Manipulation of Matrices Symbolically

Richard Fateman
Computer Science Division
University of California, Berkeley

June 18, 2001

Abstract

Traditionally, matrix algebra in computer algebra systems is "implemented" in three ways: Numeric explicit computation, symbolic explicit computation, and implicit matrix computation of symbols defined over a (non-commuting) ring. Manipulations which involve matrices of indefinite size ($n \times m$) or perhaps have components which are block submatrices of indefinite size have little or no support in general-purpose computer algebra systems, in spite of their importance in theorems, proofs, and generation of programs. We describe some efforts to design and implement tools for this mode of thinking about matrices in computer systems.

1 Introduction

The kinds of matrix calculations supported by computer algebra systems tend to be limited by those areas in which there are fairly straightforward representations of data, and in which at least rudimentary algorithms can be found in the numerical computation framework. The symbolic computation aspect is sometimes a simple generalization of the numeric, but often the introduction of indeterminates changes the nature of the computation.

1.1 Traditional Computer Algebra Computations

Certain approaches to matrix computation tend to be well-supported in computer algebra systems.

(a) Numeric explicit matrix computation involving matrices of specific sizes (say 6×6 or 1000×600) with entries which are single/double/complex floating-point numbers. This kind of facility is standard in purely numeric packages such as Matlab. Typically these are implemented as dense arrays, but could use other representations for structured or sparse matrices.

(b) Explicit matrix computation where the entries may be more general expressions including exact integer arbitrary-precision or rational numbers, symbolic expressions such as polynomials in several variables, algebraic, or transcendental functions. Some algorithms, if expressed in a suitably generic language, can be transferred with little or no change from the numeric domain. Other numeric algorithms are not feasible "symbolically" or for efficiency purposes have to be reconsidered entirely. Additional computations are possible in this domain which require symbolic entries to be meaningful. Still, any given matrix has a fixed known size during computation.

(c) Implicit matrix computation where each matrix is represented by a single symbol, and the computer algebra system principally implements arithmetic over a noncommutative ring.

1.2 New Possibilities

A model which has *not* been available in computer algebra but may be useful for the mathematician, programmer, or student is one in which matrices of *indeterminate* dimensions are manipulated. These may be populated by symbolic block submatrices, diagonals of specified values, elements specified algebraically ($h_{i,j} = 1/(i + j - 1)$ is a Hilbert matrix), or by rows and columns which are filled by some rules. We would like to support the development and execution of generalized matrix algorithms, reasoning about matrices, and the reduction or optimization of algorithms into conventional code, as appropriate, while keeping free of unnecessary particularities. This paper describes steps toward the kind of extensions necessary to fulfill these objectives.

1.3 Why Bother?

> Science is what we understand well enough to explain to a computer. — Donald E. Knuth [1]

Manipulation of matrices is a widely used paradigm in science. In theorems and texts, all matrices are symbolic except in numerical examples, so it would seem that a computer system for "doing" matrices should start with the symbolic.

Of course our current view of matrices and computers is colored by our experience in rapid manipulating numerical matrices. For many people the only framework for discussion is numeric, and we must back-construct to symbolic matrices[1].

To the chase, then. The first rationale is that the computations found so useful for numerical computation: solution of linear systems, computation of determinants, eigenvalue computations, etc. have an immediate corresponding computation with symbols in the matrices. The computation can proceed with polynomials or rational functions as elements nearly as well as with numbers, up to a point. In fact this approach is rife with disappointment. Calculations routinely done with numbers can be impossible with general symbols. Here are some of the problems encountered.

1. Determination of a zero element. In various algorithms based on row and column operations it is often required that we avoid division by a zero pivot element. Numerically it is sometimes difficult to know whether an element is zero or not, but given a selection of elements, it is generally possible to avoid using the smallest one. If one is doing exact integer or rational arithmetic, or finite-field arithmetic, zero-testing is exact and we are ahead of the game. In the symbolic domain, this decision can be impossible. Division by a pivot element of $x - y + 1$ looks safe enough unless subsequently we deduce that $x = y - 1$.

2. Solution of algebraic problems (root-finding). The notion of convergence of a Newton iteration to a root of a polynomial in one variable is a well-studied problem in numerical analysis. If one is dealing with approximate complex numbers, computer representation of floating-point values provides a usually satisfactory representation for a set of approximations to roots of a real or complex polynomial, or matrix eigenvalues[2]. A symbolic approach can rapidly run out of steam when eigenvalues of a large enough (5×5) symbolic matrix cannot in general be represented in exact terms of radicals. There are other symbolic approaches, some of which amount to giving the eigenvalues "names" and side-relations and continuing the computation as far as possible. It may also be plausible to use symbolic

[1] We back-construct to "snail mail" because today mail refers to electronic transmission, to corded phone because so many are now cordless. Cordless screwdriver is odd... it refers to battery-powered. All sufficiently old screwdrivers are cordless.

[2] Higher precision arithmetic implemented in software is an option as well.

approximations in the form of series. For some matrices which are primarily numeric but with one or a few small perturbations, it is sometimes possible to proceed a good deal farther, especially if Taylor-series computations are appropriate.

3. Memory usage. One can depend upon a double-precision floating-point number to occupy no more or less than 64 bits of memory. This provides convenient figures for the storage of dense matrices, and even for structured matrices, some bounds. Finding a particular matrix element is supported by hardware instructions supporting array indexing. (Though cache memory makes the cost of addressing elements dependent on locality of reference in time and space.) In our more general setting, array elements can be any size: they can be very long integers, rational numbers, polynomials or worse. "Heap allocation" is usually required, with its attendant possibilities of running out of available memory; some kind of indirection in storage is needed in addition to indexing.

4. Canonicality. One can always determine if two exact integer matrices (of finite size) are equal by comparing their entries. Floating-point entries may make such a determination uncertain, but given a measure of distance, still computable. Determining if two symbolic matrices are the same generally requires testing to see if the difference of two formulas is identically zero. While the problem can be solved for many simple kinds of expressions such as multivariate polynomials, the introduction of trigonometric and algebraic functions makes the task difficult (and in theory, undecidable).

For many people the initial attraction in merging computer algebra and matrix computation is the possibility of directly running standard textbook algorithms (if written out in a language that supports symbolic math). This is handy for pedagogical examples somewhat more ambitious than those typically exhibited in texts. Note that computing determinants, of *symbolic* matrices can be a challenge on even rather small dimension matrix examples. Determinants of 25 by 25 matrices can be difficult or quite impossible with ordinary computer resources, since the answer for a 25 by 25 matrix of unique symbols constitutes a polynomial of huge size (over $25! \approx 1.55 \times 10^{25}$ terms.) Computing such determinants can be done only if there is considerable sparseness and/or special structure to assure that the answer as well as intermediate steps is relatively small.

