Excerpts from a proposal to the National Science Foundation
on Programming Environments and Tools
for Advanced Scientific Computation

Richard J. Fateman
Computer Science Division
Department of Electrical Engineering and Computer Sciences
Electronics Research Laboratory

1 Summary

We propose to concentrate on investigations of technology in support of advanced scientific computation, based on a higher-level view of the relationship of mathematical modeling to computation. This is based on the following considerations:

1. Symbolic mathematical manipulation and computer algebra systems: These are tools and techniques that support problem solving, code development, comprehension, debugging and partial evaluation (the principle behind many optimization techniques).

2. Data modeling: in particular, the use of IEEE floating-point number representations and operations.

3. Closed form or exact solutions: For some sub-problems of scientific interest, numerical solutions are not the only paradigm. The availability of exact solutions, or symbolic-approximation solutions has advanced along with computing technology too.

4. The use of scripting languages: the linkage of high-performance library routines to graphical interfaces for data analysis and model setup can be enhanced by symbolic support and high-level scripting languages.

5. Extending the use of remote computation. We are now using standard network connections for the acquisition of complex data (in this case, symbolic solution of definite integrals).

2 New Experiments

In this section we discuss a variety of topics of of current interest.

2.1 Mathematical Modeling and Symbolic Computing

The (by now) traditional approach to mathematical modeling is to formulate "by hand" mathematical models whose consequences can be simulated by running computer programs. These programs are typically written in Fortran, but are increasingly being written in C, C++, or other languages that have better tools for abstraction and data structures.

In part, the computational approach we advocate involves the use of computers in that earlier "by hand" stage of model formulation, using symbolic mathematics. The blossoming of this area via commercial programs typified by Mathematica and Maple, as well as some less widely used but still viable competitors (Axiom, Macsyma, MuPAD) might suggests that this approach is (a) successful and (b) needs no more academic research.

Actually, the relatively higher level of activity (and funding) in Europe has demonstrated that important results remain to be found in advancement of algorithms and building systems. Work at RISC-Linz (Austria), ETH (Zurich), CAN (Netherlands), INRIA (France) and the multi-national POSSO project are just a few sample programs. Their activities overshadow the limited work in US universities, and point to the fact

that academic research and training of graduate students in the design, construction, and advanced use of computer algebra systems is lagging in the USA.

Enough of politics; what should we be doing?

In the following sections we consider how one might advance the state of the art in computer algebra systems such tools by (a) support for modeling, as one area, computation in the complex plane, and (b) support for code development, partial evaluation and optimization. We also describe how these systems can be applied to improve the state of the art in scientific computation by assisting in designing and coding critical procedures.

Here is an example of how we propose to improve the modeling capabilities of such systems by adding to the unlying representation capability.

2.1.1   Singularities and Branch Cuts for Complex Functions

Computer algebra systems are rather weak in dealing with singularities. Asked what $\sin(x)/x$ is, when $x = 0$, most systems report a division by zero. Most systems can compute the limit as x approaches 0 correctly to be 1, but none apply this correction automatically. Continuing to compute automatically by observing that there is a removable singularity at $x = 0$ is an interesting, and (as far as we can tell) unimplemented idea for computer algebra systems. We think that this may provide an interesting experiment in stepping toward a more ambitious program of dealing more generally with singularities and branch cuts. It is unlikely to be a panacea.

A collection of papers in the June, 1996 SIGSAM Bulletin (no. 116) including one by Richard Fateman, describes the trials and tribulations of such computing, and some papers attempt to provide remedies through Corless's `winding number` and Patton's `Unln` function. Unfortunately these are attempt to prescribe algebraic answers to topological questions. They in effect resolve only a "first level" of problem: how to reasonably locate the branch cuts of specific elementary function inverses. They do not provide tools that can be used on those occasions when other branch cuts might be better chosen, or the need to ascribe branch cuts to functions more generally. As an example, they do not allow one to tell that $\sqrt{1-z}\sqrt{1+z} - \sqrt{1-z^2}$ is zero but $\sqrt{-1+z}\sqrt{1+z} - \sqrt{-1+z^2}$ is only zero some places.

