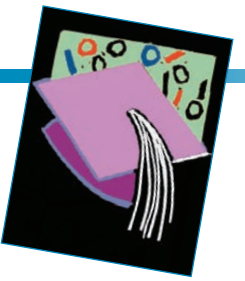
Editors: Michael Dennin, mdennin@uci.edu
Steven F. Barrett, steveb@uwyo.edu



COMPUTER-GUIDED SOLUTIONS TO PHYSICS PROBLEMS USING PROLOG

By Thomas J. Bensky and Catherine A. Taff

By posing a continual stream of pertinent questions, a nonmathematical computer program can prod freshman physics students toward an analytical solution to one-dimensional kinematics problems.

The use of computers in physics pedagogy is almost entirely focused on "parameter-based" software: students focus on a set of initial variables, which result in a subsequent output. For example, we can investigate the effects of launch angle, speed, and air drag (the parameters) on a projectile's motion[1] (the output) in several ways, including the PHeT Web-based projectile simulator (http://phet.colorado.edu), Physics Academic Software's Windows-based simulator (http://webassign.net/pas), or, for an example in Mathematica, Wolfram's Projectile Motion (http://demonstrations.wolfram.com/ProjectileMotion). However, despite vast differences in software form (symbolic, numeric, graphical, and so on), perhaps spanning generations of computer technology, the user experience remains fixed in this parameter-based mode, giving much of the physics pedagogical software the same "feeling." From a programming perspective, this commonality results from the procedural programming languages used to build the software. That is, the programmer identifies the problem to be solved, chooses a desired implementation scheme, and gives the computer specific step-by-step instructions on how to proceed, subject to some terminal condition(s).

Such software is fundamentally weak, however, where input parameters, data, or desired outcomes are not well defined.[2,3] Consider a typical end-of-chapter physics problem, such as problem 1.34 in Randall Knight's book: [4]

> A Porsche at rest accelerates from a stop light at 5 m/s^2 for 5 seconds, then coasts for 3 more seconds. How far has it traveled?

Or take problem 2.47 from a book by Douglas Giancoli:[5]

> A falling stone takes 0.30 s to travel past a window 2.2 m tall. From what height above the top of the window did the stone fall?

Any number of available software packages can numerically, symbolically, or graphically simulate such problems, reinforcing relevant concepts in the procedural programming domain. But there's another desirable software mode not being addressed. What about software that "understands" the laws of physics to the point where it can generate a textual, step-by-step, analytical solution to such problems? (By "analytical" we mean solutions that are found through direct, often symbolic, manipulation of the relevant physical laws, using algebra, trigonometry, and calculus.) This question motivated our work for two primary reasons.

First, there's a lack of software available to tutor students in this manner. Developing heuristic software is apparently an untapped research area in computational physics pedagogy. A literature search yielded volumes on "intelligent tutoring systems,"[6,7] but very little on solving physics problems.[8,9] (Mastering Physics is a homework submission system tied to many current physics textbooks; while it appears to offer heuristic-like help with end-of-chapter problems, it's not a generalized problem solver. See www.masteringphysics.com.)

Second, current physics texts continue to be written in an analytical fashion, with hundreds of problems calling for analytical solutions following hundreds of pages of analytical lessons. A disconnect appears to exist between parameter-based physics pedagogical software and the analytical emphasis of current physics texts.

We've developed a software tool that can generate a textual, step-by-step, analytical solution to an end-of-chapter physics problem without any prior knowledge of the problem itself. Our software has successfully generated solutions to one-dimensional kinematics (1DK) problems involving a single object—such as the Porsche and the stone in the problems above. The software works by asking questions about the problem and, based on the user's responses, offers instructions that lead to a solution—similar

to how an expert system works.[10,11] In short, our software simulates, at a simple level, the professor-student interaction that might occur during a one-on-one tutorial session.