The second rationale is that symbolic computation with matrices can substitute for the kinds of manipulations more often done by humans with pencil or chalk. Among these tasks we include

1. Teaching about linear algebra and running simple examples.

2. Proving theorems by explicit computation.

3. Writing algorithms and converting them to computer programs.

One might hope for a third kind of rationale for symbolic computation with matrices, except CAS neglect this:

1. Storing theorems about matrices.

2. Retrieving and applying theorems about matrices in the course of proving (and then storing) new theorems.

3. proving the equivalence of programs and mathematical formulations.

4. Generating specializations of algorithms involving matrices that take advantage of matrix shapes (e.g. sparse, banded, upper-triangular) or entry-types (single/double/complex).

2 Reasoning about matrices

> As J. R. Pierce has written in his book on communication theory, mathematics and its notations should not be viewed as one and the same thing [2]. Mathematical ideas exist independently of the notations that represent them. However, the relation between meaning and notation is subtle, and part of the power of mathematics to describe and analyze derives from its ability to represent and manipulate ideas in symbolic form.[3]

The technology for automated proofs has a life of its own, rather distinct from computer algebra systems, but the two worlds have occasional contact[4]. My view of what can and should be done with theorems in computer algebra systems is rather straightforward: If we start with a theorem of the form: *"Given conditions* $c_1,\ c_2,\ \cdots,\ c_n$*, then statement* S *is true."* then a CAS proof should consist of setting up in a computer the context in which the theorem is stated (the context which is not explicit in the theorem itself but is perhaps understood because the theorem occurs in a textbook about, for example, real analysis). Next, each of the conditions $\{c_i\}$ is asserted. Finally, the computer program simplifies the expression S. If it simplifies to the Boolean value **true**, then the theorem is proved.

Note that the steps in the proof are not given, and some of the implicit assumptions made by the CAS in the context may subsequently be falsified (e.g. in complex analysis), in which case the alleged theorem may be false under these new circumstances. We illustrate this below. Typically the generality of the CAS domains and the breadth of data types, where infinities, intervals, and other entities can be used, can produce inconsistencies.

It might be useful if the result of the computation included a list of all conditions used during the simplification process but not stated as hypotheses. This are each substantial challenges, probably requiring restructuring of existing CAS programs.

Even more challenging would be, in case the statement S were not true, the production of a list of additional conditions which, if shown to hold, would complete the proof. Finding a minimal set of such conditions except in trivial cases would be the goal. These conditions may be

1. defects in the statement of the theorem, which should therefore be corrected before the theorem can be proved,
2. conditions which falsify the statement (and thus show the alleged theorem is not true),
3. or an indication of a failure of the simplification process which was unable to deduce the truth of needed conditions.

We do not expect this to be addressed in any general way in the near future, although incremental approaches are possible.

Before entering into the water of matrix calculations, here is a simple challenge: Can we prove that $(a+b)\times(a-b) \;=\; a^2-b^2$?

To a skilled high school student this would seem to be obviously true, and in fact it is easily proven by any CAS with a command (usually named "expand"). But it is not true for all possible values of a and b known to some CAS. For example, if the CAS allows interval values $a=[0,1]$ meaning "any real number between 0 and 1" and similarly, $b=[0,1]$ then the left-hand side is $[-2,2]$ but the right-hand side is $[-1,1]$.

[3] http://www.w3.org/TR/REC-MathML/chapter1.html#sec1.2

[4] See the Calculemus interest group `http://www.mathweb.org/calculemus/`

Systems providing floating-values can also produce examples which differ from semantics of exact arithmetic. Even the commutativity of addition is violated, where $a + b - a$ can differ from b when $a = 1$ and $b = 1.0 \times 10^{-20}$

Thus the proof offered cannot be "For all assignments of values to a and b in this CAS, the following theorem holds". What can be proven is that, making some global assumptions on primitive domains (integers, rationals, variables assuming values in those domains), abstract mathematical objects built in standard ways on top of these domains (matrices), the relevant operations, definitions of special values, and also assuming the correctness of the algebraic transformations and simplications, *then* the theorem holds. In particular, proofs involving the important domain of floating-point calculations may require special consideration in cases of overflow, underflow, loss of precision, etc. The book $A{=}B$ [1] considers the building of computer proof machines in the area of hypergeometric identities, producing not only proofs, but short certificates of proofs. These are very nice results which should be extended to other areas of mathematical endeavor.

3 Representing and Operating on Symbolic Matrices

For our illustration here we are simplifying our notation and representation in several ways[5]. In particular, we assume that square matrices of size $n \times n$ (but unknown n) are used in this section.

3.1 Representation

A matrix A is a *function* from a pair of positive integer indices (i, j) to a value. If we cannot specify the value more particularly, we can just notate it as $a_{i,j}$ or some computer-language equivalent.

Of the computer algebra systems we are familiar with, only Macsyma allowed us to carry through some of the essential computations below in a reasonably straightforward way. Macsyma provides *contexts* for assumptions of inequalities and properties, and reduces sufficiently simple questions about additional inequalities to true values. Here is how we can define two matrices:

```
am   :  lambda([i,j],a[i,j]) $
bm   :  lambda([i,j],b[i,j]) $
```

The `lambda` notation, familiar to students of programming languages, provides a technique to specify the names and positions of locally bound variables (arguments) to functions. `am(3,4)` evaluates to `a[3,4]`, normally displayed as $a_{3,4}$. The unit matrix (reminder: we again assume size $n \times n$) is

```
unitm:  lambda([i,j],k_delta(i,j)) $
```

where `k_delta` is Macsyma's name for Kronecker delta, defined by **if** $i = j$ **then** 1 **else** 0. This may seem rather vacuous, but given nothing more, the system already knows, *for symbolic indeterminate* `k` that `unitm(1,1)=unitm(k,k)=1`, that `unitm(k,k+1)=0` and furthermore, `unitm(k+m,k)= k_delta(m,0)`.

3.2 Multiplication

We can now multiply any two square matrices in time independent of n with a simple program. (Inevitable! We don't know what n is!). We present it first and then explain afterward.

