THE MACSYMA SYSTEM

W. A. Martin and R. J. Fateman
Project MAC, M.I.T, and Harvard University
Cambridge, Mass.

Summary

MACSYMA is a system for symbolic manipulation of algebraic expressions which is being developed at Project MAC, M.I.T. This paper discusses its philosophy, goals, and current achievements.

Drawing on the past work of Martin [6], Moses [9], and Engelman [1], it extends the capabilities of automated algebraic manipulation systems in several areas, including

a) limit calculations
b) symbolic integration
c) solution of equations
d) canonical simplification
e) user-level pattern matching
f) user-specified expression manipulation
g) programming and bookkeeping assistance

MACSYMA makes extensive use of the power of its rational function subsystem. The facilities derived from this are discussed in considerable detail.

An appendix briefly notes some highlights of the overall system.

Contents

I -Introduction and Goals

Computers have an important role to play in applied mathematics. Their ability to accurately carry out large numbers of computational steps has revolutionized the field of numerical analysis. Many classical methods for hand calculation of approximate solutions (such as higher-order quadrature formulas) have been abandoned. It may turn out, however, that some years from now this will be only a minor part of the computers' contributions to applied mathematics. By extrapolating from the successful applications of symbolic computer methods already reported, and from the current research in problem-solving by computer, one can imagine a computer system which serves the mathematician as a tireless, capable, knowledgeable servant, co-worker, and occasionally, mentor. The system would know and be able to apply all of the straightforward techniques of mathematical analysis. In addition, it would be a storehouse of the knowledge accumulated about many specific problem areas, such as treatments of differential equations or series. In some areas, such as symbolic integration, it would apply complex and tedious algorithms to produce results considered to be in the domain of unstructured problem-solving only a few years ago.

If such a system can be constructed, its impact on applied mathematics would be substantial. Books would still be used, but only for tutorial exposition. It would be possible for the casual mathematician at a time-shared computer terminal to bring to bear on his problem a wider and more current range of methods and information. It seems reasonable to expect that a mathematician's thinking and productivity would be stimulated when he could quickly work out the consequences of his ideas. The way would be opened for the discovery of new problem-solving techniques.

These goals are not new, nor are they unique to mathematics. There are clear parallels in systems design, medical diagnosis, and interactive problem solving in many fields. We mention them here because we plan to extend the MACSYMA system until it becomes clear whether or not such goals can be attained. We feel that we will be able to do this. Our rough hypothesis is that a mathematician knows perhaps 10,000 mathematical facts. For example, if a student learned four facts an hour, four hours a day, five days a week, nine months a year for four years, he would learn some 12,000 facts. The average mathematician may not be able to sustain this pace with complete retention, but he has been learning for a longer time period. At present, the MACSYMA system contains perhaps 500 mathematical facts. For the system to be generally acceptable as a mathematical co-worker, we might estimate that it is necessary to expand the knowledge content of the current system by a factor of 20. The current knowledge is embodied in a program of about 30,000 computer words, embedded in a 60,000 word system. The expanded program might then be around 600,000-1,000,000 words, assuming that the growth will be roughly linear with the number of facts added.

This roughly linear estimate for expansion is based on system programming considerations which arise in the construction of large systems of this type. If the interaction of every fact with every other fact had to be explicitly represented, then the size of the program would tend to grow as the square of the number of facts. Our experience to date indicates that this will not occur. The MACSYMA system currently consists of about 20 principal modules. The interactions are of two types, inter-module and intra-module. As a module is asked to communicate with an increasing number of other modules, its internal complexity does increase, but the increase does not continue indefinitely as more modules are added. There is a limit to the number of really distinct and valuable facets which we have been able to find for a module to present to the world.

It may be argued that a system consisting of many independent modules will lack global understanding; that the facts will be in the system, but the mathematician will not know how to obtain them. This is certainly the case in the large programs which have been developed for time-sharing and operating systems. These might be characterized as being ineffectively hierarchical, with many duplicate functional units. The FORTRAN and PL/I subroutine libraries may be similar, but incompatible; also, there are many possible (perhaps superfluous) communication links between modules. These systems are, however, very useful. We picture the MACSYMA system as more hierarchical and more closely knit than current time-sharing or operating systems. Just how useful a personality it can be made to present to the world is one of the objects of our research.

We have grave doubts about the usefulness of large systems constructed through the haphazard contributions of unsophisticated users. Every new bit of the system must be carefully integrated with the old.

In addition, we rely heavily on the work of our colleagues in the field for the analysis and development of new symbolic algorithms. It is their work which makes us feel that our goal of amassing the fastest and most powerful techniques can be realized.

While systems like MACSYMA must be carefully integrated, they must not be restricted to an inflexible language, a single data representation, or a minimal set of transformations. A powerful algebraic manipulation system must respond to a variety of demands and constraints, both from internal modules, and external users. We attempt to do this by providing a small number of carefully chosen alternative approaches, rather than one very general one.

The more specifically a data representation and algorithm is tailored to a given application, the greater power the program has for that application. On the other hand, such programs require a great deal of effort to write and are less generally applicable. Our opinion is that the greatest gain in power comes from applying, in each case, an algorithm which fully exploits the mathematical properties of the problem to be solved. A smaller gain, which tends to be independent of problem size, comes from optimum selection of the data representation. We are employing a three part approach: (a) We provide a general language and data representation so that a user may code any algorithm he wishes, although the execution may be inefficient; (b) We try to provide all of the necessary basic algorithms, along with special data representations, if they are appropriate to make the algorithms efficient; and (c) We are initiating research in automated algorithm and data-representation improvement.

To expand on point (a), our approach to user language is also based on a combination of, rather than a compromise between, ease of use and efficiency. We want to make it possible for the user to use notation natural to his problem, although we must restrict him somewhat to the framework of notations we have chosen for our user-computer interface. On the other hand, we are willing to make substantial transformations on the input in computing the internal form of an expression.

It is clear that the system we are trying to create is a very ambitious one. Nevertheless, we think it is appropriate because it focuses our thinking and efforts on the important computer science questions of the day.

One of these is the creation of intelligent systems and the modeling of thinking and problem solving. Several years ago it was optimistically predicted that computers would by now be proving important new theorems. This has not come to pass for several reasons. The general techniques available for improving search strategies have not turned out to be very powerful, and it is not as easy to invent methods for programs to learn new information as had been hoped. Nevertheless, the performance of programs which have had information added "by hand" has been impressive. For example, there are programs which converse in English, and answer questions concerned with a suitable nontrivial data-base. The question of what could be done by a program possessing the same knowledge of a subject as the best human problem solvers has been raised. In the past few years the techniques and languages for writing such programs have been greatly improved.

The related question of computer aided instruction involves bringing people into a controlled environment which "understands" what they are thinking about. To do this, the environment must know something of value which the person wishes to learn, so that it can interact with him in depth. The MACSYMA system already contains features, and has solved problems, which have proved of interest to applied mathematicians (a group of skilled problem solvers) and has experienced continually increased demand as more features have been added. The MACSYMA system offers a concrete example for study of man-machine interactive problem solving.

Another problem area is the management of large programs and the automation of the programming process. Experience in the design of systems of the hierarchical nature of algebraic manipulation programs has lead to several conclusions discussed earlier. Problems of intelligent information retrieval and management of large complex data bases also arise in this area.

Finally, the careful analysis of mathematical algorithms, which has been an important input to our design effort, constitutes new data for those interested in building a useful theory of computation.

II - An Overview of the Current MACSYMA System

The principal modules of the current MACSYMA system are shown in Figure 1. Modules are shown linked to those other modules on which they depend heavily. Those modules still under development are shown as squares.

The initial development goal of MACSYMA was to combine the results of the doctoral research of Martin [6] and Moses [9] with the ideas in Engelman's MATHLAB [1]. This suggested an interactive system written in LISP which emphasized step-by-step problem solving. We also wanted to add new algorithms for handling arbitrary precision arithmetic, polynomials, rational functions, and integration. Except for the integration algorithm, which does not include all the capabilities we know how to implement, this initial goal has largely been met, and the system has reached new levels of performance in the following areas:

a) Obtaining limits and some cases of infinite integrals
b) Solving equations
c) Canonical simplification
d) User-level pattern matching capabilities
e) User specification of transformations on expressions
f) Programming and bookkeeping assistance.

The ideas used for simplification [8], integration [7], display [5], limits [11] and pattern matching [2] have been presented in detail in separate papers. In this paper we will give an overview of MACSYMA by demonstrating the facilities for user specification of transformations, algorithms, and simplifications. We will then describe in some detail the routines for handling rational functions and the solution of equations.

We expect that one use of MACSYMA will be the symbolic recasting of a problem as a prelude to numerical solution. Appendix II gives a specific example. It outlines the steps required to transform some differential equations describing a boundary layer so that they are expressed in terms of a more interesting set of independent and dependent variables. The recasting involves a knowledge of the magnitudes of certain quantities and how fast they may vary. We use this information to apply truncated series expansions (or neglect small terms altogether). We also apply various side relations.

