Chapter 1

# PROBLEM SOLVING ENVIRONMENTS AND SYMBOLIC COMPUTING

Richard J. Fateman
*University of California, Berkeley*

**Abstract** What role should be played by symbolic mathematical computation facilities in scientific and engineering "problem solving environments"? Drawing upon standard facilities such as numerical and graphical libraries, symbolic computation should be useful for: The creation and manipulation of mathematical models; The production of custom optimized numerical software; The solution of delicate classes of mathematical problems that require handling beyond that available in traditional machine-supported floating-point computation. Symbolic representation and manipulation can potentially play a central organizing role in PSEs since their more general object representation allows a program to deal with a wider range of computational issues. In particular Numerical, graphical, and other processing can be viewed as special cases of symbolic manipulation with interactive symbolic computing providing both an organizing backbone and the communication "glue" among otherwise dissimilar components.

**Keywords:** PSE, "computer algebra", CAS, optimization, "symbolic computation" Mathematica, Maple, Macsyma, Axiom, glue

## 1. INTRODUCTION

We can point to many examples of complete and successful "user-friendly" problem-solving **programs** where the problem domain is sufficiently restricted. Consider the multiple-choice, non-programmable user interface of an Automatic Teller Machine: you state your queries, and the ATM demonstrates expert responses. Achieving even modest success in a more ambitious setting such as solving engineering problems in a broad class is considerably more challenging. This leads us to the evolving concept of a "problem solving environment" (PSE [13]). The notion is to computationally support a user in defining a problem clearly, help search for its solution, and understand that

solution[1]. This paper explores some concepts and tools available in symbolic computation that appear appropriate for PSE construction.

The search for tools to make computers easier to use is as old as computers themselves. In the few years following the release of Algol 60 and Fortran, it was thought that the ready availability of compilers for such high-level languages would eliminate the need for professional programmers — instead, all educated persons, and certainly all scientists, would be able to write programs whenever needed.

While some aspects of programming have been automated, deriving programs from specifications or models remains a difficult step in computational sciences. Furthermore, as computers have gotten substantially more complicated it is more difficult to get the fastest performance from these systems. Advanced computer programs now depend for their efficiency not only on clever algorithms, but also on constraining patterns of memory access. The needs for advanced error analysis has also grown as more ambitious computations on faster computers combine ever-longer sequences of computations, potentially accumulating and propagating more computational error. The mathematical models used in scientific computing have also become far more sophisticated.

Symbolic computation tools (including especially computer algebra systems) are now generally recognized as providing useful components in many scientific computing environments.

## 1.1 SYMBOLIC VS. NUMERIC

What makes a symbolic computing system distinct from a non-symbolic (or numeric) one? We can give one general characterization: the questions one asks and the resulting answers one expects, are irregular in some way. That is, their "complexity" may be larger[2] and their sizes may be unpredictable. For example, if one somehow asks a numeric program to "solve for $x$ in the equation $\sin(x) = 0$" it is plausible that the answer will be some 32-bit quantity that we could print as $0.0$. There is generally no way for such a program to give an answer "$\{n\pi \mid \mathrm{integer}(n)\}$". A program that *could* provide this more elaborate symbolic, non-numeric, parametric answer dominates the merely numerical

[1] We can also consider meta-PSE: a PSE for programmers of PSEs. Such a meta-PSE would help the scientific programmer, as well as the designer of interfaces and other components, to put together a suitable "application PSE."

[2] Although some numeric programs deal with compound data objects, the complexity of the results are rarely more structured than 2-dimensional arrays of floating-point numbers, or perhaps character strings. The sizes of the results are typically fixed, or limited by some arbitrary maximum array-size in Fortran COMMON allocated at "compile-time".)

from a mathematical perspective. The single numerical answer might be a suitable result for some purposes: it is simple, but it is a compromise.

If the problem-solving environment requires computing that includes asking and answering questions about sets, functions, expressions (polynomials, algebraic expressions), geometric domains, derivations, theorems, or proofs, then it is plausible that the tools in a symbolic computing system will be of some use.

## 1.2 A SELECTED HISTORY OF CAS

This decade has seen a flurry of activity in the production and enhancement and commercial exploitation of computer algebra systems (CAS), but the field has roots going back to the earliest days of computing.

As an easily achievable early goal, the computer system would be an interactive desk-calculator with symbols. At some intermediate level, a system would be a kind of "robotic graduate student equivalent": a tireless, algebra engine capable of exhibiting some limited cleverness in constrained circumstances. The grand goal would be the development of an "artificially intelligent" automated problem solver, or the development of a "symbiotic" human-machine system that could be expert analyzing mathematical models or similar tasks [25].

W. A. Martin [26] suggested that by continuing to amass more "mathematical facts" in computer systems, we would eventually build a reasonably complete *problem solver*: one that can bridge the gap between a problem and a solution – including some or all of the demanding intellectual steps previously undertaken by humans. Indeed, Martin estimated that the 1971 Macsyma CAS contained 500 "facts" in a program of 135k bytes. He estimated that an increase of a factor of 20 in facts would make the system "generally acceptable as a mathematical co-worker" (a college student spending 4 hours/day 5 days/week 9 months/year for 4 years at 4 new facts an hour, forgetting nothing, would have about 12,000 facts). Although Martin did not provide a time line for these developments, it is clear that the *size* of the programs available today far exceed the factor of 20. Yet however clever and useful these packages, 30 years later none is likely to be considered "generally acceptable as a mathematical co-worker."

Martin's oversimplification does not account for the modeling of the workings of a successful human mathematician who manages to integrate "facts" (whatever they may be) into a synthesis of some kind of model for solving problems.

The currently available CAS, in spite of the occasional sales hyperbole, are in reality far less ambitious than Martin's "artificial mathematician". Yet they are more ambitious in scope than a symbolic calculator of algebra. They are already billed as scientific computing environments, including symbolic and numeric algorithms, elaborate graphics programs, on-line documentation,

menu-driven help systems, access to a library of additional programs and data, etc.

Will they reach the level of a clever graduate student? I believe there are barriers that are not going to be overcome simply by incremental improvements in most current systems. *A prerequisite for further progress in the solution of a problem is a faithful representation of all the features that must be manipulated.* Lacking such abstraction makes manipulation of concepts difficult or impossible.

What can be expected in the evolution of CAS? The last few decades of system builders, having succeeded in collected the "low hanging fruit" of the novel (symbolic) computations, have stepped back from some of the challenges and are instead addressing "the rest" of scientific computing. *The grand goal is no longer to simulate human mathematical problem solving. The goal is to wipe out competing programs!* This requires providing a comfortable environment that allows the user to concentrate on solving the necessary intellectual tasks while automatically handling the inevitable but relatively mindless tasks—including transporting and storing files, converting data from one form to another, and executing tedious algebra and number crunching[3].

For the coming decade, we predict that ambitious CAS builders will redouble their efforts to envelop all modes of scientific computation. We already see numerical systems such as Matlab and MathCAD with added symbolic capability kits. Computer algebra systems such as Macsyma, Maple, Mathematica, and Axiom [7] have added more-efficient numerical libraries.

Not to be left in the dust, systems that started as scientific-document word-processors are also potential players. This seems only natural since most of the CAS provide "notebooks," world-wide-web interfaces, documentation. The question, once again, seems to be who will be eaten by whom. This is not solely a technical issue, but can be seriously affected by financial and political considerations.

In order to better understand what can be offered by symbolic computation in the eventual convergence of such systems, in our next section we discuss component technology for computer algebra.

## 2. SYMBOLIC MATHEMATICS COMPONENTS

Computer Algebra Systems (CAS) have had a small but faithful following through the last several decades, but it was not until the late 1980's that this software technology made a major entrance into the consumer and education

[3]Since the early 1970's CAS have often tried to use the supposedly superior numerical compilers of the "outside" computing world by producing numerical source code to compiled externally. Bringing such activities "in house" helps provide a more robust environment.

market. It was pushed by cheap memory, fast personal computers and better user interfaces, as well as the appearance of newly engineered programs.