[5]There is no fundamental problem other than complexity itself, in making a more accurate representation design, and we do so in a subsequent section.

```
matmul(R,S):=   /* An overly simplified version */
 block([rrow:part(R,1,1),
        scol:part(S,1,2),
        index: ?gensym()],
   apply(lambda, [[rrow,scol],
         mysum (R(rrow,index)*S(index,scol),index,1,n)]))$
```

A perfectly general example: `matmul(am,bm)` $\Rightarrow$ `lambda([i, j], sum(a[i, g1] * b[g1, j],g1,1,n))`

As displayed by Macsyma, this answer is actually

$$\lambda\left(\{i,j\}, \sum_{\mathrm{g1}=1}^{n} a_{i,\mathrm{g1}}\, b_{\mathrm{g1},j}\right)$$

Tracing through the example, what has happened is that `rrow` is set to `i`, `scol` is set to `j`, and a new variable name `g1` is produced by `?gensym`. This program, borrowed from the underlying Lisp system, produces a new name each time it is used. The name `g1` is actually different from any other name, even `g1` typed in by the user[6]. The next time `?gensym` is called, another unique name, probably `g2` will be produced. The name `index` is set to `g1`. Through some machinations, we produce a new `lambda` expression whose body consists of a summation of the appropriate product terms over the appropriate limits, namely `1` to `n`. We use the function `mysum` instead of the built-in Macsyma function `sum` because we wished to modify the simplification process of the built-in function. We use our own, but when it suits our purpose, we finally convert it to the Macsyma `sum`, which is nicely typset and has various simplifications. Note also that we are taking advantage of the fact that in Macsyma, an expression which cannot be evaluated because its components are indefinite, such as the product `*` above, is generally left alone. ()

3.3 Theorem and Proof

We can now offer our first proof:

Theorem: *The matrix product of the unit matrix with an arbitrary matrix A is A.*

Proof:

```
is ((matmul(am,unitm)-am)=0)
```

Because this computation return **true**, we are done.

We must fix `matmul` in a few ways, and present a repaired version `matmul1` below. A careful student of the lambda-calculus will know that we can suffer from "variable capture" in the `matmul` program; we fix it to make sure that there are not conflicts with bound/global variables.

In subsequent computations, it is important that we insert into the Macsyma environment the notion that the `index`, here `g1`, is not going to assume arbitrary values as it appears in the first argument to `mysum`. Indeed we know that it can only assume integer values from 1 to `n`. (Recall, once again, we have not specified what `n` is. Furthermore, we do not intend to do so!)

Here is a more careful version of the program which guarantees safe lambda manipulation and, within the currently instantiated Macsyma `context`, asserts several facts about our data. We then compute and simplify the answer. If we were constructing a complete proof, we would instantiate a context, allowing it to inherit from the global context of our domain of discourse. As a courtesy to our memory management

[6] They are "interned" differently in the Lisp system. They merely share the same print name.

system we should kill the context just before returning from the proof. This work is, however, done outside our `matmul` program:

```
matmul1(R,S):=   /* more careful lambda variables */
 block([index1: if part(R,1,1)=part(S,1,2)
                  then ?gensym() else part(R,1,1),
        index2: part(S,1,2),
        index3: ?gensym()],
   assume(1<=index3, index3<=n, /* new information */
          1<=index2, index2<=n, /* should be redundant */
          1<=index1, index1<=n),
   (apply(lambda, [[index1,index2],
          mysum (R(index1,index3) * S(index3,index2),
                 index3,1,n) ]))
         )$
```

Under some circumstances, say if we multiply the unit matrix by an expression with different index sets, `qm: lambda([k,i],q[k,i])`, we get

$$\lambda\left(\{\mathrm{g3}, i\}, q_{\mathrm{g3},i}\right)$$

and must do some matching to see that this is the same as `qm`. We can retain the same index names under other more general circumstances than the test above; we need only know that the index variables of R are not used free in S and vice-versa.

We have ignored a subtlety in this program: it assumes that the inner multiplication in the scalar product is properly encoded in Macsyma's "`*`". This is true only if these matrices' elements are members of a commutative ring e.g. real numbers. If this is not true, and this could be the case if we were dealing with block matrices, the appropriate multiplication would be a dot: "`.`" instead. The characterization of this multiplication in a computer algebra system is handled informally and occasionally to the regret of its users. One solution would be to have an extra argument to `matmul` specifying how to multiply elements. Another would be to record as part of the definition of the matrix, the type(s) of the elements allowed, and from this to decide how to multiply them (An object-oriented approach we describe subsequently). Another solution would be to look at the elements actually supplied, and choose the right multiplication only when needed.

3.4 The Inverse of the Hilbert Matrix

We give the definition of the Hilbert matrix and the putative exact inverse of the Hilbert matrix, and wish to prove that their product is the identity matrix.

```
hilbertm: lambda([i,j],1/(i+j-1))$
hinv:lambda([i,j],(-1)^(i+j)/( i+j-1)*
  (n+i-1)!*(n+j-1)!/(( (i-1)!*(j-1)!)^2*(n-i)!*(n-j)!))$
```

Their product is

$$\lambda\left(\{i,j\}, \frac{(n+j-1)!\,\sum_{\mathrm{g3}=1}^{n} \frac{(-1)^{\mathrm{g3}+j}\,(\mathrm{g3}+n-1)!}{(n-\mathrm{g3})!\,(\mathrm{g3}+i-1)\,(\mathrm{g3}+j-1)\,(\mathrm{g3}-1)!^2}}{(j-1)!^2\;(n-j)!}\right)$$

If we evaluate this expression at $(1,1)$ and ask for its `closedform` in Macsyma we get

$$\frac{n!^2}{n^2\ (n-1)!^2}$$

which according to our hypothesis should be 1. Applying the simplification command `factcomb` "proves" it. We need to show that all on-diagonal elements are 1 and all off-diagonal ones are 0. While our current simplification routines seem to be inadequate to prove that $i \neq j$ (in a context where $i \leq n,\ j \leq n$) implies the above expression is 0, we can easily and rapidly show that any particular off-diagonal, say $(3,4)$ is zero, regardless of n. Alternatively, we can fix n and set $i = j + 1$ and easily show that this expression reduces to zero.

An on-diagonal expression, say $(3,3)$ reduces to

$$\frac{n!\ (n+2)!}{(n-1)!\ (n+3)! - 3\ (n-1)!\ (n+2)!}$$

which turns into 1 when we apply `minfactorial` followed by `ratsimp` or `factor`.