Thwaites [10] suggests that the basic concern of applied mathematics is the approximate solution of canonical equations such as the ones we will be dealing with. The approximations can be motivated by either mathematical or physical considerations. If a machine is to automatically apply approximations based on physical considerations, it must know the relevant physics as well as the mathematics. It seems unlikely that such problems will be solved in the near future without the step-by-step guidance of someone familiar with the relevant physical interpretations. In order to work with the computer, the mathematician must be explicit and precise. The challenge to systems designers is to reduce the burden which the system places on its users.

1. Step-by-Step Problem Solutions - Two Examples

Let us preview some of the aspects of problem solving which must be considered in designing an interactive language.

First, an error in a command can be lexical (e.g. spelling), syntactic or semantic. Lexical and syntactic errors are often most easily corrected by operations on character strings, while semantic errors can often be corrected using mathematically meaningful operations.

Second, the user must specify precisely when and how much evaluation of operations is to be done. He must also be careful about the scope of the assignment of specific values to variables or operators.

Third, a user may extend the system language in several ways:
a) By renaming concepts already known to the system, or specifying which of a set of alternatives is to apply
b) By modifying and extending known concepts such as supplying the derivatives of certain functions
or c) By defining his own concepts.
Each of these operations presents special problems.

Finally, the user and the machine must be able to discuss the problem solving process itself. For example, in MACSYMA, each command and result is automatically given a name. The user can supply his own names by use of the ":" assignment. Furthermore, the most recent expression can always be referred to as "%".

To illustrate the difficulties of solving a problem interactively, let us carry out a few steps of the calculation outlined in Appendix II. First we input the physical equations.

MACSYMA types the label (C1). The user then types a command in the MACSYMA input language:

```
(C1) RHO*(U*DERIVATIVE(U,X)+V*DERIVATIVE(U,Y))
=DERIVATIVE(MU*DERIVATIVE(U,Y),Y)-DERIVATIV
(P,X)+J*B)@

INPUT ERROR AT THE POINT MARKED BY ******

RHO*(U*DERIVATIVE(U,X)+V*DERIVATIVE(U,Y))=DERI
VATIVE(MU*DERIVATIVE(U,Y),Y)-DERIVATIV(P,X)+J*
B******)

PLEASE REPHRASE OR EDIT
```

The command is terminated by typing an "@" which indicates that the command is to be read, evaluated, and the result displayed. Terminating the command with a "$" instead of the "@" suppresses the display of the result.

The command is incorrect because it has an extra right parenthesis at the end. It is not necessary to retype the expression, because the line may be edited by a string editor at the top level of MACSYMA. The system retypes the line label, and we respond with "#", which loads the previous (erroneous) string into a buffer and displays the contents of the buffer. The character "_" acts as a cursor which marks the place currently being examined by the editor. We want to delete the last ")", so we use a command to search for J*B, and delete the character following the cursor. Typing an extra "#" exits from the editor, and "@" then executes the new command string. The rest of the computer conversation looks like:

```
(C1) #
_RHO*(U*DERIVATIVE(U,X)+V*DERIVATIVE(U,Y))=DER
IVATIVE(MU*DERIVATIVE(U,Y),Y)-DERIVATIV(P,X)+J
*B)
SJ*B#
RHO*(U*DERIVATIVE(U,X)+V*DERIVATIVE(U,Y))=DERI
VATIVE(MU*DERIVATIVE(U,Y),Y)-DERIVATIV(P,X)+J*
B_)
D
RHO*(U*DERIVATIVE(U,X)+V*DERIVATIVE(U,Y))=DERI
VATIVE(MU*DERIVATIVE(U,Y),Y)-DERIVATIV(P,X)+J*
```

```
B_
#
RHO*(U*DERIVATIVE(U,X)+V*DERIVATIVE(U,Y))=DERI
VATIVE(MU*DERIVATIVE(U,Y),Y)-DERIVATIV(P,X)+J*
B
@
```

```
           DU         DU     D     DU
(D1) RHO ((--) V + U (--)) = --MU (--)
           DY         DX     DY    DY

                       - DERIVATIV(P,X) + B J
```

The command has been read in and evaluated as a mathematical expression by the MACSYMA evaluator. The evaluator lets any symbol which has not been assigned a value previously, stand for itself. Operations such as integration, differentiation, and exponentiation are not carried out automatically, but they can be evaluated in several ways. For example, if we had typed

```
       DIFF(MU*DERIVATIVE(U,Y),Y)
```

instead of

```
       DERIVATIVE(MU*DERIVATIVE(U,Y),Y)
```

we would have obtained

```
                         2
                        D U
                    MU -----
                          2
                        DY
```

since MU has not been declared dependent on X. Returning to expression D1, we see that DERIVATIVE has been spelled incorrectly. There are two ways we can fix this. One is to type the command STRING(D1)$, which will convert D1 into a character string in the input buffer; we can then edit it as before. The second way is to use the SUBSTITUTE command:

```
(C2) SUBSTITUTE(DERIVATIVE,DERIVATIV,%)@
```

```
            DU         DU     D     DU      DP
(D2)  RHO ((--) V + U (--)) = --MU (--) - (--)
            DY         DX     DY    DY      DX

                                        + B J
```

Note that a product is indicated by a "*" on input, but by a space on output. The user can call for *'s on output by typing NOSTAR(FALSE)$.

We have now succeeded in inputting the first of our two equations. In order to reduced the chance of more typing errors, we will tell the input routine that by D we mean DERIVATIVE. That is, we give DERIVATIVE the alias D. using this alias, we input the second equation on line C4.

```
(C3) ALIAS(D,DERIVATIVE)$
```

```
(C4) RHO*C[P]*(U*D(T,X)+V*D(T,Y))=
D(K*D(T,Y),Y)+MU*(D(U,Y))**2+U*D(P,X)
+J**2/SIGMA@
```

```
              DT       DT        D    DT
(D4) C  RHO ((--) U + (--) V) = --K (--)
      P       DX       DY        DY   DY
                            2              2
                         DU       DP      J
                   + MU (--)  + (--) U + -----
                         DY       DX     SIGMA
```

The next step in the calculation, deriving the free stream equations, consists of setting the derivatives with respect to Y equal to zero. We can define an operator which will do just that by using the pattern matching subsystem.

```
(C5) DECLARE(AA,TRUE)$
```

```
(C6) DEFRULE(DYZERO,D(AA,Y),0)$
```

We first declare that AA should match any expression. We then define the rule DYZERO to match any subexpression which is a derivative with respect to Y, and replace it with zero. We now apply this rule to our equations.

```
(C7) APPLY1(D2,DYZERO)@
                      DU             DP
(D7)           RHO U (--) = B J - (--)
                      DX             DX
```

```
(C8) APPLY1(D4,DYZERO)@
                               2
                     DT       J       DP
(D8)         C  RHO (--) U = ----- + (--) U
              P      DX      SIGMA    DX
```

Equations D7 and D8 are satisfied by the free stream values of RHO, SIGMA, U, and T. Let us call these values RHO[INF], SIGMA[INF], U[INF], and T[INF]. To avoid confusion, we should substitute the latter values for the former in D7 and D8. There are several ways we can do this; one of them is the SUBSTITUTE command illustrated earlier. To make a point about the interaction between variables and the environment, we will use the EV or "evaluate" command to perform an evaluation in a local environment. If we wished to make a global substitution, we would type the commands RHO:RHO[INF]$, SIGMA:SIGMA[INF]$, U:U[INF]$, and T:T[INF]$. Then by evaluating D7 and D8, we can cause the new values of RHO (etc.) to be substituted in. The disadvantage of this method is that we have set values in the top level environment which will remain after the time we have used them. We could remove the values by executing ATOM(RHO):TRUE$ (etc.), which would return RHO (etc.) to the status of standing for itself; but, since we will be working further with both RHO and RHO[INF] this could lead to confusion. Let us therefore evaluate D7 and D8 in a local environment:

```
(C9) EV(D7,RHO=RHO[INF],U=U[INF])@

                       D                DP
(D9)      RHO    U     (--U   ) = B J - (--)
             INF  INF  DX INF            DX
```

```
(C10) EV(D8,RHO=RHO[INF],U=U[INF],SIGMA
=SIGMA[INF],T=T[INF])@
                                        2
                          D            J
(D10)   RHO     U     C    (--T   ) = --------
           INF   INF   P   DX INF     SIGMA
                                           INF

                                           DP
                                   + U    (--)
                                      INF  DX
```

The next step is to substitute the left side of D10 for the last two terms of D2. We can talk about the left side of D2 by using the PART command. PART(exp,n1,...,nk) obtains a subexpression of exp which is specified through the indices (ni). The index n1 selects the n1th argument of the top level operator; the index n2 picks up the n2th argument of the result of PART(exp,n1). Thus PART(D2,1) is the left side of D2, while PART(D2,2) is the right side. In order to use PART on, for example, a quotient, we must know that the numerator is considered the first argument. The function DPART will display the expression with the specified subexpression enclosed in a box. This gives the user a simple method of verifying his specification. To check our specification of the last two terms of D2 we type:

```
(C11) DPART(D2,2,(2,3))@
               DU          DU
(D11)    RHO ((--)  V + U (--))  =
               DY          DX

                               __________________
                 D     DU     !       DP         !
                 --MU (--) +  ! -    (--) + B J  !
                 DY    DY     !_______DX_________!


(C12) SUBSTPART(PART(D10,1),D2,2,(2,3))@

             DU          DU      D     DU
(D12) RHO  ((--)  V + U (--))  = --MU (--)
             DY          DX      DY    DY

                                      .D
                       + RHO     U    (--U     )
                            INF   INF  DX INF
```

We leave the problem at this stage, because we have illustrated our points; further steps in the solution can be found in [6].

