MATHLAB: A program for on-line machine assistance in symbolic computations*

CARL ENGELMAN

The MITRE Corporation
Bedford, Massachusetts

THE PURPOSE OF MATHLAB

A mathematical scientist experiments. Today, his test tube and his breadboard are blackboard and paper. He may, it is true, have available a computer, but its role is numerical and its results are delivered not today or tomorrow but the day after the final programming bug is corrected. The computer is not present during the most creative phases of the scientist's labor. The purpose of MATHLAB is to provide the scientist with computational aid of a much more intimate and liberating nature.

What sort of aid? The basic goal is to provide facilities for those operations which are mechanical. Among the most common of these are the addition of expressions and equations, the substitution of subexpressions into a larger expression, differentiation, integration, Laplace Transforms, multiplication of matrices, and the solution of simple equations. Although the greater part of a scientist's time is spent on these mechanical pursuits (in fact, an appreciable portion is probably spent in simply checking answers and in the eternal bookkeeping problems of getting minus signs and 2 π's right), we must keep in mind that most of the tedious computations associated with the creative aspects of his work are of a symbolic, rather than a numerical, nature.

If we are to free the scientist from his routine mathematical chores and conserve his energies for the more properly human activities of interpretation, analysis, planning and conjecture, then we must mechanize the passage from $r^2/r$ to $r$ in addition to that from $1 + 1$ to $2$.

REQUISITES FOR A MATHEMATICAL LABORATORY

I should like to outline here the properties I feel are required of a mathematical laboratory, not in terms of the range of mathematical operations available, but rather in terms of its spirit and feel.

1. It should be capable of ordinary numerical computation. This implies the ability to perform arithmetic, to compute functions or to look up their values in tables, and to draw graphs.

2. It should be capable of a wide spectrum of symbolic computations.

3. It should respond to simple user commands. MATHLAB is intended for the physicist, not the programmer. The commands should be no more complicated than his thoughts. If he wishes to enter an equation into the computer, he should need only to type the equation in a notation like that of ordinary mathematics. If he should then wish to differentiate that equation with respect to $x$, he should have to give a command no more complicated than "differentiate $(x)$".

4. It must be expandable by the expert. The language, functions, and subroutines of the laboratory must be such that it will grow as an organism. If today we write programs for symbolic differentiation, we should expect, tomorrow, to employ them in programs for power series expansions. The opportunity to expand the pro-

*This project was supported by The MITRE Corporation Independent Research Program. It was also supported in part, through access to its computer facilities, by Project MAC, a M.I.T. Research Program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract No. NONR-4102(01).
grams should be open to anyone who masters a well-defined and common computer language.

5. It should be extensible by the user. Although the ability of the physicist to augment the existing programs will no doubt be severely limited compared to that of the programming expert, he should be provided tools for doing certain simple things for himself, such as changing notational conventions or teaching the machine the derivatives of his favorite functions.

6. The computer, as viewed by the user, must be intimate and immediate. The user should have next to his desk a console consisting of a typewriter or, preferably, a typewriter and a scope. Economy might, in some cases, dictate the substitution of a plotter for the scope. These are connected to a large, fast, on-line, time-shared digital computer. He communicates with that computer by typing messages on his typewriter or by means of a light-pen on the scope. The computer replies by means of the same machines. It types both messages and equations. On the scope it displays both equations and graphs. Above all, the response time to the user's requests must be short.

BRIEF SUMMARY OF EXISTING PROGRAMS FOR SYMBOLIC MATHEMATICS

The first program we should mention was written for Whirlwind I by J. H. Laning and N. Zierler and was not really a program for symbolic computation. But, even though the program was capable only of numerical computation, it could accept programs written as simple symbolic mathematical expressions and perform them for a user with no machine language experience.

A later program that could accept instructions in the form of symbolic expressions, but which was also limited to numerical computations, was, of course, FORTRAN.

The ALPAK system at the Bell Laboratories, designed by W. S. Brown, J. P. Hyde, and B. A. Tague, is a program for the manipulation (multiplication, division, differentiation, etc.) of polynomials and rational functions. It is not a system of symbolic computation in the sense we are using that expression. Although its instructions are FORTRAN-like, its input/output, as well as its internal representations of polynomials and rational functions are lists of numbers representing the relevant coefficients and exponents. Owing to its entirely numerical nature, the program is quite rapid.