We hope to explore this further, providing perhaps graphical and topological tools for improved human understanding, and an improved notation for computer understanding.

In brief, the problem is this: many standard functions, such as the logarithm and square root functions, cannot be defined continuously on the complex plane. When working with such functions, arbitrary lines of discontinuity, or *branch cuts*, must be chosen. For example, the conventional branch cut for the complex logarithm function lies along the negative real axis, so that $\log(-1) = \pi i$ but when ϵ_1 is small, real, and positive, $\log(-1 - \epsilon_1 i) = -\pi i + \epsilon_2$ for some small, complex ϵ_2.

Most computer algebra systems provide little assistance in working with expressions involving functions with complex branch cuts. Worse, by their ignorance of the existence of branch cuts, algebra systems sometimes produce incorrect answers. With graduate student Adam Dingle, we have demonstrated that, over an interesting class of functions of a single complex variable, we can mechanically determine an expression's branch cuts, and we can continue to compute (for example, simplify) such an expression with appropriate attention to its branches. Even when an expression cannot be simplified, its branch cuts may yield useful geometric insights. The damage done to algebraic manipulation systems' reputations by their incorrect implementation of $a^r b^r \to (ab)^r$ and its consequences, is not inevitable.

Adam Dingle's masters' project, a summary of which appeared in [7] implements a simple portion of our branch cut computation algorithm. The program can provide representations of results that are still lacking in computer algebra system today, even though it would seem to be quite basic to many advanced algorithms. Dingle's work (incidentally, written using Mathematica, but not without considerable hacking), can solve many problems such as the following, which appear as exercises in a complex analysis text [4] (page 24, exercises 13a and 13b)

2

Discuss the branch-cut and Riemann-surface situation for each of the following functions: $\sqrt{1+\sqrt{z}}$ and $\ln(1+\sqrt{z^2+1})$

The `Cuts` program works as follows (the somewhat obscure syntax is inherited from *Mathematica*):

```
In[1]:= Cuts[Sqrt[1 + Sqrt[#]]&]
Out[1]= {{Cut[-Infinity, 0, Identity], Sqrt[1 - Sqrt[#1]] & }}

In[2]:= Cuts[Log[1 + Sqrt[#^2 + 1]]&]
                                          2
Out[2]= {{Cut[-Infinity, -1, -Sqrt[#1] & ], Log[1 - Sqrt[1 + #1 ]] & },

                                         2
>    {Cut[-Infinity, -1, Sqrt[#1] & ], Log[1 - Sqrt[1 + #1 ]] & }}
```

The notation `Sqrt[1 + Sqrt[#]]&` is used to represent the function $\lambda z . \sqrt{1 + \sqrt{z}}$ in Mathematica. The `&` character constructs a function; the `#` character represents the argument of the function. The notation `{Cut[-Infinity, 0, Identity], Sqrt[1 - Sqrt[#1]] & }` represents the branch cut $\{x | -\infty < x <= 0\}$, with alternate branch function $\lambda z . \sqrt{1 - \sqrt{z}}$.

Our implementation will eliminate many instances of what we call "removable branch cuts," where cuts cancel. Consider this example:

```
In[3]:= Cuts[Log[# + 1] - Log[# - 1]&]

Out[3]= {{Cut[-1, 1, Identity], 2 I Pi - Log[-1 + #1] + Log[1 + #1] & }}
```

An example demonstrating the usefulness of conformal mapping is that of the well-known Joukowski "airfoil." Manipulations of this transform pose a puzzle to every existing computer algebra system: if $R(z) = (z + 1/z)/2$ and $S(w) = w + \sqrt{w+1}\sqrt{w-1}$, simplify $S(R(z))$. (Notice that S is a "weak inverse function" for R as $R(S(z)) = z$ for all z.) Expansion of $S(R(z))$ yields

$$S(R(z)) \quad = \quad \frac{z^2+1}{2z} + \sqrt{\frac{(z+1)^2}{2z}}\sqrt{\frac{(z-1)^2}{2z}} \tag{1}$$

$$= \quad \frac{z^2+1}{2z} \pm \frac{(z+1)(z-1)}{z} \tag{2}$$

This last expression simplifies by case analysis to z or $1/z$. We may conclude that $S(R(z))$ is equivalent either to z or to $1/z$ in each of its branches.

