

Generation and Optimization of Numerical Programs by Symbolic Mathematical Methods

Richard J. Fateman *
University of California, Berkeley

November, 1999

Contents

1	Introduction	3
1.1	Symbolic or numeric	3
1.2	Recent CAS activities	4
2	Symbolic Mathematics Components	5
2.1	Programs that manipulate programs	6
2.1.1	Example: Preconditioning polynomials	7
2.1.2	Example: Generating perturbation expansions	9
2.2	Derivatives and Integrals	11
2.3	Semi-symbolic solutions of differential equations	13
2.4	Exact, high-precision, interval or other novel arithmetic	14
2.5	Finite element analysis, geometry, and adaptive precision	14
2.6	Code generation for special architectures	15
2.7	Support for proofs, derivations	15
2.8	Interchange and production of text	15
2.9	Display of data	17
3	Symbolic Manipulation Systems as Glue	17
3.1	Exchange of values	18
3.2	Why Lisp? Why not?	18
4	Two short-term directions for symbolic computing	20
4.1	Learning from specifics	20
4.2	The top-down approach	21

*Invited talk at Research Institute for Mathematical Science (RIMS), Kyoto University, Conference on Program Transformation, Symbolic Computation and Algebraic Manipulation, November 29 – December 1, 1999. Some material first appeared in a talk at THIRD IMACS INTERNATIONAL CONFERENCE ON EXPERT SYSTEMS FOR NUMERICAL COMPUTING, MAY 17-19, 1993

5	The Future	21
5.1	Symbolic tools available in some form	21
5.2	Tools not typically in any CAS	22
5.3	Scientific Visualization	22
5.4	Abstraction, representation, communication	23
5.5	System and human interface design issues	24
5.6	Miscellany	25
6	Acknowledgments	26

Abstract

Symbolic mathematical methods and systems

- support scientific and engineering “problem solving environments” (PSEs),
- The specific manipulation of mathematical models as a precursor to the coding of algorithms
- Expert system selection of modules from numerical libraries and other facilities
- The production of custom numerical software such as derivatives or non-standard arithmetic code-generation packages,
- The complete solution of certain classes of mathematical problems that simply cannot be handled solely by conventional floating-point computation.

Viewing computational objects and algorithms from a symbolic perspective and then specializing them to numerical or graphical views provides substantial additional flexibility over a more conventional view.

We also consider interactive symbolic computing as a tool to provide an organizing principle or glue among otherwise dissimilar components.

1 Introduction

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 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.

Of course the need for programmers has not disappeared: 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 fastest performance from these systems. Advanced computer programs now depend for their efficiency not only on clever algorithms, but also constrained 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 or 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 expects, are irregular in some way. That is, their “complexity” may be larger¹ 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 “ $\{n\pi \mid \text{integer}(n)\}$ ”. A program that *could* provide this more elaborate symbolic, non-numeric, parametric answer majorizes the merely numerical from a mathematical perspective. The single numerical

¹Although 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”.)

answer might be a suitable result for reasons of size, simplicity, speed, price, or portability, but it is a compromise.

What other characterizations might there be for uses of symbolic systems? In brief: if the problem-solving environment requires computing that includes asking and answering questions about sets, functions, expressions (polynomials, algebraic expressions), geometric domains, derivations, theorems, proofs, ..., then it is plausible that the tools in a symbolic computing system will be of some use.

1.2 Recent CAS activities

This decade has seen a flurry of activity in the production and enhancement and commercial exploitation of computer algebra systems (CAS). What have we learned in this time about prospects for automation of mathematics?

The grand goal of some early CAS research was the development of either an automated problem solver, or the development of a “symbiotic” human-machine system that could be used in analyzing mathematical models or similar tasks [23].

As an easily achievable 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.

In the optimistic pioneer days of the 1960s, some researchers (e.g. W. A. Martin [24]) felt that by continuing to amass more “mathematical facts” in computer systems, we would eventually see a computer system that, in important application areas, would be 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 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. However clever these packages appear at particular computations, they are unlikely 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.

Of course 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 than a symbolic calculator of algebra. They are 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 abstraction and representation and solution of a problem is a faithful rendition of all the features that must be manipulated.* The failure to adequately represent a feature makes manipulation of it difficult if not impossible².

What can be expected in the evolution of CAS? The last few decades of system builders, having accomplished what appeared to be the hard or at least novel (symbolic) part, have now begun to address “the rest” of scientific computing. *The grand goal is no longer to simulate human mathematical problem*

²We reserve judgment on the AXIOM system, which at least has a chance.

solving or to replace humans. The goal is to wipe out competing programs! This is pursued by providing a comfortable environment that allows the user to concentrate on solving the necessary intellectual tasks while automatically handling the inevitable but relatively mindless components — including transporting and storing files, converting data from one form to another, and tedious algebra and number crunching. A concession that emerged in the early 1970's was the idea that symbolic computation should take advantage of the numerical computing world by producing numerical source code to be executed outside the symbolic system. For the next decade (2000), we predict that ambitious system builders will emphasize their efforts to envelop all modes of scientific computation. Numerical systems such as Matlab and MathCAD have recently added symbolic capabilities through add-in packages. Computer algebra systems such as Macsyma, Maple, and Mathematica have added more-efficient numerical libraries. Axiom, a computer algebra system owned by a premier supplier of numerical algorithms (NAG), seems to be a minor player today, but has had an interest in easy-to-use interfacing [5].