Another interesting development in the field of numerical computations is the STL On-Line Computer designed by G. Culler and B. D. Fried at the Space Technology Laboratories. Although the programming facilities are similar to FORTRAN, the instructions are entered, in an on-line environment, through the pushing of buttons labeled “+,” “-,” “sqrt,” etc. After subroutines are debugged, it is possible, by pushing buttons, to define the effect of a short sequence of button pushes to be the same as any sequence of first-level pushes. These subroutines of higher-level button pushes can themselves be incorporated in the definition of button pushes at a still higher level.

Probably the most fundamental development, to date, for the adaptation of computers to symbolic computation is the design by J. McCarthy, for just such purposes, of a language called LISP (for List Processor). A great majority of the symbolic computation programs extant are written in this language.

The foremost problem of content facing the construction of a mathematical laboratory today is probably the writing of a satisfactory program for the simplification of mathematical expressions. Simplification is the unconscious of mathematics. We all simplify expressions every day, making choices appropriate to the occasion but of which we are almost totally unaware.

For this reason, it is much simpler to program a more advanced formal computation, such as differentiation, which has exact rules of which we are conscious, than it is to have the machine simplify the answer after it completes the differentiation. We cannot afford to enter into a detailed discussion of the current attempts to solve this problem, but we would like to mention, in chronological order, the authors of three LISP programs for the simplification of mathematical expressions: T. Hart, D. Wooldridge, Jr., and W. A. Martin.

A LISP program for symbolic differentiation was written by K. Maling. Although it suffered from some weakness of simplification and from its input/output being restricted to well-formed LISP expressions (which are not nearly as legible as those written in ordinary mathematical notation), it was certainly a dramatic early demonstration of the ability of LISP to handle formal mathematical computation.

In my opinion, the most impressive example of symbolic mathematics yet performed by machine is the formal (or indefinite) integration program of J. Slagle, written as a doctoral thesis under Prof. M. Minsky at M.I.T. Perhaps the most interesting aspect of this program is that it was heuristic rather than algorithmic. It possessed a small table of integrals and tried to re-
duce the problem to one or several found in that table by the same bag of tricks possessed by a good college freshman.

The remaining programs as we shall discuss do not, like those above, represent attempts to perform particular symbolic processes (simplification, differentiation, integration) on a computer. They are, rather, attacks on the whole problem of building a system of symbolic computations.

The first such program we should like to mention is the mathematical laboratory project of W. A. Martin, a work-in-progress as a doctoral thesis at M.I.T. under Prof. M. Minsky. This work is by far the closest known relative to our MATHLAB. At present it appears that the main difference of emphasis will be that Martin's work will stress very broad input/output capabilities. He is, for example, working on input from scopes achieved by signing with a light-pen an interesting subexpression of a previously "printed" expression, as well as anticipating using the scopes for handwritten input. The emphasis in our program is more in the direction of continually increasing mathematical powers. Since both programs are written in LISP, a good deal of exchange should be possible.

Another approach to the mathematical laboratory problem is a project, under the direction of L. C. Clapp, while he was employed at Bolt, Beranek, and Newman. The programs are not written in LISP but in a much simpler and weaker list-processing language of Bolt, Beranek, and Newman's own invention, called SIMPLIST. While this language and program might serve well for rapid performance of simple computations, e.g., adding two symbolic equations or evaluating a function entered symbolically, it seems unlikely that they will be capable of difficult symbolic procedures, such as a powerful simplification program or symbolic integrations. These limitations stem primarily from the fact that their work is married to the PDP-1, a small computer.

The next system we should like to discuss is the IBM FORMAC system. This is an extension of FORTRAN which provides it with the ability to perform a certain amount of symbolic computation. It is under development in Cambridge under a group headed by J. E. Sammet. It is quite capable, possessing such abilities as simplification, substitution, expansion of products of sums, factoring out powers of a given variable, and differentiation. For a number of reasons, however, it does not seem well suited to a mathematical laboratory. It is program oriented. One does not give a simple command, such as "differentiate" but rather one writes a program. True, the language, like FORTRAN, is easy for a nonprogrammer to learn, but it will not have the universal accessibility of, say, the simple commands of our MATHLAB. The FORMAC programs can, of course, be run on-line, but they cannot be run line by line, nor were they intended to be. It is necessary to write an entire program, for the program has to be compiled as an entity, as does a FORTRAN program (in fact, it employs the FORTRAN IV compiler). Thus, although a useful tool for symbolic computation when you know at the outset what you want to do, FORMAC would be quite inconvenient for experimentation.

