Computer Algebra and Operators

Richard Fateman*
The University of California, Berkeley

Robert Grossman†
University of Illinois at Chicago

January, 1989

*supported in part by NSF grant number CCR-8812843, DARPA order #4871, contract N00039-84-C-0089, the IBM Corporation; the State of California MICRO program.

†supported in part by grant NAG2-513 from the NASA-Ames Research Center.

1 The Symbolic Computation of Operators

After defining the two expansions

$$\exp(A) = \sum_{i=0}^{N} A^i/i!$$

$$\log(1+A) = \sum_{i=1}^{N} (-1)^{i+1} A^i/i,$$

a computer algebra system such as MACSYMA or Maple will quickly compute

$$\log(\exp(A)\exp(B)) = A + B + O(N+1).$$

Here $O(N+1)$ denotes terms containing a product of $N+1$ or more A's and/or B's. This computation depends crucially upon the fact that $AB = BA$; for objects for which this not true, certain correction terms enter. For example, if A and B are matrices, then in general $AB \neq BA$ and

$$\log(\exp(A)\exp(B)) = A + B - \frac{1}{2}AB + \frac{1}{2}BA + \ldots.$$

The symbolic computation of operator expansions ~~such as these differs~~ in a number of ways from the symbolic computation of expressions in commuting variables. The papers in this volume consider various aspects of such computations. In this introduction, we first discuss some of the capabilities that prove useful when performing computer algebra computations involving operators. These capabilities may be broadly divided into three areas: the algebraic manipulation of expressions from the algebra generated by operators; the algebraic manipulation of the actions of the operators upon other mathematical objects; and the development of appropriate normal forms and simplification algorithms for operators and their actions. We then conclude the introduction by giving a little background and a brief description of the problems considered by the authors in this collection.

1.1 Algebra of Operators

Let $E_1, E_2, \ldots, E_M$ denote operators and α a number. Then $E_1 + E_2$, $E_2 E_1$, and αE_1 are all operators of the same type. That is, a finite set of operators naturally generate an algebra of operators under addition and composition. Let $\mathbf{R}\{E_1, \ldots, E_M\}$ denote this algebra. This is just the free associative algebra over $\mathbf{R}$ generated by the symbols $E_1, \ldots, E_M$. The first capability of a computer algebra system for operators, then, is that it support the algebraic operations of addition and composition of operators.

The first issue this raises is how to represent operators and operations on them in a context which has already usurped most of the notation for an algebra of expressions. Is it possible to use some multiplication operator (typically "*") for operators, or should one use another notation? Maple[2] uses "@" and Macsyma[4] uses "." for composition, but juxtaposition for application. Mathematica[6] takes no special notice of this, but allows the use of juxtaposition for multiplication. (It thereby maps `f(a)` into the same expression as `a*f`; if you intend to convey the notion $f(a)$, you type `f[a]`.) In fact, the notation and semantics of operators has at best been a patch added on to conventional general-purpose algebra systems; by contrast, the

most effective computer implementation of operator algebras has been quite application specific, as for example, in the case of manipulation of perturbation series.

Yet the need for combining operator notation with the generality of widely available computer algebra systems dictates that we seriously address the representation and manipulation of operators. A good test for any system is to take some simple and familiar concepts and see if they can be naturally expressed and manipulated. For example, a operator that expresses "differentiation with respect to x" should be natural. Extensions to represent the 2nd or the nth derivative should be natural. Mixed derivatives (with respect to x and y) should be possible: a natural operation might be the interchange of orders of differentiation, and the combination of common variable "powers". Evaluation of the derivative at a point, say the origin (the un-obvious $f'(0)$) is a challenge [2]. Because algebra systems are generally built to be extensible by procedure definition, data-type extension, transformation-rule (or pattern-match) extension, and aggregations of data, it is natural for one to hope that new operators can be defined, and their significant properties encoded, in a natural form.

