Problem Solving Environments and Symbolic Computing

Richard J. Fateman*
University of California, Berkeley

July 14, 1997

Contents

*based in part on a talk at THIRD IMACS INTERNATIONAL CONFERENCE ON EXPERT SYSTEMS FOR NUMERICAL COMPUTING, May 17-19, 1993

Abstract

What role should be played by symbolic mathematical computation facilities in scientific and engineering "problem solving environments"? Most observers agree with us that in conjunction with numerical libraries and other facilities, symbolic computation should be useful for the creation and manipulation of mathematical models, the production of custom numerical software, and the solution of certains classes of mathematical problems that are difficult to handle by traditional floating-point computation. Even further, though, symbolic representation and manipulation can potentially play a more central role – with more general representations a program can naturally deal with computational objects of a more general nature. Numerical, graphical, and other processing can be viewed as special cases of symbolic manipulation. Therefore interactive symbolic computing can provide an organizing backbone as well as the glue among otherwise dissimilar components.

1 Introduction

We can point to successful examples of complete, "user-friendly" problem-solving programs *if the problem domain is sufficiently restricted.* Consider the multiple-choice, non-programmable user interface an Automatic Teller Machine provides for you to state your queries, and the expertise the ATM demonstrates in its responses.

Of course, achieving such success in a more ambitious setting of solving engineering problems in a broad class is considerably more challenging. In fact, we generally have to shift gears considerably so as to consider the design of a system that is relatively "open" and expressive. We turn to the currently evolving concept: that of a "problem solving environment" (PSE [8]) to help a user define a problem clearly, search for its solution, and understand that solution. We are also concerned with a kind of 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." **This paper surveys some concepts and tools available in symbolic computation both for the meta-PSE and the application PSE.**

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

What makes a symbolic computing system distinct from a non-symbolic (or numeric) one? One general characterization, the questions one asks (and especially the resulting answers one can expect to obtain) are irregular in some way. That is, their "complexity" may be larger[1] 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 and "non-numeric" answer majorizes the merely numerical, and might be preferable overall[2].

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.

This decade has seen a flurry of activity in the production and enhancement 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 [18].

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 cleverness in limited circumstances.

In the optimistic view of pioneers in the 1960s, some researchers (e.g. W. A. Martin [19]) 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

[1] 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 (typically by an array in Fortran COMMON allocated at "compile-time".)

[2] However one may still prefer the numerical program sometimes for reasons of size, simplicity, speed, price, or portability.

– including some or all of the demanding 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; howeve, I do not believe Martin would find any of them as yet "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) to synthesize some kind of model for solving problems.

Of course the currently available CAS are in reality far less ambitious than Martin's "artificial mathematician". They are, however, more ambitious than just a symbolic calculator. They are beginning to look like 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 by 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 impossible[3].

What can be expected by natural incremental evolution of CAS? The system builders, having accomplished what appeared to them to be the hard or at least novel (symbolic) part, proceed to "the rest" of scientific computing as they see it. The grand goal is now to augment human problem solving, not to replace humans. The goal is to provide a comfortable PSE that allows the user to concentrate on solving the task at hand and automatically handles those inevitable but relatively mindless necessities – including transporting files or other forms of data from one form to another, as well as tedious algebra.

The incremental evolution of existing symbolic math systems is affected by their construction. Bby virtue of their finite size they each take a partic-

[3]We reserve judgment on the AXIOM system, which at least has a chance.

ular subset of facilities as foundational. Much of this basic "kernel" is common across most systems, and ordinarily includes some linguistic facilities for adding in more capabilities in a relatively natural fashion[4]. The kernel provides efficiently implemented critical algorithms and defines base representations. A design for a general system must balance efficiency against extensibility. From the PSE perspective, this kernel plus extension philosophy makes perfect sense. In fact, one might hope that a meta-PSE, as indicated earlier, would provide a kind of kernel which can, by suitable extensions, be make into various application-PSEs.

In the next section of this paper we try to be more concrete and discuss symbolic components that might provide specific utility to a PSE, but we then return to the notion that symbolic computation has a "higher calling" in PSEs: by providing the framework for representing mathematical problems in a more natural (i.e. symbolic) notation ([8] section 3.3), such facilities can form the connections between previously dissimilar and separately composed sub-systems. After all, if symbolic processing encompasses everything "non-numeric" and indeed also includes "numeric" processing as a subset, what is left?

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 newcomers (e.g. Axiom) are also worth noting.

