
C     tion-24<1>, position-24<0>} mass-3 position-15<2> + 6.672E-11 dcub
C     ed{position-15<2>, position-15<1>, position-15<0>, position-24<2>,
C      position-24<1>, position-24<0>} position-6<2> mass-3) mass-12))/(
C     dcubed{position-15<2>, position-15<1>, position-15<0>, position-24
C     <2>, position-24<1>, position-24<0>} dcubed{position-6<2>, positio
C     n-6<1>, position-6<0>, position-15<2>, position-15<1>, position-15
C     <0>})
      yp(6)=((((  6.672E-11  * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(1
     c1)))) * Y(20)) * Y(15) + ((  -6.672E-11  * (dcubed(Y(3), Y(4), Y(5)
     c, Y(9), Y(10), Y(11)))) * Y(9)) * Y(20)) * Y(19) + (((  -6.672E-11
     c * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) * Y(18)) * Y
     c(9) + ((  6.672E-11  * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(
     c17)))) * Y(3)) * Y(18)) * Y(20)) * ((dcubed(Y(9), Y(10), Y(11), Y(
     c15), Y(16), Y(17))) * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(11)
     c))) **  -1

C **  D{momentum-13<1>, time} = ((6.672E-11 dcubed{position-6<2>, positi
C     on-6<1>, position-6<0>, position-15<2>, position-15<1>, position-1
C     5<0>} mass-12 position-24<1> + -6.672E-11 dcubed{position-6<2>, po
C     sition-6<1>, position-6<0>, position-15<2>, position-15<1>, positi
C     on-15<0>} position-15<1> mass-12) mass-21 + ((-6.672E-11 dcubed{po
C     sition-15<2>, position-15<1>, position-15<0>, position-24<2>, posi
C     tion-24<1>, position-24<0>} mass-3 position-15<1> + 6.672E-11 dcub
C     ed{position-15<2>, position-15<1>, position-15<0>, position-24<2>,
C      position-24<1>, position-24<0>} position-6<1> mass-3) mass-12))/(
C     dcubed{position-15<2>, position-15<1>, position-15<0>, position-24
C     <2>, position-24<1>, position-24<0>} dcubed{position-6<2>, positio
C     n-6<1>, position-6<0>, position-15<2>, position-15<1>, position-15
C     <0>})
      yp(7)=((((  6.672E-11  * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(1
     c1)))) * Y(20)) * Y(16) + ((  -6.672E-11  * (dcubed(Y(3), Y(4), Y(5)
     c, Y(9), Y(10), Y(11)))) * Y(10)) * Y(20)) * Y(19) + (((  -6.672E-11
     c  * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) * Y(18)) *
     cY(10) + ((  6.672E-11  * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16),
     cY(17)))) * Y(4)) * Y(18)) * Y(20)) * ((dcubed(Y(9), Y(10), Y(11),
     cY(15), Y(16), Y(17))) * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(1
```

```
     c1)))) **  -1

C **  D{momentum-13<0>, time} = ((6.672E-11 dcubed{position-6<2>, positi
C     on-6<1>, position-6<0>, position-15<2>, position-15<1>, position-1
C     5<0>} mass-12 position-24<0> + -6.672E-11 dcubed{position-6<2>, po
C     sition-6<1>, position-6<0>, position-15<2>, position-15<1>, positi
C     on-15<0>} position-15<0> mass-12) mass-21 + ((-6.672E-11 dcubed{po
C     sition-15<2>, position-15<1>, position-15<0>, position-24<2>, posi
C     tion-24<1>, position-24<0>} mass-3 position-15<0> + 6.672E-11 dcub
C     ed{position-15<2>, position-15<1>, position-15<0>, position-24<2>,
C      position-24<1>, position-24<0>} position-6<0> mass-3) mass-12))/(
C     dcubed{position-15<2>, position-15<1>, position-15<0>, position-24
C     <2>, position-24<1>, position-24<0>} dcubed{position-6<2>, positio
C     n-6<1>, position-6<0>, position-15<2>, position-15<1>, position-15
C     <0>})
      yp(8)=((((  6.672E-11  * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(1
     c1)))) * Y(20)) * Y(17) + (( -6.672E-11  * (dcubed(Y(3), Y(4), Y(5)
     c, Y(9), Y(10), Y(11)))) * Y(11)) * Y(20)) * Y(19) + ((( -6.672E-11
     c  * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) * Y(18)) *
     cY(11) + (( 6.672E-11  * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16),
     cY(17)))) * Y(5)) * Y(18)) * Y(20)) * ((dcubed(Y(9), Y(10), Y(11),
     cY(15), Y(16), Y(17))) * (dcubed(Y(3), Y(4), Y(5), Y(9), Y(10), Y(1
     c1)))) **  -1