There is a tension between expressiveness and precision. In different contexts, we have different expectations. Should the algebra system be expected to treat identical notations in different ways? Consider the notation $(L + a)(y)$. This might mean (say if L is a derivative operator) $dy/dx + ay$. Yet in other circumstances we might prefer that a constant a be an operator equivalent to a function which returns a, regardless of its argument. In that case, $(L + a)(y) = Ly + a$. If an algebra system requires unambiguous notation, it may be unnatural in nearly all mathematical contexts. For the two variations above, Maple would use `<L@x+a*x|x>@y` and `<L@x+a|x>@y`, respectively while Macsyma would require a declaration `declare(a,opscalar)` then `(L+a)y` for the first and would use `(L+constop(a))y` for the latter. A proposal and implementation for Macsyma by T. H. Einwohner [3] would use the notation `L+a*I | y` to mean $L(y) + a * y$ (Unfortunately, the use of "|" is at odds with the Maple convention). Einwohner's design is somewhat more regular in reflecting the duality between expressions involving operators (which can be quite arbitrary), and the results of applying the operator expression to arguments. The use of parameters is an important component of the design. For example, `f(a)|y` is an alternative notation for `f(y,a)`. Finally, he does not further overload the use of "." as non-commutative multiplication by using it for operator composition.

Another issue is how to best structure the data and the manipulative

algorithms for handling expressions from free associative algebras. These issues have been examined since the earliest days of computer algebra systems, and are perhaps the aspects of the computer algebra of operators which can be attacked entirely by conventional means; this usually constitutes a mapping into algebraic tree representations, where most algorithms can be treated as tree-traversals (as described in any text on data structures). On the other hand, truly efficient manipulation may dictate more compact representations, including so-called string-polynomial manipulations, matrix encodings, or other schemes that have been devised for hand manipulation.

1.2 Actions of Operators

The usefulness of operators to scientists and engineers derives not from their properties as abstract algebraic objects but rather from their interesting actions on other mathematical objects. For example, matrices act as linear transformations on vector spaces, vector fields act as derivations on spaces of functions, the physicist's raising and lowering operators act on the space of polynomials, the algebra generated by shift operators acts on the space of sequences, and the algebra of maps acts on the space of domains.

This brings us to the second major requirement on a computer algebra system for operators. They must support the action of operators on objects from the natural domains of definition of the operators (and presumably the natural domains of the algebra system). For example, having formed the matrix expression $E = A + A^2/2! + \ldots + A^4/4!$, it is useful to be able to apply E to vectors. The merging of the two notations leads to a complexity and variety of actions that is probably the single most important barrier to satisfactory implementations of operators in computer algebra systems. The operator E above certainly looks like a polynomial in A; for some purposes it can best be treated as a polynomial; in other contexts as in section 1, it certainly is not equivalent.

There is a more subtle, yet related issue. There is no natural mathematical notation to specify the various interlocking operations possible on the mixed domain of operators and operands. For example, consider an operator F, and its inverse which we will conventionally write as F^{-1} (even though it may have rather little to do with $1/F$) and an expression $F\ F^{-1}\ x$. It is natural for us to apply simplification and "cancel" the operators, yielding x. It is not so obvious to us, or to the computer, though, whether, in more complex situations to apply or simplify. Is FFx computed faster if F is

"squared" first? Or is $F(Fx)$, where the parentheses suggest some ordering, preferable?

Sometimes partial simplification is useful. Consider a definition of a new operator $F :=< \sum_{i=1}^{n} x \mid x,\ n >$ where we use Maple's notation to indicate that x and n are the formal names used for two actual "parameters" to the operator. If G is another operator (presumably operating on positive integers) then $F(G,3)$ is $G(1) + G(2) + G(3)$. Consider $< F(I,k) \mid k >$, where I is an identity operator. At what time (if ever) would a computer algebra system realize that this is equivalent to $< k(k+1)/2 \mid k >$?

What kind of syntax do we supply for the user to define such a simplification? Furthermore, how do we deal with an action of a particular operator on a domain which is not supported? For example, should the syntax describing the action of shift operators acting on sequences be the same as the syntax describing the action of vector fields as derivations on the space of functions?

How can the implementation of operators as algebraic objects be best merged with the implementation of operators as objects acting on other objects in the computer algebra system?