The final program we would like to discuss before turning to our own MATHLAB is the FORMULA ALGOL of Perlis and Iturriaga. In one sense, this language is similar to FORMAC; namely, the ability to manipulate formulas, i.e., mathematical expressions, is embedded in what was previously a purely numerical compiler, in this case ALGOL rather than FORTRAN. Viewed this way, its built-in, as opposed to programmable, facilities are quite meager: restricted mainly to formula substitution and some rudimentary simplification. However, this view of FORMULA ALGOL has been superseded by the creation of a powerful programming language which combines the numerical facilities of ALGOL not only with some ability to manipulate mathematical expressions but also with the ability to create and manipulate list structures. In this role of combined algebraic and list-structure compiler, FORMULA ALGOL stands as a predecessor of LISP 2, the type of language in which programs such as MATHLAB will probably be written in the future.

MATHLAB—ON THE SURFACE

MATHLAB is our current attempt at realizing a mathematical laboratory of the sort we have been discussing. The program, which has been developed on the time-shared system of Project MAC at M.I.T. and on the IBM 7030 at The MITRE Corporation, is continually growing, and the following description is accurate as of June 30, 1965. We do not feel that MATHLAB has, as yet, sufficient mathematical powers to be of aid to a general user, except with respect to special and occasional problems. It does, however, possess most of the qualities postulated in the previous section as requisites for a mathematical laboratory.

1. Numerical computations. It is very weak in this department because we decided at first to study symbolic computation as it represented the crux of our problem. It cannot draw graphs or evalu-
Mathlab

ate common transcendental functions. We can evaluate algebraic expressions with numerical arguments in a variety of ways.

2. Symbolic computations. Here we can perform many common tasks. We can simplify, substitute, add equations, differentiate, integrate a little, solve equations, etc.

3. The user commands are simple. If the user has stored an equation called "e1" and wishes to differentiate both sides of it with respect to "x" and call the resulting equation "e2", he need only type: differentiate (e1 x e2).

4. The program can be expanded by any LISP programmer. In fact, we are doing this all the time.

5. MATHLAB can be extended a little by the user. He can teach the machine the derivatives of functions and change the names of system commands.

6. It is intimate. The user types in some initial equations; the computer acknowledges them. The user requests certain symbolic manipulations; the computer performs them and types back the answer. The user types in some expressions or numbers and requests the computer to substitute these for certain variables in a previous equation; the computer types back the answer, etc.

Mathematical Notation

In this section and the next we wish to describe MATHLAB as it appears to the user. There are some minor differences between MATHLAB as it exists at Project MAC and on the STRETCH at MITRE. Where such differences occur, we shall describe the situation at MITRE. First, what sort of expressions may a user type to denote mathematical quantities? The answer is: those expressions, composed in the ordinary way, of the following entities:

1. Numbers: 1, 5/2, 2.5
2. Words, representing symbolic variables, composed of strings of letters and digits, the first of which is not a digit:
   x, distance, x1, x1sub2.
3. Operation symbols:
   + (addition)
   - (subtraction or minus)
   * (multiplication)
   / (division)
   † (exponentiation)
4. Parentheses: ( and ). These have two functions. The first is to ensure the desired interpretation of certain expressions, e.g., to distinguish 5*(x + y) from 5*x + y. The second use of parentheses is for functional notation, e.g., sin(x).
5. Comma: The comma, besides its end-of-message role to be discussed shortly, serves to separate the arguments of a function of several variables, e.g., f(x, y, z).

All blanks are ignored. The rules of precedence of the operational symbols are conventional (FORTRAN) except that, in the absence of parentheses, e†x†2 denotes e†(x†2), not (e†x)†2 = e†(2*x).

For example, if the user wishes to enter the mathematical expression (in conventional notation): sin(5x + y²), he need only type (in MATHLAB's notation): sin(5*x + y 2).