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 Fortran is somewhat

[4]Some level of kernel plus extended system is what most users ordinarily see when they start up a system

naive – what will the Fortran program do with a symbolic expression? Since Fortran deals with numbers or arrays of numbers, what Fortran expression can store an expression?)

Even if one cannot 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, there is 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 fragile[5] 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. Especially if noise on the phone line could result in one or both of the mathematicians losing all memory of the problem and erasing the blackboard.

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 terfaces 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?

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

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 close[6].

For example, in an interesting and influential thesis project, Warren Teitelman [22] at MIT 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.

By contrast, other researchers of about that time prevail on their systems not so much for generality as for specificity: systems are aimed at solving particular classes of problems by patching together new pieces with old in a stereotypical way, using templates, expressions to be evaluated, derivatives of expressions, etc. We can date back to at least the late 1970s, ambitious efforts using symbolic mathematics systems (e.g. M. Wirth [24] who used Macsyma to automate work in computational physics).

While we are quite optimistic about the usefulness of some of the tools,

[6]Common Lisp has actually moved away from this kind of dual representation, and makes use of the distinction between a function and the list of symbols that in some data form describe it.

we also feel an obligation to voice cautions about some of the "obvious" uses that proponents of CAS may identity, but which may actually be questionable.

2.1.1 Example: Preconditioning polynomials

A well-known yet 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 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 [20] (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, is need not be ugly, and partly to aid in further manipulation, we can rewrite this as Lisp, abstracting the polynomial evaluation operation, as:

```
(setf
 bessi0
 (* (/ (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))))
```

And we now can reformulate this – by symbolic manipulation: into a *preconditioned* version of the above `(poly-eval ...)`

```
(let* ((z (+ (* (+ x -0.4474202468916 62d0) x) 0.5555574445841143d0))
       (w (+ (* (+ x -2.180440363165497d0) z) 1.759291809106734d0)))
  (* (+ (* x (+ (* x (+ (* w (+ -1.7459865688143 45d0 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 you insist) is that it uses 6 multiplies and 7 adds while the original Fortran program uses 8 multiplies and 8 adds. (Horner's rule also uses 8 multiplies and 8 adds).

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 not exactly trivial, but we suspect that it would be much harder in a purely numeric language than in Lisp.

As long as we are working on polynomials, we should point out that the symbolic system can also in principle, provide program statements (in Lisp or Fortran) to insert into the program to compute the maximum error in the evaluation at run-time, as a function of the machine-epsilon for the floating-point type, and other parameters. This is the kind of program-writing assistance

Even better we can imagine a situation in which the "program manipulation program" could provide an error bound for evaluation of a polynomial *BEFORE it is RUN*: given a range of values for x (in this Bessel function evaluation context we know that $0 \leq x \leq 3.75$) it could 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 Bessel function evaluation routine with an *a prior* error bound.

This is the kind of practical expertise in a somewhat arcane subject (error analysis) that should be provided in a PSE where the "problem" to be solved is to write the "best" (fastest, most accurate) possible numerical program for a task.

2.1.2 Example: Generating perturbation expansions

A common task for some computationa scientists is the generation (perhaps at considerable mental expense) of a formula to be inserted into a Fortran or other program. An example from celestial mechanics (or equivalently, accelerator physics[4]) is the solution to the "Euler equation"

$$E = u + e \sin(E)$$

which is 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 $A_{k+1} = e \sin(u + A_k)$ where the right hand side is expanded as a series in $\sin nu$ to order $k+1$ in e. This is a natural operation, and takes only a line or two of programming in a suitable computer algebra system (one that supports Poisson series). The Macsyma system computes A_4 (and incidentally, converts it into TEX for inclusion as a typeset formula) in this paper as:

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

If you insist that it be expressed in Fortran, as any of several computer algebra systems offers, you get a variety of somewhat appropriate responses. 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
```

In general, Mathematica may be outright wrong in conversion into Fortran since version 2.1 at least produces forms like 1/3 (this is known to be 0 in Fortran). We can correct this tendency by calling `Expand[N[]]`

```
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)
```

Apparently Mathematica's Fortran has some other errors: it produces text with multiplications by 1, and has apparently guessed exactly how many 3's are relevant for our Fortran compiler.