It appears that linguistically, two approaches are possible, and these are not addressed in Maple or Macsyma. One is to require unambiguous specification of operations such as operator simplification (so they would occur on command), a distinct notation for application of operators, and an explicit translation (back and forth) from operator to functional notation.

Another approach is to use the domains of objects (themselves possibly other operators) to disambiguate the meanings of operators, at least whenever possible. This may require fairly sophisticated pattern-matching support which checks the types of arguments. An example used by Gonnet [2] illustrates this. Consider the expression $a \times D \times D \times y$. If we assume D is an operator, each of the three "multiplications" may be a different concept. The first is symbolic multiplication by a constant. The second is composition, and the third is application. Yet we were able to disambiguate this by looking at the types of the "arguments" to $\times$. A constant on the left, a, means traditional multiplication; a non-operator on the right, y means that the operator to the left is being applied to it. A multiplication between two operators means composition. Note that it would be an error to simply work from the right to the left, applying as we go, although for this expression it might work. Consider a non-integrable form z, and the integration operator D^{-1}. Then $D\ D^{-1}\ z$ could not be simplified, because the application of the integration operator would not "work" (unless D were specially coded to

"unwrap" an integral).

There are many open questions, and it appears to us that the best process for making headway in the introduction of operator calculi in algebraic manipulation is to explore serious applications, ones which are challenges for humans and computers. Without further experience such as we see in this volume of papers, it is too easy to make false starts and empty promises. We do not know, for example, if it make sense for the language describing the action of matrices on vectors to be the same as the language describing the action of differential operators on the space of functions.

While Maple proposes a methodology primarily based on an extension of procedures, with some original notation for operators, Macsyma uses (for the most part) already existing notation for non-commutative multiplication; we expect that users of the Mathematica system will primarily use pattern matching and notations which look fundamentally like functional application. Each approach has its advantages and advocates.

1.3 Normal Forms and Simplification

Data structures and algorithms to manipulate operators depend crucially on the action of the operators upon objects from the natural domains of definition of the operators. Normal forms for expressions built from matrices are probably not the best normal forms for expressions built from maps. Questions about normal forms, simplification, and efficient evaluation of operator expressions are by and large open.

It appears that another significant area of application of a computer algebra system is the manipulation of operator expressions to produce simplified or if possible normal forms for the various types and combinations of supported operators by the computer algebra system. This may involve transformation of operator expressions to a well-supported domain (typically polynomials), or collections of rules which drive expressions algorithmically or heuristically, toward equivalent but simpler forms.

Given the state of the art, it seems inevitable that a successful general-purpose system will have to provide some facility for users to implement their own normal forms and simplification algorithms for more specialized types of operators and actions.

2 Examples of Operators and their Domains

In this section we give brief descriptions of the computer algebra computations that arise when working with various operators and their actions. The papers in this collection give careful and complete descriptions of how the authors dealt with some of the issues mentioned above.

2.1 Perturbation Theory

A classical problem in perturbation theory is to compute approximate solutions to differential equations containing small parameters. Consider van der Pol's equation

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + x - \epsilon(1-x^2)\frac{\mathrm{d}x}{\mathrm{d}t} = 0,$$

where $\epsilon \neq 0$ is a small parameter. The starting point is to expand a putative solution $t \longrightarrow x(t)$ in a power series in ϵ

$$x(t) = x_0(t) + \epsilon x_1(t) + \epsilon^2 x_2(t) + \ldots,$$

and then substitute this series into the original differential equation to obtain a sequence of differential equations (one for each power of ϵ) for the unknown coefficient functions $x_i(t)$. These auxiliary differential equations have the property that the ith equation involves only the coefficient functions $x_0(t)$, ..., $x_i(t)$, so that the sequence of differential equations can be solved recursively. In the example above, the equations are

$$\begin{aligned}
\frac{\mathrm{d}^2}{\mathrm{d}t^2}x_0(t) + x_0(t) &= 0 \\
\frac{\mathrm{d}^2}{\mathrm{d}t^2}x_1(t) + x_1(t) &= \left(\frac{\mathrm{d}}{\mathrm{d}t}x_0(t)\right)(1 - {x_0}^2(t)) \\
&\vdots
\end{aligned}$$