The System Commands

The program as we have developed it accepts three types of symbolic quantities, called expressions, functions, and equations, which are stored in the computer and which can be referred to and manipulated by name. An expression is a mathematical quantity or expression referred to by a (one word) name. A function is a mathematical expression and an ordered list of bound or dummy arguments and is also referred to by a (one word) name. An equation is really two mathematical quantities, one for the left and the other for the right side of the equation, referred to by a (one word) name. The initial expressions, functions, and equations in an experiment are entered into the computer by a function called "denote". For example, one might type:

denote nil
d = 1/2*a*t²,
el == r|² = x|² + y|²,
f (x,y) = x + y,

The first three commas signify the end of individual definitions and the fourth comma tells the computer that this is all the information we choose to give it for the time being. The word "nil" is a vestige of the hidden fact that "denote nil" is really a couple for the LISP evalquote operator. The effect of this input is to store in the computer an expression whose name is d and whose value is 1/2*a*t², an equation whose name is el, whose right side is r|², and whose left side is r|² + y|² and a function whose name is f, whose list of dummy arguments is (x, y), and whose value is x + y. From this point on the user never again has to type in r|² = x|² + y|² but can simply refer to el.
Incidentally, the response to the above instruction (we shall, from now on, use the convention that we speak in lower case and the computer in capitals) is:

THANKS FOR THE EXPRESSION D, THE EQUATION E1, AND THE FUNCTION F.

In terms of these basic constructs of expression, function, and equation, we shall describe the various system commands.

**repeat (x)**

This command repeats x to the user. x may be the name of an expression, a function, or an equation. The format for an expression or a function is similar to that of denote. For an equation, if x were e1 above, then “repeat” would print:

(E1) R1^2 = X1^2 + Y1^2

This same format is used when any of the succeeding commands is called upon to print an equation.

**Please simplify (x y)**

This command simplifies x and names it y. In this, as in the following commands, the name “y” may be the same as the name “x”. In that case, the old x is lost. If “x” is the name of an equation, both sides are simplified independently. For a detailed discussion of the scope of simplification, the reader is referred to Ref. 10, which discusses the simplification program we employ.

**forget (x)**

If x is a complicated expression, function or equation, its storage might be burdensome. The command “forget” allows the user to retrieve the space when it is no longer needed.

**substitute ((v1 v2 ... vn) x y)**

The first argument “(v1... vn)” must be a list of names of expressions and functions in any order. The value of each “vi” is substituted in x at each occurrence of the name “vi”. The new equation, expression, or function (depending on x) is named “y”.

At this point we should like to state more precisely the meaning of the denote and substitute instructions. Should we give the command:

denote nil

z = x + y,

x and z would be quantities of quite different natures. We shall refer to “x” as a formal symbol; it is without meaning. “z”, on the other hand, is the name of an official expression; its meaning, which we normally refer to as its “value”, is the quantity x + y constructed of the formal symbols x and y.

If we now type the instructions:

denote nil

x = 5*t,

an expression whose name is x and whose value is 5*t would be created, but this would in no way affect the status of x in the value x + y of the expression whose name is z. That x remains a formal symbol. The fundamental connection that can be established between these two occurrences (of different types) of the character “x” is through the instruction “substitute”.

If we now type:

substitute( (x) z w)

the program will look for an expression whose name is x, find that its value is 5*t, look for an expression whose name is z, discover that its value is x + y, substitute the quantity 5*t (containing the formal symbol t) for all occasions of the formal symbol x in x + y (obtaining the quantity 5*t + y), simplify this to 5*t + y, and create a new expression whose name is w and whose value is 5*t + y.

Substitutions take place only upon command, never automatically. This is as it should be. The user may have previously informed the computer that x = r*cos(t), but he might like to type x without having it automatically changed to r*cos(t). Automatic substitution schemes are not only undesirable, but are also prone to interminable loops.

The substitution of a function in a function is defined recursively so that it will operate to any depth. For example, if

\[ g(u, v) = f(f(u, v), f(u, v)) \]

and

\[ f(x, y) = x + y, \]

then the command:

substitute( (f) g h)

would yield:

\[ H(U, V) = 2*U + 2*V. \]

It is probably better to think of substitute in this circumstance as a command to unwind the functional definitions.