By now the reader has probably observed that these operations might be carried out more easily by hand. This is true for these first few steps, but as the calculation progresses, the size of the expressions grow, (so that the hand solution of the rest of the problem required three months) while the commands remain at about the same level of complexity. We feel that the commands shown above are actually quite easy to use, provided that the user has a thorough knowledge of the MACSYMA language. Since it may not be possible for a mathematician to work closely with a system without knowledge of a suitable language, our goal has been to make that language natural, expressive, and powerful.

Some less obvious aspects of MACSYMA should also be noted. To avoid filling fast core memory with previous results, the system automatically moves old data to secondary storage. The user can control the rate of transfer of expressions, and can, if he wishes, move all but the current command string out of core. At the end of a session with MACSYMA, the user may want to create a file containing the main steps in a calculation which has been performed. MACSYMA gives him facilities to do this. Furthermore, the data can be translated into "human-readable" form and displayed on high-speed line printers with a larger line width. Many parts of this text were produced directly by MACSYMA, and altered only slightly for the sake of the text justification program used for printing this paper.

Our next example is the solution of a classical problem in algebraic manipulation, the so-called F and G series problem of dynamical astronomy. The series Fi and Gi are given by recursion relations and initial values. The results are polynomials in EPSILON, MU, and SIGMA.

In MACSYMA, subscripted variables and functions are represented by arrays. The arrays need not have their dimensions declared, and may have the value of their entries defined by some function. The function will be evaluated whenever the value of a specific entry is called for, and that value has not been found previously. At any time the value of an entry is specified by the user or found by evaluating the function, it is saved. Thus to specify the F and G computation we use lines C1 through C5 to specify the derivatives and initial values, and C6 and C7 to define the recursion relations. Line D8 prints out one of the values.

```
(C1) D(MU,T):-3*MU*SIGMA$

(C2) D(SIGMA,T):EPSILON-2*SIGMA**2$

(C3) D(EPSILON,T):-SIGMA*(MU+2*EPSILON)$

(C4) F[0]:1$

(C5) G[0]:0$

(C6) F[I]:=EXPAND(-MU*G[I-1]+DIFF(F[I-1],T))$

(C7) G[I]:=EXPAND(F[I-1]+DIFF(G[I-1],T))$

(C8) F[5]@
                 3        2
(D8) 105 MU SIGMA  - 15 MU  SIGMA

                           - 45 EPSILON MU SIGMA
```

Rather than continue this informal discussion to include all of the currently implemented and proposed commands and facilities, we have relegated most of these details to a brief treatment in Appendix I. Detailed discussions of many commands can be found in the other MACSYMA papers in these proceedings. We will, however, touch upon a few commands to make some points about our philosophy, and then proceed to discuss in detail a major set of facilities not treated elsewhere.

2. Extension of facilities

One of the most difficult problems of designing algebraic manipulation systems is allowing for new knowledge to be added to old. Several efforts along these lines which demonstrate our approaches deserve some attention.

The TELLSIMP [2] facility allows the user to define new simplification rules which will be applied by the built in simplification routine. It is expected that most of these new rules will specify the simplification of expressions containing functions previously unknown to MACSYMA. Thus the MACSYMA environment can be significantly altered in response to new problem areas.

In a similar vein, the assignment operation ":" checks to see if its left operand is an operator or a function. For example, if it is a derivative, it is assumed that a value is being assigned for future use by the differentiator. Thus derivatives of variables or arbitrary functions may be defined in a simple manner. We have already seen this used in the F and G series calculation example, where derivatives of EPSILON, MU, and SIGMA were defined.

As was pointed out in the Introduction, MACSYMA seeks to make available most basic algorithms implemented in the most efficient manner possible. Many of the modular arithmetic algorithms described in this proceedings have been or are currently being implemented in MACSYMA, some to replace their traditional, inefficient predecessors. Among these Berlekamp's polynomial factorization algorithm, and the modular greatest common divisor algorithm are foremost.

New developments and extensions in the host language for MACSYMA, LISP, will further influence our product. A new LISP compiler which approaches in the quality of its generated code an optimizing FORTRAN compiler will allow us to prove to the legions of LISP nay-sayers that, properly done, LISP is an entirely appropriate programming formalism and system for this type of work.

III - Rational Function Commands

This section concerns one of the critical design decision in MACSYMA which we believe contribute greatly to its usefulness. This decision was that algorithms should have special data types when it is necessary for their proper operation. The rational function package embodies the essentials of a special data type, which, by suitable treatment, has yielded a number of new results. Because of their significance in the design and philosophy of MACSYMA, and in their practical implications, the rational function commands are treated in greater detail here.

Moses [8] distinguishes between the "radical" approach to algebraic manipulation, and the "conservative" approach. According to this classification, a radical system will transform a user-supplied expression into an internal format which consists of an encoding of the expression in a special unique simplified form. This transformation generally destroys superficial resemblances between the input and output. The only attribute necessarily preserved is the functional value of the expression. Polynomial and rational function systems generally fall in the "radical" category. The contrasting "conservative" approach does almost nothing but that which is specified by the user; it keeps the internal form as nearly the same as the external form as is possible, and generally accepts a wide variety of expressions (wider than polynomials and rational functions).

The top-level (i.e. "liberal" in Moses' terminology) "general" simplifier in MACSYMA takes a stance in the middle, yet allows certain subsystems to explore the far reaches of the "political" spectrum. Because of the conjunction of different approaches, radical simplification algorithms can be applied to expressions which would not ordinarily be considered proper inputs. For example, the ability to manipulate e**(2*x) + 2*e**x + 1 as a quadratic in e**x (and apply polynomial "radical" processing) is quite useful, even though the expression is not quite fair game for ordinary polynomial systems. MACSYMA is capable of factoring the above expression into (e**x+1)**2, and treating it as a polynomial in general; however, it is also capable of noticing that e**x can reduce to y when x=log(y). Polynomial or rational function systems are rarely aware of such possibilities in their data.

1. Data types and conventional rational simplification. These sections discuss the "radical" data handling facilities of MACSYMA, and their relation to the MACSYMA command level. In one particular instance (the SOLVE command) we show how radical and conservative handling of different parts of the same expression can lead to an end result which could not be produced with either approach alone. Other commands where rational simplification or other radical approaches are essential to programming effective algorithms are also discussed.

In order to clarify the discussion, it is necessary to distinguish between the two major internal forms for expressions in MACSYMA. Ordinary MACSYMA form is a variant of the Polish prefix form which is typical of many list-processing implementations of algebraic manipulation systems. For example, 3*x**2 would be represented (glossing over inessential details) as (times 3 (expt x 2)), and x+y as (plus x y). By contrast, the canonical rational expression (CRE) form in MACSYMA is an internal form especially suitable for rapid manipulation of sparse polynomials and rational functions. In CRE form, 3*x**2 is represented, (again, glossing over details) as (x 2 3). The first element of the list is the variable, the second is its highest exponent, and the third, the coefficient of the just preceeding exponent. Thus 6*x**2+4 is represented as (x 2 6 0 4), and, allowing coefficients themselves to be polynomials, x**2*y +7*x*z is (x 2 (y 1 1) 1 (z 1 7)). Since (y 1 (x 2 1) 0 (x 1 (z 1 7))) is an equivalent CRE representation, it should be clear that the ordering of variables must be specified to insure that equivalent CRE's are identical, that is, they are in canonical form.

CRE's in general represent rational expressions, that is, ratios of polynomials, where the numerator and denominator have no common factors, and the denominator is positive. Thus a CRE has three essential parts: a variable list (VARLIST), specifying the ordering of the variables, and two polynomial parts. With these preliminaries, we can describe the actions of the rational function commands.

RATVARS(a,b,...) orders the variables listed in its argument list on a global variable list (VARLIST) so that the rightmost element of the list a,b,... will be the main variable of future rational expressions in which it occurs, and the other variables will follow in sequence. If a variable is missing from the RATVARS list, it will be given lower priority than the leftmost element. If several variables are missing, they will be ordered by the MACSYMA function GREAT, which uses an implementation of the ordering algorithm described in [8]. The arguments to RATVARS can be either variables or non-rational functions (e.g. SIN(X)).

RATSIMP(EXP) rationally simplifies the expression EXP. That is, EXP is converted into a single fraction, whose numerator and denominator are polynomials over the integers, with no common factors. EXP is written in a recursive form: a polynomial in the main variable whose coefficients are polynomials in the next-higher-order variable, ..., whose coefficients are integers. This is accomplished by converting EXP into CRE, and then converting back to ordinary MACSYMA form for display.