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 computation systems also provide “notebooks,” world-wide-web interfaces, documentation, and even networking. The question seems to be who will be eaten by whom. This is not solely a technical issue of course.

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, it was not until the late 1980's that this software technology made a major entrance into the consumer and education market. It was pushed by cheap memory, fast personal computers and fancier user interfaces, as well as the appearance of newly engineered programs.

Now programs like Mathematica, Maple, Derive and Theorist are fairly well known. Each addresses at some level, symbolic, numerical, and graphical computing. The predecessor (and still active) systems like Macsyma, Reduce, SAC-2 as well as some relative newcomers (e.g. Axiom, MuPAD) are also worth noting for their occasional fresh insights.

Yet none of the commercial systems is designed to provide components that can be *easily* broken off and called by, for example, “Fortran” programs. (The initially appealing idea of calling a CAS from a numerical computation system is somewhat naive—what will the Fortran program do with a symbolic expression? 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 pieces, most CAS make an efforts to enable a programmer or user to somehow communicate in two directions with other programs or processes. These existing tools generally require a delicate hand.

A few non-commercial systems have been used successfully (no doubt in part because of delicate hands at work) when investigators needed separable components in the context of (for example) expert systems dealing with physical reasoning, or qualitative analysis. In particular, we have at hand evidence that those CAS whose host language is Lisp can be re-cycled to be used with other Lisp programs.

Lisp provides something of a base representation that can be used as a lingua franca between programs. Yet without any common internal data structures, other systems still claim to be useful as components. They typically offer some interconnection strategy which amounts to the exchange of character streams: one

program prints out a question and the other answers it. Not only is this clumsy and fragile³ it may make it impossible to solve significant problems whose size and complexity make character-printing impractical. It also inhibits interactions whose solution requires many branching decision steps whose nature cannot be easily predicted. The 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 use only oral communication. By talking on the 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 a way of teasing out components, or more plausible, the input/output interfaces for separate modules, from a computer algebra system. 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 code in assembly language (instead of binary!). We are used to manipulation of programs by compilers, symbolic debuggers, and similar programs. Today some research is centered on language-oriented editors and environments. These usually take the form of 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. In fact, another model of the program is being maintained by the environment 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 find computers can play a much more significant role in program manipulation. Often we see experimentation in program manipulation within the Lisp programming language because the data representations and the program representations have been so close⁴.

For example, in an interesting and influential thesis project, Warren Teitelman [29] 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 take (sqrt(abs ~s))"
      (first arglist))
    (setf arglist (list (abs (first arglist))))))
```

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

³For example, handling errors: streams, traps, messages, return-flags, etc. is difficult.

⁴Common Lisp has actually moved away from this kind of dual representation, and makes use of the distinction between a function and the lists of symbols that in some data form describe it. Some systems compile programs as soon as they are read in.

```
input: (sqrt -4)
sqrt given negative number. we take (sqrt(abs -4))
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
  (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 was 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 to automatically translate such English advice into Lisp, by *advising the advise program*. He also shows how to give 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 [31] 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. We include a selection of references to such work [21, 12, 30, 5, 27, 3, 1].

While we are quite optimistic about the potential usefulness of some of the tools, whether they are in fact practical 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 identify 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.

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 [25] (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))))))
...

```

Partly to show that Lisp, in spite of its 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
  bess10
  (* (/ (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, (which also can be translated back to Fortran if needed) 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.*

Computing this new 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, but would be much harder to program as part of a general programming-language compiler. Referring to this operation as `poly-eval` simplifies our considerations for the moment.

An additional caution: this rearrangement should in some sense be “licensed” by the designer of the program. It could be that in some programs the *exact sequence of operations as originally written* were cleverly constructed through approximations and series economization so as to cancel some of the round-off error. (This is not likely here.)

As long as we are working on polynomials, we should point out that the symbolic system can also provide source program statements to insert into the program to compute the maximum error in the evaluation at


```
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 using evalf

```
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 (in Maple, the command is `convert(expression, horner, [e])`) 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
```

Macsyma's facilities are approximately the same as Maple's in generating Fortran.