There is an important restriction on the substitution of expressions within functions. One may only substitute for the unbound variables or parameters of a function. For example, if \( f(x) = x + t \), one may substitute for t but not for x. It is expected that the casual user will attempt to violate this rule and instructive error messages have been prepared. This restriction is dictated by the meaning of a dummy variable of a function, but
any desired result can surely be achieved by a short sequence of commands. In this case, we could, for example, denote \( h(y) = f(x) \) and then substitute first \( f \) and then \( x \) in \( h \).

This description may seem too detailed, but an understanding of the distinction between an expression and a formal symbol as well as the function of the substitute instruction is fundamental to an understanding of MATLAB. We shall presume the extension of these concepts to equations and to other commands, e.g., differentiate, is apparent.

\[ \text{add}((q_1\ q_2\ \ldots\ q_n)\text{name}) \]

The \( q \)'s can be equations, functions, expressions, or numbers in any order. If there is at least one equation among them, name is an equation; if not and some \( q \) is a function, name is a function; otherwise it is an expression. Equations are added by adding left sides and adding right sides independently. Expressions, functions, and numbers are added to an equation by adding their values to both sides of the equation. If functions are added to form a new function, the list of dummy variables of the new function is the union of the lists of dummy variables of the old functions.

\[ \text{multiply}((q_1\ \ldots\ q_n)\text{name}) \]

Similar to above.

\[ \text{subtract}(x\ y\ \text{name}) \]

Similar to above, but only two equations, expressions, functions, or numbers are subtracted instead of an indefinite number as in add and multiply.

\[ \text{division}(x\ y\ \text{name}) \]

Similar to above.

\[ \text{raise}(x\ y\ \text{name}) \]

Similar to above. name = \( x^y \).

\[ \text{negative}(x\ y) \]

\( y = -x \).

\[ \text{invert}(x\ y) \]

\( y = 1/x \).

\[ \text{flip}(x\ y) \]

\( x \) must be the name of an equation. \( y \) becomes the name of that equation with the left and right sides interchanged. This is useful if we wish, say, to add the left side of one equation to the right side of another.

\[ \text{makeequation}(x\ y) \]

\( x \) must be the name of an expression or function. If \( x \) is an expression, an equation is formed whose name is \( "y" \), whose left side is the name \( "x" \), and whose right side is the value of \( x \). For example, using the expression \( "d" \) of denote above, if the instruction: "makeequation (d e2)" were given, the computer would respond: (E2) \( D = 1/2*AT^{-2} \). Should \( x \) be a function, the effect is best described by an example. Employing the function \( "f" \) of denote above, "makeequation (f e3)" would yield: (E3) \( F(X, Y) = X + Y \).

\[ \text{makeexpression}(e) \]

e must be an equation whose left side is a single word. Then an expression is formed whose name is the left side of \( e \) and whose value is the right side of \( e \). For example, "makeexpression(e2)" would now produce: \( D = 1/2*AT^{-2} \), which is where we started.

\[ \text{makefunction}(e) \]

e must be an equation whose left side looks like \( "f(x_1, x_2, \ldots, x_n)" \) with all the \( x_i \) simple (one word) formal symbols. We get a new function \( "f" \) whose dummy variable list is \( "(x_1 x_2 \ldots x_n)" \) and whose dummy variable list is \( "(x_1 x_2 \ldots x_n)" \) and whose value is the right side of \( e \).

The purpose of these three "make" commands is twofold. The first purpose is to guarantee the legality of certain succeeding commands. In the example occurring in the description of "makeexpression" above, \( e_2 \) could not be substituted in another equation but \( d \) could. The second is to ensure the accessibility of certain results. E.g., taking \( e_2 \) and \( f \) as above, compare the effect of the two following sequences of commands:

First:

\[ \text{add}((e_2 \ f)\ e) \]

(E) \( D + X + Y = 1/2*AT^{-2} + X + Y \)

Second:

\[ \text{makeequation} (f \ e_{101}) \]

(E101) \( F(X, Y) = X + Y \)

\[ \text{add}((e_2 \ e_{101})\ e) \]

(E) \( D + F(X, Y) = 1/2*AT^{-2} + X + Y \)

\[ \text{expand}((x_1\ x_2\ \ldots\ x_n)) \]