For example:

```
(C1) (X**2-Y**2)*(Z**2+2*Z)/((X+Y)*W)@

                 2    2     2
               (X  - Y ) (Z  + 2 Z)
(D1)           --------------------
                    W (Y + X)

(C2) RATSIMP(D1)@
                   2
          (X - Y) Z  + (2 X - 2 Y) Z
(D2)      --------------------------
                      W

(C3) RATVARS(X)$

(C4) RATSIMP(D1)@
             2              2
          X (Z  + 2 Z) - Y Z  - 2 Y Z
(D4)      ---------------------------
                      W
```

FACTOR(EXP) factors the expression EXP into factors irreducible over the integers. If EXP is a rational expression (with a denominator not 1) both numerator and denominator are factored. If FACTORFLAG is set to TRUE, the integer multiplier, if any, is factored also. The algorithm can be used to factor polynomials in any number of variables; however, factorization with respect to some of the variables can be avoided by setting the global variable DONTFACTOR to a list of such variables.

For example,

```
(C5) FACTOR(X**6+1)@

                 2         4    2
(D5)           (X  + 1) (X  - X  + 1)
```

SQFR(EXP) is similar to FACTOR except that the polynomial factors are "square-free" that is, have no multiple roots. This algorithm, which is also used by the first stage of FACTOR, utilizes the fact that a polynomial has in common with its nth derivative all its factors of degree > n. Thus by taking derivatives with respect to each variable in the polynomial, all factors of degree > 1 can be found.

PARTFRAC(EXP,VAR) expands the expression EXP in partial fractions with respect to the main variable, VAR. The algorithm employed is based on the fact that the denominators of the partial fraction expansion (the factors of the original denominator) are relatively prime. The numerators can be written as linear combinations of denominators, and the expansion falls out.

```
(C6) PARTFRAC(X/(X**2-1),X)@

                1         1
(D6)         ------- + -------
             2 X - 2   2 X + 2
```

2. Contagious CRE Commands. The above commands represent no new capabilities; MATHLAB [5] has almost identical facilities, although its internal equivalent of our CRE's is less efficient for sparse polynomials. Other systems, by limiting their universe of discourse to canonical representations, make these commands unnecessary.

The commands in this and the following sub-sections represent significant departures from the usual use of rational function routines.

RAT(EXP) is indistinguishable on command level from RATSIMP; however, RAT leaves its internal result in rational function (CRE) form, so that operations used by the rational function commands described here can be more rapidly performed on it. Furthermore, any time the user adds to or multiplies by a CRE, the result is a CRE. That is, the CRE form is "contagious." This enables a user to easily force his entire calculation to be done in CRE form by converting one of his inputs into CRE by simply multiplying by RAT(1). Some problems require excessive amounts of storage and/or time if intermediate results are converted back into prefix form at each step of the calculation. The RAT facility, by being integrated into the simplifier, permits a user to compose a program and try it out (without any changes) on ordinary prefix form arguments or on CRE arguments.

RATDISREP(EXP), which appears to do nothing on the command level, changes its argument from rational function form (CRE) to ordinary MACSYMA form. This is sometimes necessary in order to use some of the other MACSYMA commands. If RATDISREP is not given a CRE for an argument, it does nothing.

3. The Rational Coefficent Program. RATCOEF(EXP,PART) returns the coefficient, C, of the expression PART in the expression EXP. C will be free (except possibly in a non-rational sense) of the variables in PART. If no coefficient of this type exists, zero will be returned. RATCOEF will give reasonable answers to reasonable requests, and will often produce reasonable answers to poorly stated requests. Generally, when PART includes a "+" or a "/", results may seem odd. (see lines D7, D8, D10, and D11 in the examples to follow). Since EXP is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned. The effect of RATCOEF should be clarified by the following examples.

```
(C1) S:A*B*X**2+B*X+2*X+5@
                    2
(D1)          A B X  + B X + 2 X + 5

(C2) RATCOEF(S,X)

(D2)                B + 2

(C3) RATCOEF(S,A*B)@
                          2
(D3)                     X

(C4) RATCOEF(S,B)@
                        2
(D4)                 A X  + X

(C5) RATCOEF(S,2*X)@
```

```
                   B + 2
(D5)               -----
                     2

(C6) RATCOEF(S,B/2)@
                         2
(D6)              2 A X  + 2 X

(C7) RATCOEF(A*X+B*X+C,A+B)@
(D7)                     X

(C8) RATCOEF(3*A+2*B,A+B)@
(D8)                     2

(C9) RATCOEF(S,-A)@
                           2
(D9)                  - B X

(C10) RATCOEF( (A*B+C)/D,B/D)@
(D10)                    A

(C11) RATCOEF(3*A/D+A/D**2, A/D**2)@
(D11)                    0
```

Let us first define RATCOEF(EXP,PART) where EXP is a polynomial and PART has the form v**k for v a variable, k a number. This case is clear: we expand EXP as a CRE, and pick off the coefficent of v**k. If there is no occurrence of v**k, the coefficent is 0. If EXP is not a polynomial, but a ratio of polynomials, then we must make a decision about how to treat occurrences of v in the denominator.

Let EXP =num/denom, where num = Σai*v**i. If the coefficient of v**k, namely ak, is zero or if ak/denom depends on any variable in the original PART, then the response is zero. Otherwise the response is ak/denom.

RATCOEF of a product can be defined recursively as follows. Consider RATCOEF(EXP,PART). If PART = v1**n1*v2**n2*...*vk**nk, then RATCOEF(EXP,PART) = RATCOEF(RATCOEF(EXP,vk**nk),v1**n1*...*v(k-1) **n(k-1)).

If PART = A/B then RATCOEF(EXP,PART) = RATCOEF(EXP*B,A).

If PART = -A, RATCOEF(EXP,PART) = RATCOEF(-EXP, A).

If PART = ΣAi**i (possibly after removing multipliers, as above), then EXP is divided by PART with respect to the main variable in PART. If the quotient depends on any variable in the original PART, the response is zero. Otherwise the answer is the quotient.

The coefficient produced in this manner may depend, in the last case, on the ordering of the variables within EXP. For example, the coefficient of (Y+Z)*X in Z**2*X**2+(Y+Z)*X+A is clearly 1. The similar problem of finding the coefficient of X*Z+X*Y in X**2*Z**2+X*Z+X*Y+A yields the answer 0, since X**2*Z**2+X*Y+A divided by X*Z+X*Y is X*Z+1, with remainder -X**2*Y*Z+A. The quotient depends on X, and thus the coefficient is taken to be zero.

This illustrates both the ability of the user to ask for coefficients of sums, and the ability of RATCOEF to sometimes answer correctly. We could have defined RATCOEF only for products, but it seems more in keeping with the spirit of an interactive system to avoid such restrictions on the user. Note that if the user were disappointed with the answer 0 to the above request, first executing RATVARS(X) would correct the situation.

In summary, RATCOEF will find the coefficient of PART when PART is a factor of the expression, or of some part of the expression such that the other factor has none of the same variables. RATCOEF cannot be used to pick out the coefficient of a number.

The returned value is in CRE form.

An alternative to RATCOEF is available in situations where its flexibility is not needed. The COEFF command can operate on CRE forms or on ordinary MACSYMA forms which have been expanded. COEFF(EXP,VAR,POWER) will extract the coefficient of VAR**POWER (where POWER may be 0) from EXP. COEFF returns a CRE form if and only if it is given a CRE form.

4. Extensions to Rational Simplification.

FULLRATSIMP(EXP) is an expanded version of RATSIMP which is recursive on the arguments of non-rational functions. It also removes zero exponents, and converts forms like (x**y)**z to x**(y*z). Although these last two operations are generally performed by the simplification program, FULLRATSIMP must repeatedly simplify the results of such transformations until no more rational simplifications can be made. FULLRATSIMP is no more time-consuming than RATSIMP if EXP is an algebraic expression with no non-rational functions.

Since any equation has a non-rational function, namely "=", in it, FULLRATSIMP, rather than RATSIMP should be used on equations.

A more extensive expansion of the concept of global simplification is embodied in RADCAN. While FULLRATSIMP does not apply any identities concerning logs, radicals, and non-numeric exponents, RADCAN does.

RADCAN(EXP) converts the expression EXP into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, RADCAN produces a normal form; that is, all forms equivalent to zero are mapped into zero. For purely rational expressions, RADCAN is no more time-consuming than RATSIMP or FULLRATSIMP; however, for more general expressions including radicals, logs, and non-integer exponents, RADCAN can be quite expensive. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents.

A description of the method, and proofs of the canonical properties of the RADCAN algorithm are discussed in [6]. Examples should give a rough feel for the capabilities of RADCAN (% always refers to the just-previously displayed expression, %E is the base of the natural logarithms):

```
(C1) SQRT(98)@
(D1)                SQRT(98)

(C2) RADCAN(%)@
(D2)                7 SQRT(2)
```

```
(C3) (SQRT(X**2-1))/(SQRT(X-1))@
                         2
                 SQRT(X   - 1)
(D3)             ------------
                 SQRT(X - 1)

(C4) RADCAN(%)@
(D4)               SQRT(X + 1)

(C5) (LOG(A**(2*X)+2*A**X+1))/(LOG(A**X+1))@
                     2 X      X
              LOG(A    + 2 A  + 1)
(D5)          --------------------
                      X
                 LOG(A  + 1)

(C6) RADCAN(%)@
(D6)                  2

(C7) (%E**X-1)/(%E**(X/2)+1)@
                     X
                  %E  - 1
(D7)              ---------
                     X/2
                  %E    + 1

(C8) RADCAN(%)@
                     X/2
(D8)              %E    - 1
```

5. The RATSUBST (rational substitution) Commands. RATSUBSTn(A,B,C) where n = 1, 2, 3, 4 is a set of similar commands to substitute A for each occurrence of B in the expression C. In those cases where it is clear where B occurs, the result will correspond to the intuitive notion of substitution.