The last formula (from Maple) has multiple occurrences of $\sin u$ that should be noticed by a decent Fortran compiler, and precomputed. However a numerical compiler probably 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, as show below.

$$s_2 = \sin 2u = 2 \cdot s \cdot c, \quad c_2 = \cos 2u = 2c^2 - 1, \quad s_3 = \sin 3u = s \cdot (2c_2 + 1), \quad s_4 = \sin 4u = 2s_2c_2, \text{ etc.}$$

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 the *de novo* calculation S by calculation of $S[\mathbf{z}]$ as an update to 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. (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 [32].)

More specifically, if we return to the Euler equation program, we can compute $\sin(a + nd)$ and $\cos(a + nd)$ for fixed a and d for $0 \leq n \leq k$ in various ways. Probably for $k > 3$ the best is using a recurrence based on two previous values which, for each additional sin and cos pair, requires only two adds and two multiplies. Using the facts that

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

we can construct a simple program whose inner loop looks something like

$$\begin{aligned} s_i &:= s_{i-2} + c_{i-1} \\ c_i &:= c_{i-2} - k_1 * s_{i-1} \\ s_{i-2} &:= k_2 * s_{i-2} \end{aligned}$$

We have chosen this illustration because it shows that

1. Symbolic computing can be used to produce not only large expressions, but also relatively small ones. Furthermore, the important contribution of these small expressions is to form sections of numerical computation programs. By extension, of course, larger expressions can be subjected to similar manipulation.
2. Some relatively small expressions may nevertheless be costly to compute, especially in inner loops; there are tools to make this faster.
3. Having computed expressions symbolically, prematurely converting them to Fortran (etc.) is a mistake.
4. It would be nice (and in some cases possible) to produce auxiliary expressions or programs related to the main computation. In particular, keeping track of errors can be done for polynomial evaluation. This can be done as well as for the Euler equation, although we haven't given details here.
5. There is a use for the expressions: at least small ones can be routinely transferred into typeset form for inclusion in papers such as this. Hand-massaging of large forms requiring custom line-breaking may be painful in current systems. These expressions are different from ones you would compute with.

Incidentally, although CAS vendors boast about their systems' capability of producing hugely-long expressions, it is usually a bad idea to dump these into a source file. Not only are such expressions likely to suffer from instability, but their size causes problems in many compilers: they may strain common-subexpression elimination "optimizers" and exceed the input-buffer sizes or stack sizes. Breaking expressions into smaller "statement" pieces for evaluation is not difficult; even better may be the systematic replacement of expressions by calculation schemes such as `polyeval` discussed previously.

2.2 Derivatives and Integrals

Many students who having studied the use of a "symbolic" language like Lisp will have seen differentiation as a small exercise in tree-traversal and transformation. They will likely view closed-form symbolic differentiation as trivial, if for no other reason than it can be expressed in a half-page of code (We've shown a 14 line program [9]).

Unfortunately the apparent triviality of such a program is purchased by peddling misconceptions on several levels. A serious CAS will have to deal efficiently with far more complicated expressions than $x \sin x$. 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 solving these problems still doesn't address the real application for scientific programmers, an application which for computer science students, defies intuition. The application is often stated as the differentiation of a Fortran "function subroutine." Approached from a general perspective, 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 compute "derivatives of programs" efficiently and accurately, this can be an engrossing and worthwhile challenge. The alternative is using a "numerical" derivative of $f(x)$ at a point a computed by choosing some small Δ 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 [14] considers a wide range of tools: from numerical differentiation, through pre-processing of languages to produce Fortran, to new language designs entirely (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,

Hessian s, etc.) rapidly. Tools such as ADIFOR www.mcs.anl.gov/adifor are receiving more attention. The use of explicit symbolic manipulation as in lisp is rarely a consideration in these programs.

To show why, consider how one 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. In that case, 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.)

While it seems handy to not have to write this, far more useful is a pre-processor or compiler-like system which automatically and without even exhibiting such code, produced implicitly the computation of $f'(x)$ in addition to $f(x)$. Consider that every scalar variable v is transformed to a vector $(v(x), v'(x), v''(x), \dots)$ 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? Can we integrate them? Surely the closed form versions of integrals of expressions appear in programs: the programmer has inserted the formula and thereby avoided numerical quadrature programs. Does it make sense to leave the integral “unevaluated” in the program?

How would this work? While we are used to some “constant folding” by an optimizing compiler, few programmers would expect a compiler to “closed-form” fold, say by noticing $\sum_{i=1}^n i$ can be replaced by $n(n+1)/2$ or replacing an integral by its closed form. Certainly this can be done by CAS intervention.

It is perhaps less obvious that such compilation should be attempted, given the large sizes one can encounter: the closed form may be more painful to evaluate than the quadrature. Example:

$f(x) = 1/(1 + z^{64})$ whose integral is

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

where $c_k := \cos((2k-1)\pi/64)$ and $s_k := \sin((2k-1)\pi/64)$. [18]

Other examples can be found when the closed form, at least under usual numerical evaluation rules, is either not especially stable for computing, or inefficient.