Now programs like Mathematica, Maple, Macsyma, Derive, Reduce, Axiom, MuPad, and Theorist are fairly well known. Each addresses at some level symbolic, numerical, and graphical computing. Yet none of the commercial systems is designed to provide components that can be *easily broken off and re-used* —called by, for example, "Fortran" programs. (but then since Fortran deals with numbers or arrays of numbers, there is no *natural* way for a Fortran program to accept an answer that is an expression!)

Even if one cannot easily break off modules, most CAS make an efforts to enable a procedure to communicate in two directions with other programs or processes. These existing tools generally require a delicate hand, in spite of considerable effort expended to try to support communication protocols and interconnections.

There have been examples, mostly in research systems when investigators needed certain separable components in the context of (for example) expert systems dealing with physical reasoning, or qualitative analysis. Our own experiences suggest that CAS whose host language is Lisp can be re-cycled in fairly large modules to be used with other Lisp programs. Lisp provides a common base representation serving as a lingua franca between programs running in the same system. Another approach is to provide a relatively primitive interconnection strategy amounting to the exchange of character streams: one program prints out a question and the other answers it. This this is clumsy and fragile[4]. It also makes it difficult to solve significant problems whose size and complexity make character-printing impractical. The communication situation could be likened to two mathematicians expert in different disciplines trying to solve a problem requiring both of their expertises, but restricting them to oral communication by telephone. They could try to maintain a common image of a blackboard, but it would not be easy.

We will get back to the issue of interconnection when we discuss symbolic math systems as "glue".

Let us assume we have overcome this issue of re-use, either by teasing out components or via input/output interfaces for separate command modules. What components would we hope to get? What capabilities might they have, in practical terms? How do existing components fall short?

## 2.1 PROGRAMS THAT MANIPULATE PROGRAMS

The notion of symbolically manipulating programs has a long history. In fact, the earliest uses of the term "symbolic programming" referred to writing

[4]For example, handling errors: streams, traps, messages, return-flags, etc. is difficult.

code in assembly language (instead of binary!). We are used to manipulation of programs by compilers, symbolic debuggers, and similar programs. Language-oriented editors and environments have become prominent in software engineering: tools for human manipulation of what appears to be a text form of the program, with some assistance in keeping track of the details, by the computer. Usually another coordinated model of the program is being maintained by the system to assist in debugging, incremental compiling, formatting, etc. In addition to compiling programs,there are macro-expansion systems, and other tools like cross-referencing, pretty-printing, tracing etc. These common tools represent a basis that most programmers expect from any decent programming environment.

By contrast with these mostly record-keeping tasks, we computers can play a much more significant role in program manipulation. Historically the Lisp programming language has figured prominently here because the data representations and the program representations have been so close[5].

For example, in an interesting and influential thesis project, Warren Teitelman [31] in 1966 described the use of an interactive environment to assist in developing a high-level view of the programming task itself. His PILOT system showed how the user could "advise" arbitrary programs—generally without knowing their internal structure at all — to modify their behavior. The facility of catching and modifying the input or output (or both) of functions can be surprisingly powerful. While ADVISE is available in most lisp systems, it is unknown to most programmers in other languages. For example if one wished to avoid complex results from `sqrt` one can "advise" `sqrt` that if its first argument is negative, it should instead print a message and replace the argument by its absolute value.

```
(advise sqrt :before negativearg nil
    (unless (>= (first arglist) 0)
        (format t "sqrt given negative number.
                  We return (sqrt(abs ~s))"
           (first arglist))
        (setf arglist (list (abs (first arglist))))))
```

With this change, `(sqrt -4)` behaves this way:

```
input: (sqrt -4)
sqrt given negative number. We return (sqrt(abs -4))
output: 2.0
```

and if you wish to advise sqrt that an answer that "looks like an integer" should be returned as an integer, that is, $\sqrt{4}$ should be 2, not 2.0, then one can put advice on the returned value.

```
(advise sqrt :after integerans nil nil
```

[5]Recently, Common Lisp has actually modified this stand on duality by generally making a distinction between a function and the lists of symbols (data) that describe it.

```
(let* ((p (first values))
       (tp (truncate p)))
  (if (= p tp) (setf values (list tp)))))
```

The details illustrated here are unappealing to the non-Lisp reader, but it is about as simple as it deserves to be. If we were writing the first piece of advice in English, we might say "Label this advice "negativeans" in case you want to remove it or change it later. Advise the system that before each call to the square-root function it must check its argument. If it is less than zero, the system should print a message `sqrt given a negative number...` and then return the result of computing the square-root of the absolute value of that argument." (In fact Teitelman shows how he could have the program translate such English advice as given above into Lisp, by *advising the advise program*. He also demonstrates that he could advise his program to take advice in German as well!) Notice that absolutely no knowledge of the interior of `sqrt` is needed, and that in particular `sqrt` need not be written in Lisp (Indeed it is probably taken from a system library!).

We mention this to emphasize the generality that such flexibility is possible and often lost in the shuffle, when programmers attempt to constrain solutions to "pipe" or "bus" architectures, or even traditionally compiled languages.

The favorite paradigm for linking general computer-algebra systems with specific numerical solution methods is to try to define a "class of problem" that corresponds to a solution template. In this template there are "insertions" with symbolic expressions to be evaluated, perhaps derivatives or simplifications or reformulations of expressions, etc. We can point to work as early as the mid-1970s: ambitious efforts using symbolic mathematics systems (e.g. M. Wirth [33] who used Macsyma to automate work in computational physics). This paradigm is periodically re-discovered, re-worked, applied to different problem classes with different computer algebra systems [23, 14, 32, 7, 29, 5, 2].

While we are quite optimistic about the potential usefulness of some of the tools, whether they are in fact useful in practice is a complicated issue. An appropriate meeting of the minds is necessary to convince anyone to use an initially unfamiliar tool, and so ease of use and appropriate design are important, as is education, and availability. We also feel an obligation to voice cautions that there is a clash of cultures: the application programs and the CAS designers may disagree: some of the "obvious" uses that proponents of CAS may identity may be directed at parts of a computation that do not need automation. It is also clear that generating unsatisfactory (inefficient, naive) solutions to problems that have historically been solved by hand-crafted programs is a risky business. We must find improvements, not just alternatives.

**2.1.1 Example: Preconditioning polynomials.** A well-known and useful example of program manipulation that most programmers learn early in

their education is the rearrangement of the evaluation of a polynomial into Horner's rule. It seems that handling this rearrangement with a program is like swatting a fly with a cannon. Nevertheless, even polynomial evaluation has its subtleties, and we will start with a somewhat real-life exercise related to this. Consider the Fortran program segment from [27] (p. 178) computing an approximation to a Bessel function:

```
      ...
      DATA Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9/0.39894228D0,-0.3988024D-1,
     *    -0.362018D-2,0.163801D-2,-0.1031555D-1,0.2282967D-1,
     *    -0.2895312D-1,0.1787654D-1,-0.420059D-2/
      ...
        BESSI1=(EXP(AX)/SQRT(AX))*(Q1+Y*(Q2+Y*(Q3+Y*(Q4+
     *        Y*(Q5+Y*(Q6+Y*(Q7+Y*(Q8+Y*Q9)))))))
      ...
```

(In this case we know $|x| > 3.75$, `AX=ABS(X)` and `Y=3.75/X`)

Partly to show that Lisp, notorious for many parentheses, need not be ugly, and partly to aid in further manipulation, we can rewrite this as Lisp, abstracting the polynomial evaluation operation, as:

```
(setf
 bessi1
 (* (/ (exp ax) (sqrt ax))
    (poly-eval y
      (0.39894228d0 -0.3988024d-1 -0.362018d-2  0.163801d-2
       -0.1031555d-1 0.2282967d-1 -0.2895312d-1 0.1787654d-1
       -0.420059d-2))))
```

An objection might be that we have replaced an arithmetic expression (fast), with a subroutine call, and how fast could that be? Indeed, we can define `poly-eval` as a program that expands in-line, via symbolic computation, before compilation into a *pre-conditioned* version of the above. That is we replace `(poly-eval ...)` with

```
(let* ((z (+ (* (+ x -0.447420246891662d0) x)
             0.5555574445841143d0))
       (w (+ (* (+ x -2.180440363165497d0) z)
             1.759291809106734d0)))
  (* (+ (* x (+ (* x (+ (* w (+ -1.745986568814345d0 w z))
                        1.213871280862968d0))
                9.4939615625424d0))
        -94.9729157094598d0)
     -0.00420059d0))
```

The advantage of this particular reformulated version is that it uses fewer multiplies (6 instead of 8). While it uses 9 additions, *at least two of them can be done at the same time*. This kind of form generalizes and saves more work for higher degree.