Software Implementation

Procedural languages themselves stand as an enormous hindrance in developing the type of heuristic software that we propose. Such software is vague in terms of inputs and outputs and raises many questions: What is the student confused about? What is the nature of the problem? How should we present the problem's facts to the computer? What solution do we need? Attempting to code this software in a rigid, step-by-step procedural language would be unnecessarily labored. Clearly, we need another programming approach.

Why Prolog?

A long-ago interest in AI led to our awareness of Prolog and its ability to manipulate symbols while emphasizing a strict logical plan. Prolog enables an entirely different approach to computer programming, using a declarative rather than procedural method. The declarative method doesn't require the programmer to supply the computer with step-by-step instructions. Instead, we must present rules to satisfy and data to be tested against those rules. The Prolog "engine" then works to find relationships among the data that are consistent with the rules. Already this sounds a lot like how we solve a physics problem.

Luckily, in the limited domain of freshman-level 1DK problems, the text of each problem always contains sufficient logical connections to enable a step-by-step analytical solution. This approach doesn't perform any numeric or mathematical computation, nor does it deliver a specific result; it's decidedly nonparameter-based. Instead, our approach focuses on three key elements for analytically solving a physics problem:

- understanding the proper interrelationships between kinematic variables,
- eliciting all possible knowledge from the problem text, and
- organizing knowledge learned along the way.

Figure 1 shows a session with our software, in which a student solves the "Porsche" problem above (the student's input is shown in bold).

We wrote the Prolog program to contain the rules, which are essentially the 1DK equations.[4] The student provides the data that the rules are applied to when prompted by the software; the data itself emerges from facts and other clues found in the problem text. The software never requires specific numbers. Instead, it works with either yes/no questions (such as, "Do you know the acceleration of the Porsche?") or short-answer questions (such as, "Where is the Porsche when it starts?" for which the answer might be "at the traffic light"). This data set can grow internally as the program learns things along the way. Prolog is uniquely suited to these tasks, as it's rule-based, with built-in database capabilities. The rule-based character provides a natural structure for translating the laws of physics (the rules) into the computer, while the database element permits careful knowledge organization as the solution unfolds. The entire program—which can lead to a solution for any freshman-type, single-object, 1DK problem—consists of approximately 100 lines of Prolog code.

How Prolog Works

Prolog reached its most widespread adoption in the late 1980s as the language of "artificial intelligence," but it never went mainstream. It has largely become an academic computer language, although several commercial implementations exist including Visual Prolog (www.visual-prolog.com), Quintus Prolog (www.sics.se/quintus), Logic Programming Associates Prolog (www.lpa.co.uk), and Arity Prolog (www.arity.com). There are two open-source Prologs: Gnu Prolog (www.gprolog.org) and Swi Prolog (www.swi-prolog.org), which we used in our work. Here, we provide a brief synopsis of the language; interested readers can refer to the books *Programming in Prolog*[12] or *Prolog Programming for Artificial Intelligence*[13] for more details.

Prolog is a remarkably simple language with almost no formal constructs. At its core, Prolog has a string, called an *atom*, that can't be further processed. An atom is a sequence of lowercase letters, such as `position`, `acceleration`, or `car`. From atoms, developers can build predicates. For example, if we know a car's position in a physics problem, we might declare a `know` predicate which, when connected with the car's position, would be `know(car,position)`. This predicate is a valid Prolog statement, ending in a period, and represents the most basic representation of knowledge about a system (that is, "we know the position of the car.").