If B is an atom, occurrences of B are obvious. The action taken is simply substitution followed by simplification.

If B is a quotient, say b1/b2, then RATSUBSTn(A,B,C) is entirely equivalent to RATSUBSTn(A*b2,b1,C).

If B is a product, all coefficients of powers of B can be detected in C by a technique similar to that used by RATCOEF. (Hearn [4] suggests this approach) If B is a sum, we must define what we mean by an occurrence of an expression B in a polynomial expression C. (If C is not a polynomial, we can consider its numerator and denominator separately.)

If C = $\sum$Si*B**i, then B is said to occur in C with coefficient S1 and exponent 1, coefficient S2 and exponent 2, ..., and remainder S0. If B occurs in such a fashion we wish to replace C by $\sum$Si*A**i. Unfortunately, finite power series expansions for an expression in terms of a non-atomic subexpression are not unique. For example, C = x**2+3*x*y+y**2 has (among others) the following expansions in (x+y):

1. 1*(x+y)**2 + 0*(x+y)**1 + x*y*(x+y)**0

2. 1*(x+y)**2 + x*(x+y)**1 - x**2*(x+y)**0

3. 1*(x+y)**2 + y*(x+y)**1 - y**2*(x+y)**0

What is needed is a set of restrictions on the coefficients Si so that the expansion is unique and appropriate to the problem at hand. This is the basic problem in substitution for simplification, and this solution is based on a set of heuristics for achieving what appear to be, in some instances, more desirable results than have been possible in the past. We will separate out only the highest power of B, and discuss at each stage (recursively on lower powers of B) the situation C = S*B**n + r, where r contains the lower order terms.

As we have pointed out earlier in our discussion of RATCOEF, the ordering of variables is sometimes quite critical. "Sum"-hood, which is a property of a form, not of a function, sometimes depends on ordering. For example, x*z+y*x is a sum, but (z+y)*x is (for purposes of RATSUBST) not a sum, but a product, although the two expressions are functionally equivalent.

Let B be a polynomial containing variables v1,v2,...,vn, where the highest power of each vi is mi. For all but condition 2 below, the only restriction on r, the remainder consisting of lower order terms, is that it has lower degree than C does in some particular variable (namely, the most important on the varlist that is also in B). The conditions below are embodied in the commands RATSUBST1,2,3, and 4, respectively. Their effects can best be gauged by frequent reference to the examples in figure 2.

1. The highest power of some vi in S that appears in B is less than the corresponding mi.
2. The highest power of each vi in S that appears in B is less than the corresponding mi, and the highest power of each vi in r that appears in B is less than the corresponding mi**n.
3. S is a polynomial
4. S contains no sum.

The value of n ranges from the highest possible (the ratio of the highest coefficient of some vi in C which is also present in B, to the corresponding maximum coefficient of that vi in B, namely mi) to the lowest possible (when some vi in B is no longer present in C to a power as high as it is in B, or 1.). To avoid the possibility of looping, occurrences of B in C are replaced, as found, by a special dummy variable, which is subsequently replaced by A. Cases in which B occurs in A (probably an error on the user's part) or where simplification of C results in new occurrences of B can be treated with repeated calls to RATSUBST. This can be easily programmed in MACSYMA.

If C contains non-rational functions, substitution proceeds on the arguments of the non-rational functions, recursively. Thus A, B, and C need not be rational expressions.

By noting when B has non-rational components (e. g., e**x, or x**(1/2)), RADCAN can be called on B and C, and they can be left in a special expanded format, which tends to reflect more clearly the similarities of the two expressions. Thus RATSUBST(A,E**X,E**(2*X)) is A**2.

An example of an extension to the RATSUBST framework might serve to illustrate its generality. If there is a canonical ordering on all expressions submitted to RATSUBST, and on all intermediate expressions, then a RATSUBST5 could be programmed with the following condition:

5. S*A**n + r has a lower canonical order

("is simpler") than S*B**n + r.

By using the RATSUBST commands selectively, such substitutions as sin(x)**2 + cos(x)**2 ⇒ 1 can be performed more nearly in the sense in which they are intended. If one RATSUBST command does not do the job, perhaps another will.

6. The SOLVE Program. The SOLVE command in MACSYMA uses several techniques for solving for a given variable in an equation. Each of these techniques is open to extension in a straightforward manner. The roots and their multiplicities are available to other programs, and are used as building blocks for more complicated facilities, such as contour integration.

The format of the SOLVE command is:

SOLVE(equation, variable)@

where the equation may also be an expression (which is assumed to be set equal to zero).

SOLVE(E,X) puts its first argument E, in radical canonical form, and attempts to factor it with respect to the variable X, and all non-rational functions in E containing X. Each factor is examined for being linear, quadratic, cubic, or biquadratic with respect to X and the non-rational functions containing it. If the factor is of degree five or more, then it is considered unsolvable. Such unsolved factors and their multiplicities are put on a list which is returned along with the roots.

Linear terms of the form F(X)-C are examined to see if C, the constant term, is actually free of elements containing X; if so, USOLVE is called. Otherwise the term is added to the list of unsolved factors. USOLVE knows the inverses of SIN, COS, ARCSIN, ARCCOS, TAN, ARCTAN, LOG, and powers of e. It could be extended to other functions. Once the inverse has been applied, a new equation results. It may be of the form X = FINVERSE(C), in which case the term has been solved, or it may be of the form G(X) = FINVERSE(C), in which case SOLVE is called again. This recursive algorithm allows for solution of, for example, SIN(COS(X)) = 0 for X.

The quadratic (cubic, biquadratic) formula is applied to quadratic (etc.) factors, and the same sort of recursive treatment as described above is used if the equation is, for example, quadratic in SIN(X) instead of X.

The simplification done by the quadratic (etc.) routines is of some interest, in that the roots in the formulae are simplified by a special program (SIMPNRT) which takes out perfect n*k powers of a kth root. (i.e. even powers in a square root, multiples-of-three powers in a cube root, etc.) Thus SQRT(8) is simplified to 2*SQRT(2). SIMPNRT calculates a square-free factorization of the radicand, and takes appropriate multiple factors, if any, outside the radical.

The following examples illustrate the capabilities of SOLVE:

```
(C1) SOLVE(Y**(2*X)-3*Y**X+2=0,X)@
SOLUTION


(E1)                      X = 0

                              LOG(2)
(E2)                      X = ------
                              LOG(Y)

(D2)                         (E1,E2)


(C3) A:X**2-12*X+3@
                          2
(D3)                     X  - 12 X + 3

(C4) SOLVE(SIN(A)**2-5*SIN(A)+3,X)@

SOLUTION
                               5   SQRT(13)
(E4)   X = 6 - SQRT(ARCSIN(- - --------) + 33)
                               2      2

                           5   SQRT(13)
(E5)   X = SQRT(ARCSIN(- - --------) + 33) + 6
                           2      2

                           SQRT(13)   5
(E6)   X = 6 - SQRT(ARCSIN(-------- + -) + 33)
                              2       2

                       SQRT(13)   5
(E7)   X = SQRT(ARCSIN(-------- + -) + 33) + 6
                          2       2


(D7)              (E4,E5,E6,E7)

(C8) SOLVE(ARCSIN(COS(3*X))*(F(X)-1),X)@

SOLUTION

                          ARCCOS(0)
(E8)                  X = ---------
                              3
THE ROOTS OF

(E9)                      F(X) = 1

(D9)                      (E8,E9)

(C10) SOLVE(5**X=125,X)@

(D10)                       X=3
```

Note that SOLVE has taken advantage of radical approaches but is still able to step back and treat fairly general expressions. In order to use the "radical" polynomial factoring program, it uses RADCAN to expand unlikely-looking expressions into polynomials. Thus the expression Y**(2*X)-3*Y**X+2 in C1 is expanded into a polynomial in Z, where Z=Y**X (actually Z=e**(X*log(Y))), which is then factored into (Z-1)*(Z-2). By setting each of these factors equal to zero, the following sequence of steps is followed:

1. e**(X*log(Y))-1 = 0 is converted by USOLVE to X*log(Y) = log(1)

2. The simplifier changes this to X*log(Y) = 0.

3. SOLVE is called recursively, and factors X*log(Y):

a. SOLVE throws out the log(Y) factor since it does not depend on X, and

b. the factor "X" is recognized as a linear expression of the form a*X+b where a=1 and b=0, which has solution X=-a/b, or in this case, X=0.

The other root is handled in an analogous fashion.

Acknowledgements

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency (ARPA), Department of Defense, under Office of Naval Research Contract N00014-70-A-0362-0002. One of the authors (Fateman) has been supported by Harvard University, sponsored by ARPA under Air Force contract F19628-68-0101, and by the National Science Foundation under their Graduate Traineeship program.