The algorithms we have devised can determine that the branch cut of the function $S(R(z))$ is the unit circle (if an appropriate conformal-mapping rule is present). The unit circle partitions the plane into two regions; testing a point in each region reveals that $S(R(z)) = z$ outside the circle and that $S(R(z)) = 1/z$ inside the circle. No other existing system we are aware of can manage this. (In fact, we too are negligent: we fail to resolve what is true about points *on* the circle.)

There is substantial additional work that can be done on this topic. In our current implementation we do not distinguish between closed and open intervals. More careful attention to the endpoints of intervals would allow us to represent and manipulate singularities, which we can consider to be degenerate branch cuts.

Extensions to wider classes of functions, a more complete treatment of multiple values, and a larger catalog of standard descriptions of cuts (conformal mapping rules) will make this work more useful as a tool for correct manipulation of complex-valued functions by computer systems as well as applications such as the integral table results from TILU.

2.1.2 Code Development

Can we build programs that are experts in writing code? Can we re-use expertise in formulating programs from models? Certainly we can write programs that manipulate programs; in computer science, compilers (and even debuggers, loaders, in a trivial way) do this. But typically a compiler has only a limited model of transformations that are legal on programs generally, and not the transformations that might be legal on a particular program: it has very little mathematical semantic information at its disposal, and no physical modeling information. What little expertise an optimizing compiler may have (e.g. regarding rearranging expressions) may in fact be *incorrect in detail* in the floating-point arithmetic model. It depends on the expertise of the programmer, and some luck, to avoid snares.

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[1].

For example, in his PhD dissertation, Warren Teitelman [40] 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. Commercial Lisp systems provide an advising facility as a matter of course.

By contrast, other classic early successes of symbol-manipulating systems aimed not for generality but for specificity: such systems solved particular classes of problems by joining together newly generated pieces using templates. We can date some significant efforts back to at least the late 1970s, with ambitious efforts using symbolic mathematics systems (e.g. M. Wirth's PhD dissertation [43] who used Macsyma to automate work in computational physics).

Here we give some sample manipulations from our recent work.

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 to use "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 nearly real-life exercise related to this. Consider the Fortran program segment from [39] (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+
```

---

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

```
*         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 have rewritten 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 *pre-conditioned* version of the above
(poly-eval ...)

```
(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) 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 or earlier* 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 a 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. An assist in this kind of program-writing can be very important because the human modeler is unlikely to spent the time to write this out—and in most cases would not even know how to do it correctly and efficiently.

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 (e.g. in this Bessel function evaluation context we know that $0 \leq x \leq 3.75$.) One can determine the maximum error in the polynomial for *any x* that region. We could in principle extend this kind of reasoning to produce, in this symbolic programming environment, a Bessel function evaluation routine with an *a prior* error bound; a bound computed, in effect, **by the "compiler"**.

This is the kind of practical expertise in a somewhat arcane subject (error analysis) that should be provided in a problem solving environment whose focus is writing the "best" (fastest, most accurate) possible numerical program for a task.

Before leaving this topic, we realize that on today's computer architectures, it may very well be the case that evaluation of polynomials can best be done by observing and optimizing for the pattern of access to memory and registers. In this case the actual number of operations is less important. Yet in this case we believe that—to the extent that a careful human program has access to such niceties—the automated code generator can also utilize patterns and idioms that may be most efficient for evaluation.

Example: Generating perturbation expansions

A common task for some computational 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 [9]) 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 text as:

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

If you insist that it be expressed in Fortran, as any of several computer algebra systems offers, you get a variety of more or less appropriate responses. One can coerce Mathematica to produce