C **  D{position-15<2>, time} = (momentum-13<2>)/(mass-12)
      yp(9)=Y(6) * Y(20) **  -1

C **  D{position-15<1>, time} = (momentum-13<1>)/(mass-12)
      yp(10)=Y(7) * Y(20) **  -1

C **  D{position-15<0>, time} = (momentum-13<0>)/(mass-12)
      yp(11)=Y(8) * Y(20) **  -1

C **  D{momentum-22<2>, time} = (((-6.672E-11 dcubed{position-24<2>, pos
C     ition-24<1>, position-24<0>, position-6<2>, position-6<1>, positio
C     n-6<0>} mass-12 + -6.672E-11 dcubed{position-15<2>, position-15<1>
C     , position-15<0>, position-24<2>, position-24<1>, position-24<0>}
C     mass-3) position-24<2> + (6.672E-11 dcubed{position-24<2>, positio
C     n-24<1>, position-24<0>, position-6<2>, position-6<1>, position-6<
C     0>} position-15<2> mass-12 + 6.672E-11 dcubed{position-15<2>, posi
C     tion-15<1>, position-15<0>, position-24<2>, position-24<1>, positi
C     on-24<0>} position-6<2> mass-3)) mass-21)/(dcubed{position-24<2>,
C     position-24<1>, position-24<0>, position-6<2>, position-6<1>, posi
C     tion-6<0>} dcubed{position-15<2>, position-15<1>, position-15<0>,
C     position-24<2>, position-24<1>, position-24<0>})
      yp(12)=((((  -6.672E-11  * (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4),
     c Y(5)))) * Y(20) + ( -6.672E-11  * (dcubed(Y(9), Y(10), Y(11), Y(1
     c5), Y(16), Y(17)))) * Y(18)) * Y(15) + (( 6.672E-11  * (dcubed(Y(1
```

```
     c5), Y(16), Y(17), Y(3), Y(4), Y(5)))) * Y(9)) * Y(20) + (( 6.672E-
     c11  * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) * Y(3)) *
     c Y(18)) * Y(19)) * ((dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5))
     c) * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) **  -1

C **  D{momentum-22<1>, time} = (((-6.672E-11 dcubed{position-24<2>, pos
C     ition-24<1>, position-24<0>, position-6<2>, position-6<1>, positio
C     n-6<0>} mass-12 + -6.672E-11 dcubed{position-15<2>, position-15<1>
C     , position-15<0>, position-24<2>, position-24<1>, position-24<0>}
C     mass-3) position-24<1> + (6.672E-11 dcubed{position-24<2>, positio
C     n-24<1>, position-24<0>, position-6<2>, position-6<1>, position-6<
C     0>} position-15<1> mass-12 + 6.672E-11 dcubed{position-15<2>, posi
C     tion-15<1>, position-15<0>, position-24<2>, position-24<1>, positi
C     on-24<0>} position-6<1> mass-3)) mass-21)/(dcubed{position-24<2>,
C     position-24<1>, position-24<0>, position-6<2>, position-6<1>, posi
C     tion-6<0>} dcubed{position-15<2>, position-15<1>, position-15<0>,
C     position-24<2>, position-24<1>, position-24<0>})
      yp(13)=(((( -6.672E-11  * (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4),
     c Y(5)))) * Y(20) + ( -6.672E-11  * (dcubed(Y(9), Y(10), Y(11), Y(1
     c5), Y(16), Y(17)))) * Y(18)) * Y(16) + (( 6.672E-11  * (dcubed(Y(1
     c5), Y(16), Y(17), Y(3), Y(4), Y(5)))) * Y(10)) * Y(20) + (( 6.672E
     c-11  * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) * Y(4))
     c* Y(18)) * Y(19)) * ((dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5)
     c)) * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) **  -1

C **  D{momentum-22<0>, time} = (((-6.672E-11 dcubed{position-24<2>, pos
C     ition-24<1>, position-24<0>, position-6<2>, position-6<1>, positio
C     n-6<0>} mass-12 + -6.672E-11 dcubed{position-15<2>, position-15<1>
C     , position-15<0>, position-24<2>, position-24<1>, position-24<0>}
C     mass-3) position-24<0> + (6.672E-11 dcubed{position-24<2>, positio
C     n-24<1>, position-24<0>, position-6<2>, position-6<1>, position-6<
C     0>} position-15<0> mass-12 + 6.672E-11 dcubed{position-15<2>, posi
C     tion-15<1>, position-15<0>, position-24<2>, position-24<1>, positi
C     on-24<0>} position-6<0> mass-3)) mass-21)/(dcubed{position-24<2>,
C     position-24<1>, position-24<0>, position-6<2>, position-6<1>, posi
C     tion-6<0>} dcubed{position-15<2>, position-15<1>, position-15<0>,
C     position-24<2>, position-24<1>, position-24<0>})
      yp(14)=(((( -6.672E-11  * (dcubed(Y(15), Y(16), Y(17), Y(3), Y(4),
     c Y(5)))) * Y(20) + ( -6.672E-11  * (dcubed(Y(9), Y(10), Y(11), Y(1
     c5), Y(16), Y(17)))) * Y(18)) * Y(17) + (( 6.672E-11  * (dcubed(Y(1
     c5), Y(16), Y(17), Y(3), Y(4), Y(5)))) * Y(11)) * Y(20) + (( 6.672E
     c-11  * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) * Y(5))
     c* Y(18)) * Y(19)) * ((dcubed(Y(15), Y(16), Y(17), Y(3), Y(4), Y(5)
     c)) * (dcubed(Y(9), Y(10), Y(11), Y(15), Y(16), Y(17)))) **  -1