The Project MAC Artificial Intelligence Group time-sharing system was of immeasurable assistance in preparing and debugging the programs, and incidentally, in preparing this paper.

Participating in the original design work for MACSYMA (beginning in July, 1968) were W. A. Martin, C. Engelman, and J. Moses.

Programming began in July, 1969. The expression evaluator and input-output (i.e. string editor, parser, 2-D display, language) were programmed by W. A. Martin, P. Loewe, and T. Williams.

Of the other major modules in MACSYMA, W. A. Martin designed and programmed the polynomial arithmetic package; R. Fateman designed and programmed the rational function package and its extensions (including the radical simplifier); J. Moses designed and programed the simplifier (a major overhaul of the Korsvold program), many of the commands (e.g. differentiation, substitution), and the integration facility. E. Tsiang and W. A. Martin designed and programmed the power series expansion routines.

P. Wang designed and implemented the limit programs, and the secondary storage control. R. Fateman designed and implemented the semantic pattern matching system. The improved LISP compiler is the work of J. Golden. Others who have contributed to the programming include D. Hill and S. Saunders.

In debugging these programs and in interfacing the different modules, it often became necessary for one programmer to add to or considerably modify another's work. In this sense, many of the modules are joint efforts.

Within this paper, sections I, II, and the Appendix are the work of W. A. Martin and R. Fateman; the section on rational function commands is by R. Fateman.

References

1. Engelman, C., "MATHLAB 68," in A. J. H. Morrell (Ed.) Information Processing 68, North-Holland, Amsterdam, 1969, pp. 462-467.

2. Fateman, R., "The User-level Semantic Matching Capability in MACSYMA," these proceedings.

3. --- "Essays in Algebraic Manipulation," doctoral dissertation, Harvard University, Div. of Engineering and Appl. Physics, 1971.

4. A. Hearn, "The Problem of Substitution," Stanford Artificial Intelligence Report, Memo No. AI-70, Stanford University, Stanford Calif., Dec., 1968. (Also appears in Proceedings of the 1968 Summer Institute on Symbolic Mathematical Computation, R. Tobey, editor, IBM Boston Programming Center, Cambridge, Mass., pp. 3-19, 1969.)

5. Martin, W. A., "Computer Input/Output of Mathematical Expressions," these proceedings.

6. --- "Symbolic Mathematical Laboratory," doctoral dissertation, M.I.T., E. E. Dep't, 1967.

7. Moses, J., "Symbolic Integration: The Stormy Decade," these proceedings.

8. --- "Algebraic Simplification: A Guide for the Perplexed," these proceedings.

9. --- "Symbolic Integration," doctoral dissertation, M. I. T., Math Dep't, 1967.

10. Thwaites, B., "1984: Mathematics <=> Computers?" Presidential Address to the Institute of Mathematics and its Applications, Bull. I. M. A., Dec. 1967.

11. Wang, P., "Automatic Calculation of Limits," these proceedings.

Appendix I: The Language and Commands of MACSYMA

Commands to MACSYMA are strings of characters representing mathematical expressions, equations, arrays, functions, and programs. Spaces and carriage returns are ignored. Commands are terminated by @ or $. @ causes the command to be evaluated and the result displayed. $ causes the command to be evaluated but the display of the result is suppressed. When typing commands, "rubout" deletes (and echoes back) the previous character; ?? deletes the whole command, and causes the line number to be redisplayed.

The Input Stream Editor

At any point while he is inputting a comand, the MACSYMA user can enter the input-stream editor by typing #. The editor is given the string of characters typed so far in the current command. In the case of a detected syntax error, the entire previous command string will be given to the editor.

All the commands to the editor reference a cursor which is displayed within (or at either end of) the string of characters under edit. In the description to follow, n stands for a positive or negative integer. The default value of n is +1. If n is positive, the commands operate toward the right of the cursor; if n is negative, they operate toward the left.

nC moves the cursor n characters.

nL moves the cursor to the right of the nth carriage return (e.g. L moves to the next line)

Sstring# moves the cursor to the right of the first occurrence of the string of characters "string" searching toward the right. (-S implies left)

nD deletes n characters.

nK deletes all the characters through the nth carriage return. (e. g., K deletes the remainder of this line)

Istring# inserts the characters "string"

# leaves the editor and returns to inputting from the user's console.

System Control

Lines are consecutively numbered, except that the input line Ci will be followed by an output line (if one is generated) named Di. The next input-output pair will be labelled C(i+1) and D(i+1), respectively. If one command produces several lines of output, the line number will be incremented for each additional line. A user can refer to any command or expression by its line label. The most recently outputted expression may be referred to as "%".

The system automatically writes old expressions onto secondary storage. The process is controlled by the following variables which can be set by the user. (e.g. FILESIZE:10$ would set FILESIZE.)

variable	default value	purpose
FILESIZE	10	Expressions are written out with FILESIZE expressions in each file.
RETAINNUM	8	When the number of expressions in core reaches FILESIZE + RETAINNUM, a file is written.
FILENAME	username	The first name of the file written out. The second names (our filing system requires two names for a file) are 1,2,....
INCHAR	C	The prefix character for inputted line numbers.
OUTCHAR	D	The prefix character for outputted line numbers.

When an expression is written out, the name of the file containing it is attached to the expression name in core. Thus when the expression is referenced in a later step, it can be automatically retrieved from the file.

At the end of the session, the secondary storage files can be deleted by the command FINISH(). The command FINISH(TRUE) allows the user to retain some or all of the expressions on his file. In order to specify the form and contents of the retained file, he must answer a series of questions:

question	response	meaning
OUTPUT DEVICE?	(file spec)	The name of the file on which the output will be saved.
EDIT?	N	Save the files as they are. This response will cut off further questions.
	Y	Read the files back into memory, one expression at a time, so that selected expressions can be saved on the previously specified file.
INTERNAL?	Y	Save the expressions in machine readable form. In this form they may be read back into a fresh system using RESUME.
	N	Save only the two dimensional display forms.
SAVE?	Y	(This is asked for each expression.) Include the expression currently displayed.
	N	Do not include it.

RESUME (file specification) reads a file previously outputted through FINISH, displaying the commands and recomputing the results. BATCH(file specification) reads an input text from the designated file, command by command. When the end of the file is reached, further commands may be supplied by the user at his console. This batch-processing mode in time-sharing has been surprisingly useful in generating demonstrations free from typing errors.

Rules for Expression Evaluation

The philosophy of evaluation used in MACSYMA is that expressions should be evaluated as much as the user would normally desire, given the information available at evaluation time.

A:X assigns A the value of X. This is the way a user would typically assign a value to a variable. Values are also assigned when the variables are used as labels for expressions on input and output.

A variable which does not have a value stands for itself. Numbers always stand for themselves. The functions DERIVATIVE, INTEGRAL, SUM, and the transcendental functions are not automatically evaluated. Other defined functions are evaluated unless their names are quoted. The arguments of undefined functions are evaluated, but, obviously, the function itself cannot be evaluated. As an expression is evaluated, it is also simplified.

If a name is subscripted (a subscript, recall, is enclosed in square brackets on input), then its value is stored in an array. The size of an array may be declared by the command ARRAYSIZE(name,size)$. An array need not have its dimensions declared, but if it has been declared, it will be permitted to have only numerical subscripts. At the first attempt to store a value in an undeclared array, a mechanism will be set up to describe the entries and their values in terms of a hash-coded list. The hash code can be computed from the subscripts whether or not they are numerical. If an array is subsequently declared, the values in the hash table are transferred to the new (true array) organization. The value of an array entry can be a number, expression, equation (etc.) regardless of whether it is a hash array or a true array. A hashed array is organized as follows: It is initially allocated a hash table with four entries. Each table entry contains a list of subscripts and values which hashes into that entry. Whenever the number of entries with values is equal to the size of the hash table, the size of the hash table is doubled. Whenever the operation ":" is executed, a check is made to see if the name is subscripted. If so, the appropriate array entry is set.

A::X assigns the value of A the value of X. The value of A must be a variable in this situation.

Function Definitions and Arrays

MACSYMA incorporates a programming syntax resembling Algol-60 for use on the top (command) level and in function definitions. The parser is entirely syntax directed, so that modifications to the grammar can be easily included; also, an exact definition of the acceptable forms (and their interpretations in terms of LISP and MACSYMA functions) can (but will not) be given. The syntax is illustrated in figure 3. Each of these constructions has fairly conventional interpretation, except when symbolic and traditional numeric notions conflict. One such instance is in inequalities, and is discussed in the next section in more detail.

The first argument to ":=" (the function definition operator) may take one of three forms: f(x), f[i] or f[i](x). Let the second argument to ":=" (that is, the right hand side) be y. In the first case, the variable f denotes a function, with value lambda(x)y. In the second case, a function definition is being associated with an array. The name f is denoted an AEXPR with value lambda(i)y. An AEXPR is used as follows. If a particular value of an undeclared array (it is an array if the variable is subscripted or if the name has previously been subscripted and assigned a value) is not present in the associated hash table, a check is made to see if the name also denotes an AEXPR. If so, this function is evaluated and the resulting value is stored in the hash table and also returned. If no value is present and no AEXPR is present, the expression is handled as though it were an undefined function.