Perhaps the most trivial example is $\int_a^b x^{-1} dx$ which most computer algebra systems give as $\log b - \log a$. Even a moment’s thought suggests a better answer is $\log(b/a)$. Or if we are going to do this carefully, a

numerically preferable formula would be something like “if $0.5 < b/a < 2.0$ then $2 \operatorname{arctanh}((b - a)/(b + a))$ else $\log(b/a)$.” [7]. 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 these cases we do not mean to argue that the symbolic result is mathematically wrong, but only that the result is the correct answer to the wrong question. The tool is not adequate to determine the appropriateness of the answer. Certainly a slightly level higher approach to some of these problems can be programmed in a CAS — one that might deliberately choose numerical quadrature over symbolic integration even though the latter is possible. And returning $\log(b/a)$ ordinarily would not be difficult. A more ambitious 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 are a variety of areas of mixed algebraic and numeric techniques. We discuss one area of application in this section.

There is a large literature on the solution of ordinary differential equations. For a compendium of methods, Zwillinger’s handbook on CD-ROM [33] 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.

There is a detailed exposition by Henrici[15] 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* [4]. In brief, if you have a system of differential equations $\{y'_i(t) = f_i(t, y_1, \dots, y_n)\}$ with $\{y_i(t_0) = a_i\}$ 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 rather plodding program we gave for taking a second derivative above, is of this nature.) The resulting power series could be a solution, but its validity in some region depends on that region being within the radius of convergence of the series. A truncated power series will become less accurate in as one approaches singularities. 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 values that are computing from the 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 function at all! To quote from Barton [4], “[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: solution of differential equations.

2.4 Exact, high-precision, interval or other novel arithmetic

Sometimes using ordinary finite 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 precision is another feature, useful not so much for routine calculations (since it too is slow), but for the critical evaluation of key expressions. This can also involve non-rational functions, making it more versatile than the exact arithmetic. 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: 262537412640768743.999999999992500726. 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.

We do not see the occasional NaN encoding as a substitute for the high-speed floating-point computation usually needed for scientific work. It is an adjunct to test for benchmark values for computations.

There is a substantial literature of competing computation 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 [30] 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 [28] 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 uncertainty of the result.

Finite element code is but one example of an area where symbolic manipulation seems plausible as an adjunct to numerical code generation. Other systems (e.g. [3]) aimed at other application techniques or even specific problems are under investigation, and there is a substantial literature developing here.

The interested reader can pursue this via the references.

2.6 Code generation for special architectures

Especially within the context of high-performance computing it seems plausible that we can move from a high-level mathematical model of a problem to a specific variation of code for specialized environments.

That is, 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, but need not be.

Such unintentional constraints are the biggest problem in retaining old source languages (typically Fortran, but perhaps we now have legacy code in newer systems like Matlab). Merely changing the compiler has the attraction that the application programmer may be spared reprogramming effort, but cannot get the best programs. The best compiler technology now requires that a program be run 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. Among other practitioners of software-writing software, SciComp Inc. has looked at PDE solving in this way [1], although they seem now to be targeting financial computations.

2.7 Support for proofs, derivations

Of the computer algebra systems now available, only Theorist makes some attempt to maintain a correct line of transformations. While it is in practice possible to use computer algebra in proofs, only specialized “theorem proving” systems have, to date, taken this task seriously. Indeed serious work at proving transformations correct by representing functions and domains in the complex plane, has been proposed at various times, (work by A. Dingle, UC Berkeley) but remains a substantial challenge.

2.8 Interchange and production of text

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 type-set 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 stored in Lisp or perhaps in XML/MathML web forms, and then perhaps 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 specifications⁵.

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, at least three computer algebra systems (Macsyma, Maple, Mathematica) convert their internal representations into \TeX on command. Macsyma’s internal representation is in Lisp, and Mathematica’s representation, as illustrated by its `FullForm` printout, is essentially Lisp with square brackets instead of parentheses.

While none of these systems 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.)

A suitable representation of a problem can (and 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

⁵We 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 `{(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.

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 disastrous 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 spite of efforts to develop them, computer algebra “killer applications” are still lacking. It seems to us that the major organizing paradigm should be centered more effectively around a class of important problems, or even a particular problem. That is, “a notebook” is not going to be as convincing as “the world’s most expert generator of finite-element code” or even “the world’s best financial model generator”.

2.9 Display of data

The problem of paging through output from a simulation, or viewing a collection of plots as animation, has been approached in many systems. Some systems have been dismissed as “toys” because they require that the plots all be present in main memory before any work can be done on them. This is clearly something that can be remedied: The locations of secondary files, or even the labels of back-up tapes can certainly be made part of a compound object representing a problem. The timely retrieval of this data may be a legitimate problem to address as a component of a problem solving environment⁶.