C **  D{position-24<2>, time} = (momentum-22<2>)/(mass-21)
      yp(15)=Y(12) * Y(19) **  -1
```

```
C **  D{position-24<1>, time} = (momentum-22<1>)/(mass-21)
      yp(16)=Y(13) * Y(19) **  -1

C **  D{position-24<0>, time} = (momentum-22<0>)/(mass-21)
      yp(17)=Y(14) * Y(19) **  -1
      return
      end
```

A.4 The particle trajectories



B Rigid body dynamics

Section 2 derives a SIMLAB model for particle mechanics, although it does not use the actual SIMLAB syntax; this is given in Appendix A. This appendix shows, in explicit detail, how SIMLAB is used to create a simulator for rigid body dynamics. The description in this section is intended to provide a detailed account of how the model was created, and what decisions were made in creating it.

In this model, the primitive object is generalized from a single particle to a *rigid body*, which may be viewed as a set of particles whose relative position is constant though time. For the model we derive here, the only interaction between rigid bodies is the *ball-and-socket* hinge, which models connections such as the hip joint in a person, or a ball joint in a car.

The model specification, which is given in it's entirety at the end if this section, defines a model named *Rigid-body-dynamics*. The model contains a single kind of *primitive*, the *rigid-body*.

```
DEFINE-MODEL rigid-body-dynamics
```

We now proceed to define the quantities and laws that define the behavior of a rigid body.

The position (and thus velocity and acceleration) of a rigid body is defined by six parameters. In addition to the three that define the position of a particle in space, three parameters are required to define its *orientation*. This means that we must add quantities to account for the additional degrees of freedom. Thus the analog of position in the particle model is the pair:

```
(position :domain R3 :synonyms r)
(orientation :domain unit-quaternion :synonyms p)
```

Here we see that the (generalized) position of a rigid body is given by two vectors, the *position* and *orientation*, which relate the position and orientation of the body to a *global coordinate frame*. In this case, we have chosen to represent the orientation of a body with a unit quaternion.

Here we have specified *synonyms* for position and orientation, r and p respectively. This allows the use of both descriptive names, such as *position* and traditional symbols, such as r.

A natural consequence of partitioning the generalized position is that velocity and acceleration (as well as momentum and force) are similarly partitioned. Thus:

```
(velocity :domain R3 :synonyms v)
(angular-velocity :domain R3   :synonyms omega)
(acceleration :domain R3   :synonyms a)
(angular-acceleration :domain R3 :synonyms omega-dot)
```

In the case of particle mechanics, the momentum and velocity are related by a scalar valued quantity, the *mass*. In the case of rigid bodies, the "mass properties" are again partitioned into the translational and rotation parts. The density, which in this model is assumed to be constant throughout a given rigid body, represents the mass per unit volume. The mass and moment of inertia are computed by integrating over the geometry of the body, as defined below in the laws.

```
(density R :synonyms rho :constant)
(mass :domain R :synonyms m :constant)
(moment-of-inertia :domain R3x3 :synonyms J)
```

In the case of the translational momentum and velocity, the relationship is independent of direction, and may be represented with a single scalar. On the other hand, the rotational relation is more complicated. The *moment of inertia* relates the angular momentum to the angular velocity. In this case it is represented by a three by three matrix, which specifies the inertia tensor with respect to the global coordinate frame.

In addition to the forces of particle mechanics, rotational forces, or *torques* are included in the model. Also, we have further divided the forces into *applied* and *constraint* forces, foreshadowing the introductions of the two kinds of connections defined in this model. The laws defined later will specify how they should be summed to form their respective resultants.