```
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)
```

There are better ways of computing the collection of $\sin(nu)$ terms, but even an optimizing Fortran compiler is not likely to notice what a symbolic math system might:

From the initial values $s = \sin u$ and $c = \cos u$ one can quickly compute $s_2 = \sin 2u = 2 \cdot s \cdot c$, $c_2 = \cos 2u = 2c^2 - 1$, $s3 = \sin 3u = s \cdot (2c_2 + 1)$, $s4 = \sin 4u = 2s_2 c_2$, etc.

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. 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.). Notice that Mathematica above had to determine how many 3's to put in 0.333...3.

4. It would be nice (and in some cases possible) to produce another expression for the error in the derived approximation. This can be done for polynomial evaluation under a model of floating point arithmetic, as well as for the Euler equation using mathematical grounds, although we do not have room for 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.

We have occasionally encountered coding techniques using computer algebra that consists of dumping hugely-long unexamined expressions into a program text. There is a danger that such unexamined expressions suffer from instability; but sometimes their size alone causes problems: they may strain common-subexpression elimination "optimizers" and/or sometimes exceed the input-buffer sizes in one's compiler.

Derivatives and the like

The notion of computing a closed-form derivative expression may strike some readers as a trivial exercise for a computer algebra system. 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 [32]. 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

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

is correct and could be printed in Fortran if necessary. In that case however, a more useful form 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 even smaller.)

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 $s_k := \sin((2k-1)\pi/64)$. [29]

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 $\log(b/a)$." [26]. Consider, in IEEE double precision, $b = 10^{15}$ and $a = b + 1$: the first formula gives $-7.1 \cdot 10^{-}15$, the second gives the far more accurate $-10. \cdot 10^{-15}$.

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.

Other areas we have looked at tentatively include the use of asymptotic series, and other symbolic approximative techniques in particular as solutions for differential equations. While these mathematical techniques, applied "by hand", tend to be more human-effort intensive and hence non-competitive with numerical simulation, the development of automated tools may change this.

2.2 Closed Form Solutions, Exact or High-precision Computations

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. There is a substantial literature on such testing, as well as tools based on interval arithmetic (e.g. [38]). CAS provide an easy handle to go beyond ordinary fixed-precision floating-point arithmetic that may be 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 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" 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 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.

2.3 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. 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 [28] 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* [2]. 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 [2], "[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. 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.

2.4   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 [42] 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. [6]) aimed at other application techniques or even specific problems are under investigation, and there is a substantial literature developing here.

2.5 Data Modeling and IEEE floating-point number representations

An idea that has emerged from discussion with earth-science numerical simulation consumers has been the idea that keeping track of program exceptional data could be of enormous value in debugging program, and in debugging data.

Consider the prospect of running conventional programs – unaltered – but in a software environment that supported more fully the hardware capabilities of the floating point numbers (IEEE exceptional operands). So-called "retrospective diagnostics" can provide encodings in the results of numerical runs that fail. The errors can be of the form: "$Z(43)$ is an error datum caused by the use of uninitialized data in program XXX, line YYY using input $A(101)$". Similar messages can encode exceptions like division by zero. The notion of retrospective diagnostics approaches in some respects the impossible dream of "running programs backwards to the source of the error".