3 Symbolic Manipulation Systems as Glue

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

Gallopoulos *et al* [11] 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 backbone). 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 such as Perl, Tcl, Python. Lisp as well as Computer Algebra Systems are not usually in the list, though we think they should be there. They do not figure prominently partly because they are “large” and they are thought of as specialized for mathematics. (Our view is that they are quite general, and probably have all the relevant features plus more.) 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. Combined, these features tend to provide an opportunity 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 program development is part of the objective, the inability of some scripting languages to be able to handle complicated objects is a critical failure. The long history of effectively treating Lisp programs as

⁶To say that this is a database problem is not particularly helpful. The capabilities offered by conventional database management systems are typically a poor match; existing systems tend to squander resources on mostly irrelevant issues such as atomicity of transactions.

data and data as programs is a particular strength of this language.

3.1 Exchange of values

We would actually 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 particular Common Lisp implementation with a “foreign function” interface. (These are not standardized, but several full-scale implementations have similar appearances and capabilities).

On the workstation at which I am typing this paper, and using Allegro Common Lisp⁷, if I have developed a Fortran-language package 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 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 data that made sense to Fortran. A system in which everything would be converted to character strings, as in the UNIX operating system pipes convention, would open up “non-numeric” data, but would be quite inefficient for numeric data. 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. It is possible for the “foreign” functions to violate integrity for Lisp data in ways that Lisp itself would be unlikely to attempt, like addressing past the end of an array, but such transgressions are possible in those languages already (especially 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 in most commercial and even academic Lisp systems, providing access to (depending on your host machine and operating system) X-window, Macintosh, Microsoft-Windows, and other interfaces.

I have myself hooked up Lisp to an arbitrary-precision floating-point package, and others have interfaced to the Numerical Algorithms Group (NAG) library, and the library from *Numerical Recipes*. Interfaces to SQL and database management systems have also been constructed at Berkeley and apparently elsewhere.

3.2 Why Lisp? Why not?

The linkage of *Lisp-based* symbolic mathematics tools such as Macsyma and Reduce into Lisp naturally is in a major sense “free.”

Linkage from a PSE to symbolic tools in other languages is also possible, at least according to vendor information: if one were to wish to make use of (say) Mathematica or Maple, each has a well-defined interface, albeit via a somewhat narrow channel. Yet one would might have considerable difficulty tweezing out just a

⁷the details differ for other brands

particular routine like our Fortran 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.

If you were to need only a few commands, these could be set up, as it appears MathCAD and Matlab programs have done (to link with Maple).

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 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, the possibility of compiling programs for efficiency is sometimes an important concern. The prototyping and debugging environments are dramatically superior to those in C, even though interpretive C environments 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. There are fringe cases, even with lists, that have to do with structure sharing, or even exact binary-to-decimal conversion, but these too can be accommodated. 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 the structures.

Modern Lisps have abandoned this principle of built-in universal read/write capabilities: Although every structure has some default form for printing, it may not be enough for the reader to reconstruct it. Arbitrary “objects” may have print-methods associated with them that are “lossy”. Structures in Common Lisp such as hash-tables or compiled functions may also, with justification, have no default full-information printout.

In spite of our expressed preference, there other possible glues: a popular interactive system nicely balanced and specialized to the X-window based system is TCL/TK. Other possibilities are “smaller” variants of Lisp including Xlisp, Dylan, Elfin, and a number of Scheme dialect systems. Languages like Forth or RESX may help. Their advantages are primarily in the size of the delivery system and the simplicity of their data and programs: By contrast, Common Lisp typically uses several megabytes of memory more than these smaller systems (4 or even 8 megabytes might be expected). In some senses, you get what you pay for. Although Common Lisp is larger than needed for any single application, some of the features that are included may later prove useful. Some programmers using C++, for example, seem to realize that memory allocation and freeing via “garbage collection” is not just a frill. Oddly enough, we have encountered some “easy-to-use” glues are even larger than Lisp – heavily elaborated visualization systems like AVS are hardly lightweight.

Arguing for small size when full-featured computer algebra systems with graphics (etc) like Mathematica or Maple are already bulky, seems less important these days.

Serious consideration has also been given to building scientific programming environments in a few other (non-C) languages, of which the most interesting are probably Smalltalk and Prolog. My opinion is that these alternatives are less likely for a host of reasons, but certainly significantly less advantageous because

they lack the library of programs for symbolic scientific computing. Maple and Mathematica have been promoted as a glue languages, and they have interesting prospect of attracting, through widespread use, a thorough library. The generally spotty quality of contributed user software, and the inefficiency of the interpreted language for numerics makes them somewhat less attractive; its proprietary internals and cost are also barriers.

These days the major language of system implementation seems to be C, or some variant of C; it has the advantage that the UNIX operating system is written in C, and thus the C programmer tends to get nearly the same level of access to facilities as the system builder. This is also one of its principal disadvantages: without discipline or type-checking at interfaces, (and C imposes very little) the programmer can get into deep trouble.

We hope to benefit from the current increase in exploration and design of languages for interaction, scripting, and communication.