If the first argument of ":=" is f[i](x), the third case, then f is denoted an AEXPR as above, but this AEXPR evaluates to a function of x. For example, given f[i](x):=x**i, evaluating f[3](5) would cause the AEXPR to be evaluated to lambda(x)x**3 and this value would be stored as the value of f[3] and also applied to 5 to yield 5**3. A subsequent evaluation of f[3](7) would cause the value lambda(x)x**3 of f[3] to be retrieved and applied to 7.

Predicates and Conditionals

The comparison operators ">", "<", and "=" are not evaluated in ordinary contexts; however, these operators, along with AND and OR are evaluated when they are in the predicate position of the IF-THEN-ELSE construction. If the predicate (the IF clause) evaluates to TRUE, the THEN clause is evaluated and returned. If the predicate evaluates to FALSE, the ELSE clause is evaluated and returned. If the predicate cannot be evaluated completely (e.g. a comparison operator was given a non-numeric argument), the construction is returned unevaluated.

General Purpose Commands

INTEGRATE(exp,var) integrates exp with respect to var or returns an integral expression if it cannot perform the integration.

LIMIT(exp,var,val,dir) finds the limit of exp as the real variable var approaches the value val from the direction dir. Dir may have the value PLUS for a limit from above, MINUS for a limit from below, or may be omitted (implying a two-sided limit is to be computed). LIMIT uses the following special symbols: INF (positive infinity) and MINF (negative infinity). On output it may also use UND (undefined) and IND (indefinite but bounded).

SUBSTITUTE(a,b,c) substitutes a for b in c. b must be an atom or a function with arguments, rather than a function with only some of its arguments. When b does not have these characteristics, one may sometimes use SUBSTPART or RATSUBST. SUBSTITUTE((eq1,...,eqk),exp) is another permissible form. The eqi are equations indicating substitutions to be made. For each equation, the right side will be substituted for the left in the expression exp (if the left side is non-atomic, and the right side is, the equation will be "flipped")

EXPAND(exp) will cause an expansion of the argument. The MACSYMA variables MAXNEGEX and MAXPOSEX (originally set to 6) control the maximum negative and positive exponents, respectively, which will expand. EXPAND(exp,p,n) expands exp, but uses p for MAXPOSEX and n for MAXNEGEX.

SIMPLIFY(exp) simplifies its argument, thus overriding the value of the MACSYMA variable SIMP which if set to FALSE stops simplification.

PART(exp,n1,...,nk) obtains a subexpression of exp which is specified by the indices ni. The index n1 which (like all the indices is a non-negative integer) selects the argument of the top level operator of exp corresponding to its value. Thus PART(Z+Y,2) yields Y. The index n2 (if specified) picks up an argument of the result of PART(exp,n1). Thus PART(Z+2*Y,2,1) yields 2. The operator is considered to be argument 0.

In exponentiation, the base is considered argument 1, and the exponent argument 2. In a quotient, the numerator is argument 1, and the denominator is argument 2. A minus sign appearing in the display is considered as an operator. For example

```
(C1) X+Y/Z**2@
                                   Y
(D1)                               -- + X
                                    2
                                   Z
(C2) PART(D1,1,2,2)@

(D2)                                2
```

DPART(exp,n1,...,nk) selects the same subexpression as PART, but instead of just returning that subexpression as its value, it returns the whole expression with the selected subexpression displayed inside a box. Thus in the example above,

```
(C2) DPART(D1,2,2,1)@
```

```
(D2)                  Y
                     ---- + X
                       2
                     z
```

SUBSTPART(x,exp,n1,...,nk) substitutes x for the subexpression picked out by the rest of the arguments. It returns the new value of exp.

DERIVDO(exp,var1,...,varn) forces the derivatives of exp with respect to the vari to be evaluated.

DIFF(exp,var1,n1,...,vark,nk) differentiates exp with respect to vari ni times. If k=1 and n1=1, n1 may be omitted: DIFF(exp,var).

DEPENDENCIES(f1,...,fn) declares functional dependencies used by DIFF. Each fi (i=1,n) has the format f(v1,...,vm) where each vj (j=1,m) is a variable on which f depends. Thus DIFF(Y,X) is 0, initially. Executing DEPENDENCIES(Y(X))$ causes future differentiations of Y with respect to X to be displayed as

```
DY
--
DX
```

KILL (arg1,...,argn) eliminates its arguments from the MACSYMA system. If argi is a variable, a function name, or an array name, the designated item is removed from core and the storage it occupies is reclaimed. argi = "HISTORY" eliminates all input and output lines to date (but not other named items). argi = a number, n, deletes the last n lines.

STORE(arg1...,argn) is similar to KILL in that it reclaims core storage (but not quite as much). The values of the arguments to STORE are removed from core and saved on a secondary storage device. Special indicators left in core allow MACSYMA to read back these items whenever referenced. The arguments can be variables, function names, or array references. Numbers or "HISTORY" are not acceptable, since storage of the input and output lines is automatic and controlled by RETAINNUM.

COEFF(exp,var,n) obtains the coefficient of var**n in exp. For best results, exp should be expanded. Coefficients of var**n which are functions of var are ignored. This command is less powerful than RATCOEF, but is sometimes convenient in interactive situations.

```
(C2) COEFF(Y+X*%E**X+1,X,0)@
(D2)                  Y + 1
```

DOSUM(ind,lo,hi,exp) performs a summation of the values of exp as the index ind varies from lo to hi.

```
(C3) DOSUM(I,1,4,I!)@
(D3)                   33
```

EV(exp,arg1,...,argn) causes the expression exp to be evaluated and simplified with switches set according to the values of the argi.

EVAL reevaluates the expression so that variables in it which have values will be evaluated.

SIMP overrides the setting of the SIMP switch.

EXPAND causes expansion. EXPAND(n,m) set the values of EXPOP and EXPON.

DIFF causes all differentiations indicated to be performed. DIFF(var1,...,vark) causes only differentiations with respect to the indicated variables.

NUMER causes SIN, COS, LOG, and "**" with numerical arguments to be evaluated.

v=exp causes the substitution of exp for v. v must be an atom.

The arguments following the first (exp) may be given in any order. It should be understood that EV performs a single evaluation and simplification; thus all of the functions are performed in one scan. This is possible because the simplifier is used to perform expansions, differentiation, and numerical evaluations by the setting of switches. For example:

```
(C4) SIN(X)+COS(Y)+(W+1)**2
+DERIVATIVE(W,1,SIN(W))@

                                D                 2
(D4)  COS(Y) + SIN(X) + --SIN(W) + (W + 1)
                               DW

(C5) EV(%,NUMER,EXPAND,DIFF,X=2,Y=1)@

                             2
(D5)        COS(W) + W  + 2 W + 1.425324
```

WHEN conditional DO identifier = expression e.g., WHEN I=2 DO K=%@. The value of the identifier is determined by evaluating the conditional. If it evaluates to TRUE, then the expression is evaluated and used for the value of that use of the identifier. If the conditional evaluates to FALSE, then the identifier's value is itself. In effect, the identifier becomes a function of no arguments which evaluates the conditional, and if TRUE, returns the expression as its value.

SOLVEX((lhs1,...,lhsn),(v1,...,vn)) solves a system of linear algebraic equations. It takes two lists as arguments. The first list (lhsi, i=1,n) represents the left-hand-sides of the equations to be solved; the right-hand sides are 0. The second list is a list of the unknowns to be determined. If the given equations are not compatible, the message SINGULAR will be displayed. The solutions are exact, not subject to round-off error, and may involve symbolic variables. The solution set consists of a list of numbered equations and an index to the list, as in the SOLVE command.

DISPLAY(exp1,...,expn) prints equations whose left-hand-sides are the expi, and whose right-hand-sides are the values of each expression. The value of DISPLAY is a list of the labels of the equations displayed.

```
(C7) DISPLAY(D3,I)@
(E7)          D3 = X + Y
(E8)             I = 5
(D9)            (E7,E8)
```

Rational Function Commands

The rational function commands have been discussed earlier. For the sake of completeness, we briefly list them, along with an indication of their purposes.

RATVARS provides a method for specifying the ordering of variables in CRE form.

RAT converts an expression to CRE form.

RATDISREP converts a CRE to a normal prefix expression.

RATSIMP, FULLRATSIMP, and RADCAN are simplifiers.

FACTOR factors a polynomial or rational function (numerator and denominator).

PARTFRAC expands a rational function in partial fractions.

RATCOEF picks out coefficients.

RATSUBST substitutes.

SOLVE solves an equation for a variable.

The Matching Subsystem

(for details, see [2])

DECLARE(*var*,*pred*) declares *var* to match only expressions satisfying the predicate *pred*, when *var* is used in a pattern.

DEFMATCH (*name*,*exp*, *var1*,...,*vark*) defines a pattern matching program with name *name*.

DEFRULE(*name*,*exp*,*repl*) defines a transformation rule with name *name* which matches the pattern *exp* and transforms it to the replacement *repl*.

APPLY1(*exp*,*r1*,...,*rk*) (and similarly for APPLY2) applies the rules *r1*,...,*rk* to the expression *exp*, and returns the transformed expression. The difference between APPLY1 and APPLY2 is in the sequencing through the expression and rules.