We have written prototype programs to support use of such encodings for Sun SPARC computers using Common Lisp. A serious implementation would not, however, depend intimately on the hardware or the software particulars, except to the point of requiring IEEE standard hardware and some software support for IEEE NotANumbers in the languages.

We believe these techniques can be especially useful in "marking" data—perhaps data that is especially uncertain—and seeing how it is handled arithmetically. It is perhaps not widely known that in the IEEE standard you have 2^{52} or so different Not-a-Numbers. These can easily encode (say) program-counter information, or a pointer into a table. IEEE hardware requires that any arithmetic operation that combines a NaN with a regular number results in the NaN. Except for our own (as yet unpublished) software, and designs suggested by our colleague W. Kahan, the principal designer of the IEEE standard, we know of no software or language designs making significant use of this facility; unfortunately, some new language designs may even exclude use of this facility (e.g. Java).

2.6 Scripting Languages

Using an interactive interpreter-based language to guide the connection of (Fortran or C) programs, seems plausible as a way to experiment with prototype designs, user-interfaces, and other matters without committing to a major project. UNIX workstation programming has evolved in an unfortunate direction. The philosophy of pipe-lined workers written in mini-languages makes more sense in a small-memory system in a well-controlled sequential environment, such as was common in the early UNIX time-shared systems. It is appalling that one sees in C, C++, Perl, the repeated reprogramming of common interfaces that are missing from the language. Typically the repeated poor implementations do not include sufficient error detection and recovery to provide confidence in the package.

We have been experimenting with Common Lisp [21] as a language that provides enormous (perhaps too much) support for the many ancillary tasks for computing—including nearly every feature of sequential and parallel programming languages today—in scientific programming tasks as well. We have found in the past few months, working on new Intel Windows NT machines, that code written in Lisp, with the exception of those rare explicitly hardware-dependent pieces, can be run without change, and often at higher speed, than on UNIX systems of earlier vintage. The system we have running now comes with a rather complete and simple-to-use interactive socket-based communication package. We are, for example, communicating from lisp to WWW browsers, data-base systems, and (for speed, via direct calls) to the Windows API, scientific subroutines (coded in C), a solid-modeling facility, and extensive profiling and debugging tools.

We propose to continue and renew our effort to provide a well-defined and standard interactive interface to numerical and library code. As an example of the code that has been produced, we have been able to "write" fortran code for optimization (derivatives, etc.) and for finite-element cells, compile this code, and then invoke it "untouched by humans." This work at Berkeley dates back to before Douglas Lanam's [35] 1981 MS project; earlier significant work includes the previously mentioned work by Wirth [43].

A stream of similar programs has emerged as a major thrust in symbolic system applications [42]. We do not yet see it emerging in the larger numerical simulation community. In the future, the critical feature

may be the ability to keep track of more a more complex computational plan in numerical simulation, but with less direct human attention taken in programming.

2.7   Remote Computation and Collaboration: Integral Table Lookup

We have previously reported on TILU, a table-lookup program for integration. We have hooked this program up to a web page, and to date several thousand web queries have been handled by this prototype TILU server.

It is natural to wonder if this is necessary—Don't computer algebra systems do this? We are grateful for some testing information from Paul Zimmerman[44] (INRIA, Lorraine) who started with our sample of 2145 artificially generated TILU inputs (generated by substitutions into our current table contents), and used some 1382 of them as testing inputs to various systems. The solution rates ranged from 86% to 99%, and times from 150ms to 3600ms average on different systems. (By comparison, our average time is 6.5 ms, and our maximum time is about 20 ms.) It is interesting that some 10% of the (mostly basic) formulas represented in our table stumped Maple V.4 and Mathematica 2.2.3. We must emphasize that our table to date has very few esoteric formulas.