4 Two short-term directions for symbolic computing

Martin's [24] 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 [16]. 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 is reasonably successful for algebra, efficient implementation is difficult and it appears that compromises to unalloyed algebra may be needed in engineering mathematics.

By contrast, the bottom-up approach provides an opportunity to identify some of these compromises more directly by finding or building successful applications and then generalizing. At the moment, this latter approach seems more immediately illuminating, and likely to demonstrate application successes.

We discuss these approaches in slightly more detail below.

4.1 Learning from specifics

As an example of assessing the compromises needed to solve problems effectively, consider the work of Fritzson and Fritzson [10] 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 (for example) 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 Fritzsens' 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.

4.2 The top-down approach

The approach implicit or occasionally explicit in some CAS development has been more abstract: Make as much mathematics constructive as possible, and hope that applications (which, after all, use mathematics) will follow.

It is easy to confuse it with a more rational approach of starting with a problem, then generalizing it. Some systems started with *no* problem (or perhaps a trivialized and oversimplified example) and generalized. In addition to the challenge of being irrelevant, is that general constructive solutions may be too slow or inefficient to put to work.

Yet it seems to us that taking the "high road" of building a constructive model of mathematics is an inevitable, if difficult, approach. Of the commercial CAS today, AXIOM seems 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.

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

5 The Future

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

5.1 Symbolic tools available in some form

These capabilities are available in at least one non-trivial form, in at least one CAS.

- 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 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 (series)
- Access to numerical libraries
- Typeset quality equation display / interactive manipulation
- 2-D and 3-D (surface) plots
- On-line documentation, notebooks.

We will not discuss them further, although documentation for most CAS will cover these topics.

5.2 Tools not typically in any CAS

These tools, capabilities, or abstractions are generally not included in a typical CAS, although many of them are available in some computerized form, usually 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-base problem solving
- Qualitative analysis
- Derivations, theorems, proofs
- Various group theory and number theory calculations
- Mechanical, electronic, or other computer-aided design data

5.3 Scientific Visualization

As an example of the areas which are supported in numerical components that can be used in PSEs but could be strengthened by consolidation with CAS capabilities, 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 [8], 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; perhaps a limit calculation is appropriate. Of course these techniques are relevant to functions defined mathematically and for the most part do not pertain to plots of (sensor-derived) data, although symbolic manipulation of interpolation forms is possible.

In principle these would add to the traditional graphical or design facilities such as

- representation of geometry/intersections and algebraic linkages
- display of curved surfaces (3-D, contour)
- display of higher-dimensional objects
- lighting models
- texture models
- animation.

5.4 Abstraction, representation, communication

The future may also bring a renewed interest in multiple representations of expressions, and techniques for communication amongst programs. Currently most CAS emphasize manipulation in an environment consisting of viewpoints on *centrally defined* models. We do not object to the simultaneous use of some representation of $\sin x$ to type-setting program as a string `\sin x` appropriate for T_EX or a similar notion relayed to Mathematica as `Sin[x]`. We do, however, object to a vision of a problem-solving environment where the abstraction is missing — where all that exists is a set of 2^n communication protocols amongst the n programs.

Some environments must deal with substantial collections of data from simulations, physical sensors, statistical, meteorological, demographic, or economic data. Even textual data from electronically available publications can represent a sizable database. An environment with these collections must also have an adequate framework for manipulation of data sets. Operations like selection, searching, correlation, display, may apparently fit with the suite of facilities in a database system. Yet today’s database technology seems inadequate to meet the needs of large-scale scientific computing and visualization.

In order to provide a foundation for future growth and extension, it would be helpful if designers and implementors could agree on some representation issues. Currently we see the follow

1. Lack of agreement on acceptable compromises. In fact, one person’s minor inefficiency in the name of portability is another’s major roadblock. One person’s hard problem is another’s non-problem. As an example, consider a person laboring to develop a computer system that can tell if a differential equation is elliptic or hyperbolic or parabolic. (or none of the above). This would, to the naive, seem an imperative first step, since numerical solution techniques differ. On the other hand, any human who is working seriously on a differential equation mathematic model is quite unlikely to require computer assistance in this classification: he will know which it is.

Another example: a system that plots points on a display is built around the assumption that the data it is plotting fits in main-memory on the display computer. Large data-sets cannot be handled.