The required search for the needed simplification sequence is actually an important issue. This particular expression can be converted to a sum over hypergeometric ${}_4F_3$ functions and subsequently can be reduced to 1[7]. The challenge is to automate this kind of effort.

We revisit this challenge in a later section.

3.5 Why "mysum"?

Macsyma provides only a framework for treatment of the Kronecker delta function, and furthermore, in its treatment of `sum` tries to provide several subtlely different facilities under the guise of one function name. This causes us difficulty and so we use another function, at least temporarily.

Regarding `k_delta`, we assert a simplification rule that `k_delta(r,s)` is changed to `k_delta(r-s,0)`. To avoid looping indefinitely, we apply this rule only if `s` is non-zero. This is only a first step in canonical simplification of the Kronecker delta. (For example, applying any non-zero linear transform on both arguments does not change its value. Specifying a canonical form for symbolic arguments requires some thought!)

To effect somewhat more pointed simplifications for this form we need to identify three classes of expressions:

1. Those we can deduce to be definitely zero in the current context even if they are not syntactically identically zero.
2. Those we deduce to be not equal to zero.
3. Those which we cannot tell, and so we must leave the `k_delta` unresolved.

Here's what we did:

```
nonzero(zz):=not(is(zz=0))$  /*syntactically zero? */
nonzerop(zz):=block([prederror:false], /*zero in database*/
```

[7]Private communication from R.W. Gosper and Mizan Rahman, e-mail 22 May, 2001 and 24 May, 2001. The proof is too large to fit in the margins of this paper.

```
               is(is(equal(zz,0))=false))$
myzerop(zz):=   block([prederror:false],
               is (is(equal(zz,0))=true))$
/* set up simplification rules */
matchdeclare(ynz,nonzerop, yz,myzerop, ynsz, nonzero)$
tellsimp(k_delta(wany,ynsz),k_delta(wany-ynsz,0))$
tellsimp(k_delta(ynz,0),0)$
tellsimp(k_delta(yz,0),1)$
```

Next we turn to summation simplification. While the system routine can compute with `k_delta` within a sum with numeric limits essentially by expanding it out term by term, it fails to try this if given symbolic limits. Here is what we need: if a summand has a `k_delta` which depends on the index,`i`, say `k_delta(f(i),0)`, then we find the set of values of `i` for which `f(i)=0` and for which `i` lies between the lower and upper limits of the sum. For example if we know that $r > n$ and $s > n$

$$\sum_{i=1}^{n} a_i \text{k\_delta}((i-1)(i-r)(i-s), 0) = a_1.$$

If we know that $1 \leq r \leq n$ and $1 \leq s \leq n$ for integers r and s then the sum is $a_1 + a_r + a_s$. Fortunately Macsyma can deal with simple inequalities as in this example, and it also provides programs for distribution of $\sum$ over sums, while moving multiplicative constants (with respect to the index) outside the $\sum$. Declaring `mysum` to be `linear` does this latter task.

What if we really don't know much about (say) r. In the tradition of "lazy evaluation" we can return $a_1 + a_s +$ (if $(1 \leq s)$ and $(s \leq n)$ then a_s else 0). We must also make sure that the names used here ($r,\ s,\ n,\ a$) are bound correctly regardless of the environment in which the evaluation is done.

Defining `mysum` and the Kronecker delta simplification is somewhat complicated, and not recommended reading except for those curious about Macsyma details.

```
/* Special rules to simplify sum over Kronecker delta*/
/*first get k_delta into a more regular form, with second arg 0.*/
tellsimp(k_delta(wany,ynz),k_delta(wany-ynz,0))$
/* need 2 rules : definitely 0, definitely not 0. others are\
 unchanged */
/*definitely zero case */
tellsimp(k_delta(yz,0),1)$

/* Apply the summation/delta transformation */
 /* set simp:off to
    keep "linear" declaration of mysum
    from making a mess of the rule lhs. */
(simp:off,
  /* Look for something*k_delta inside a sum*/
tellsimp(mysum(termany*k_delta(wany,0),indany,lowany,hiany),
          sumdels(termany,wany,indany,lowany,hiany)),
 simp:on)$

/* Sumdels computes the sum of x(w) over all w such that z(w)=0.
```

```
   It allows only solutions between low<=w<=hi */
sumdels(x,z,ind,low,hi):=
  /* This requires solve to find solutions. */
    apply("+", map(lambda([h],
   subst(h,ind,x)),
  filtersol(map(rhs,solve(z,ind)),low,hi)))$

filtersol(vals,low,hi):=
 if vals=[] then [] else
   block([val:vals[1], prederror:true,
          r: filtersol(rest(vals),low,hi)],
   /*default: if we can't decide predicate: error */
        if (low <= val) and (val <= hi) then cons (val, r) else r)$

/*Arguably a better version might consider that
the Macsyma truth-value analysis includes
true, false, and unknown. If we do not
know if a value is between low and hi, we
can preserve our ignorance for later
evaluation.  In the context of sumdels' usage,
this fancier result is not helpful.*/

filtersol(vals,low,hi):=
 if vals=[] then [] else
   block([val:vals[1],prederror:false,
          r: filtersol(rest(vals),low,hi),
           epred],
 epred: (low <= val) and (val <= hi),
         if epred=false then r else
            if epred=true then cons(val,r) else
             cons ((if epred then val else unknown),r))$

/* Finally, we must change mysum(a*f(i)+g,i,1,n) to
    a*mysum(f(i),i,1,n)+g*mysum(1,i,1,n) to {since mysum --> sum}
    a*mysum(f(i),i,1,n)+g*n. Fortunately such situations
have occurred frequently enough in the past that the
Macsyma system anticipates such a need: This can be done by
the following additional command: */
declare(mysum,linear)$
```

If we give up on our transformations, we can use Macsyma's simplification by converting `mysum` to `sum`. Then commands such as `closedform` and `simpsum` can be used to simplify further. Using `apply1(Z,sumrule1)` converts forms in `Z`:

```
(simp:off,
defrule(sumrule1,mysum(termany,indany,lowany,hiany),
              apply(sum, [termany,indany,lowany, hiany])),
```

```
 simp:on)
```

4 Other Matrix algorithms

Here are a selection of computations that can be done more or less easily with the representations given.

4.1 Transpose

This is a simple trick: reverse the indices:

```
mattrans(M):= apply(lambda,[reverse(part(M,1)), part(M,2)])$
```

Theorem: *The transpose of the transpose of an arbitrary matrix* `am` *is equal to the original matrix.*
Proof: `is(0=am-mattrans(mattrans(am)))` returns **true**.