From predicates, we can define rules, as in `can_find(X,v):-know(X,v0), know(X,a), know(X,duration_a_is_applied)`. This is a valid Prolog rule; it states that we "can find" v (the speed) of object X if (`:-` means "if") we know both X's v_0 (initial speed) and a (acceleration), as well as the time over which a was applied to the object

```
?- solve(porsche).
Do you know about  a  time interval that involves the  porsche ?  y.
What is the  porsche  doing at the start of this time interval? at_rest.
What is the  porsche  doing at the end of this time interval? going_fast.
Do you know the acceleration of the  porsche  during the time interval from  at_rest  to  going_fast ?  y.
Do you know the speed of the  porsche  when  at_rest ?  [y/n]: y.
Do you know the speed of the  porsche  when  going_fast ?  [y/n]: n.
Do you know the position of the  porsche  when  at_rest ?  [y/n]: y.
Do you know the position of the  porsche  when  going_fast ?  [y/n]: n.
Is it possible that an event in this list is closely connected to (or is actually the same as) another event?
[at_rest, going_fast][y/n]: n.
Using just a few words, can you say anything else that the  porsche  does in this problem?  [y/n]: y.
Ok,what?  coasts.

Using  x=x0+v0*dt+1/2*a*dt^2  you can find  x(porsche, going_fast)  of the  porsche  by using
x(porsche, going_fast) x0(porsche, at_rest) v0(porsche, at_rest) a(porsche, at_rest, going_fast) dt(porsche, at_rest,
going_fast)

Using  v=v0+a*dt  you can find  v(porsche, going_fast)  of the  porsche  by using
v(porsche, going_fast) v0(porsche, at_rest) a(porsche, at_rest, going_fast) dt(porsche, at_rest, going_fast)

You know about a time interval between  at_rest  and  going_fast
What caused the  porsche  to  coasts ?  no_acceleration.
What happens to the  porsche  because of the  coasts ?  went_farther.
Do you know the acceleration of the  porsche  during the time interval from  no_acceleration  to  went_farther ?  y.
Do you know the speed of the  porsche  when  no_acceleration ?  [y/n]: n.
Do you know the speed of the  porsche  when  went_farther ?  [y/n]: n.
Do you know the position of the  porsche  when  no_acceleration ?  [y/n]: n.
Do you know the position of the  porsche  when  went_farther ?  [y/n]: n.
Do you know how long it took for the  porsche  to go from  no_acceleration  to  went_farther ?  [y/n]: y.
Is it possible that an event in this list is closely connected to (or is actually the same as) another event?
[at_rest, no_acceleration, going_fast, went_farther][y/n]: y.
Which event leads into the other (or comes first)?  going_fast.
And which event then takes over?  no_acceleration.

Using  v( porsche , going_fast )=v0( porsche , no_acceleration )  you can find  v0(porsche, no_acceleration)  of the  porsche
by using
v(porsche, going_fast) v0(porsche, no_acceleration)

Using  x( porsche , going_fast )=x0( porsche , no_acceleration )  you can find  x0(porsche, no_acceleration)  of the  porsche
by using
x(porsche, going_fast) x0(porsche, no_acceleration)

You know about a time interval between  at_rest  and  going_fast
You know about a time interval between  no_acceleration  and  went_farther
Do you know about  any other  time interval that involves the  porsche ?  n.
Do you know the speed of the  porsche  when  went_farther ?  [y/n]: n.
Do you know the position of the  porsche  when  went_farther ?  [y/n]: n.
Is it possible that an event in this list is closely connected to (or is actually the same as) another event?
[at_rest, went_farther][y/n]: n.
Using just a few words, can you say anything else that the  porsche  does in this problem?  [y/n]: n.

Using  x=x0+v0*dt+1/2*a*dt^2  you can find  x(porsche, went_farther)  of the  porsche  by using
x(porsche, went_farther) x0(porsche, no_acceleration) v0(porsche, no_acceleration) a(porsche, no_acceleration, went_farther)
dt(porsche, no_acceleration, went_farther)

Using  v=v0+a*dt  you can find  v(porsche, went_farther)  of the  porsche  by using
v(porsche, went_farther) v0(porsche, no_acceleration) a(porsche, no_acceleration, went_farther) dt(porsche, no_acceleration,
went_farther)
```

Figure 1. The software's input and output for the "Porsche" problem. The student's input is shown in bold.

(the "duration a is applied"). The uppercase "X" denotes a variable in Prolog, and we use it here because it's a general fact of physics: we can find any object's speed if we know its initial speed, acceleration, and the time duration over which the acceleration was applied.