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 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)*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 might very well be noticed by a decent Fortran compiler, and optimized out. A numerical compiler probably wouldn't notice that from the initial values $s = \sin u$ and $c = \cos u$ one can quickly compute $s_2 = \sin 2u = 2 \cdot s \cdot c$, $c_2 = \cos 2u = 2c^2 - 1$, $s3 = \sin 3u = s \cdot (2c_2 + 1)$, $s4 = \sin 4u = 2s_2c_2$, etc.

We chose this illustration because it shows that

1. Symbolic computing can be used to produce not only large expressions, but also relatively small ones. By extension, of course, larger expressions can be subjected to similar manipulation; examples are available in the CAS literature of large expressions.

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 them symbolically, it is not a straight path to convert them to Fortran (etc.)

4. It would be nice (and in some cases possible) to produce another expression for the error in the first. This can be done for polynomial evaluation as well as for the Euler equation, although we haven't given details here.

5. Having computed expressions symbolically, they can be routinely transferred into typeset form for inclusion in papers, although the hand-massaging of the form may be painful in current systems.

Incidentally, one should think carefully before dumping hugely-long expressions into a program text. Not only are such expressions likely to suffer from instability, but the mere size causes problems: they may strain common-subexpression elimination "optimizers" and/or might exceed the input-buffer sizes in one's compiler.

2.2 Derivatives and the like

The notion of computing a closed-form derivative expression may strike some readers as a trivial exercise for a CAS. Certainly most students who having taken a course in which they learn a bit of Lisp, will think so. Unfortunately, the triviality of this is a misconception on several levels. A serious CAS will have to deal with partial derivatives with respect to positional parameters (even the notation is not standard in mathematics), as well as simplification. Even so, that still doesn't address the quite reasonable requirements of carrying through the concept of differentiation to a whole Fortran program or other algorithmic expression of a multivariate computation. Especially if you do not wish to waste time, the task is rather more difficult. A recent book edited by Griewank and Corliss [10]. 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, Hessians, etc.) rapidly, and the use of explicit symbolic manipulation, though discussed, is not necessarily the central 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

$$e^{e^{e^x}+e^x+x}\left(1+e^x+e^{e^x+x}\right)$$

while correct and easily printed in Fortran if necessary,is not as useful as another program, also mechanically produced:

```
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
```

This program's structural simplicity could be made slightly smaller by eliminating operations of adding 0 or multiplying by 1.

What about integrals, then? Surely the closed form versions of integrals are valuable adjuncts to a numerical program – entirely avoiding numerical quadrature programs. This is sometimes true, and CAS can be very valuable for this capability. Yet we can argue the contrary. For a starter, the closed form may be more painful to evaluate than the quadrature. Example:

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

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

where $c_k := \cos((2k-1)\pi/64)$ and $s_k := \sin((2k-1)\pi/64)$. [14]

Other examples abound when the closed form, at least under usual numerical evaluation rules, is either not 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 right, a numerically preferable formula would be something like "if $0.5 < b/a < 2.0$ then $2\operatorname{arctanh}((b-a)/(b+a))$ else $\log(b/a)$." [5]. Consider, in IEEE double precision, $b = 10^{15}$ and $a = b+1$: the first formula gives $-7.1e-15$, the second gives the far more accurate $-10.e-15$.

In each of these cases we do not mean to argue that the symbolic result is wrong, but only that the tools being used may not be adequate by themselves to determine the appropriateness of the answer. Certainly a slightly 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 $\log(b/a)$ would not be difficult.

2.3 Closed form solutions

Of course there are also instances where a CAS can provide an outright closed form algebraic solution to a problem. This can reveal some inner truth that is not easily spotted by evaluation. The locations of singularities, the

values of limits, the orders of growth, and similar more qualitative properties of expressions, may all be available as the result of symbolic manipulation.

A particularly interesting kind of result is one in which numeric evaluation can never really assure you of the correctness of your result. Say that you believe an expression $f(x, y, z)$ to be zero, but numerical evaluation at various values of x, y, and z always gives you small but non-zero answers. Even if f is too complicated to manipulate accurately by hand, symbolic manipulation may help: you may be able to use the computer prove the expression is zero by exact rational evaluation, or expansion in series, or computing its derivative, or simplifying it.

2.4 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. Almost all of it 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[11] of the background including "applications" of analytic continuation. In fact his results are somewhat theoretical, but they provide a rigorous if pessimistic 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* [3]. In brief, if you have a system of differential equations $\{y_i'(t) = f_i(t, y_1, ..., y_n)\}$ 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 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. It is possible, however, to use analytic continuation to step around singularities (located by various means, including perhaps symbolic methods) by reexpanding 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 [3], "[The method of Taylor series] has been resetricted 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. Although 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."