```
(applied-force          :domain R3   :synonyms F_a)
(applied-torque         :domain R3   :synonyms tau_a)
(constraint-force       :domain R3   :synonyms F_c)
(constraint-torque      :domain R3   :synonyms tau_c)
```

Finally, the following quantities are included for computational convenience. We have now departed from defining quantities necessary for specifying the semantics of a rigid body, and are including quantities that are used below in the laws, but are redundant semantically. They are included so that SIMLAB can associate data structures with the primitives in the simulator it creates. In later versions of SIMLAB it is possible that the semantic and computational quantities will be distinguished.

```
(E :domain R3x4)
(G :domain R3x4)
(local-coordinate-frame  :domain SO3  :synonyms lcf)
```

The matrices E and G are used in the laws below, and will be described when the laws are introduced. The *local-coordinate-frame* is used to allow objects (points, vectors, etc) to be represented in the coordinate frame of the rigid body. This means, for instance, that a point defined with respect to this coordinate frame will move with the rigid body. This matrix is used to define the hinge constraints below.

B.1 A rigid body's laws

```
(CONSTRAINTS
   (= v (deriv r time))
   (= a (deriv v time))
   (= (dot-product p p) 1.0)
```

The first two constraints define the relationships $v = \frac{dr}{dt}$ and $a = \frac{dv}{dt}$, while the third specifies that p, a quaternion, has unit length.

```
   (= omega (* 2 E (deriv p t)))
   (= omega-dot (deriv omega t))
   (= m (* rho geometry.volume))

  (= J (* rho
          (* lcf (* volumetric-moment-of-inertia
                    (transpose lcf)))))
   (= E (direct-sum (- (vector (ref p 1) (ref p 2) (ref p 3)))
                    (+ (tilde (vector (ref p 1) (ref p 2) (ref p 3)))
                       (* (ref p 0) (I3x3)))))
   (= lcf (coerce p (domain-of lcf)))
```

B.2 The *connections* between rigid bodies

As in the case of particle dynamics, and indeed all lumped models of physics, the interactions between rigid bodies are defined by defining *connections* between bodies. Connections constrain the values that the quantities of a primitive. In this case, the primitive is the rigid body, and the connections are kinematic constraints between the bodies. A kinematic constraint constrains the relative positions of two rigid bodies. Examples of kinematic constraints include prismatic joints (sliding joints, such as hydraulic cylinders), and hinges.

In order to show the inheritance relationship for connections, we first define a general kinematic constraint: the *hk-constraint.*[2] We then define the a more specific kinematic constraint, the *ball-and-socket.* The more specific connection inherits quantities and behavior (laws) from the more general, in the manner of object oriented languages, such as CLOS.

```
(connection hk-constraint
            (objects
             (object1  rigid-body)
             (object2  rigid-body))
            (quantities
             (force1  :domain R3)
             (force2  :domain R3)
             (torque1 :domain R3)
             (torque2 :domain R3))
            (constraints))
```

This connection, like the gravitational constraint defined in the particle model, constrains the quantities of two primitives, in this case two rigid bodies, *object1* and *object2.* Associated with each holonomic kinematic constraint is a set of *quantities* that represent the state of the interaction between *object1* and *object2.* The quantities *force1* and *torque1* represent the force (and torque) applied by *object1* on *object2* by this connection. Conversely, *force2* and *torque2* represent the force of *object2* on *object1.*

This connection has no constraints, but the empty (CONSTRAINTS) statement is included to remind the reader that the purpose of connections is to constrain the values of primi-

[2]The *hc-constraint* connections represent holonomic kinematic constraints – constraints which constrain the positions of the primitives by functions which are integrable with respect to time, i.e., do not depend on time derivatives of position such as velocity or acceleration.

tives. In this case, the connection defines the behavior of the *class* of holonomic kinematic constraints.

We next define the *ball-and-socket* connection. This connection inherits quantities from *hk-constraint* connection described above. In addition, it defines the constraints between the quantities of the objects it constrains.

```
(CONNECTION ball-and-socket (INHERITS-FROM hk-constraint)
   (QUANTITIES
    (c1 :domain (lcf object1))
    (c2 :domain (lcf object2))
    (lambda :domain R3))
   (CONSTRAINTS
    (= (+ (r object1) (* (lcf object1) c1))
       (+ (r object2) (* (lcf object2) c2)))))
```

This connection, in addition to the forces and torques (as well as *object1* and *object2*) that it inherits from *hk-constraint*, this connection contains three quantities, *c1*, *c2*, and *lambda*. *lambda* will contain the value of the Lagrange multiplier associated with the Lagrange multiplier method of solving holonomic constraints[10].

The quantities c1 and *c2* represent points on *object1* and *object2*, respectively. The domain of *c1* is `(lcf object1)`, which is the local coordinate frame of *object1*, as described above. This means that if the coordinates of *c1* are constant, c1 moves with *object1*. Similarly, the domain of *c2* is `(lcf object2)`.

Finally, the only constraint specified for ball and socket connections is the constraint `(= c1 c2)`, which means that the points c1 and c2 must be the same at all times. Since each of these points is in a constant position with respect to the object on which it is defined, this has the effect of constraining these points (c1 on *object1* and c2 on object2) to coincide. This results in a ball and socket joint.

A third kind of constraint in this model is the *applied-force* constraint, which models forces from outside the system (in SIMLAB terms, from outside the scene), such as the gravitational force of earth.