Prolog attempts to satisfy such rules (in this case, `can_find`) by attempting to satisfy each predicate (or subgoal) of the rule, working from left to right. If a given predicate evaluates to `true`, Prolog proceeds rightward to the next predicate. If a given predicate fails, then Prolog will backtrack to the left, predicate by predicate, attempting to resatisfy each in a different way until it can again proceed to the right. Predicates are satisfied by using the facts in the internal database or by evaluating other rules. A rule succeeds (evaluates to `true`) if Prolog reaches the period ending the rule. A rule fails (evaluates to `false`) if Prolog backtracks to the left of the `:-` symbol (to the rule head) without satisfying the entire rule. If more than one version of a rule is given, attempts to satisfy the versions will proceed in the order in which they appear.

Programming in Prolog then, consists of carefully constructing rules that uphold an idea's semantics when exposed to incoming data. This highlights the key difference between declarative and procedural programming. In its declarative mode, Prolog relentlessly searches for "solutions" to rules based on other rules and any available data. Procedural languages follow a strict path through the given code until terminal conditions are met.

Prolog and 1D Kinematics Problems

To develop a physics-problem solver in Prolog, we first needed a framework.

Kinematic Equations

All 1DK problems in a freshman-level textbook (such as Knight's) can be solved with two equations:

$$x = x_0 + v_0\Delta t + \frac{1}{2}a\Delta t^2, \qquad (1)$$

and

$$v = v_0 + a\Delta t, \quad (2)$$

where the usual variable meanings apply.[4] In forming a program design, we need a strict (and ever-obvious) interpretation of these equations as follows.

Because these equations allow for advancing x_0 and v_0 forward in time, identification of the time interval Δt is of primary importance. Typical 1DK problems might include several different time intervals, each requiring specific identification and ordering to develop a solution. The program can specify these kinematic variables only after these time intervals are clearly defined. For example, the initial position of an object, x_0, is strictly the object's position at the time interval's beginning, and x is its position at the time interval's end. Analogous meanings hold for v_0 and v. It assumes acceleration, a, is constant and acts over the entire time interval. Unless they're properly connected to some time interval, these kinematic variables can play no logical role in a solution.

Prolog's natural fit with our endeavor is now apparent. We program Equation 1 as a rule into Prolog via the construct

```
  physics_law(Object,
    [x(Object,Tend),
x0(Object,Tstart),
  v0(Object,Tstart),
a(Object,Tstart,Tend),
  dt(Object,Tstart,Tend)],
'x=x0+v0*dt+1/2*a*dt^2').
```

Here `physics_law` is a Prolog rule, containing three parameters, an `Object` variable, a list of kinematic variables, and textual advice for the student. In this rule, the kinematic variables are tied to an object `Object` and a time interval ($\Delta t = t_{end} - t_{start}$) via `Tstart` and `Tend`. Prolog's built-in pattern-matching capabilities let this rule succeed only if the pattern of kinematic variables—which are linked carefully to a time interval—are matched to known data. To see how this works, consider Prolog's attempt to unify the first term in the `physics_law` rule's list above: `x(Object,Tend)`. In its knowledge base, Prolog might, for example, find `x(porsche,top_speed)`, meaning "we know the position of the Porsche when it's reached its top speed." This find will instantiate `Object` to `porsche` and `Tend` to `top_speed` for all subsequent terms in the rule. When it comes to `x0(Object,Tstart)`, `Object` has already been instantiated to `porsche`, so it searches for terms matching `x0(porsche,Tstart)`, with `Tstart` as yet uninstantiated. It might find `x0(porsche,at_rest)` instantiating `Tstart` to `at_rest`. Thus, `Object`, `Tstart`, and `Tend` are now instantiated and will remain so for the rest of the terms in the rule. Failure of any subgoal will force backtracking, in which case the program will seek alternative solutions to the `x0` and `x` goals.