Computing coefficients in this form required the accurate solution of a cubic equation, followed by some "macro-expansion" all of which is accomplished at

compile-time and stuffed away in the mathematical subroutine library. Defining `poly-eval` to do "the right thing" at compile time for any polynomial of any degree $n > 1$ is feasible in a symbolic mathematics environment using exact rational and high-precision floating-point arithmetic, and also assuming the programmer is willing to "license" suc a transformation (in a term coined by my colleague W. Kahan). Consider that the rare carefully-crafted programs the coefficients and the arithmetic sequence is specified so as to balance off round-off error *but only if the sequence of operations is not modified by "optimization"*[6] . This is not not the case here: the coefficients were taken from a 1954 paper by E.E. Allen as quoted by Abramowitz and Stegun [1].

As long as we are working on polynomials, we should point out that another possibility emerges: a simple source code template can be inserted to carefully compute other items of interest. For example, the derivative of the polynomial, or a rigorous bound on the error in the evaluation or the derivative. Running such code typically doubles the time to compute a polynomial.

In some cases if we know in advance the range of input, the "program manipulation program" could provide a useful error bound for evaluation of a polynomial *BEFORE it is RUN*. If, in this Bessel function evaluation context we know that $0 \leq x \leq 3.75$, we can determine the maximum error in the polynomial for *any* $x$ that region. We could in principle extend this kind of reasoning to produce, in this symbolic programming environment, a library routine with an *a prior* error bound. (As it happens, the truncation error in this approximation far exceeds the roundoff in this routine).

While the total automation of error analysis requires far more than routine algebra, local application of rules-of-thumb and some practical expertise in an arcane subject (error analysis) can be provided in a PSE. We foresee one difficulty in optimizing: the design objectives and limits are rarely specified formally. Is the goal to write the fastest or most accurate or most robust possible numerical program? Is the goal constrained to writing the program that will run on the widest range of possible computer architectures while giving exactly the same answers? These are open research areas.

Finally, we should comment that the original Fortran segment displayed above shows constants given in double-precision syntax The number of figures given is only about half that precision. It would be delightful to have intelligent libraries that could on command, reformulate their entries to take advantage of full accuracy or other considerations by reference to the original mathematical analysis. Instead we see programmers copying, by proxy, not very accurate code developed in 1954. In a scheme for computing improved approximations

[6]For an example of this, see the VAX trigonometric function approximations developed at UC Berkeley for 4BSD UNIX, arranged specifically for VAX roundoff.

on command we might also see the elimination of other errors, such as those caused by copying coefficients.

**2.1.2 Example: Generating perturbation expansions.** A common task for some computational scientists is the generation (at considerable mental expense) of a formula to be inserted into a Fortran or other program. A well-known example from celestial mechanics (or equivalently, accelerator physics [8]) is the solution to the Euler equation, $E = u + e\sin(E)$. This can be solved iteratively to form an expansion in powers of the small quantity $e$. Let $A_0$ be 0. Then suppose $E = u + A_k$ is the solution correct to order $k$ in $e$. Then $A_{k+1}$ is computed as $e\sin(u + A_k)$ expanded as a series in $\sin(nu)$ with coefficients of degree up to $k + 1$ in $e$. This is a natural operation in any CAS system supporting Poisson series. The Macsyma system computes $A_4$ (and incidentally, converted it to TEX for inclusion as a typeset formula) for us:

$$A_4 = \frac{e^4\,\sin(4\,u)}{3}+\frac{3\,e^3\,\sin(3\,u)}{8}+\frac{\left(12\,e^2-4\,e^4\right)\,\sin(2\,u)}{24}+\frac{\left(24\,e-3\,e^3\right)\,\sin u}{24}$$

If we insist on a version in Fortran, as any CAS provides, we get a variety of (partially) appropriate responses. For example, Mathematica:

```
      FortranForm=
     -  (24*e - 3*e**3)*Sin(U)/24 + (12*e**2 - 4*e**4)*Sin(2*U)/24 +
     -   3*e**3*Sin(3*U)/8 + e**4*Sin(4*U)/3
```

Historically, it has been important to check such results to make sure the Fortran is valid. (Mathematica 2.1 converted $1/3$ to `1/3`, which in Fortran is simply 0.) Newer versions make some default (and hence sometimes wrong) decision on how many 3's there are in $1/3$:

```
FortranForm=
     -  1.*e*Sin(U) - 0.125*e**3*Sin(U) + 0.5*e**2*Sin(2.*U) -
     -   0.1666666666666666*e**4*Sin(2.*U) + 0.375*e**3*Sin(3.*U) +
     -   0.3333333333333333*e**4*Sin(4.*U)
```

Maple produces

```
    t0 = E**4*sin(4*U)/3+3.0/8.0*E**3*sin(3*U)+(12*E**2-4*E**4)*sin(2*
   #U)/24+(24*E-3*E**3)*sin(U)/24
```

or after floating-point conversion,

```
    t0 = 0.3333333E0*e**4*sin(4.0*U)+0.375E0*e**3*sin(3.0*U)+0.4166667
   #E-1*(12.0*e**2-4.0*e**4)*sin(2.0*U)+0.4166667E-1*(24.0*e-3.0*e**3)
   #*sin(U)
```

After rearranging using Horner's rule we get

```
   t0 = (sin(U)+(sin(2*U)/2+(3.0/8.0*sin(3*U)-sin(U)/8+(-sin(2*U)/6+s
   #in(4*U)/3)*e)*e)*e)*e
```

The last formula (from Maple) has multiple occurrences of $\sin u$. Any decent Fortran compiler will compute this once and re-use it. However that compiler wouldn't notice that from the initial values $s = \sin u$ and $c = \cos u$ one can compute all the values needed with remarkably little effort: $s_2 = \sin 2u = 2 \cdot s \cdot c$; $c_2 = \cos 2u = 2c^2 - 1$; $s3 = \sin 3u = s \cdot (2c_2 + 1)$; $s4 = \sin 4u = 2s_2c_2$; etc.

In fact, any time an expression $S$ of even moderately complicated form is evaluated at regular grid points in one or more dimensions $S[\mathbf{z}]$, the calculus of divided differences may help in figuring out how to replace calculation of $S[\mathbf{z}]$ as an update to one or more previous values computed nearby. Values of $S[\mathbf{z}+\Delta]$ computed in this way will typically suffer some slight loss in accuracy, but this may very well be predictable and correctable if necessary[7].

More specifically, if we return to the Euler equation program, We can apply a more general method useful for $k > 3$. Here we use a two-term recurrence which for each additional sin and cos pair requires only two adds and two multiplies. Using the facts that

$$\begin{aligned} 0.5\sin(a+d)/\sin(d) &= 0.5\sin(a-d)/\sin(d) + \cos(a) \\ \cos(a+2d) &= \cos(a) - 4\sin^2(d)0.5\sin(a+d)/\sin(d) \end{aligned}$$

we can construct a very simple program whose inner loop computes $s_n = 0.5\sin(nu)/\sin(u)$.