```
(CONNECTION applied-force
   (OBJECTS
      (object rigid-body))
```

```
(QUANTITIES
   (force :domain R3)))
```

Simply stated, this connection specifies that a force is applied to object1. A similar connection could be defined to represent a torque applied to an object.

At this point, we have defined all the primitives and connections in the rigid body model. The next step is to define the laws that apply to systems of rigid bodies.

The first law defines the resultant of the constraint forces on a given rigid body, for each rigid body in the scene.

```
(resultant_c
   (FOR o IN rigid-body
      (= (F_c o)
         (+ (SUM h IN hk-constraint SUCH THAT (= o (object1 h))
                  (force1 h))
            (SUM h IN hk-constraint SUCH THAT (= o (object2 h))
                  (force2 h))))))
```

That is, the law states that, for every rigid body o in the scene, the value of the resultant F_c (the resultant of all constraint forces on o) should be the sum of all holonomic kinematic constraints defined on o.

B.3 The rigid body dynamics model in its entirety

```
(define-model rigid-body-dynamics

   (required-models (basic-3D-geometry))

   (primitive rigid-body (inherits-from 3D-solid)

    (quantities

       (mass                  :domain R    :synonyms m :constant)
       (moment-of-inertia     :domain R3x3 :synonyms J)
       (position              :domain R3   :synonyms r)
       (velocity              :domain R3   :synonyms v)
       (acceleration          :domain R3   :synonyms a)
       (orientation           :domain unit-quaternion :synonyms p)
```

```
    (angular-velocity          :domain R3    :synonyms omega)
    (angular-acceleration      :domain R3    :synonyms omega-dot)
    (E                         :domain R3x4)
    (G                         :domain R3x4)
    (applied-force             :domain R3    :synonyms F_a)
    (applied-torque            :domain R3    :synonyms tau_a)
    (constraint-force          :domain R3    :synonyms F_c)
    (constraint-torque         :domain R3    :synonyms tau_c)
    (density                   :domain R     :synonyms rho :constant)
    (local-coordinate-frame    :domain SO3   :synonyms lcf))

  (constraints

    (= v (deriv r 'time))
    (= a (deriv v 'time))
    (= omega (* 2.0 E (deriv p 'time)))
    (= omega-dot (deriv omega 'time))
    (= m (* rho volume))
    (= J (* rho
            (* lcf (* volumetric-moment-of-inertia
                      (transpose lcf)))))
    (= E (direct-sum (- (vector (ref p 1) (ref p 2) (ref p 3)))
           (+ (tilde (vector (ref p 1) (ref p 2) (ref p 3)))
                       (* (ref p 0) (I3x3)))))
    (= G (direct-sum (- (vector (ref p 1) (ref p 2) (ref p 3)))
           (+ (- (tilde (vector (ref p 1) (ref p 2) (ref p 3))))
           (* (ref p 0) (I3x3)))))
    (= lcf (coerce p (domain-of lcf)))))



(connection hk-constraint
            (objects
             (object1  rigid-body)
             (object2  rigid-body))
            (quantities
             (force1                    :domain R3)
             (force2                    :domain R3)
             (torque1                   :domain R3)
             (torque2                   :domain R3))
            (constraints))


 (connection ball-and-socket
             (inherits-from hk-constraint)
```

```
            (quantities
             (c1                          :domain R3)
             (c2                          :domain R3)
             (lambda                      :domain R3))
            (constraints (= (+ (r object1) (* (lcf object1) c1))
                            (+ (r object2) (* (lcf object2) c2)))))


 (connection applied-force
            (objects
             (object   rigid-body))
            (quantities
             (force                       :domain R3)))

(laws
 (resultant_fc
  (for o in rigid-body
       (= (F_c o)
          (+ (sum h in hk-constraint
                    such that (= o (object1 h))
                (force1 h))
             (sum h in hk-constraint
                    such that (= o (object2 h))
                (force2 h))))))

 (resultant_tau_c
  (for o in rigid-body
       (= (tau_c o)
          (+ (sum h in hk-constraint
                    such that (= o (object1 h))
                (torque1 h))
             (sum h in hk-constraint
                    such that (= o (object2 h))
                (torque2 h))))))

 (resultant_fa
  (for o in rigid-body
       (= (F_a o)
          (sum h in applied-force
                   such that (= o (object h))
              (force h)))))