We again emphasize the organizational structure here. The variable x can occur only at the end of a time interval, which is denoted here by the time when the Porsche has attained `top_speed`, according to the student. This end-time boundary must match any other end-time boundaries needed by other kinematic variables if this rule as a whole is to succeed. The same applies for kinematic variables involving start-time boundaries. In traditional "paper and pencil" format, this rule would be written as

$$x_{porsche,top\_speed} = x_{porsche,at\_rest} + v_{porsche,at\_rest}\Delta t_{top\_speed \rightarrow at\_rest} + \frac{1}{2} a_{top\_speed \rightarrow at\ rest}\Delta t_{top\_speed \rightarrow at\_rest}. \quad (3)$$

It's true that there's excessive subscripting, but it's this specific term-by-term organization that lies at the core of our work. In practice, the `physics_law` rule succeeds if any three of the four variables in its kinematic-variable list are known. The fourth might then be derived (by the student) using the kinematic equation the rule represents (as the software will advise). A similar written rule exists for Equation 2.

Surprisingly, this completes our kinematic equations programming into the Prolog program. Prolog's built-in inference engine handles the rest, relentlessly searching all available data as it attempts to find rule versions for which all but one of the variables is known. At that point, the unknown is assumed computable and added to the problem's knowledge base.

Time-Interval Identification

In the Porsche problem, there are two time intervals (when the Porsche is accelerating and when it's coasting). The stone problem also has two time intervals (when the stone is falling in front of, and toward, the window from above). Because of the time intervals' importance, the software aggressively attempts to learn about them as soon as possible. It does so by asking the student what the objects "are doing" at the start and end of such intervals. To answer these questions, the student is forced to examine a given object's actions and devise descriptions of what seems to occur at the beginning and end of each time interval. This effectively defines "time bounds" for the time interval.

In 1DK problems, there are two types of time intervals. The first are explicitly stated, as in the Porsche and stone problems above. The Porsche accelerates for "5 seconds" and the stone

falls for "0.3 seconds." As Figure 1 shows, the Porsche's explicit time interval is recognized and bounded by times when the Porsche is "at rest" and "going fast." This tells the software that we know a fact called `dt(porsche,at_rest,going_fast)` or $\Delta t_{at\_rest \rightarrow going\_fast}$. To the software, this means the student knows an actual value of Δt and can use it in a later computation. For the stone, the 0.3 seconds is bounded by expressions such as "top of window" and "bottom of window."

The second type of time interval is not explicitly stated, but is known to exist via a particular set of actions that an object takes or imposes. For example, in the stone problem, there's a time interval during which the stone is falling toward the window, but it's not known how long this interval lasts. However, even without knowledge of the time interval's magnitude, the student can supply descriptive time interval boundaries if the program asks "if they can say anything else about what the stone does in the problem."

The student can recognize that, for a part of the problem, the stone "falls toward the window," which is a time interval bounded by the acts of "being released by a hand" and "reaching the top of the window." With this knowledge, the software could discover a `dt(stone,released_by_hand,top_of_window)` (or $\Delta t_{released\_by\_hand \rightarrow top\_of\_window}$), which currently has no known magnitude. In the Prolog code, this is handled by a modified form of the `physics_law` construct that has the rule `dt_bounds(Object,Description,Tstart,Tend)` (as in Δt "boundaries") in place of `dt(Object,Tstart,Tend)`. The former notion is always forced to fail as a Prolog fact because the student can't compute with a time interval of unknown magnitude. The failing `dt_bounds` predicate instead triggers the software into asking specific questions about the time interval, with the goal of finding the time interval's magnitude from the kinematic equations. Such questions involve asking the student whether a is known during the time interval, if x or v are known at the time interval's end, and if x_0 or v_0 are known at the time interval's start (alternatively, it could search its internal database for the same information). After gathering such information, the software sees whether a Δt is computable from Equation 1 or 2; if so, it's added to the problem's knowledge base.