First set $k_1 = 2\sin(u)$ and $k_2 = k_1^2$, $s_0 = 0$, $s_1 = 1/2$, $c_0 = 1$, $c_1 = \cos u$. Then for $i > 1$:

$$\begin{aligned} s_i &:= s_{i-2} + c_{i-1} \\ c_i &:= c_{i-2} - k_2 * s_{i-1} \end{aligned}$$

Then $\sin(nu)$ is $k_1 * s_n$ and $\cos(nu)$ is simply $c_n$. (There is some accumulation of round-off; this can be counteracted if necessary with minor additional effort.)

We have chosen these illustrations because they show that

1. Symbolic computing can be used to produce small expressions, not just relatively large ones. Furthermore, the important contribution is to form *sequences of program statements involving clever optimization of inner loops.*

[7]Often such correction is not needed: Since computing a surface plot of $z = f(x, y)$ does not usually require full accuracy, such a method can present substantial speedups. E.V. Zima has written extensively on this [34].

2. Some relatively small expressions (just sin and cos) may nevertheless be costly to compute when they occur in inner loops; there are tools, far beyond what compilers can do, to make this faster.

3. Having computed expressions symbolically, prematurely converting them to Fortran (etc.) is generally a mistake unless you are so carefully crafting the program that the Fortran code is *exactly* what you mean: no more and no less.

4. As long as we are generating the code, it is in some cases possible to produce auxiliary program text related to the main computation. For example, keeping track of accumulated roundoff is often possible.

5. Other auxiliary information such as typeset forms can be routinely produced for inclusion in papers such as this. These expressions may be quite different from computationally optimal ones.

By contrast, CAS vendors illustrate their systems' capability of producing hugely-long expressions. It is usually a bad idea to dump large expressions into a Fortran source file. Not only are such expressions likely to suffer from instability, but their size may strain common-subexpression elimination "optimizers" or exceed the input-buffer sizes or stack sizes. Breaking expressions into smaller "prograsm statement" pieces as we advocate above is not difficult; even better may be the systematic replacement of expressions by calculation schemes such as `poly-eval` discussed previously.

## 2.2 DERIVATIVES AND INTEGRALS

Students who study a "symbolic" language like Lisp often see differentiation as a small exercise in tree-traversal and transformation. They will view programming of closed-form symbolic differentiation as trivial, if for no other reason than it can be expressed in a half-page of code (e.g. see a 14 line program [11]).

Unfortunately the apparent triviality of such a program relies on a simplistic problem definition. A serious CAS will have to deal far more efficiently with large expressions. It must deal with partial derivatives with respect to positional parameters (where even the notation is not standard in mathematics!). It must deal with the very substantial problem of simplification. Even after solving these problems the real application is often stated as the differentiation of a Fortran "function subroutine." Approached from a computer science perspective, it is tempting to dismiss this since so much of it is impossible: Can you differentiate with respect to a do-loop index? What is the derivative of an "if" statement?

However, viewing a subroutine as a manifest representation of a mathematical function, we can try to push this idea as far possible. If we wish to

indeed compute "derivatives of programs" efficiently and accurately, this can be a worthwhile endeavor for complex optimization software. One alternative is computing "numerical derivatives" where $f'(x)$ at a point $a$ is computed by choosing some small $\Delta$ and computing $(f(x+\Delta) - f(x))/\Delta$. Numerical differentiation yields a result of unknown, but probably low, accuracy.

The collection by Griewank and Corliss [16] considers a wide range of tools: from numerical differentiation, through pre-processing of languages to produce Fortran, to entirely new language designs (for example, embodying Taylor-series representations of scalar values). The general goal of each approach is to ease the production of programs for computing and using accurate derivatives (and matrix generalizations: Jacobians, Hessians, etc.) rapidly. Tools such as ADIFOR (`www.mcs.anl.gov/adifor`) are receiving more attention.

To again illustrate the importance of programs rather than expressions, consider how we might wish to see a representation of the second derivative of $\exp(\exp(\exp(x)))$ with respect to $x$. The answer $e^{e^{e^x}+e^x+x}\ (1 + e^x + e^{e^x+x})$ is correct and could be printed in Fortran if necessary. But then more useful would be the mechanically produced program below.

```
t1 := x
t2 := exp(t1)
t3 := exp(t2)
t4 := exp(t3)
d1t1 := 1
d1t2 := t2*d1t1
d1t3 := t3*d1t2
d1t4 := t4*d1t3
d2t1 := 0
d2t2 := t2*d2t1 + d1t2*d1t1
d2t3 := t3*d2t2 + d1t3*d1t2
d2t4 := t4*d2t3 + d1t4*d1t3
```

(by eliminating operations of adding 0 or multiplying by 1, this could be made smaller.)

It is entirely plausible to have a pre-processor or compiler-like system which automatically and without even exhibiting such code, produces implicitly the computation of $f'(x)$ in addition to $f(x)$. Consider that every scalar variable $v$ is transformed to a vector $(v(x), v'(x), v''(x), ...)$ and all necessary computations are carried out to maintain these vectors (to some order). For optimization programs this is a rather different approach with advantages over the derivative-free methods or ones that depend on hand-coded derivatives.

What about other operations on programs? Packages exist to altering them so they compute much higher precision answers. Another possibility is to have programs return interval answers (every result rigorously bounded).

Programming language designers like to consider alternative modes of evaluation (usually delayed) in a kind of compilation. We can consider pre-

computing some specializations at compile time such as evaluating an integral symbolically, replacing a call to a numerical quadrature program. This would be like a very high order "constant folding" by an optimizing compiler. Or noticing that the program is computing $\sum_{i=1}^{n} i$ and that it can be replaced by $n(n+1)/2$. Given an appropriate "license" for such grand substitutions, it might be possible to do this by CAS intervention.

It is not obvious that such licenses should be automatically granted since the closed form may be more painful to evaluate than the numerical quadrature! Example: $f(x) = 1/(1+z^{64})$ whose integral is

$$F(z) = \frac{1}{32}\sum_{k=1}^{16} c_k \operatorname{arctanh}\left(\frac{2c_k}{z+1/z}\right) - s_k \arctan\left(\frac{2s_k}{z-1/z}\right)$$

where $c_k := \cos((2k-1)\pi/64)$ and $s_k := \sin((2k-1)\pi/64)$. [20]

Another favorite examples where the CAS closed form needs to be considered carefully is one of the most trivial examples: $\int_a^b x^{-1}dx$ which most computer algebra systems give as $\log b - \log a$. Even a moment's thought suggests a better answer is $\log(b/a)$. It turns out that a numerically preferable formula is "if $0.5 < b/a < 2.0$ then $2\operatorname{arctanh}((b-a)/(b+a))$ else $\log(b/a)$." [9]. Consider, in IEEE double precision, $b = 10^{15}$ and $a = b+1$: The first formula (depending on your library routines) may give an answer of $-7.1e-15$ or $0.0$. The second gives $-1.0e-15$, which is correct to 15 places.

In each of two last cases we do not mean to argue that the symbolic result is mathematically wrong, but only that CAS to date tend to answer the wrong question. An effort should be undertaken to provide, as an option from CAS more generally, *computer programs* not just mathematical formulas.

## 2.3 SEMI-SYMBOLIC SOLUTIONS OF DIFFERENTIAL EQUATIONS

There is a large literature on the solution of ordinary differential equations. For a compendium of methods, Zwillinger's handbook on CD-ROM [35] is an excellent source. Almost all of the computing literature concerns numerical solutions, but there is a small corner of it devoted to solution by power series and analytic continuation. We pick up on this neglected area of application.

There is a detailed exposition by Henrici[17] of the background including "applications" of analytic continuation. In fact his results are somewhat theoretical, but they provide a rigorous computational foundation. Some of the more immediate results seem quite pleasing. We suspect they are totally ignored by the numerical computing establishment.