Zimmermann used our test suite to refine the MuPAD integrator so that it could do nearly all of the problems, but to do so, more than half are done by lookup rather than algorithms!

Traffic statistics and queries show there are many areas open for further development. Here are a few we are prepared to explore, and these are suggested in a separate proposal to NSF. Here's only one item.

2.7.1   More entries in the tables

We have about 1100 entries in our table, mostly inserted from the relatively small CRC table of integrals obtained in machine-readable form and parsed into our internal representation. We added some additional formulas for experiments. (These come from two areas: problems whose integrands involve Bessel functions and whose results are probably not amenable to solution by current algorithms, and problems with elliptic function solutions requiring certain careful simplification not likely to be available yet in computer algebra system algorithms.)

We discuss in other contexts the automatic reading of published integral tables, but there are other ways of generating entries. These include the use of computer algebra algorithms in ways that are not productive in the normal query-response mode. That is, we can generate classes of formulas, not necessarily in response to a particular question. An idea previously used by some systems is to generate "new"results by taking existing integrals and differentiating with respect to a parameter. An idea which we have not seen previously used is to take a product inside an integral and use Laplace and inverse Laplace transforms of the terms to generate another formula. That is,

$$\int_0^\infty f(x)g(x)dx = \int_0^\infty \mathcal{L}(f(x);s)\mathcal{L}^{-1}(g(x);s)ds$$

where $\mathcal{L}$ and $\mathcal{L}^{-1}$ denote the Laplace and inverse Laplace transform, respectively. As a simple application,

$$\int_0^\infty \sin x \cdot \frac{1}{x}dx = \int_0^\infty \frac{1}{s^2+1} \cdot 1 \ ds$$

Other transforms with symmetric kernels could be used (e.g. Fourier, perhaps Hankel), although conditions of validity must be checked.

Another method standardly used is expansion of generating functions, which we have begun investigating. Because we can generate these forms at our leisure and insert them into the table for very fast lookup, there is a substantial potential for saving time if the formulas are ever used.

2.7.2 Other forms, especially ODEs

Summation, simplification, integral transforms, and other symbolic table entries are plausible targets for lookup. In this section we discuss only our beginning work on quick lookup of solutions of ordinary differential equations.

In fact, we are torn between classification and solution techniques that require, in some cases, substantial computation, and a technique that accesses stored solutions. Our server relies on the fact that the time for lookup is not very heavily dependent on the size of the lookup table, so ODEs may be stored in several different transformed forms to see if one matches. This approach requires encoding material such as Zwillinger's *Handbook* [45], Polyanin's *Handbook* [34] or Kamke's *Differential-Gleichgungen* [30] in TILU-compatible electronic form. This, in turn, involves addressing some ticklish engineering issues of encoding and allowed transformations. We expect that we will have to augment the relatively uniform lookup method that has already been refined and tested for integrands. If the table fails to solve the problem, it is plausible to attempt a variety of solution methods, including heuristics and algorithms. It appears that the state of the art in ODE solving by computer and even by humans is still mixed: while some solutions are easily found, others are found almost by chance.

In such a circumstance, a shared client/server approach is plausible, and we may use a CAS framework for the canonical transformations, and TILU for lookup.

As an example of the value of this approach, we considered some 316 first-order ODES extracted from Kamke[30], and provided to us in REDUCE syntax, by Alain Mossiaux, tested them in Mathematica 2.2, 3.0, Macsyma 2 and MuPad (via Paul Zimmermann[2]) Using Kamke as a benchmark is something of a tradition in computer algebra. Peter Schmidt [37] used it in 1976, where he claimed a solution rate of 90% on Kamke's first order equations, although it appears that Schmidt counted as a solution results in terms of unresolved integrals. His program (entirely written in PL/I) has not been maintained, although some of his methods were adopted in Macsyma, and probably other systems.