Time-Interval Sequencing and Connections

At any given point in the problem, several textual descriptions might accumulate in the database, linked to the time intervals' beginnings and ends. Connecting and sequencing such expressions is another critical step toward an analytical solution. In the Porsche problem (Figure 1), the acceleration time interval ends with the description `going_fast`, while the coasting time interval begins with `no_acceleration`. Although different descriptions, these times actually represent the same time instant. The problem's story line would indicate that the `no_acceleration` just precedes the `going_fast`. In other words, the Porsche begins its coasting just as it ends its acceleration.

Likewise, the stone is seen to be "reaching the top of the window" (after being released) just as its position is known to be at the "top of window." The software constantly trolls its database for such descriptions and asks the student if any of them actually represent the same physical time instant. If it knows two time expressions are equivalent—for example t_1 and t_2, with (t_1 just preceding t_2 in the storyline)—then the software can immediately connect the two respective time intervals via kinematic variables, as in

$$x_0(t_2) = x(t_1) \tag{4}$$

and

$$v_0(t_2) = v(t_1). \tag{5}$$

These equations state that initial parameters in x or v at the beginning of the later time interval can be found from the final like-parameters at the previous time interval's end. We represent Equation 4 as

```
  physics_law(Object,
    [x(Object,Tend),
x0(Object,Tstart)],
  ['x (',Object,',',Tend,')
= x0 (',Object,',',
  Tstart,')']) :-
adjacent times(Object,Tend,
  Tstart),
```

where the rule `adjacent_times` succeeds if two times (`Tstart` and `Tend`) are found in the database and known to be related. Such work is left up to the Prolog engine: the code specifies only a required pattern between any `x` and `x0` (or `v` and `v0`). If one of the two terms is known, the other can be found and added to the database, with instructions given to the student. Knowledge of `adjacent_times` facts comes directly from the student.

Kinematic Variables

With clearly identified time intervals in hand, the software can now ask about specific kinematic variables in their proper context. For example, if it

knows the time interval magnitude—either directly or in a derivable sense—between events `Tend` and `Tstart`, it can pose a question about *v*, a kinematic variable that can exist only at the end of a given time interval. The Prolog code

```
  question(Object,v (Object,
    Tend)) :-
maybe_dt(Object,_,Tend),
  not(v(Object,Tend)),
affirm(['Do you know the speed
  of the
',Object,' when ',Tend,'? '])
```

will handle such a question. The `maybe_dt` rule succeeds if the database contains either an explicit or derivable time interval magnitude. The underbar (_) in this line is the Prolog "anonymous" or "don't care" variable. Because the code is asking about *v*, we're not concerned with the time interval's beginning, hence the placement of only `Tend` in the code. If for example, `Object` instantiates to `stone` and `Tend` to `reaching_top_of_window`, the question will read "Do you know the speed of the stone when reaching_top_of_window?" (We fine the broken, grammatically incorrect yet understandable sentences a charming aspect of AI applications.) The `affirm` predicate used to pose the question succeeds only if the student answers "yes" to the question.

Such questions are also posed for v_0, x_0, x, and a. If any of these rules succeed, then the program inserts this kinematic variable knowledge—which is carefully tied to an object and time interval—into the Prolog database as newly gained knowledge.

Other Components

Several lines of code form the software's core engine. These lines continually drive the software to ask questions, make database assertions, and display any instructions to the student. The software is started with a rule called `solve`, which takes a single object name as its parameter—as in `solve(porsche)` or `solve(stone)`. This rule starts by extracting questions to ask from any available question predicates and updates the database for questions with affirmative answers or those that might generate new time intervals data.

The rule `check_laws` is called by `solve` after the program asks appropriate questions. It sequences through all `physics_laws` rules attempting to find one that succeeds (with all but one of its required variables known). A subgoal of `check_laws`, called `check_terms`, actually checks each kinematic variable's availability against the database and returns the one that is unknown, which might now be a known (derivable) quantity.