The basic idea is quite simple and elegant, and an excellent account may be found in a paper by Barton *et al* [6]. In brief, if you have a system of differential equations $\{y_i'(t) = f_i(t, y_1, ..., y_n)\}$ with initial conditions $\{y_i(t_0) = a_i\}$

proceed by expanding the functions $\{y_i\}$ in Taylor series about $t_0$ by finding all the relevant derivatives. The technique, using suitable recurrences based on the differential equations, is straightforward, although it can be extremely tedious for a human. The resulting power series could be a solution useful within the radius of convergence of the series. It is possible, however, to use analytic continuation to step around singularities (located by various means, including perhaps symbolic methods) by re-expanding the equations into series at time $t = t_1$ with coefficients derived from evaluating Taylor series at $t_0$.

How good are these methods? It is hard to evaluate them against routines whose measure of goodness is "number of function evaluations" because the Taylor series *does not evaluate the functions at all*! To quote from Barton [6], "[The method of Taylor series] has been restricted and its numerical theory neglected merely because adequate software in the form of automatic programs for the method has been nonexistent. Because it has usually been formulated as an *ad hoc* procedure, it has generally been considered too difficult to program, and for this reason has tended to be unpopular."

Are there other defects in this approach? It is possible that piecewise power series are inadequate to represent solutions. Or is it mostly inertia (being tied to Fortran)?

Considering the fact that this method requires ONLY the defining system as input (symbolically!) this would seem to be an excellent characteristic for a problem solving environment. Barton concludes that "In comparison with fourth-order predictor-corrector and Runge-Kutta methods, the Taylor series method can achieve an appreciable savings in computing time, often by a factor of 100." Perhaps in this age of parallel numerical computing, our parallel methods are so general and clever, and our problems so "stiff" that these series methods are neither fast nor accurate; we are trying to reexamine them in the light of current software and computer architectures at Berkeley. Certainly if one compares the conventional step-at-a-time solution methods which have an inherently sequential aspect to them, Taylor series methods provide the prospect of taking both much larger steps, and much "smarter" steps as well. High-speed and even parallel computation of functions represented by Taylor series is perhaps worth considering. This is especially the case if the difficulties of programming are overcome by automation, based on some common models.

## 2.4 EXACT, HIGH-PRECISION, INTERVAL OR OTHER NOVEL ARITHMETIC

Sometimes using ordinary machine-precision floating-point arithmetic is inadequate for computation. CAS provide exact integer and rational evaluation of polynomial and rational functions. This is an easy solution for some kinds of computations requiring occasionally more assurance than possible with just

approximate arithmetic. Arbitrarily high precision floating-point arithmetic is another feature, useful not so much for routine large-scale calculations (since it is many times slower than hardware), but for critical testing of key expressions. It is more versatile than exact rational arithmetic, allowing for transcendental and algebraic function evaluations.

The well-known constant $\exp(\pi\sqrt{163})$ provides an example where cranking up precision is useful. Evaluated to the relatively high precision of 31 decimal digits, it looks like a 17 digit integer: 262537412640768744. Evaluated to 37 digits it reveals its true nature: 262537412640768743.99999999999992500726. Other kinds of non-conventional but still numeric (i.e. not symbolic) arithmetic that are included in some CAS include real-interval or complex-interval arithmetic. We have experimented with a combined floating-point and "diagnostic" system which makes use of the otherwise unused fraction fields of floating-point exceptional operands ("IEEE-754 binary floating '*Not-A-Numbers*). This allows many computations to proceed to their eventual termination, but with results that indicate considerable details on any problems encountered along the way. The information stored in the fraction field of the NaN is actually an index into a table in which rather detailed notes can be kept.

There is a substantial literature of competing interval or high-precision libraries and support systems, including customized compilers that appear to use conventional languages, but in reality "expand" the code to call on subroutines implementing software-coded long floats, etc.

## 2.5 FINITE ELEMENT ANALYSIS, GEOMETRY, AND ADAPTIVE PRECISION

Formula generation needed to automate the use of finite element analysis code has been a target for several packages using symbolic mathematics (see Wang [32] for example). It is notable that even though some of the manipulations would seem to be routine — differentiation and integration — there are nevertheless subtleties that make naive versions of algorithms inadequate to solve large problems.

Our colleague at Berkeley, Jonathan Shewchuk [30] has found that clever computations to higher precision may be required in computational geometry. That is, one can be forced to compute to higher numerical accuracy to maintain robust geometric algorithms. He has shown a technique for adaptive-precision arithmetic (to satisfy some error bound) whose running time depends on the allowable uncertainly of the result.

## 2.6 LICENSES AND CODE GENERATION FOR SPECIAL ARCHITECTURES

Starting from the same "high level" specification we can symbolically manipulate the information in a machine-dependent way as one supercomputer is supplanted by the next. Changing between minor revisions of the same hardware design may not be difficult; altering code from a parallel shared-memory machine such as has been popular in the past to a super-scalar distributed networked machine would be more challenging.

It is especially difficult to make such a conversion if the original model is overly constrained by what may appear to be sequential (or even shared-memory parallel) operations, as written by the programmer, simply because the language of the computation cannot expose all the available independence of computations.

While we advocate using a higher-level mathematical model for manipulation, there is still the issue of how to reflect our understanding of acceptable code to the numerical computing level activity. One way we have already mentioned with regard to `poly-eval` earlier, is to change languages and their compilers to recognize programmer-specified "licenses" in source code. These license are assertions about the appropriateness of using transformations on source code that are known to be true not in general, but in this particular case; these free the compiler to make optimizations that are not universally allowed. For example, if a programmer knows that it is acceptable in a certain section of code to apply distributivity when seeking common subexpressions, a local declaration license this can help the compiler. In our earlier example of re-arranging polynomial evaluation, in the absence of such a license the re-arrangement might be ill-advised.

Licenses cannot remove all constraints: unintentional constraints are among the biggest problems in retaining old source languages. Merely changing the compiler to target new architectures has the attraction that old code can still run, but this cannot get the best results. Current state-of-the art compiler technology is more aimed at architecture: compilers now run a program under controlled conditions to measure memory access patterns and branching frequencies, and the efficacy of optimizations may be judged by improved empirical timings rather than predictions encoded in a code-generator. As examples of possible improvements through dynamic compiling in new architectures such as the Intel/HP IA-64:

- Speculative execution that can be substantially aided if one can deduce that a branch "usually" goes one way. (This can be conditioned on previous branches at that location, or can be statically estimated).
- Code can be emitted to recommend the fetching into high-speed cache of certain sections of memory in advance of their use.

- Register usage and efficiency can be parameterized in various ways, and the ebb and flow of registers to memory can affect performance.

The point we wish to make here is that by committing certain methods to (Fortran) programs, with data structures and algorithms constrained to one model, we are cutting off potential optimizations at the level of the model.

We are hardly the first to advocate higher-level modeling languages, and it it interesting to see it has entered the commercial realm[8].

## 2.7 SUPPORT FOR PROOFS, DERIVATIONS

CAS have not been as widely used as specialized "theorem proving" systems in producing interesting automated theorems. We believe there is potential here, but theorem-proving software has not broken into applied mathematics, where it could have more effect on scientific computing.

Perhaps notable is an attempt to merge CAS and proof technology in several directions is a current project "Theorema" at RISC-Linz and SCORE at University of Tsukuba, Japan.

Proofs in "applied mathematics" seem to require far more structure than proofs in (say) number theory. Of the computer algebra systems now available, only Theorist makes some attempt to formalize the maintenance of a "correct" line of user-directed transformations. Systems like AUTOMATH, which support verification of proof steps rather than production of steps represent another cut through this area. (There are publications such as the Journal of Automated Reasoning covering such topics quite thoroughly. In the interests of brevity We suggest only two references to the "interface" literature [3, 4].) When the interesting problems of main-line numerical computation are addressed, they require deeper results to modeling real and complex functions and domains, as well models of computer floating-point arithmetic for numerical error analysis. This direction of work remains a substantial challenge.