 (newton
  (for o in rigid-body
       (= (* (m o) (a o)) (+ (F_c o) (F_a o)))))
```

```
(euler
 (for o in rigid-body
      (= (+ (* (J o) (omega-dot o))
            (cross-product (omega o) (* (J o) (omega o))))
         (+ (tau_c o) (tau_a o)))))

(virtual-work
 (for h in hk-constraint
      (= (force1 h)
         (* (transpose (jacobian (constraints h) (r (object1 h))))
            (lambda h))))
 (for h in hk-constraint
      (= (force2 h)
         (* (transpose (jacobian (constraints h) (r (object2 h))))
            (lambda h))))
 (for h in hk-constraint
      (= (torque1 h)
         (* (* 0.5
               (e (object1 h))
               (transpose (jacobian (constraints h) (p (object1 h)))))
            (lambda h))))
 (for h in hk-constraint
      (= (torque2 h)
         (* (* 0.5
               (e (object2 h))
               (transpose (jacobian (constraints h) (p (object2 h)))))
            (lambda h))))))


(formulation-scheme unreduced-newton-euler
   (let motion-equations =
        (union (mapover (rigid-body (objects-of-type 'rigid-body))
                        (apply-law 'newton rigid-body))
               (mapover (rigid-body (objects-of-type 'rigid-body))
                        (apply-law 'euler rigid-body))
               (mapover (rigid-body (objects-of-type 'rigid-body))
                        (apply-law 'resultant_fc rigid-body))
               (mapover (rigid-body (objects-of-type 'rigid-body))
                        (apply-law 'resultant_tau_c rigid-body))
               (mapover (rigid-body (objects-of-type 'rigid-body))
                        (apply-law 'resultant_fa rigid-body))
               (mapover (connection (connections-of-type 'hk-constraint))
                        (apply-law 'virtual-work connection))))
   (let kinematic-constraints =
```

```
      (union
       (mapover (connection (connections-of-type 'hk-constraint))
                (constraint-equations-of-connection connection))))
(let other-constraints =
      (union
       (mapover (object (objects-of-type 'rigid-body))
                (constraint-equations-of-object object))))))
```

B.4 Sample output from automatically generated rigid body dynamics simulator

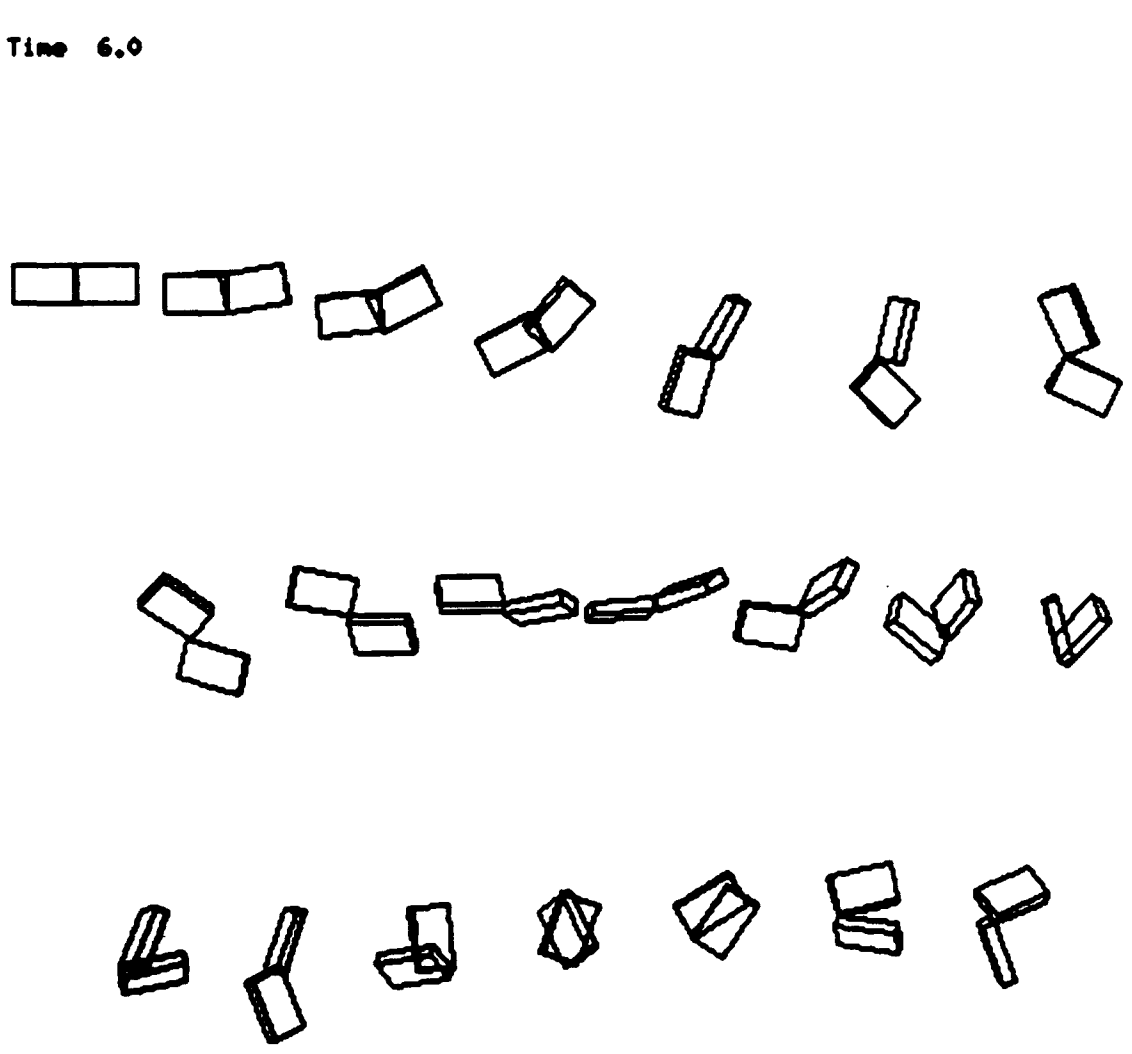
B.4.1 A block falling under gravity

In this example gravity is the only force. The rotation seen is due an initial angular velocity, as specified in the scene. The rotation axis then changes due to the Coriolis forces ($\omega \times J\omega$) term in the dynamics equations.

Time 4.5

B.4.2 A double pendulum

In this example we have two parallelepipeds, which 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.



C An AC circuit model

Lest the reader be concerned that SIMLAB is suitable only for creating mechanics simulators, we offer this simple model of an AC circuit. It's meaning is assumed to be more or less self evident. In this model we have chosen to make the "components" – the capacitors, resistors, etc. – connections, and the "nodes" primitives. Note that we could have as easily reversed these roles and created a model dual to this.