In practice, the software might terminate without the student finding an answer. If so, it's run again (and again) as necessary, forcing the student to rethink previous answers or answer new questions, until he or she unearths enough data from the problem to find what it requires. Repeated runs are generally different, as run N can benefit from data asserted into the database in run $N - 1$. We generated Figure 1 by running the software three times, with the answer finally appearing near the bottom, in the line describing how `x(porsche,went_farther)` is found.

Running the Software

The complete Prolog code for our work is available at http://ocean.physics.calpoly.edu/prolog. You can download the code, `phys.pl`, and the supporting input/output routines, `io.pl`, and save them in a common folder. With Swi Prolog installed, you can load and compile the code by typing `[phys].` in response to Prolog's `?-` prompt. You can then run the software by typing `solve(porsche).` or `solve(stone).` following the next `?-` prompt, to start the program working on what's computable for that object. Finally, you can rerun the software as needed—via additional `solve(porsche).` or `solve(stone).` entries—at each subsequent `?-` prompt.

Prolog's input/output interface is extremely fragile. All "yes/no" responses must be a single, lowercase "y" or "n" followed by a period (that is, `y.` or `n.`) and the return key. All short-response inputs must be in the form of a valid Prolog atom construct, followed by a period—that is, all lowercase letters, with no spaces or symbols other than the underbar (_), such as `at_the_top_of_the_window.` (with a period at the end).

Although we could have written similar software using a procedural language, Prolog's natural ability to handle symbolic information in a strict logical setting makes it a compelling platform for this work. We're particularly impressed with the pattern-matching ability of Prolog's inference engine, yet find it ever-difficult to write code that maximally exploits this after a lifetime of procedural programming. We're also intrigued by Prolog's `call` predicate, which allows data to be executed as code; this enables compact representation of the `physics_laws` by allowing each member of a `physics_law` list (that is, each data element) to be executed as a Prolog query (code).

Our future work includes adapting this code into a Web-ready form, so students can access it via a Web browser. In addition, we plan to adapt the code to handle kinematics problems involving more than one object, such as "two trains passing each other" and so on. We also see potential for this software to help grade-school students with the dreaded "story problems"—that is, the word problems involving age, money, simple motion, and so on that often stand as a barrier between those who "get math" and those who don't.

References

1. W. Christian and M. Belloni, *Physlets: Teaching Physics with Interactive Curricular Material*, Prentice Hall, 2001.
2. T. Khannan, *Foundations of Neural Networks*, Addison Wesley, 1990.
3. J.S.R. Jang, C.T. Sun, and E. Mizutani, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*, Prentice Hall, 1997.
4. R.D. Knight, *Physics for Scientists and Engineers*, Pearson, 2004.
5. D.C. Giancoli, *Physics: Principles with Applications*, 4th ed., Prentice Hall, 1995.
6. E.H. Houstis and J.R. Rice, eds., *Artificial Intelligence, Expert Systems and Symbolic Computing*, Elsevier Science, 1992.
7. E. Wenger, *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge*, Morgan Kaufmann, 1987.
8. A. Bundy et al., "Solving Mechanics Problems Using Meta-Level Inference," *Proc. 6th. Int'l J. Conf. Artificial Intelligence*, Univ. Press, 1979; http://dli.iiit.ac.in/ijcai/IJCAI-79-VOL2/PDF/002.pdf.
9. M. Schmidt and H. Lipson, "Distilling Free-Form Natural Laws from Experimental Data," *Science*, Vol. 324, no. 5923, pp. 81-85, April 3, 2009.
10. N.C. Rowe, *Artificial Intelligence Through Prolog*, Prentice Hall, 1988.
11. M.A. Covington, D. Nute, and A. Vellino, *Prolog Programming in Depth*, Prentice Hall, 1996.
12. W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, 3rd ed., Springer-Verlag, 1987.
13. I. Bratko, *Prolog Programming for Artificial Intelligence*, 2nd ed., Addison-Wesley, 1991.