## 2.8 DOCUMENTATION AND COMMUNICATION

We believe that the future directions of computer algebra systems are on a collision course with elaborate documentation and presentation systems. For decades now it has been possible to produce typeset quality versions of small-to-medium sized expressions from CAS. Two-way interfaces to documentation systems are also possible; we have looked at converting mathematical typeset documents (even those that have to be scanned in to the computer), into re-usable objects suitable for programming. These scanned expressions can be

[8] For example, SciComp Inc. builds models this way for PDE solving[2], and more recently various financial computations.

stored in Lisp or perhaps in XML/MathML web forms, and then re-used as computational specifications.

We would not like to depend on the automatic correctness of such optical character recognition for reliable scientific software; indeed errors in recognition, ambiguity of expression as well as typographical errors all contribute to unreliability. Yet in the push toward distributed computation, there is a need to be able to communicate the objects of interest— symbolic mathematical expressions— from one machine to another.

If we all agree on a common object program data format, or even a machine independent programming language source code (like Java), we can in principle share computational specifications.We do not believe that we can use TEX as a medium and send integration problems off to a remote server. While we have great admiration for TEX's ability in equation typesetting, typeset forms without context are ambiguous as a representation of abstract mathematics. Notions of "above" and "superscript" are perfectly acceptable and unambiguous in the typesetting world, but how is one to figure out what the TEX command `f^2(x+1)` means mathematically? This typesets as $f^2(x+1)$. Is it a function application of $f^2$ to $(x+1)$, perhaps $f(f(x+1))$? Or is it the square of $f(x+1)$? And the slightly different `f^{(2)}(x+1)` which typesets as $f^{(2)}(x+1)$ might be a second derivative of $f$ with respect to its argument, evaluated at the point $x+1$. These examples just touch the surface of difficulties.

Thus the need arises for an alternative well-defined (unambiguous) representation which can be moved through an environment – as a basis for documentation as well as manipulation. The representation should have some textual encoding so that it can be shipped or stored as ASCII data, but this should probably not be the primary representation. When it comes to typesetting, virtually every CAS can convert its internal representations into TEX on command.

While no CAS can anticipate all the needs for abstraction of mathematics, there is no conceptual barrier to continuing to elaborate on the representation. (Examples of current omissions: none of the systems provide a built-in notation for block diagonal matrices, none provides a notation for contour integrals, none provides for a display of a sequence of steps constituting a proof, none understands "Let $H$ be a Hilbert space.")

A suitable representation of a problem probably should include sufficient text to document the situation, most likely place it in perspective with respect to the kinds of tools appropriate to solve it, and provide hints on how results should be presented. Many problems are merely one in a sequence of similar problems, and thus the text may actually be nothing more than a slightly modified version of the previous problem — the modification serving (one hopes) to illuminate the results or correct errors in the formulation. Working from a "script" to go through the motions of interaction is clearly advantageous in some cases.

In addition to (or as part of) the text, it may be necessary to provide suitably detailed recipes for setting up the solution. As an example, algebraic equations describing boundaries and boundary conditions may be important in setting up a PDE solving program.

Several systems have been developed with "worksheet" or "notebook" models for development or presentation.

Current models seem to emphasize either the particular temporal sequence of commands (Mathematica, Maple, Macsyma-PC notebooks) or a spreadsheet-like web of dependencies (MathCAD, for example). One solves or debugs a problem by revisiting and editing the model.

For the notebooks, this creates some ad-hoc dependency of the results by the order in which you re-do commands. This vital information (vital if you are to reproduce the computation on a similar problem) is probably lost.

For the spreadsheets, there is a similar loss of information as to the sequence in which inter-relationships are asserted, and any successes that depend on this historical sequence of settings may not be reproducible on the next, similar, problem.

In our experience, we have found it useful to keep a model "successful script" alongside our interactive window. Having achieved a success interactively is no reason to believe it is reproducible — indeed with some computer algebra systems we have found that the effect of some bug (ours or the system's) at the $n$th step is so disasterous or mysterious that we are best served by reloading the system and re-running the script up to step $n-1$. We feel this kind of feedback into a library of solved problems is extremely important. A system relying solely on pulling down menu items and pushing buttons may be easy to use in some sense, but it will be difficult to extend such a paradigm to the solution of more complicated tasks. So-called 'keyboard macros" seem like a weak substitute for scripts in which "literate programming" combined with mathematics can describe constructive, algorithmic solution methods. It is not clear that this is the best approach, but in the absence of CAS "killer applications" it is our best guess as to how a major organizing paradigm can be presented. Centered more effectively around a document combined with world-class expertise.

## 3. SYMBOLIC MANIPULATION SYSTEMS AS GLUE

In this section we spend some time discussing our favorite solutions to interconnection problems.

Gallopoulos *et al* [13] suggest that symbolic manipulation systems already have some of the critical characteristics of the glue for assembling a PSE but are not explicit in how this might actually work. Let's be more specific about aspects of glue, as well as help in providing organizing principles (a back-

bone). This section is somewhat more nitty-gritty with respect to computing technology.

The notion of glue as we have suggested it has become associated with "scripting languages," a popular description for a collection of languages including Perl, Tcl, Python, Basic, Rexx, Scheme (a dialect of lisp). Rarely is a CAS mentioned though if the glue is intended to connect programs speaking mathematics, this suddenly becomes plausible. CAS do not figure prominently partly because they are "large" and partly because most people doing scripting are generating e-commerce web pages, or controlling data-bases. (Of course the CAS could also do these tasks, but mostly without use of any of their mathematical capabilities.) In any case, the common currency of scripting languages tends to be character strings as a lowest-common-denominator of computer communication. The other distinction for scripting languages is that they are interactive in environment and execution. They provide a mechanism to piece together a string which can then be evaluated as a program. The simultaneous beauty and horror of this prospect may be what makes scripting languages a hackers' playground.

When large-scale program development is part of the objective, the inability of some scripting languages to be able to handle complicated objects (in particular, large programs) can be a critical deficiency. The long history of effectively treating Lisp programs as data and Lisp data as programs is a particular strength of this language that causes it to rise to the top of the heap for scripting. The next two sections give some details.

## 3.1 EXCHANGE OF VALUES

We prefer that the glue be an interpretive language with the capability of compiling routines, linking to routines written in other languages, and (potentially, at least) *sharing memory space with these routines*. We emphasize this last characteristic because the notion of communicating via pipes or remote-procedure call, while technically feasible and widely used, is nevertheless relatively fragile.

Consider, by contrast, a typical Common Lisp implementation with a "foreign function" interface. (Virtually all systems have this but with minor syntactic differences).

On the workstation at which I am typing this paper, and using Allegro Common Lisp, if I have developed a Fortran-language package, or if my program has generated one, in which there is a `double-precision function` subroutine FX taking one double-precision argument, I can use it from Lisp by loading the object file (using the command `(load "filename.o")`). and then declaring

```
(ff:defforeign 'FX  :language :fortran
  :return-type :double-float  :arguments '(double-float))
```

Although additional options are available to `defforeign`, the point we wish to make is that virtually everything that makes sense to Fortran (or C) can be passed across the boundary to Lisp, and thus there is no "pinching off" of data interchange as there would be if everything had to be converted to character strings, as in the UNIX operating system pipes convention. While this would open up "non-numeric" data, it would be quite inefficient for numeric data, and quite unsuitable for structures with pointers. Lisp provides tools to mimic structures in C: (`def-c-type`) creates structures and accessors for sub-fields of a C structure, whether created in Lisp or C.

What else can be glued together? Certainly calls to produce web pages, displays in window systems, and graphics routines. In fact, the gluing and pasting has already been done and provided in libraries.

We have hooked up Lisp to two arbitrary-precision floating-point packages, LINPACK and MINPACK, and others have interfaced Lisp to the Numerical Algorithms Group (NAG) library, LAPACK and the library from *Numerical Recipes*. Interfaces to SQL and database management systems have also been constructed at Berkeley and apparently elsewhere.