4.2 Extract a Row/Column

We can produce vectors from columns.

```
matcol(M,c):=  /* return selected column c */
 block([index1: part(M,1,1)],
      apply(lambda, [[index1],M(index1,c)]))

matrow(M,r):=  /* return selected row r */
 block([index2: part(M,1,2)],
      apply(lambda, [[index2],M(r,index2)]))$
```

4.3 Matrix addition

Matrix addition is fairly simple. Again, we'd prefer not to introduce unnecessary gensyms, so we check the indices to assure the two inputs are not in conflict. In such a case we don't have to make new gensyms:

```
matadd(R,S):=
 block([index1: if part(R,1,1)=part(S,1,1)
                  then part(R,1,1) else ?gensym(),
        index2: if part(R,1,2)=part(S,1,2)
                  then part(R,1,2) else ?gensym()]
apply(lambda, [[index1,index2], R(index1,index2)+S(index1,index2)]))$
```

4.4 Create a Diagonal Matrix

Here v is a vector. The result is a matrix which is zero everywhere except on the diagonal. At position (i, i) it has value $v(i)$:

```
matdiag(v):=  buildq([ind1:part(v,1,1),ind2:?gensym(),v:part(v,2)],
        lambda([ind1,ind2],k_delta(ind1,ind2)* v))$
```

In the definition above we use `buildq` which provides a macro-substitution rather than evaluation of the lambda-form. We do not really wish to test to see if the indices are the same until later. The `buildq`, which is described in more detail in the Macsyma online reference manual, looks like a `block` but, roughly speaking, returns an *unevaluated form* with the exception of the names specified in the initial `[]` binding list whose values are inserted in the given form.

4.5 Removal of a Row/Column

First note that our global assumption that all matrices are of size $n \times n$ is now false.

Ignoring this issue for the moment, we need to construct new functions for the changed matrices.

```
matmcol(M,c):= /* matrix without column c, no nested lambdas */
 block([index1: part(M,1,1),
        index2: part(M,1,2),
        prederror:false],
  apply(lambda, [ [index1,index2],
                  M(index1,if(index2<c) then index2 else index2+1)
                  ]))$
```

Here it is critical to set `prederror:false` because without it, the predicate comparison of `index2<c` will be executed right away; in usual circumstances it will result in an error message since the relative values of the index and c are not known yet.

Suppose we wish to create a new matrix with columns 3 and 5 removed. Here is the result.

$$\lambda\left(\{i,j\}, a_{i,\textbf{if }(\textbf{if } j<5 \textbf{ then } j \textbf{ else } j+1)<3 \textbf{ then } (\textbf{if } j<5 \textbf{ then } j \textbf{ else } j+1) \textbf{ else } (\textbf{if } j<5 \textbf{ then } j \textbf{ else } j+1)+1}\right)$$

This expression is somewhat repetitive, and lambda-binding can be used as an alternative. For example,

```
matmcol(M,c):= /* matrix without column c; nested lambda result*/
 buildq([index1: part(M,1,1),
        index2: part(M,1,2),M,c],
      lambda([index1, index2],
        M(index1,if(index2<c) then index2 else index2+1)))$
```

results in the following expression, which (statically speaking) seems to require less space, but more storage dynamically.

$$\lambda\left(\{i,j\}, \lambda\left(\{i,j\}, \lambda\left(\{i,j\}, a_{i,j}\right)\left(i, \textbf{if } j<3 \textbf{ then } j \textbf{ else } j+1\right)\right)\left(i, \textbf{if } j<5 \textbf{ then } j \textbf{ else } j+1\right)\right)$$

Useful in Cramer's rule is the replacement of the column with index `c` by a given alternative column, `newc`, shown here:

```
/* replace a column... */
matreplacecol(M,c,newc):=
 block([index1: part(M,1,1),
        index2: part(M,1,2),
        prederror:false],
```

```
 buildq([M:M(index1,index2),
         new:newc(index2), index1, index2,c],
     lambda([index1,index2],
                  if equal(c,index2) then M else new)))$
```

4.6 Minors

Here is a way to compute minors, stacking up lambdas

```
matmin(M,r,c):= /* a minor with row r and col c removed*/
 buildq([index1: part(M,1,1),
         index2: part(M,1,2),M,r,c],
    lambda([index1, index2],
        M(if(index1<r) then index1 else index1+1,
          if(index2<c) then index2 else index2+1
)))$
```

Another version which does more computation in subscripts is

```
matmin1(M,r,c):=
 block([index1: part(M,1,1),
        index2: part(M,1,2),
        prederror:false],
  apply(lambda, [ [index1,index2],
                  M(if(index1<r) then index1 else index1+1,
                    if(index2<c) then index2 else index2+1)
                  ]))$
```

This seems pretty mindless, but it can be helpful. For example, using `matmin1`, we can show that the minor of the unit matrix with row/column 1 removed is the unit matrix.

In the next section we show that one can feed this in to other programs for the calculation of a determinant.

Aside: Showing that the minor q of the unit matrix with row/column k for *arbitrary* k is still a unit matrix is a good deal trickier, and can't be done automatically by our current program. It requires that we determine

$$q = \lambda\left(\{i, j\}, \text{k\_delta}\left((\textbf{if } i < k \textbf{ then } i \textbf{ else } i + 1) - (\textbf{if } j < k \textbf{ then } j \textbf{ else } j + 1), 0\right)\right)$$

Note that $q(r, r)$ *with* r *unspecified* returns 1; $q(k + n, k)$ returns k_delta $(n, 0)$. If we look at the values of $q - I$ by case analysis, considering $i < k$ or not, and j, k or not, we find this is always 0. If we are given only q and challenged with simplifying it to some possible known function or a "simpler" combinations of functions at our disposal seems difficult. (Undoubtedly such questions in general form are undecidable [3]).

4.7 Determinants

We promised that given a way to express minors, we can proceed on to the determinant. Let us expand by minors along the first column. Then a simple formula would seem to be

```
matdet(M):= apply(sum,[(-1)^k*M(k,1)*
          matdet(matmin(M,k,1)),k,1,n])$
```

Unfortunately this expresses a recursive computation proceeding from n to $n-1$ but with no way of reaching a termination condition. Alternatively, we can view this as an induction step without a basis. One way around this is to proceed one or two steps at a time.