```
(define-model dual-ac-circuit

    (primitive node
       (quantities
          (voltage :domain R)))

    (connection 2-terminal
       (objects (n1 node)
                (n2 node))
       (quantities
        (current :domain R)
        (delta-V :domain R))
       (constraints
           (= delta-V  (- (voltage n1) (voltage n2)))))

    (connection capacitor (inherits-from 2-terminal)
       (quantities
          (capacitance :domain R)
          (charge :domain R))
       (constraints
          (= current (deriv charge 'time))
          (= delta-V (/ charge capacitance))))

    (connection resistor (inherits-from 2-terminal)
       (quantities
          (resistance :domain R))
       (constraints
          (= delta-V (* resistance  current))))

    (connection inductor (inherits-from 2-terminal)
       (quantities
          (inductance :domain R))
       (constraints
          (= delta-V (* inductance (deriv current 'time)))))
```

```
(connection ac-generator (inherits-from 2-terminal)
   (quantities
      (E-max :domain R)
      (omega :domain R))
   (constraints
      (= delta-V (* E-max (sin (* omega time))))))

(laws
   (current-law
      (for n in node
           (= 0.0
              (+ (sum terminal in 2-terminal
                      such that (= n (n2 terminal))
                      (current terminal))
                 (sum terminal in 2-terminal
                      such that (= n (n1 terminal))
                      (- (current terminal))))))))