## 3.2 MORE ARGUMENTS FOR LISP

The linkage of *Lisp-based* symbolic mathematics tools such as Macsyma and Reduce into Lisp naturally is in a major sense "free" and doesn't require any glue at all. It is clear that Fortran can't provide the glue. C or Java can only provide glue indirectly: you must first write a glue system in them. (You could, like most other people, write a Lisp system in C, but you would not exactly be breaking new ground.)

We return in part to an argument in our first section. Linkage from a large PSE to symbolic tools in CAS is typically supported via a somewhat narrow character-string channel. Yet one would might have considerable difficulty tweezing out just a particular routine like our Euler Fortran example function above. The systems may require the assembling of one or more commands into strings, and parsing the return values. It is as though each time you wished to take some food out of the refrigerator, you had to re-enter the house via the front door and navigate to the kitchen. It would be preferable, if we were to follow this route, to work with the system-providers for a better linkage – at least move the refrigerator to the front hall.

Yet there are a number of major advantages of Common Lisp over most other languages that these links do not provide. The primary advantage is that *Common Lisp provides very useful organizing principles for dealing with complex objects, especially those built up incrementally during the course of an interaction.* This is precisely why Lisp has been so useful in tackling artificial intelligence (AI) problems in the past, and in part how Common Lisp features

were designed for the future. The CLOS (Common Lisp Object System) facility is one such important component. This is not the only advantage; we find that among the others is the posibility of compiling programs for additional space and time efficiency. The prototyping and debugging environments are dramatically superior to those in C or Java even considering the interpretive C environments that have been developed. There is still a vast gap in tools, as well as in support of many layers of abstraction, that in my opinion, gives Lisp the edge: Symbolic compound objects which include documentation, geometric information, algebraic expressions, arrays of numbers, functions, inheritance information, debugging information, etc. are well supported.

Another traditional advantage to Lisp is that a list structure can be written out for human viewing, and generally read back in to the same or another Lisp, with the result being a structure that is equivalent to the original[9]. By comparison, if one were to design a structure with C's pointers, one cannot do much debugging without first investing in programs to read and display each type of structure.

In spite of our expressed preference, what about other possible glues? Script languages seem to have little special regard for symbolic mathematics, although several mentioned in our earlier list are quite interesting. Ambitious CAS vendors certainly would like to come forward; using proprietary code is one barrier. Nevertheless, we hope to benefit from the current surge in exploration and design of languages for interaction, scripting, and communication as a reaction to the dominant "thinking in C" of previous decades.

## 4. TWO SHORT-TERM DIRECTIONS FOR SYMBOLIC COMPUTING

Martin's [26] goal of building a general assistant, an artificially intelligent robot mathematician composed of a collection of "facts" seems, in retrospect, too vague and ambitious. Two alternative views that have emerged from the mathematics and computer science (not AI) community resemble the "top-down" vs "bottom-up" design controversy that reappears in many contexts. A top-down approach is epitomized by AXIOM [18]. The goal is to lay out a hierarchy of concepts and relationships starting with "Set" and build upon it all of mathematics (as well as abstract and concrete data structures). While this has been shown to be reasonably successful for a kind of constructive algebra, an efficient implementation of higher constructs seems to be difficult. Perhaps

[9]Modern Lisps tend to back away from this principle of built-in universal read/write capabilities for non-list structures: Although every structure has some default form for printing, information-preserving print and read methods may have to be programmed.

this reflects an underlying problem relating to approximations to continuum engineering mathematics.

By contrast, the bottom-up approach attempts to build successful applications and then generalizing. At the moment, this latter approach seems more immediately illuminating.

We discuss these approaches in slightly more detail below.

## 4.1 THE TOP-DOWN APPROACH

As we have indicated, the approach implicit or occasionally explicit in some CAS development has started from the abstract: Make as much mathematics constructive as possible, and hope that applications (which, after all, use mathematics) will follow.

In addition to the challenge of overcoming such humble beginnings, is that general constructive solutions may be too slow or inefficient to put to work on specific problems.

Yet it seems to us that taking the "high road" of building a constructive model of mathematics is an inevitable, if difficult, approach. Patching it on later is not likely to be easy. Of the commercial CAS today, AXIOM sees to have the right algebraic approach, at least in principle. Software engineering, object-oriented programming and other buzzwords of current technology may obscure the essential nature of having programs and representations mirror mathematics, and certainly the details may change; the principles should remain for the core of constructive algebra.

We hope this is not incompatible with the view of the next section; with perseverance and luck, these two approaches may converge and help solve problems in a practical fashion.

## 4.2 BOTTOM UP: LEARNING FROM SPECIFICS

As an example of assessing the compromises needed to solve problems effectively, consider the work of Fritzson and Fritzson [12] who discuss several real-life mechanical design scenarios. One is modeling the behavior of a 20-link saw chain when cutting wood, another is the modeling of a roller-bearing. To quote from their introduction.

> "The current state of the art in modeling for advanced mechanical analysis of a machine element is still very low-level. An engineer often spends more than half the time and effort of a typical project in implementing and debugging Fortran programs. These programs are written in order to perform numerical experiments to evaluate and optimize a mathematical model of the machine element. Numerical problems and convergence problems often arise, since the optimization problems usually are non-linear. A substantial amount of time is spent on fixing the program to achieve convergence.

> Feedback from results of numerical experiments usually lead to revisions in the mathematical model which subsequently require re-implementing the Fortran program. The whole process is rather laborious.
>
> There is a clear need for a higher-level programming environment that would eliminate most of these low-level problems and allow the designer to concentrate on the modeling aspects.

They continue by explaining why CAD (computer aided design) programs don't help much: These are mostly systems for specifying geometrical properties and other documentation of mechanical designs. The most general systems of this kind may incorporate known design rules within interactive programs or databases. However such systems *provide no support for the development of new theoretical models or the computations associated with such development... [the] normal practice is to write one's own programs*.

The Fritzsons' view is quite demanding of computer systems, but emphasizes, for those who need such prompting, the central notion that the PSE must support a single, high-level abstract description of a model. This model can then serve as the basis for documentation as well as computation. All design components must deal with this model, which they have refined in various ways to an object-oriented line of abstraction and representation. If one is to make use of this model, the working environment must support iterative development to refine the theoretical model on the basis of numerical experiments.

Thus, starting from an application, one is inevitably driven to look at the higher-level abstractions. These are not likely to be initially a close match to modern algebra but there may be some common ground nevertheless.

## 5. THE FUTURE

What tools are available but need further development? What new directions should be explored? Are we being inhibited by technology?

There are some impressive symbolic tools available in at least one non-trivial form, in at least one CAS. Often these can and should be extended for use in PSEs.

- Manipulation of formulas, natural notation, algebraic structures, graphs, matrices
- Categories of types that appear in mathematical discourse.
- Constructive algorithmic mathematical types, canonical forms, etc.
- Manipulation of programs or expressions: symbolic integrals and quadrature, finite element calculations: dealing with the imperfect model of the real numbers that occurs in computers.
- Exact computation (typically with arbitrary precision integer and rational numbers).

- Symbolic approximate computation (Taylor, Laurent or asymptotic series, Chebyshev approximations)
- Access to numerical libraries
- Typeset quality equation display / interactive manipulation
- 2-D and 3-D (surface) plots
- On-line documentation, notebooks.

There are tools, capabilities, or abstractions fundamentally missing from today's CAS, although many of them are available in some partial implementation or are being studied in a research context. They seem to us to be worthy of consideration for inclusion in a PSE, and probably fit most closely with the symbolic components of such a system.

- Assertions, assumptions
- Geometric reasoning
- Constraint-based problem solving
- Qualitative analysis (Reasoning about physical systems)
- Derivations, theorems, proofs
- Mechanical, electronic, or other computer-aided design data

As an example of another area in which CAS can support PSEs in the future, consider plotting and visualization.