There are several ways of approaching these types of computations. Letting $\dot{x}(t) = y(t)$, Van der Pol's equation can be written as the first order system

$$\begin{aligned}
\dot{x}(t) &= y(t) \\
\dot{y}(t) &= -x(t) + \epsilon(1 - x^2(t))y(t).
\end{aligned}$$

Let E_1 denote the vector field $y\partial/\partial x - x\partial/\partial y$ and let E_2 denote the vector field $(1-x^2)y\partial/\partial y$. Then Van der Pol's equation becomes

$$\dot{z}(t) = (E_1 + \epsilon E_2)(z(t)),$$

where $z(t) = (x(t), y(t))$. Notice that $E_2E_1 \neq E_1E_2$.

Let T_n denote the operator which acts upon functions $t \to z(t)$ and returns the first nth terms in a power series expansion in ϵ of the function. Then the ith auxiliary equation is equivalent to the system of equations which is the coefficient of ϵ^i in the expansion of

$$(\mathrm{d}/\mathrm{d}t - (E_1 + \epsilon E_2))\,T_n(z(t)) = 0,$$

for n sufficiently large. From this point of view, perturbation theory is concerned with the symbolic algebra of noncommutative operators such as E_1 and E_2 acting on the domain of power series expansions of the form

$$f(x,y) = f_0(x,y) + \epsilon f_1(x,y) + \epsilon^2 f_2(x,y) + \cdots.$$

A different, but related point of view is used by R. Rand in his contribution "Perturbation Methods and Computer Algebra". In this paper Rand describes a computer algebra system built using MACSYMA, which in a systematic fashion can perform serious perturbation computations, including the computation of normal forms and center manifold reductions.

2.2 Finite Difference Operators and Domains

Consider the heat equation in a bounded region Ω of the Euclidean plane

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, \quad \text{for } x \in \Omega \text{ and } t \geq 0.$$

To compute the temperature $u(x,t)$ numerically using finite differences, we need to discretize the domain Ω, the function $u(x,t)$, and the differential operator

$$\frac{\partial}{\partial t} - \frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2}.$$

This can be done in many ways. Let D denote a finite mesh of points

$$\begin{aligned} x_i &= x_0 + i\triangle x \quad \text{for } i = 0, \ldots, I \\ y_j &= y_0 + j\triangle y \quad \text{for } j = 0, \ldots, J \end{aligned}$$

covering the region Ω, and let $u^n_{i,j}$ denote the temperature at time $n\triangle t$ at the mesh point $(x_0 + i\triangle x, y_0 + j\triangle y)$.

With the shift operators

$$\begin{aligned} E_x(k)u^n_{i,j} &= u^n_{i-k,j} \\ E_y(k)u^n_{i,j} &= u^n_{i,j-k}, \end{aligned}$$

we can define the difference operators

$$\begin{aligned} \delta^2_x u^n_{i,j} &= (E_x(-1) - 2E_x(0) + E_x(1))\, u^n_{i,j} \\ &= u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j} \\ \delta^2_y u^n_{i,j} &= (E_y(-1) - 2E_y(0) + E_y(1))\, u^n_{i,j} \\ &= u_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}, \end{aligned}$$

and compute the temperature $u^{n+1}_{i,j}$ given the temperature $u^n_{i,j}$ implicitly using the scheme

$$u^{n+1}_{i,j} = u^n_{i,j} + \frac{1}{2}\left(\frac{\triangle t}{\triangle x^2}\right)\left(\delta^2_x u^{n+1}_{i,j} + \delta^2_x u^n_{i,j} + \delta^2_y u^{n+1}_{i,j} + \delta^2_y u^n_{i,j}\right).$$

Notice that the basic objects are: domains, such as D; functions defined on domains, such as u; operators defined on the functions, such as δ^2. The latter two objects can both be thought of as maps: a function on a domain can be thought of as a map from the domain to the range of the function; an operator on functions can be thought of as a map from the space of functions on domains to itself. In the paper "FIDIL: A Language for Scientific Programming" by P. Hilfinger and P. Colella, the language FIDIL (FInite DIfference Language) is introduced which makes domains and maps basic data types and provides for the efficient computation of objects built from them. This makes the translation of standard numerical algorithms into programming statments very simple.