We put a quote mark in front of the internal `matdet` which means "do not evaluate this function" but just display it. The program is further complicated by renaming our index variable to avoid name conflicts, inserting an appropriate assumption about the index variable, and also putting in a simplification function on lambda expressions. Macsyma ordinarily keeps its simplification procedure out of the body of lambda expressions, viewing them as programs to be examined only when they are applied. We, on the other hand, are using the lambda bodies as expressions that should be manipulated, compared, and simplified. We need to pull off the body, evaluate it with respect to knowledge of global variables, and put it back together. The program `lamsimp` does this.

```
matdet(M):=block([k:?gensym(),prederror:false],
  assume(1<=k,k<=n),
    lamsimp(mysum((-1)^(k+1)*M(k,1)*
   'matdet(matmin(M,k,1)),k,1,n)))$

defrule(lambdar1, lambda(wany,termany),
          apply(lambda,[wany, ev(termany)]))$
lamsimp(x):=apply1(x,lambdar1)$
```

If we wish to convert one level of the quoted `matdet` to unquoted, we can do this in Macsyma by converting the noun form to a verb form and evaluating.

```
onestep(z):=apply_nouns(z,'matdet)$
```

Now computing the determinant of the diagonal matrix v created earlier, and running `onestep` on it a few times gives:

$$v_1\, v_2\, v_3\, \mathrm{matdet}\left(\lambda\left(\{i, \mathrm{g29}\}, v_{i+3}\, \mathrm{k\_delta}\left(i - \mathrm{g29}, 0\right)\right)\right).$$

While this does not tell you that the determinant of a diagonal matrix is the product of the diagonal elements, it gets close. We can take the determinant of the unit matrix and run `onestep` any number of times and still get

$$\mathrm{matdet}\left(\lambda\left(\{i, j\}, \mathrm{k\_delta}\left(i - j, 0\right)\right)\right).$$

This is a bit disconcerting; actually we have ignored an aspect of this minor computation, and that is the size of the matrices is reduced by one row and column. What we have really shown is that the determinant of an $n \times n$ unit matrix is the same as the determinant of an $(n-1) \times (n-1)$ unit matrix.

4.8 Equality

A necessary program we have neglected to this point is one to compare two matrices to see if they are equal in the case when they are not identical with respect to argument list. The way to do this is to make a canonical "gensym" argument list and substitute these into the two function bodies to be compared. If the two bodies are then `equal` then so were the lambda expressions. The test looks like this:

```
/* Tell the simplifier to recursively apply the following
test whenever comparing two lambda expressions */
tellsimp(equal(lambda(args1,body1),lambda(args2,body2)),
 testlameq(args1,args2,body1,body2))$

testlameq(a1,a2,b1,b2):=
 block([k1:length(a1), k2:length(a2),
        gens: lambda([g1,g2],?gensym()),
        g:[], eqs:[] ],
  if k1#k2 /* mismatched arg lists */
         then return(false),
  g: part(genmatrix(gens,1,k2),1), /*row of gensyms */
  eqs: map("=",a1,g),/* list of equations   */
  b1: subst(eqs,b1), /*substitute in body 1 */
  eqs: map("=",a2,g),   b2:subst(eqs,b2),
     equal(b1,b2))$
```

4.9 Miscellany

Some additional programs we have written are simpler versions of the ones above, such as multiplying matrices by vectors or scalars. For example, multiplying a scalar times a matrix in constant time and space:

```
scalmatmul(s,m):= /* scalar X matrix -> matrix */
   apply(lambda, [part(m,1), s* part(m,2)])$
```

Multiplying a vector by the unit matrix produces the same result as `matdiag`.
Oddly enough, we can display infinite matrices. Thus the diagonal matrix `v`:

$$\lambda\left(\{i, \mathrm{g2}\}, \mathrm{k\_delta}\left(i, \mathrm{g2}\right)\, v_i\right)$$

can be displayed as

$$\begin{pmatrix} v_1 & \ldots & 0 \\ \vdots & v_i\,\mathrm{k\_delta}\left(i-j,0\right) & \vdots \\ 0 & \ldots & v_n \end{pmatrix}$$

by the program

```
matdisp(m):= matrix([ m(1,1),"...",m(1,n)],
        [":",m(i,j),":"],
        [m(n,1),"...",m(n,n)])$
```

or if we know that we are interested in the (i,i) diagonal entry, we can produce a display like this:

$$\begin{pmatrix} v_1 & \ldots & \ldots & \ldots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ \vdots & \ldots & v_i & \ldots & \vdots \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \ldots & \ldots & \ldots & v_n \end{pmatrix}$$

Returning to computational issues, some programs are even "lazier" than the determinant calculation in the sense of usually doing nothing but setting up a computation for later execution. A solution to an infinite system of linear equations could be computed by Cramer's rule using only the tools given (ratio of determinants), though one would presumably be concerned about the whether the matrix of coefficients was invertible, and the representation – in the case where there is no simplification available – rapidly becomes unwieldy.

5 Relaxing the Fixed Size Assumption

The observation in the previous section is actually critical: we should keep track of the size of matrices. Not all the matrices are indeed of the same size after we manipulate them. Here are some of the issues: Vector and matrix representation must include the row/column sizes, even if they are sizes like $n-1$. The programs that we use must also consider the row/column aspects of their input and output.

It then becomes possible to (say) consider the result of iterating `onestep` n times, and terminating a recursion. Rather than pursue the details using Macsyma's user language, we discuss a different overall "object oriented" representation in this section.

5.1 Generalizing the representation

It becomes necessary to associate sizes with objects.

A given matrix consists of the functions we have used up to this point, plus two more parameters which may be symbols, height = number of rows, and width = number of columns. We mentioned in passing that we need to consider the component multiplication, which seems to plausibly reside in the description of the elements.

A object-oriented approach makes sense here, where a matrix object now has encoded more that our function mapping. It must also encode height, width and element-domain. It may also encode other information (sparsity, rank, preferred display format, ...) The element-domain is itself an object which declares the nature of the domain (e.g. field of rational functions, GF[2], non-commuting symbolic block matrices) as well as appropriate methods for element addition, multiplication, inverse. Each algorithm (say `matmul`) must now look at most of these entries to judge the conformance of the matrices and their elements. It must also set up a new result object with appropriate dimensions and element type, as well as the functional mapping we have (up to now) used as the sole representative of the matrix.

The Macsyma language we have been using was designed in 1967, based on Algol-60 and the Lisp of that time.

Although various paradigms were used in Macsyma system and user code, including data-directed code, inheritance played a role only in some internal routines, visible to those (rare) users who made use of managing contexts and assumptions.