To date, most of the tools in scientific visualization are primarily numerical: ultimately computing the points on a curve, surface or volume, and displaying them. In fact, when current CAS provide plotting, it is usually in two steps. Only the first step has a symbolic component: producing the expression to be evaluated. The rest of the task is then essentially the traditional numerical one.

Yet by maintaining a hold on the symbolic form, more insight may be available. Instead of viewing an expression as a "black box" to be evaluated at some set of points, the expression can be analyzed in various ways: local maxima and minima can be found to assure they are represented on the plot. Points of inflection can be found. Asymptotes and other limiting behaviors can be detected (e.g. "for large $x$ approaches $\log x$ from below). By using interval arithmetic [10], areas of the function in which additional sampling might be justified, can be detected. In some cases exact arithmetic, rather than floating-point, may be justified. These techniques are relevant to functions defined mathematically and for the most part do not pertain to plots of measured data.

Finally, we feel that a the recent interest in communication in the MathML/XML OpenMath communities may provide a foundation for a better appreciation of the merits of symbolic computation in a broad context of PSEs.

## 6. ACKNOWLEDGMENTS

Discussions and electronic mail with Ken Rimey, Carl Andersen, Richard Anderson, Neil Soiffer, Allan Bonadio, William Kahan, Bruce Char and others have influenced this paper and its predecessor notes. Some of this material appeared in an unpublished talk at the Third IMACS Conference on Expert Systems for Numerical Computing, 1993.

This work was supported in part by NSF Infrastructure Grant number CDA-8722788 and by NSF Grant number CCR-9214963 and CCR-9901933.

## References

[1] M. Abramowitz and I.A. Stegun (eds.), *Handbook of Mathematical Functions*, Dover publ. 1964.

[2] R. Akers, E. Kant, C. Randall, S. Steinberg, and R. Young, "SciNapse: A Problem-Solving Environment for Partial Differential Equations," IEEE Computational Science and Engineering, vol. 4, no. 3, July-Sept. 1997, 32–42. (see `http:/www.scicomp.com/publications`)

[3] C. Ballarin, K. Homann, and J. Calmet. "Theorems and Algorithms: An Interface between Isabelle and Maple." *Proc. of ISSAC'95*, ACM Press, (1995). 150–157.

[4] A. Bauer, E. Clarke, and X. Zhao. "Analytica, an Experiment in Combining Theorem Proving and Symbolic Computation,"*J. of Autom. Reasoning* vol. 21 no. 3 295–325, (1998)

[5] G.O. Cook, Jr. "Code Generation in ALPAL using Symbolic Techniques," *Proc. of ISSAC'92* ACM Press (1992), 27–35.

[6] D. Barton, K. M. Willers, and R. V. M.Zahar. "Taylor Series Methods for Ordinary Differential Equations – An evaluation," in *Mathematical Software* J. R. Rice (ed). Academic Press (1971) 369–390.

[7] M.C. Dewar., Interfacing Algebraic and Numeric Computation, Ph. D. Thesis, University of Bath, U.K. available as Bath Mathematics and Computer Science Technical Report 92-54, 1992. See also Dewar, M.C. "IRENA – An Integrated Symbolic and Numerical Computational Environment," *Proc. ISSAC'89*, ACM Press (1989) 171 – 179.

[8] R. Fateman. "Symbolic Mathematical Computing: Orbital dynamics and applications to accelerators," *Particle Accelerators 19* Nos.1-4, pp. 237–245.

[9] R. Fateman and W. Kahan. "Improving Exact Integrals from Symbolic Algebra Systems," Ctr. for Pure and Appl. Math. Report 386, U.C. Berkeley. 1986.

[10] R. Fateman. “Honest Plotting, Global Extrema, and Interval Arithmetic,” *Proc. ISSAC’92* ACM Press, (1992) 216–223.

[11] R. Fateman. “A short note on short differentiation programs in lisp, and a comment on logarithmic differentiation,” *ACM SIGSAM Bulletin* vol. 32, no. 3, September, 1998, Issue 125, 2-7.

[12] P. Fritzson and D. Fritzson. The need for high-level programming support in scientific computing applied to mechanical analysis. *Computer and Structures 45* no. 2, (1992) pp. 387–395.

[13] E. Gallopoulos, E. Houstis and J. R. Rice. “Future Research Directions in Problem Solving Environments for Computational Science,” Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science, April, 1991 Washington DC Ctr. for Supercomputing Res. Univ. of Ill. Urbana (rpt 1259), 51 pp.

[14] B.L. Gates, *The GENTRAN User’s Manual : Reduce Version,* The RAND Corporation, 1987.

[15] K. O. Geddes, S. R. Czapor and G. Labahn. Algorithms for Computer Algebra. Kluwer, 1992.

[16] A. Griewank and G. F. Corliss (eds.) *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* Proc. of the First SIAM Workshop on Automatic Differentiation. SIAM, Philadelphia, 1991.

[17] P. Henrici. *Applied and Computational Complex Analysis vol. 1 (Power series, integration, conformal mapping, location of zeros)* Wiley-Interscience, 1974.

[18] Richard D. Jenks and Robert S. Sutor. *AXIOM, the Scientific Computation System*. NAG and Springer Verlag, NY, 1992.

[19] N. Kajler, “A Portable and Extensible Interface for Computer Algebra Systems,” *Proc. ISSAC’92* ACM Press, 1992, 376–386.

[20] W. Kahan. “Handheld Calculator Evaluates Integrals,” *Hewlett-Packard Journal 31,* 8, 1980, 23-32.

[21] E. Kant, R. Keller, S. Steinberg (prog. comm.) AAAI Fall 1992 Symposium Series Intelligent Scientific Computation, Working Notes. Oct. 1992, Cambridge MA.

[22] D. E. Knuth. *The Art of Computer Programming, Vol. 1*. Addison-Wesley, 1968.

[23] D.H. Lanam, “An Algebraic Front-end for the Production and Use of Numeric Programs”, Proc. ACM-SYMSAC-81 Conference, Snowbird, UT,August, 1981 (223—227).

[24] E.A. Lamagna, M. B. Hayden, and C. W. Johnson "The Design of a User Interface to a Computer Algebra System for Introductory Calculus," *Proc. ISSAC'92* ACM Press, 1992, 358–368.

[25] J. C. R. Licklider, "Man-Computer Symbiosis," *IRE Trans. on Human Factors in Electronics*, March 1960.

[26] W. A. Martin and R. J. Fateman. "The MACSYMA System" *Proc. 2nd Symp. on Symbolic and Algeb. Manip* ACM Press, 1971, 59–75.

[27] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. *Numerical Recipes (Fortran)*, Cambridge University Press, Cambridge UK, 1989.

[28] J.' Purtilo. "Polylith: An Environment to Support Management of Tool Interfaces," |em ACM SIGPLAN Notices, vol. 20 no. 7 (July, 1985) pp 7–12.

[29] Sofroniou M. *Symbolic And Numerical Methods for Hamiltonian Systems*, Ph.D. thesis, Loughborough University of Technology, UK, 1993.

[30] J.R. Shewchuk, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Predicates for Computational Geometry," *Discrete and Computational Geometry 18:305-363*, 1997. Also Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996.

[31] W. Teitelman. *Pilot: A Step toward Man-computer Symbiosis*, MAC-TR-32 Project Mac, MIT Sept. 1966, 193 pages.

[32] P. S. Wang. "FINGER: A Symbolic System for Automatic Generation of Numerical Programs in Finite Element Analysis," *J. Symbolic Computing* vol. 2 no. 3 (Sept. 1986). 305–316.

[33] Michael C. Wirth. *On the Automation of Computational Physics* PhD. diss. Univ. Calif., Davis School of Aplied Science, Lawrence Livermore Lab., Sept. 1980.

[34] E.V. Zima. "Simplification and optimization transformation of chains of recurrences." *Proc. ISSAC-95*, ACM Press (1995) 42–50.

[35] D. Zwillinger. *Handbook of Differential Equations*, (CD-ROM) Academic Press, Inc., Boston, MA, 1989.