It would seem that in this age of parallel numerical computing, the solution of numerical ODEs could be re-examined. The conventional step-at-a-time solution methods have an inherently sequential aspect to them. Taylor series methods provide the prospect of taking much larger steps, and perhaps much smarter steps as well. High-speed and even parallel computation of functions represented by Taylor series is perhaps worth considering.

2.5 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 looks like 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" sys-

tem 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 this as a substitute for the high-speed floating-point computation usually needed for scientific work, but as an adjunct, to test for benchmark values for computations. Distinguishing the consequences of accumulated round-off error from a physical simulation result can be difficult when you have exhausted your fixed maximum double-precision format, and are still uncertain.

2.6 Finite element analysis, and similar environments

Formula generation needed to automate the use of finite element analysis code has been a target for several packages using symbolic mathematics (see Wang [23] 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. Precisely which integrations can and should be done by a computer algebra system, and in what form (symbolically or numerically), is not obvious at the outset. Furthermore the algebraically "correct" form may nevertheless be inadequate for numerical computation – inefficient re-computation of common subexpressions being one obvious problem, numerical stability being another. And finally, it is reasonable to expect that engineers or other users would appreciate a well-designed system that does not require specialized computer-system knowledge that is irrelevant to the problem at hand.

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. [2]) 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.

2.7 Code generation for special architectures

Especially within the context of high-performance computing it is desirable to incorporate into software systems some understanding of how to generate variations of code for specialized environments, starting from the same "high level" specification. That is, only the code generator changes as one supercomputer is supplanted by the next generation. Changing between minor revisions of the same design may not be difficult; altering code from a parallel shared-memory machine to a distributed networked machine would be more challenging. It is especially difficult to make such a conversion if if the original model is overly constrained by what may appear to be sequential (or even shared-memory parallel) operations, but need not be.

2.8 Support for proofs, derivations

Of the computer algebra systems now available, only only Theorist makes some attempt to maintain a correct line of transformations. While it is in 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, is quite recent (work by A. Dingle, UC Berkeley). Work by S. Dooley at UC Berkeley may proceed further along this path, and also into other mathematical domains.

2.9 Interchange and production of text

Interfaces to documentation systems are also possible; converting expressions stored in Lisp to TEX format and or writing programs in Lisp to generate PostScript. How do such ideas fit in a problem solving environment?

Could one, for example,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, 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 a mock-up of 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 manipulable expressions 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. In spite of efforts to do so, the major organizing paradigm is probably not centered around a class of problems, or even a particular problem. 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 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 nth 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.

2.10 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 environment[7].

3 Symbolic Manipulation Systems as Glue

Gallopoulos *et al* [8] 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.

[7] To 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.

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 Lisp[8], 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 were converted to data that made sense to Fortran. Even more restrictive would be a system in which everything would be converted to character strings, as in the UNIX operating system pipes convention. While there are some subtleties that may be difficult to mimic exactly with interactions with C, there are tools like the `def-c-type` declaration that makes it possible to create structures and access 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 already possible in those languages already (especially C).

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

[8] the details differ for other brands

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 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 posibility 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 accomodated. 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 allocationn 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 [19] 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 [12]. 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 sucesses.

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 [7] 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 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.

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 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.

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 detect (e.g. "for large x approaches $\log x$ from below). By using interval arithmetic [6], 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 consistenting 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 TEX 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 – buth 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 accomodated. 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 yourself[9].

   (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 errors[10].

   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

[9] As 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.

[10] Of course your own code will not have such problems!

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 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? [21] CAS/PI [13] We have direct calls, multiple interconnected processes, input/ouput 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] V. Anupam and C. Bajaj. Collaborative Multimedia Scientific Design in SHASTRA. Proc. of the 1993 ACM SIGGRAPH Symposium on Multimedia. Anaheim, CA.

[2] 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.

[3] 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.

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

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

[6] R. Fateman. "Honest Plotting, Global Extrema, and Interval Arithmetic," *Proc. Int'l Symp. on Symbolic and Algebraic Computation* (ACM Press), (ISSAC-92) Berkeley, CA. July, 1992. 216-223.

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] 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.

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

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

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

[17] 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.

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

[19] 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.

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

[21] James Purilo. Polylith: An Environment to Support Management of Tool Interfaces, ACM SIGPLAN Notices, vol 20 no. 7 (July, 1985) pp 7–12.

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

[23] 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.

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