Related work is contained in [1].

2.3 Automatic Derivation of Dynamical Equations

The time evolution of a mechanical system consisting of a system of rigid bodies joined together by massless ball and socket joints can be quite complicated. It is difficult to write down the correct equations of motion can be difficult. It would be very useful if a program could automatically

derive the equations of motion of this type of mechanical system. Similarly, an electrical circuit consisting of resistors, capacitors, and voltage sources can exhibit interesting dynamical behavior. It is an interesting problem to write a program whose input consists of a description of a mechanical or electrial system and whose output consists of the differential equations governing the time evolution of the state variables of the system.

A description of the system would include the following:

System parameters. System parameters must be defined and specified. For example, the moment of inertia of a body is defined to be $\int_{body} Q \cdot Q \mathrm{d}m(Q)$, where $m(Q)$ is the mass distribution of the body. To describe the system, the mass distributions and topology of the connections of the rigid bodies must be given, and the moments of inertia must be computed.

Dynamical variables. The dynamical variables must be defined. For example, the rotation matrix $B(t)$ of a rigid body, which specifies how the body is turning in space, must be defined. Using this, the angular velocity $\Omega(t)$ of the body can be defined via $\Omega(t) = \dot{B}(t)B^{-1}(t)$. As a another example, Newton's Law $F(t) = \mathrm{d}p(t)/\mathrm{d}t$ defines the force $F(t)$ acting on a body in terms of the momenta $p(t)$ of the body. Both these example are typical, in the sense that dynamical variables are often defined by differentiating other dynamical variables. Notice that this gives rise to differential identities satisfied by the dynamical variables.

Algebraic relations. The dynamical variables not only satisfy differential relations, but typically algebraic relations determined by the geometry of the particular system. For example, if $r_1(t)$ and $r_2(t)$ denote the positions of the center of mass of bodies 1 and 2 respectively, $r(t)$ denotes the position of the center of mass of the ensemble of the two bodies, $d_1(t)$ and $d_2(t)$ their initial displacements, and $B_1(t)$ and $B_2(t)$ the rotation matrices describing the rotation of the bodies, then

$$r(t) = B_1(t)d_1(t) + B_2(t)d_2(t).$$

State variables. Because of the algebraic and differential relations satisfied by the dynamical variables, it is possible to select some of the dynamical variables, called state variables, and to construct all of the other dynamical variables from these. For example, for two bodies, only the position and velocity of the two bodies and their center of mass need be specified. Alternately, the position and momenta of the two bodies, and of their center of mass, can be specified.

Evolution of state variables. Finally, differential equations giving the time evolution of the state variables need to be given, together with the corresponding initial conditions. For example, the initial positions and velocities of each of the bodies and of each of the joints must be specified in order for the dynamical equations giving the time evolution of the state variables to be integrated.

The contribution "Multibody Simulation in an Object Oriented Programming Environment" by N. Sreenath and P.S. Krishnaprasad undertakes the automatic derivation of the equations of motion for systems of coupled rigid bodies in the plane, while the contribution "The Dynamicist's Workbench: I. Preparation of Numerical Experiments" by H. Abelson and electrical networks consisting of resistors, capacitors, and sources. Both papers start with a description of the system, and by using a variety of symbolic and symbolic/numeric techniques eventually compute numerical simulations of the differential equations which the programs automatically compute.

The papers treat the problem of finding state variables differently. The paper by Abelson and Sussman finds state variables using a symbolic Newton iteration to eliminate relations among dynamical variables. The paper by Sreenath and Krishnaprasad uses symmetries of the mechanical system and a technique from geometric mechanics called reduction to derive equations of motion on a smaller dimensional phase space consisting of state variables.