(formulation-scheme ac-circuit
   (let equations =
      (union
        (mapover (node (objects-of-type 'node))
                 (apply-law 'current-law node))
        (mapover (connection (connections-of-type '2-terminal))
                 (constraint-equations-of-connection connection))))))
```

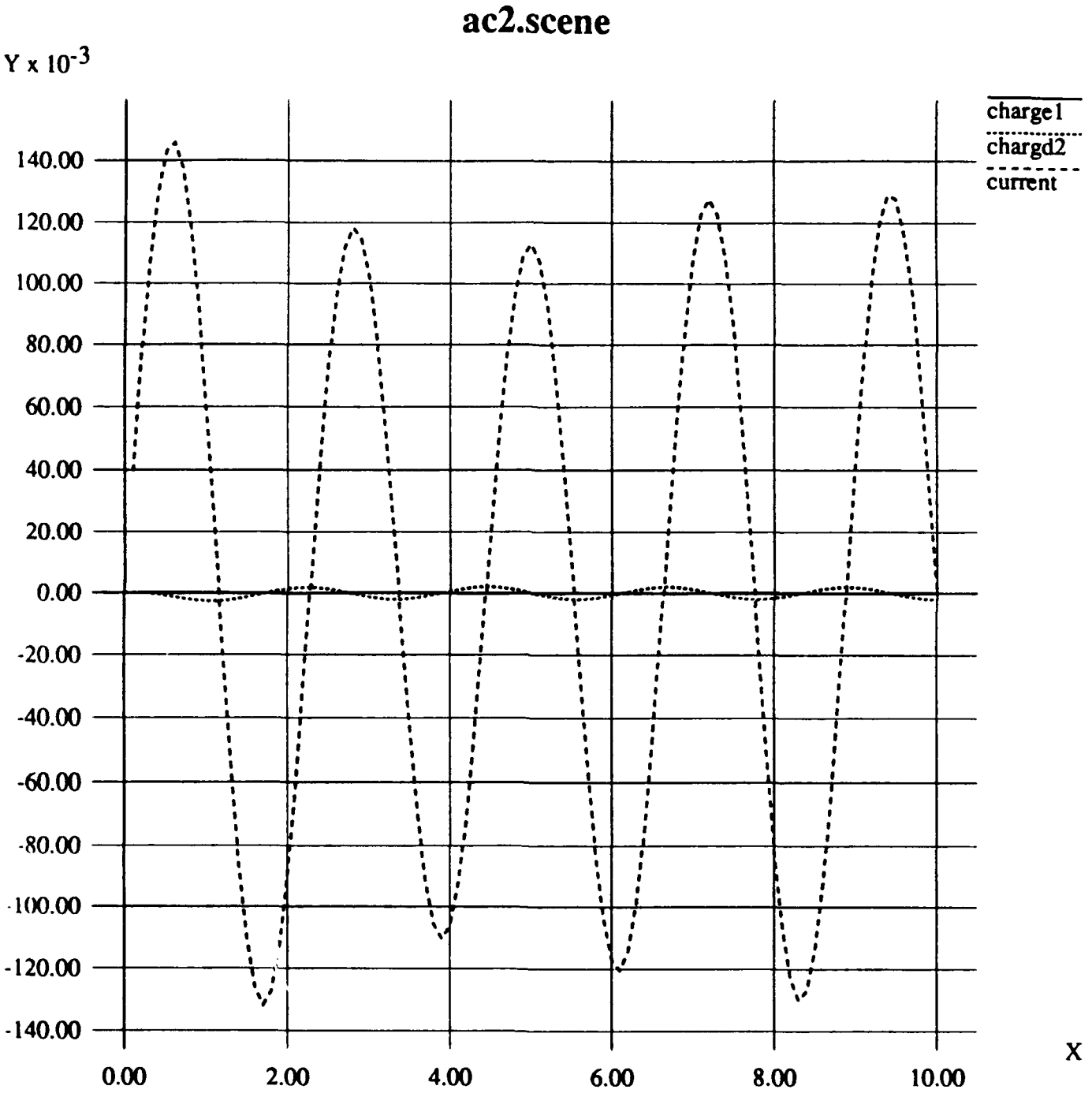
C.1 A circuit scene

This scene contains 4 nodes, an ac-generator, an inductor, a resistor, and two capacitors. A graph of the charges on the capacitors and the current through the inductor follows.

```
(define-scene

  (primitive node
     (name node1)
     (properties
      (voltage 1.0)))

  (primitive node
     (name node2)
     (properties))
```

```
(primitive node
   (name node3)
   (properties))

(primitive node
   (name node4)
   (properties))

(connection ac-generator
    (name generator1)
    (objects (n1 node1) (n2 node2))
    (properties
     (E-max 120.0)
     (omega 60.0)))

(connection capacitor
    (name capacitor1)
    (objects (n1 node2) (n2 node3))
    (properties
     (capacitance 0.001)))

(connection inductor
    (name inductor1)
    (objects (n1 node3) (n2 node4))
    (properties
     (inductance 0.001)))

(connection capacitor (name cap2)
    (objects (n1 node3) (n2 node4))
    (properties
     (capacitance 0.01)))

(connection resistor
    (name resistor1)
    (objects (n1 node4) (n2 node1))
    (properties
     (resistance 1000.0))))
```

C.2 The charge and current graphs



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] Gesellschaft für Mathematik und Datenverarbeitung mbH (German National Research Center for Computer Science), D-5205 Sankt Augustin 1, Federal Republic of Germany. *CLM — A Language Binding of Common Lisp and OSF/Motif User Guide and Manual*, 1991.

[8] 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.

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

[10] Cornelius Lanczos. *The Variational Principles of Mechanics.* Dover, New York, 1970.

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

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

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

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

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

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

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

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

[19] Guy L. Steele, Jr. *Common Lisp, the language.* Digital Press, second edition, 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 E. Zippel. The Weyl computer algebra substrate. Technical Report 90–1077, Department of Computer Science, Cornell University, Ithaca, NY, 1990.