Even excluding these cases that reveal bugs in these various systems, and excusing some failures as really failures of the system to solve generated implicit equations, the differential equations that *are* solved can easily take tens of seconds or minutes. Furthermore, some of the answers are quite large, and sometimes unnecessarily so.

We anticipate that our lookup program would solve these and similar (though admittedly cookbook) problems in very short times (say 10 ms) and, of course, would provide the answers in text-book form. In addition to the quick times, the compact results generally benefit any attempt to actually use the answers. We anticipate that the hierarchical search ODE program (we are calling it ANODE) will generally provide appropriate conditions on the answers, maintaining some hope that invalid regions will be tagged in further computations.

On the other hand, ANODE can still fail to find solutions from user input for several reasons.

- The user can pose equations differing in form yet equivalent to those stored in the table. Since we use weak methods (for example, we do not even compute polynomial greatest common divisors) we may fail to match a form that is only modestly different from ours. Clearly a combination of approaches would push us further in solving this class of problems. If we combine approaches, we would have to abandon our goal of solving each problem in 10—20ms.

- The problem posed may be more general than the specific examples stored in the table. For example the problem can be linear and inhomogeneous. We think it might be possible, however, to use table lookup to find the Green's function for the homogeneous equation. Maintaining our low-cost criterion, we may still be able to multiply the answer by the inhomogeneous term and express the answer in quadratures, and perhaps even solve that further with TILU.

- The solution is not previously known, although algorithmically obtainable.

---

[2] email, 11/19/96

- The solution is not tabulated, cannot be found with current algorithms, and/or simply does not exist in terms of known functions and/or quadratures. Or perhaps one can show that no solution exists at all.

We are looking at algorithmic approaches for backup. One could be a version of Bronstein's system mentioned previously; this would provide a decision procedure for the solution of linear differential equations. His system could be brought up locally or as another process, executed remotely. It may in turn may use TILU for resolving integrals. We are also looking at other parts of the EU-funded CATHODE project, newly funded for phase II. In particular we are engaged in discussions with Dr. Fritz Schwarz of the GMD (St. Augustin, Germany) whose work on Janet Bases and symmetries of ODEs may provide insight into problems not easily looked up[3]. See also related work, also at GMD, and also in Lisp[4].

Finally, with regard to lookup, it is possible, by analogy with the integration program, to generate additional ODEs to which we have solutions, by transformations of existing ODEs. We suspect that some of the cataloged ODEs already fall into this kind of expansion program.

References

[1] Yannis Avgoustis. "Symbolic Laplace Transforms of Special Functions" in Proc. of the 1977 Macsyma Users' Conference, Berkeley. CA 1977. (NASA CP-2012). 21–41.

[2] 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.

[3] Benjamin Berman and Richard Fateman. "Optical Character Recognition for Typeset Mathematics," *Proc. of Int'l Symp. on Symbolic and Algebraic Computation* (ACM Press) (ISSAC-94) Oxford, UK. July, 1994. 348—353.

[4] G. F. Carrier, M. Krook, and C. E. Pearson. *Functions of a Complex Variable: Theory and Techniques*, McGraw-Hill, 1966.

[5] B. Char, K. Geddes, G. Gonnet, and S. Watt. *Maple User's Guide*, 4th ed. WATCOM Publ. Ltd., Waterloo, Ontario, Canada, 1985.

[6] 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, 27—35.

[7] Adam Dingle and Richard Fateman. "Branch Cuts in Computer Algebra" *Proc. of Int'l Symp. on Symbolic and Algebraic Computation* (ACM Press) (ISSAC-94) Oxford, UK. July, 1994. 250—257.

[8] Richard J. Fateman. "Advances and Trends in the Design of Algebraic Manipulation Systems," in: S. Watanabe, M. Nagata (eds), *Proc. Int'l Symp. on Symbolic and Algebraic Computation* (ACM/ Addison-Wesley), (ISSAC-90) invited paper, Tokyo, August, 20-24, 1990, 60-67.