Studies such as these make heavy use of symbolic computation of operators. For example, the dynamical equations of a system of coupled rigid bodies can be written as $\dot{F}(t) = \{F, H\}$, where H is the Hamiltonian of the system, and both F and H evolve on the appropriate phase space P. Here $\{\cdot,\cdot\}$ is a noncommutative operator defined on

$$\mathbf{C}^\infty(P) \times \mathbf{C}^\infty(P) \to \mathbf{C}^\infty(P).$$

2.4 Lie Brackets and Vector Field Systems

Recall that a smooth vector field $F(x)$ defined in a neighborhood of the origin in $\mathbf{R}^N$ is of the form

$$F(x) = \sum_{i=1}^{N} a_i(x) \frac{\partial}{\partial x_i},$$

where $a_1(x), \ldots, a_N(x)$ are smooth functions defined in a neighborhood of the origin. The Lie brackets of two vector fields $F(x)$ and $G(x)$ is defined

to be

$$[F, G] = F(x)G(x) - G(x)F(x).$$

It is a fundamental property of Lie brackets that although they appear to be second order differential operators they are in fact first order differential operators because of the cancellation of the second order partial derivatives. In other words, the Lie bracket of two vector fields is a vector field. It therefore makes sense to form iterated Lie brackets such as $[[F, G], G]$.

Observe that the Lie bracket of two vector fields is a measure of how far from commuting the two vector fields are. For example, the Lie brackets of the coodinate vector fields $F_i = \partial/\partial x_i$ are all zero, indicating that these vector fields all commute with each other. In other words, $F_i F_j = F_j F_i$. On the other hand, the Lie bracket of the vector fields $F_1 = x_2 \partial/\partial x_1$ and $F_2 = \partial/\partial x_2$ is equal to

$$[F_2, F_1] = \partial/\partial x_1.$$

Let $F(x)$ and $G_1(x), \ldots, G_m(x)$ denote smooth vector fields defined in a neighborhood of the origin in $\mathbf{R}^N$. It turns out that the local behavior of the nonlinear control system

$$\dot{x}(t) = F(x(t)) + \sum_{i=1}^{m} u_i(t) G_i(x(t)), \qquad x(0) = x_0 \in \mathbf{R}^N,$$

is determined by the algebraic properties of iterated Lie brackets of the F and G's. This is analogous to the fact that the local behavior of a smooth function is determined by its Taylor coefficients. Because of this fact, questions about the dynamical behavior of control trajectories can be reduced to symbolic questions about the algebraic properties of the noncommutative operators $F, G_1, \ldots, G_m$ acting on the domain $\mathbf{C}^\infty(\mathbf{R}^N)$ of smooth functions on $\mathbf{R}^N$.

The contribution "Symbolic Computations in Differential Geometry Applied to Nonlinar Control Systems" by O. Akhrif and G. L. Blankenship describes a software package written in the computer algebra system MACSYMA, which through the symbolic computation of the proper Lie bracket expressions, can analyse system theoretic properties of a nonlinear control system, such as feedback equivalence, left-invertibility, or the design of control laws.

The contribution "Vector Fields and Nilpotent Lie Algebras" by M. Grayson and R. Grossman considers conditions on the Lie brackets which guarantee that the trajectories of a vector field system can be explicitly integrated in closed form. In general this cannot be expected and the game

is to find as large a class of such systems as possible. The paper gives some simple examples of such systems which were computed used the computer algebra system Maple.

References

[1] David Balaban, "The Automatic Programming Project," Lawrence Livermore National Laboratory Technical Report, 1987.

[2] Gaston Gonnet, "An Implementation of Operators for Symbolic Algebra Systems," in *Proc. ACM SYMSAC '86 Conf.*, Waterloo, Ontario, Canada, 1986, 239–243.

[3] Theodore H. Einwohner, "A Vaxima Implementation of Operator Calculus," Lawrence Livermore National Laboratory Computer Science Report TS88–60, Dec. 1988.

[4] Jeffery P. Golden, "An Operator Algebra for MACSYMA," in *Proc. ACM SYMSAC '86 Conf.*, Waterloo, Ontario, Canada, 1986, 244–246.

[5] Stephen M. Watt and Jean Della Dora, "Algebra Snapshot: Linear Ordinary Differential Operators," Scratchpad II Newsletter, (1986) Vol. 1, No. 2, 14–18.

[6] Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, 1988.