Thomas J. Bensky is an associate professor of physics at California Polytechnic State University in San Luis Obispo, California. His research interests include optics of the ocean and the use of computer graphics and artificial intelligence in undergraduate education. Bensky has a PhD in physics from the University of Virginia. Contact him at tbensky@calpoly.edu.

Catherine A. Taff is a mission systems engineer at The Boeing Company. Her research interests are in computer modeling and simulation. Taff has an MS in physics from San Francisco State University. Contact her at ctaff@sfsu.edu.

TIMELY, ENVIRONMENTALLY FRIENDLY DELIVERY

DIGITAL EDITIONS

Subscribe to the interactive digital versions of *Computer* and *IEEE Security & Privacy*, and access the latest news and information whenever and wherever you want it.

Computer

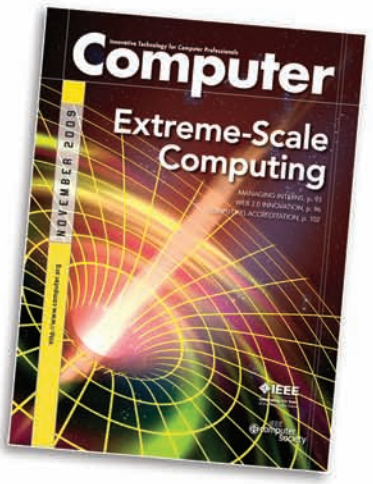
The IEEE Computer Society's flagship publication, *Computer* magazine publishes peer-reviewed technical articles on all aspects of computer science, computer engineering, technology, and applications.

Industry professionals, researchers, and managers rely on *Computer* to keep current on research developments, trends, best practices, and changes in the profession.

Upcoming theme issues include:

- Extreme-scale computing,
- Multi- and many-core,
- Biometric identification, and
- Nano-architecture.

To see what you're missing, check out selected *Computer* articles for free in Computing Now, and then subscribe to the digital edition to get full access right away.



$29.95
for 12 issues!

IEEE Security & Privacy

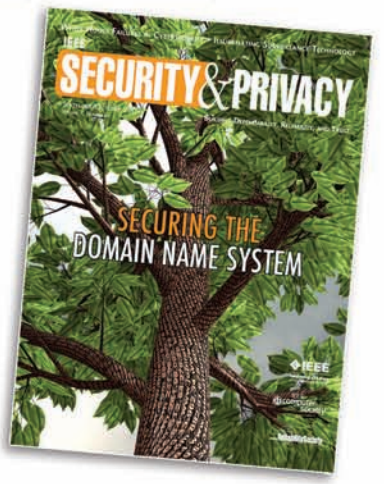
IEEE Security & Privacy brings together the practical and the leading edge advances in security, privacy, and dependability.

IEEE Security & Privacy covers and influences policy in the enterprise and the government—from basic training and attack trends to the US's cyberattack policy and telephone wiretapping, *S&P* brings guidance from some of the leading thinkers in the field. Bruce Schneier, Steve Bellovin, Gary McGraw, and Mike Howard have you in mind when writing their columns!

Upcoming theme issues include:

- The insider threat,
- Mobile device security, and
- The security and privacy of cloud computing.

Sample free *IEEE Security & Privacy* articles and the Silver Bullet podcast series from Computing Now, and subscribe to the digital edition today.



$19.95
for 6 issues!

The latest content at your fingertips within minutes.

Email notification. Receive an alert as soon as each digital edition is available. Links will take you directly to the enhanced PDF edition OR the web browser-based edition.

Quick access. Download the full issue in less than two minutes with a broadband connection.

Convenience. Read your digital edition anytime -- from your home PC, at work, or on your laptop while traveling.

Digital archives. Subscribers can access the virtual archive of digital issues dating back to Jan./Feb. 2007.

To subscribe, go to: computer.org/digitaleditions

IEEE computer society