2. Disagreement on technical issues: Examples: When are problems exact (root isolation vs root finding?), when are closed-form integrals required instead of quadratures; are matrices likely to be dense (for choosing optimal algorithms);
3. Lack of good standards. There may be standards, e.g. X11R5 Window system, that are themselves inadequate – both then how many distinct and non-standard “standard” interfaces are there to X11? Does the interface also work with MS Windows, and hence involve additional compromises? Does the system have to run on machines with different arithmetic, word size, file system naming conventions etc? If Postscript is used for portable graphical display, then all the deficiencies of Postscript may have to be accommodated. In fact, since backward compatibility may be important, future systems may not even be able to take advantage of revisions of Postscript!
4. The desire to build your own. It is frequently tempting to write a new system from scratch because
 - (a) This is (thought to be) better training for students.
 - (b) If you use a pre-existing system, then you may have to pay for it. You, and your potential software “customers” will have to make this investment. While building on public domain (or freely available) source code has advantages, generally it is less robust than commercial code. (Most people concede that they have to use a commercial operating system, but this is not necessary.)
 - (c) The only way to understand some systems is to try to build them yourself⁸.
 - (d) You can get funding for the early phases of a project – design and prototype; you cannot get funding for refining your own or someone else’s program. A succession of prototypes is the result.
 - (e) If you try to use someone else’s code, you may find out that the code contains errors, is unreadable and undocumented, and can’t be changed without a high probability of introducing errors⁹.

The notion of re-usable software has become a touchstone for software engineering. Everyone wants it, but no one knows how to build re-usable software above a rather low level. Even the carefully designed scientific subroutine libraries are challenges – who can use a Fortran subroutine with 18 arguments?

The consequences of competing groups continuing to build incompatible systems includes a substantial lack of quality control for components, and a failure to produce re-usable modules. Furthermore, such operations make it nearly impossible to have quality documentation.

5.5 System and human interface design issues

At Berkeley, we thought that the availability of the first Sun Microsystems workstations (in 1982 or so) running the computer algebra system Macsyma, would drastically change the notion of computer algebra user interfaces. Although Symbolics workstations had previously run Macsyma on graphics systems, the hardware was expensive and the software apparently neglected. The combination of a relatively low-cost and high-powered graphics workstation plus symbolic math seemed like a synergistic combination. Plotting, typesetting, menu-selection programs etc. were written and then obsoleted as the underlying software support

⁸As an instructor in a programming language class, it becomes clear that students writing a compiler gain a very good understanding of the behavior and semantics of their target programming language. Almost all ambiguity is removed by building a compiler.

⁹Of course your own code will not have such problems!

was removed in successive incompatible waves. We were able, however, think more about the choices open to us for implementation of capabilities.

In electronic mail, then-graduate student Ken Rimey provided something of checklist of considerations for an environment (in this case primarily to support interactive mathematics itself), based on experiences of our group (and especially students Gregg Foster, Richard Anderson, and Neil Soiffer.)

This was, in fact, at least four years into our explorations, and it seems that the answers to questions posed in 1986 are elusive still. Although we were concerned with computer algebra at least as a unifying theme, it seems that our concerns are really the same as for PSE design, or user interfaces generally.

- How elaborate should the display be? Can we simultaneously provide ease of use for novice vs. comprehensive and compact display for expert? How can you display the “state” of a complex system?
- Given menus, functional command style, handwritten or mouse/stylus-based expression of manipulation. Which is preferable and when? How large a menu can you stand? Can you come up with icons for everything? (We’ve since seen a number of systems with truly obscure icons. Several color painting/drafting systems and visualization systems stand out for high obscurity.)
- Can you effectively tailor a system to an application, moving the important stuff up front with the (temporarily) less useful stuff to background or off line?
- What number of commands is the system being designed for? Will there be more than can be memorized? Will we need “shortcuts”? Is “command completion” a better choice?
- How powerful is the typical command? The standards for computer algebra systems like “simplify” or “solve” are highly unlikely to do everything expected of them; even “integrate” may be illusory. How much effort from the system is required? For example, in the absence of a closed form from integration, should a numeric answer be offered?
- Using a mouse for selecting text strings, positions, or motions (put the file in the trash) can be intuitive or not. If it is necessary to specify more than one (or perhaps two) arguments, are you stuck with some cumbersome and/or unforgiving and/or error-prone mechanism? Does the order of the arguments need to be memorized?
- Is system response time critical, as it might be when popping up a menu?

5.6 Miscellany

We have already meandered into byways of problem-solving with symbolic environments, but have hardly touched on a number of issues that are also of concern: Education: how do people learn to use these computers? Can they be used as teaching tools for disciplines such as mathematics or engineering? , Library interfaces: not just to numerical libraries, though that is important – what about on-line catalogs of journals, data, etc.?

Do we really even understand how to link programs together? [26] CAS/PI [17] We have direct calls, multiple interconnected processes, input/output streams, files, etc. Is there a “software bus” structure that would work? Do we have good ideas for taking advantage of a parallel symbolic computation environment?

These issues, which in our experience were first raised in the computer algebra context, are certainly as relevant in the exploration of PSEs generally.

6 Acknowledgments

Discussions and electronic mail with Ken Rimey, Carl Andersen, Richard Anderson, Neil Soiffer, Allan Bonadio and others have influenced this paper and its predecessor notes.