TELLSIMP (*pat*,*repl*) (and similarly for TELLSIMPAFTER) changes the simplifier, so that in all subsequently simplified expressions, an occurrence of the pattern *pat* will be replaced by the expression *repl*.

Several additional predicates and testing programs are provided for use in constructing patterns and their predicates. SIGNUM(*x*) returns -1,0, or +1, depending on whether the sign of *x* is negative, zero, or positive. If *x* is not a number, its signum is computed from the coefficient of the leading term in a rationally simplified expression equivalent to *x*. FREEOF(*x*,*y*) returns TRUE if *y* does not depend explicitly on *x*. This is accomplished by searching through *y* for an occurrence of *x*, and assumes that *x* is not, for example, used as a dummy variable of integration. INTEGER(*x*) returns TRUE if *x* is an integer.

Appendix II

This problem is taken from Chapter 3 of a 1963 Masters Thesis by J. S. Draper for the MIT Department of Aeronautics and Astronautics. This thesis investigates the laminar compressible boundary layer on the electrode walls of a direct-current crossed field plasma accelerator under very special physical conditions.

The solution procedure begins as follows.

1. Write down 5 non-linear partial differential equations for momentum, state, continuity, energy, and electron mobility as a function of temperature. These equations relate
U, the stream velocity
V, the lateral velocity
t, the temperature
ρ, the density
μ, the electron mobility
P, the pressure
in terms of the independent variables, x and y. The constants are
j, the current
B, the magnetic field
C_p, the specific heat
K, the compressibility
G, the conductivity
R, the gas constant

2. The absence of variation in the y direction in the free stream is used to find the momentum and state equations there. These two reduced equations are solved for D_xP which is eliminated from the five main equations, since P is not a function of y.

Further steps in the solution procedure are discussed by Martin in [6].

Figure 1.

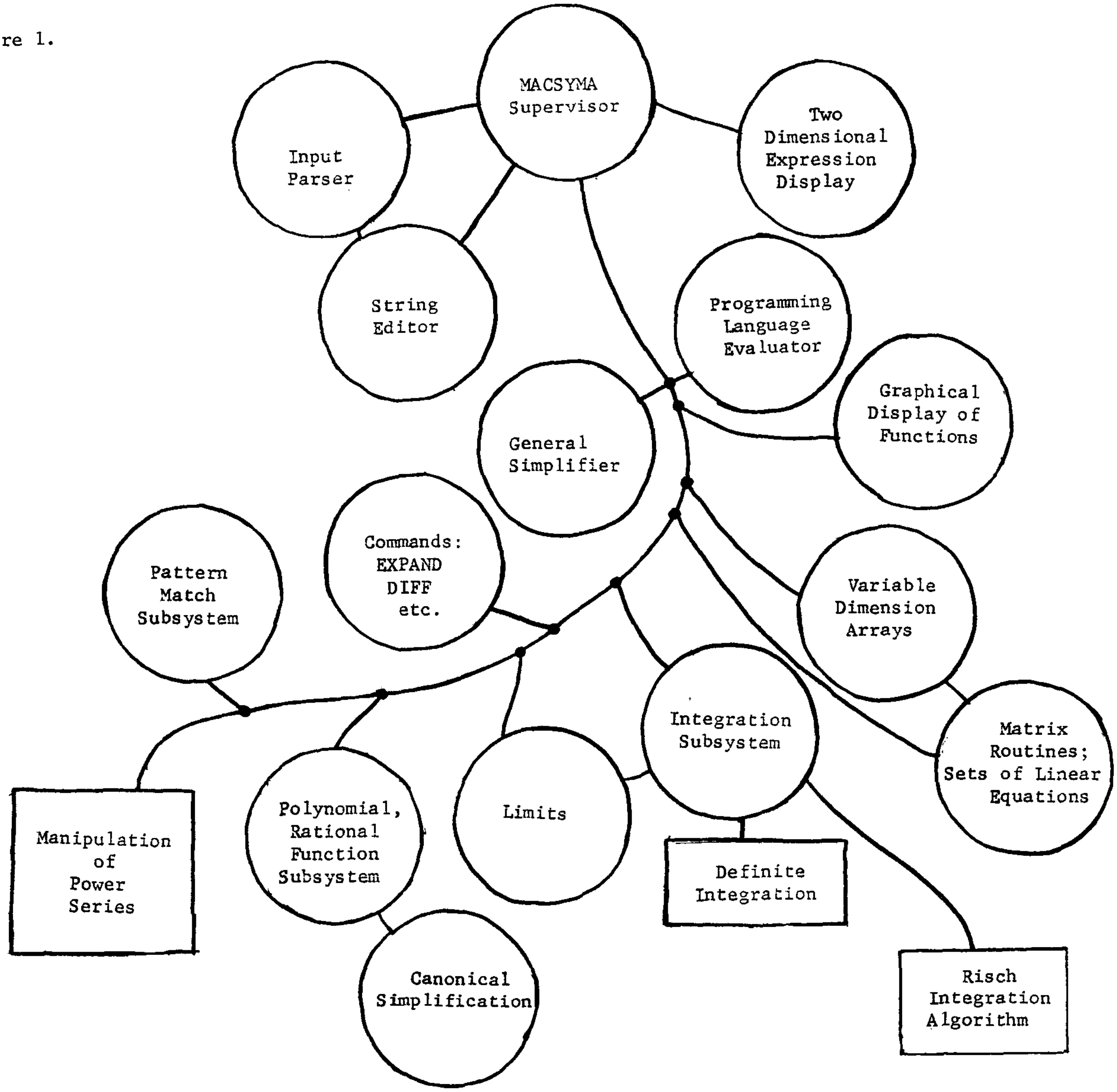
MACSYMA Supervisor
Input Parser
Two Dimensional Expression Display
String Editor
Programming Language Evaluator
Graphical Display of Functions
General Simplifier
Commands: EXPAND DIFF etc.
Pattern Match Subsystem
Variable Dimension Arrays
Integration Subsystem
Matrix Routines; Sets of Linear Equations
Manipulation of Power Series
Polynomial, Rational Function Subsystem
Limits
Definite Integration
Canonical Simplification
Risch Integration Algorithm

Figure 2. Examples of RATSUBST

Argument 1	Argument 2	Argument 3	RATSUBST Versions	Result
A	$X\ Y^2$	$X^4Y^8+X^4Y^3$	1,2 3,4	$X^4Y^3+A^3$ $A\ X^3Y+A^4$
1	S+C	S	1,3,4 2	-C+1 S
A	B (X+Y)	$B^2+BX+BY+1$	1,3,4 2,with RATVARS(Y) 2,with RATVARS(X) 2,with RATVARS(B)	B^2+A+1 $BY+BX+B^2+1$ $BX+BY+B^2+1$ B^2+A+1
A	$X\ Y^2$	X^2Y	1,2,3 4,with RATVARS(X) 4,with RATVARS(Y)	X^2Y A^2/Y^3 X^2Y
A	X+Y	(X+Y) (Z+W)	1,2,3 4	(Z+W) A (Z+W)Y+(Z+W)X
-1	I^2	I^4+1	1,2,3,4	2

Figure 3. Syntax of Expressions: Examples of the legal input expressions and the corresponding two dimensional display form are shown below. W, X, Y, and Z stand for any expressions; U and V for variables. (Some of these forms can be extended to take an arbitrary number of arguments in the obvious manner.)

input	display	meaning
AB	AB	variable
'AB	'AB	quoted variable
1	1	integer
1.2	1.2	floating point number
F[X,Y]	$F_{X,Y}$	subscripted variable
F(X,Y)	F(X,Y)	function invocation
F[X,Y](W,Z)	$F_{X,Y}(W,Z)$	subscripted function invocation
X!	X!	factorial
X**Y	X^{Y}	exponentiation
X/Y	$\frac{X}{Y}$ or X/Y	quotient
-X	-X	negation
X+Y	X+Y	sum
X-Y	X-Y	difference
X*Y	X*Y	product
X=Y	X=Y	equality predicate or equation
X < Y	X < Y	less than predicate or inequality
X > Y	X > Y	greater than predicate or inequality
X AND Y	X AND Y	logical AND or Boolean operator
X OR Y	X OR Y	logical OR or Boolean operator
'X	'X	quoted expression
(X,Y)	(X,Y)	list of expressions
IF X THEN Y	IF X THEN Y	conditional
IF X THEN Y ELSE W	IF X THEN Y ELSE W	conditional
FOR I:1 STEP 1 UNTIL 1 > 3 DO X	FOR I:1 STEP 1 UNTIL I > 3 DO X	DO loop
A:X	A:X	assign A value X
A(V):=X	A(V):=X	define function A(V)
A(V):Y FOR ALL W	A(V):Y FOR ALL W	define function A(V)
SUM(I,1,100,X)	$\sum_{I=1}^{100} X$	summation
INTEGRAL(Y,V,W,X)	$\int_{W}^{X} Y\ DV$	integration
DERIVATIVE(Z,U,2,V,3)	$\frac{D^{5}Z}{DU^{2}DV^{3}}$ or Z_{UUVVV}	differentiation
X.Y	X.Y	non-commutative product

An example of a program:

```
F(V):= BLOCK (IF V  > 0 GO (A),
              RETURN (0),
              A, RETURN (1) )
```