Object-oriented programming was not a consideration. In fact the language objectives were directed toward an elusive goal, namely naturalness of the communication (for mathematicians/ non-computer specialists). The designers of Macsyma were neither the first nor the last to underestimate the difficulty of achieving this in a computer algebra system. The levels of discourse in its design are blurred: the notion of a mathematical function (e.g. cosine) is conflated with the program that sometimes computes its value, but sometimes does not. Thus `cos(1)` is ordinarily left alone, but `cos(1.0)` is converted to approximately 0.54030230586814. The mathematical expression $x + x$, given a programming context value for x say $x = 2$ *evaluates* to 4. If there is no value for x, the expression *nevertheless can be evaluated*, but results in $2x$.

Conditional expressions such as **if** $f(a) > b$ **then** c **else** d suggest that $f(a) > b$ must evaluate to either **true** or **false**, when it could also be the case that computation of $f(a)$ causes an error, or perhaps the truth or falsity of the expression $f(a)$ cannot be deduced from information known to the system (i.e. result `UNKNOWN`), or that it is consistent with what is known, but not necessarily true (`MAYBE`). Also confusing the programmer is the fact that one can sometimes defer, in Macsyma, the computation of the conditional until one has values for $f(a)$ and b. The rules for when such evaluation is delayed depend, in some cases, on the input values as well as the user's efforts in applying the evaluate (`ev`) command. Macsyma seems to deal with `lambda` as though it were the name of a function, but simply one which does not evaluate its arguments. The notions of variable renaming and lambda reduction are implemented in some parts of the system but not others (e.g. `subst`).

A clearer semantic model is maintained in ANSI Standard Common Lisp, where all the variables are either bound or not bound in the programming context. If one attempts to evaluate a variable/name which has no value associated with it, it is an error. Some values may be symbolic data: symbols, lists, strings, etc. The symbolic data can be handled in a variety of ways, including inquiring whether a symbol has associated with it a variable and value binding. In spite of the often-touted feature of Lisp that one can CONS together a list and execute it as a program, in reality the use of the Lisp function EVAL is quite unusual. (One does APPLY or FUNCALL functions on arguments, but EVAL, given the need for care in environments, is not the right procedure.) This is interesting because Macsyma programmers tend to use roughly equivalent notion of EV often.

Lisp also provides clean macro expansion for program-construction. It also provides the Common Lisp Object System (CLOS) through which classes and methods can be defined, and whose use is briefly sketched below. We do not use any of the sophistication available. This representation includes height, width, and element-type entries.

```
;; Example definition of a matrix class

(defclass mat()  ;no superclasses
  ((height :initarg :height :accessor matheight :initform 'n)
   (width  :initarg :width :accessor matwidth :initform 'n)
   (element :initarg :element :accessor matelement :initform 't)
     ;; fun is mapping function from (i,j) to an expression.
     ;; This declaration provides a default
   (fun :initarg :matfun :accessor matfun
        :initform #'(lambda(i j)(list 'm i j)))))

;; (ref m i j) extracts the i,j element of matrix m.

(defmethod ref((m mat)i j)(funcall(matfun m) i j))

;; matrix multiply

(defmethod mul( (r mat) (s mat))
  (let* ((rlam (function-lambda-expression (matfun r)))
         (slam (function-lambda-expression (matfun s)))
         (rrow (car(cadr rlam)))
         (scol (cadr (cadr slam)))
```

```
         (n (matheight r '))
         (ind (gensym)))
    (assert (equal (matwidth r)(matheight s)));;check for conformance
    (assert (equal (matelement r)(matelement s)))
    (make-instance 'mat
                   :matfun
                   `(lambda(,rlam ,slam)
                      (mysum (* (ref ,r ,rrow ,ind)
                                (ref ,s ,ind  ,scol))
                             ,ind 1 ,n))
                   :height (matheight r)
                   :width n (matwidth s)
                   )))

;; an alternative that meta-multiplies
(defmethod mul( (r mat) (s mat))
  (let* ((rlam (function-lambda-expression (matfun r)))
         (slam (function-lambda-expression (matfun s)))
         (rrow (car(cadr rlam)))
         (scol (cadr (cadr slam)))
         (n (matheight r '))
         (ind (gensym)))
    (make-instance 'mat
                   :matfun
                   `(lambda(,rrow ,scol)
                      (mysum (* ,(subst ind rrow rbody)
                                ,(subst ind scol sbody))
                             ,ind 1 ,n))
                   :height (matheight r)
                   :width n (matwidth s)
                   )))

;; Note that if we have mat, vec, scalar etc
;; we can use the generic-function mechanism
;; and call all multiplication programs with the
;; same name, mul.  (There is a question when
;; multiplying a matrix by a symbol v,
;; should v be treated as a scalar or a matrix v[i,j].
;; This would be resolved by our choice/convention.):

(defun mul((r mat)(s mat)) ... )
(defun mul((r vec)(s mat)) ... )
(defun mul((r sca)(s mat)) ... )
;...
```

Dealing with loops in a normal programming language situation, the limits must be made explicit before

running the loop to completion. Looking at the summation in the matrix multiplication, one could insist on summing over some explicit range, say 1 to 100, and compute 100 values. This is what `sum` does below.

However, to meet our needs for an indefinite summation, we can mechanically produce the body of a program which consists of a loop directive. This is done by the lisp macro `mysum` below, which itself does not try to add anything. Instead it produces a piece of list structure that, if later is interpreted as a program and executed, adds together some number of items.

```
(defmacro sum(s i low hi) ;; works only if low and hi are numbers
  `(loop for ,i from ,low to ,hi sum ,s))

(defmacro mysum(s i low hi) ;; returns "loop" form symbolically
                    ``(loop for ,i from ,low to ,hi sum ,s))
```

Re-implementation of our symbolic matrix computation more carefully using the these ideas and tools provided in Common Lisp eventually hits the barrier that Common Lisp does not know useful facts that are built in to Macsyma, namely simplification of algebraic expressions. Fortunately it is possible to use (free) versions of the Macsyma source code when necessary, loading them into any Common Lisp. We would not wish to re-implement Macsyma.