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

[10] R. Fateman and W. Kahan. "Improving Exact Integrals from Symbolic Computation Systems," Tech. Rept. Ctr. for Pure and Appl. Math. Univ. Calif. Berkeley, PAM 386, 1987.

[11] E.L. Ince, *Ordinary differential equations*, Dover, NY, 1956.

---

[3] http://www.rrz.uni-koeln.de/REDUCE/spde/spde.html
[4] http://www.gmd.de/SCAI/alg/cade/lodef/lodef.html

[12] R. Fateman, T. Tokuyasu, B. Berman, N. Mitchell. Optical Character Recognition and Parsing of Typeset Mathematics, *J. Visual Commun. and Image Representation* vol 7 no 1, March, 1996. 2—15.

[13] R. Fateman, "FRPOLY: A Benchmark Revisited," *Lisp and Symbolic Programming, 4* (1991) 153—162.

[14] R. Fateman, *Lisp and Symbolic Programming, 4* (1991) 153—162. "Review of Mathematica," *J. Symbolic Comp. 13* no. 5 (May 1992) 545—579.

[15] R. Fateman. "Canonical Representations in Lisp and Applications to Computer Algebra Systems" *Proc. Int'l Symp. on Symbolic and Algebraic Computation* (ACM/ Addison-Wesley), (ISSAC-91) Bonn, Germany, July, 1991. 360-369.

[16] R. Fateman. Review of "A Guide to Computer Algebra Systems," D. Harper, C. Wolf, D. Hodgkinson (J. Wiley). *Math. Comp.* (book review).

[17] R. Fateman and Derek T. Lai. "A Simple Display Package for Polynomials and Rational Functions in Common Lisp," SIGSAM Bulletin 98 (vol 24 no. 4 Oct. 1991) 1—3.

[18] 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.

[19] R. Fateman. Review of *Automatic Differentiation of Algorithms: Theory, Implementation, and Application,* A. Griewank, G. Corliss (SIAM) *SIAM Rev 35,* no 4. (Dec. 1993) 659-660. (book review).

[20] R. Fateman. Review of *The Maple V Handbook* (Abel, Braselton), *Computing Reviews*, Feb. 1995. (book review)

[21] R. Fateman, Kevin A. Broughan, Diane K. Willcock, and Duane Rettig. "Fast Floating-Point Processing in Common Lisp", *ACM Trans. on Math. Software*, vol 21 no. 1, March 1995, 26—62.

[22] R. Fateman and T. H. Einwohner. "Searching Techniques for Integral Tables," *Proc. of Int'l Symp. on Symbolic and Algebraic Computation* (ACM Press) (ISSAC-95) Montreal CA. July, 1995. 133—139.

[23] R. Fateman and H-C (Phil) Liao. "Evaluation of the Heuristic Polynomial GCD." *Proc. of Int'l Symp. on Symbolic and Algebraic Computation* (ACM Press) (ISSAC-95) Montreal CA. July, 1995. 240—247.

[24] "Symbolic Mathematical System Evaluators", *Proc. of Int'l Symp. on Symbolic and Algebraic Computation* (ACM Press) (ISSAC-96) Zurich, Switzerland. July, 1996. 86—94.

[25] R. Fateman, T. Tokuyasu, B. Berman, N. Mitchell. Optical Character Recognition and Parsing of Typeset Mathematics, *J. Visual Commun. and Image Representation* vol. 7 no 1, March, 1996. 2—15.

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

[27] Simon Gray, Norbert Kajler, Paul Wang. "MP: A Protocol for efficient exchange of mathematical expressions" *Proc. ISSAC 94* 330–335.

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

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

[30] E. Kamke. *Differentialgleichungen, Losungsmethoden und Losungen,* Akademische Verlagsgesellschaft, Leipzig 1961.

[31] I. S. Gradshteyn and I. M. Ryzhik. *Table of Integrals, Series, and Products,* Corrected and Enlarged, 5th edition, Academic Press, 1996.

[32] 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.

[33] A.C. Hearn. "Computer Algebra on the Net," Proc. DISCO 96,

[34] Andrei D. Polyanin and Valentin F. Zaitsev, *Handbook of Exact Solutions for Ordinary Differential Equations,* CRC Press, 1995.

[35] 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).

[36] H-C (Phil) Liao. *Automated Techniques for Proving Geometry Theorems* 52 pages, typeset (MS Report, UC Berkeley).

[37] Peter Schmidt. "Automatic symbolic solution of differential equations of first order and first degree" ACM Proc. SYMSAC 76, 114–125 Karlsruhe, Germany, 1996.

[38] Ramon E. Moore. *Methods and applications of interval analysis.* SIAM studies in applied mathematics, 1979 also, *Reliability in computing: the role of interval methods in scientific computing,*Academic Press, 1988.

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

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

[41] A. P. Prudnikov, Yu.A. Brychkov, O. I. Marichev. *Integrals and Series,* three volumes, Gordon and Breach Science Publishers, N.Y. 1986-1990.

[42] 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.

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

[44] Paul Zimmermann. "Using Tilu to improve the MuPAD integrator," INRIA, Lorraine, Sept. 1996.

[45] Daniel Zwillinger. *Handbook of Differential Equations,* Harcourt Brace, 1996.