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

References

- [1] 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, pp 32-42. (see <http://www.scicomp.com/publications>)
- [2] V. Anupam and C. Bajaj. Collaborative Multimedia Scientific Design in SHASTRA. Proc. of the 1993 ACM SIGGRAPH Symposium on Multimedia. Anaheim, CA.
- [3] Grant O. Cook, Jr. *Code Generation in ALPAL using Symbolic Techniques*, in Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1992, P. Wang, Ed., Berkeley CA, 1992, ACM, New York, pp. 27-35.
- [4] 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.
- [5] Dewar, M. C., 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," Proceedings of the ACM-SIGSAM International Symposium on Symbolic and Algebraic Computation (ISSAC'89), Portland, Oregon, ACM Press 171 – 179, 1989
- [6] R. Fateman. "Symbolic Mathematical Computing: Orbital dynamics and applications to accelerators," *Particle Accelerators 19* Nos.1-4, pp. 237-245.
- [7] R. Fateman and W. Kahan. Improving Exact Integrals from Symbolic Algebra Systems. Ctr. for Pure and Appl. Math. Report 386, U.C. Berkeley. 1986.
- [8] R. Fateman. "Honest Plotting, Global Extrema, and Interval Arithmetic," *Proc. Int'l Symp. on Symbolic and Algebraic Computation* (ACM Press), (ISSAC-92) Berkeley, C. July, 1992. 216-223.
- [9] R. Fateman. "A short note on short differentiation programs in lisp, and a comment on logarithmic differentiation," *ACM SIGSAM Bulletin* Volume 32, Number 3, September, 1998, Issue 125, 2-7.
- [10] 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.
- [11] 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.

- [12] Gates, B. L. , 1987, The GENTRAN User’s Manual : Reduce Version. The RAND Corporation.
- [13] K. O. Geddes, S. R. Czapor and G. Labahn. Algorithms for Computer Algebra. Kluwer, 1992.
- [14] 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.
- [15] P. Henrici. *Applied and Computational Complex Analysis vol. 1 (Power series, integration, conformal mapping, location of zeros)* Wiley-Interscience, 1974.
- [16] Richard D. Jenks and Robert S. Sutor. *AXIOM, the Scientific Computation System*. NAG and Springer Verlag, NY, 1992.
- [17] N. Kajler, *A Portable and Extensible Interface for Computer Algebra Systems*, in Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1992, P. Wang Ed., Berkeley CA, 1992, ACM New York, pp. 376–386.
- [18] W. Kahan. “Handheld Calculator Evaluates Integrals,” *Hewlett-Packard Journal* 31, 8, 1980, 23-32.
- [19] E. Kant, R. Keller, S. Steinberg (prog. comm.) AAAI Fall 1992 Symposium Series Intelligent Scientific Computation, Working Notes. Oct. 1992, Cambridge MA.
- [20] D. E. Knuth. *The Art of Computer Programming, Vol 1*. Addison-Wesley, 1968.
- [21] Douglas 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).
- [22] Edmund A. Lamagna, M. B. Hayden, and C. W. Johnson The Design of a User Interface to a Computer Algebra System for Introductory Calculus, in Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1992, P. Wang Ed., Berkeley CA, 1992, ACM New York, pp. 358–368.
- [23] J. C. R. Licklider, “Man-Computer Symbiosis,” IRE Trans. on Human Factors in Electronics, March 1960.
- [24] W. A. Martin and R. J. Fateman. “The MACSYMA System” Proc. 2nd Symp. on Symbolic and Algeb. Manip. March, 1971, Los Angeles, CA. p. 59–75.
- [25] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. *Numerical Recipes (Fortran)*, Cambridge University Press, Cambridge UK, 1989.
- [26] James Purilo. Polyolith: An Environment to Support Management of Tool Interfaces, ACM SIGPLAN Notices, vol 20 no. 7 (July, 1985) pp 7–12.
- [27] Sofroniou M. *Symbolic And Numerical Methods for Hamiltonian Systems*, Ph.D. thesis, Loughborough University of Technology, UK, 1993.
- [28] Jonathan 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. <http://www.cs.cmu.edu/quake/robust.html>.

- [29] Warren Teitelman. Pilot: A Step toward Man-computer Symbiosis, MAC-TR-32 Project Mac, MIT Sept. 1966, 193 pages.
- [30] P. S. Wang. "FINGER: A Symbolic System for Automatic Generation of Numerical Programs in Finite Element Analysis," *J. Symbolic Computing* 2 no. 3 Sept. 1986). 305–316.
- [31] Michael C. Wirth. On the Automation of Computational Physics. PhD. diss. Univ. Calif., Davis School of Applied Science, Lawrence Livermore Lab., Sept. 1980.
- [32] E.V. Zima. "Simplification and optimization transformation of chains of recurrences." *Proc. ISSAC-95*, ACM, Montreal, Canada, 1995 42–50.
- [33] Daniel Zwillinger. *Handbook of differential equations*, (CD-ROM) Academic Press, Inc., Boston, MA, 1989.