6 More Display Options

We can consider a user interface that allows such input as

```
HilbertMatrix:=
ConstructGeneralizedMatrix
   (rows->m, columns->n,
    element->lambda(i,j).(1/(i+j-1))
    element_type->Rational)
```

We can elaborate on our earlier display technique and construct a display of this form:

$$\text{Hilbert} \;=\; \begin{pmatrix} 1 & \cdots & \frac{1}{j} & \cdots & \frac{1}{n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{1}{i} & \cdots & \frac{1}{i+j-1} & \cdots & \frac{1}{i+n-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{1}{m} & \cdots & \frac{1}{m+j-1} & \cdots & \frac{1}{m+n-1} \end{pmatrix}$$

TEX supplies an option for labeling borders of a matrix that may clarify the situation, at the risk of confusing the left-label with multiplication.

$$\text{Hilbert} \;=\; i \begin{array}{c} \\ 1 \\ \vdots \\ \frac{1}{i} \\ \vdots \\ \frac{1}{m} \end{array} \begin{array}{c} \begin{array}{cccc} & & j & \\ \end{array} \\ \begin{pmatrix} 1 & \cdots & \frac{1}{j} & \cdots & \frac{1}{n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{1}{i} & \cdots & \frac{1}{i+j-1} & \cdots & \frac{1}{i+n-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{1}{m} & \cdots & \frac{1}{m+j-1} & \cdots & \frac{1}{m+n-1} \end{pmatrix} \end{array}$$

This smaller border-matrix display may be adequate to show a useful amount of redundancy: the general trend in the matrix, and a few key values.

$$\text{Hilbert} \;=\; \begin{array}{c} \\ 1 \\ i \\ m \end{array} \begin{array}{c} \begin{array}{ccc} 1 & j & n \end{array} \\ \begin{pmatrix} 1 & \cdots & \frac{1}{n} \\ \vdots & \frac{1}{i+j-1} & \vdots \\ \frac{1}{m} & \cdots & \frac{1}{m+n-1} \end{pmatrix} \end{array}$$

We have raised this output issue once again to make it clear that one could spew out some other language, typically Fortran or C, from these matrix specifications, and that one could replace the body of the lambda-expressions by any mechanism that related two indices to a value, including conventional 2-d array indexing, sparse indexing, or other computational methods. One could also compile-away the formalism we have used and replace in-line the function invocations and replace them with direct memory access, if that becomes appropriate for optimization.

If we talk about Fortran, we should mention that we have dealt with matrices as functional objects, not as state-based entities. That is, we have provided programs for returning a new matrix based on one or more other matrices. We have not provided a mechanism to alter in place a given matrix. This can certainly be supplied, but using such manipulations substantially clouds the mathematical statements that one can make about matrices. (For example, we can prove properties of `unitm`, the unit matrix defined previously. What if someone can alter its entries?) Nevertheless, what can be done in a case where a matrix is really a storage mechanism, is to provide two associated functions: the one we have already used which maps from an index set to a particular memory location, and returns the value stored there, and another (setting) function which maps from that index set plus a value and alters the memory location. One can then set a new value there. The Common Lisp convention is to define `setf` methods for a structure or part of it, as a mechanism for this second operation.

7 Simplification and Proof Techniques

Earlier we pointed out that it would be unreasonable to anticipate that our proofs by simplification will always be easy. To prove that $A = B$ (by simplifying $A - B$ to zero), or disproving by simplifying to a distinctly non-zero expression, is a classic problem in Computer Algebra systems. On one hand, Richardson's results [3] offer an explanation for us not to find a uniform procedure. On the other hand, we can point to fully effective methods over polynomials in any number of variables, and heuristic methods over a rather large domain. That is, if $f(x, y, z) := A - B$, then choosing random values for x, y, and z in some domain and

evaluating f repeatedly would tend to give us a hint of validity, although one might miss the points in a set of measure zero which disprove an allegation.

We are more encouraged by proof by successive identity rule-transformations. The exact sequence of transformations to reduce $A - B$ to zero (or show that it is not) may benefit from heuristic search. Some commands may enlarge the expression to allow cancellations. This was the case with `minfactorial`.

Well known transformations on trigonometric forms sum-of-angles versus product of functions sometimes work at cross-purposes, but are not inverses. Drastic methods such as removing all trignometric forms in favor of complex exponentials can produce canonical forms for some classes of expressions.

Totally, or nearly, unguided heuristic search among transformations to find a minimum size expression (assuming zero is about as small as one can get) is a possibility. Ordinarily I would be skeptical of trying genetic algorithms or simulated annealing in such a domain, but in the absense of any hints of an organized approach, one could perhaps assemble a collection of such trial identity transformations. Naturally there are significant hints available in the nature of most mathematical expressions, suggestive of viable transformations. Relationships among continguous hypergeometric functions can, for example, be systematically exploited.

8 Conclusions

We have sketched out the beginning of a way of thinking within a symbolic computer language system, either a computer algebra system or a language like Common Lisp, about matrices more generally as maps, without tying them to particular dimensions, or to explicit numeric entries. That is, one should be able to manipulate such objects constructively without making them any more concrete than necessary.

We have demonstrated some of our simpler objectives: proofs of some theorems are reduced to calculation. Other proofs seem to require considerable additional effort in enhancing computer algebra systems: case analysis, (hyper) geometric reasoning. We have illustrated that we can, contrary to expectations, solve certain problems with indefinite parameters, without resorting to any form of induction, or forward/backward chaining, or lengthy and error-prone derivations of proofs from first principles based on weak theories. Without doubt we are relying on the correctness of the computer algebra system, as well as the correctness of the rules and programs we write. We see no alternative to this since the even the correctness of automated theorem proving systems must also rely on correctness of (for example) C compilers and underlying hardware.

As should be evident, perfecting the routines for simplification allow us to take substantial leaps in reasoning. Further work will not only grow the collection of simplifications, but work on assuring that the order in which they are applied produce the desired proofs. This requires that there be a systematic approach to applying identities towards reaching a goal. In some simple algebraic cases we can succeed in proving that $a = b$ because our procedure reduces $a - b$ to zero in some computational domain with effective zero-equivalence procedures. This cannot, in general, be required [3], and so we must deal with sub-domains or partially effective procedures [1].

Acknowledgments

Thanks to R.W. Gosper for email discussions on simplification and Macsyma. This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley.

References

[1] Petkovsek, M., Wilf, H.S. and Zeilberger D., $A{=}B$, A.K. Peters, Wellesley, Massachusetts, 1996.

[2] Pierce, John R.; "An Introduction to Information Theory". Symbols, Signals and Noise.", Revised edition of "Symbols, Signals and Noise: the Nature and Process of Communication" (1961). Dover Publications Inc., New York, 1980

[3] Richardson, D. "Some Undecidable Problems Involving Elementary Functions of a Real Variable", *J.Symb. Logic, 33* no. 4 p. 514–520. Dec., 1968.