This produces no immediate result but affects all succeeding simplifications. Whenever one of the \( x \)'s (which are formal symbols) occurs in a product of sums, that product is multiplied out. The following dialogue will clarify this.

\[ \text{denote nil} \]

\( e_3 \) == \( y^3 + y*y = (x + z) * (u + v) \), THANKS FOR THE EQUATION E3

\[ \text{pleasesimplify}(e_3\ e_4) \]

(E4) \( Y^3 + Y^2 = (X + Z)*(U + V) \)

\[ \text{expand}((x u)) \]

YES

\[ \text{pleasesimplify}(e_3\ e_5) \]

(E5) \( Y^1 + Y^2 = X*U + X*V + Z*U + Z*V \)

The "expand" command affects not only the command "pleasesimplify" but other commands, such as...
"substitute" or "add", which always simplify their answers.

factor((x y . . .)

Like "expand", this produces no immediate result but affects all future simplifications. It causes the collection of all terms containing, as a factor, a power of x and similarly for y. The order of formal symbols in the list "(x y . . .)" implies precedence. In this case, the term "x*y" will count as y occurrences of x rather than x occurrences of y.

We give an example:

denote nil
mitre = a*x + 2*x + x*y + y + b*y + 4*x
⊹2 + c*x⊹2,

THANKS FOR THE EXPRESSION MITRE.
pleasesimplify(mitre bedford)
BEDFORD = A*X + 2*X + X*Y + Y + B*Y + 4*X⊹2 + C*X⊹2
factor ((x y))
YES
pleasesimplify(mitre bedford)
BEDFORD = (2 + A + Y)*X + (4 + C)*X⊹2 + (1 + B)*Y

By calling the functions expand and factor at different times with different arguments, the user can maintain control over the form of his answers.

differentiate(y x yprime)...

Differentiates y with respect to x and calls the resulting expression or equation (depending on y) yprime. At present, all differentiation is explicit, i.e., a term is considered as constant in x, unless x appears in it explicitly.

learnderivative...

Allows the user to teach the computer the derivative of a new function. The format of this command is best explained by an example.

learnderivative nil
arctan,
1/(1 + x⊹2).

The need for this command has, to some extent, been obviated by an improvement in the differentiation program. If the computer is asked to differentiate arctan (x)⊹2, it will decide that the answer is twice arctan(x) times the derivative of arctan(x). If it should then discover that it does not know the derivative of arctan, it will try to obtain it from the typewriter. If it succeeds, it will complete the differentiation and remember the derivative of arctan for future use.

integrate(v x w)...

v must be an expression which is a rational function of x with (rational) numerical coefficients. w is its indefinite integral. For a more precise discussion, see the following section. We give an example. If:

v = (x + 1)/(((x⊹2 + 1)*(x⊹2 + x + 1)⊹3),
then the command
integrate(v x w)
yields the result:

W = (2/3*X⊹3 + 1/2*X⊹2 + 1/3*X - 1/2)/
(X⊹4 + 2*X⊹3 +3*X⊹2 + 2*X + 1) +
1/2*LOG(X⊹2 + 1 - ARCTAN(X) -
1/2*LOG(X⊹2 + X + 1) +
7/(3*SQRT(3))*ARCTAN((2*X + 1)/SQRT(3))

solve(e x)...

e must be an equation that is equivalent to a rational function (with symbolic coefficients) of x. At present "solve" can handle only those equations which are really (although not necessarily explicitly) quadratic or linear in x. The reply to this command, excepting those cases which the program cannot solve, takes one of three forms depending on whether the computer analyzes the equation to be linear, quadratic with distinct roots, or quadratic with a double root. The following three examples illustrate these different responses.

1. If e is the equation: y = a*x + b, then "solve(e x)" yields:

   THIS EQUATION HAS A SINGLE ROOT.
   X = (Y-B)/A

2. If e is the equation: a*x⊹2 + b*x + c = 0, then "solve(e x)" yields:

   THIS EQUATION HAS TWO ROOTS, NAMED FIRSTROOT AND SECONDRoot.
   FIRSTROOT = 1/2*(- B + SQRT(B⊹2 - 4*A*C))/A
   SECONDRoot = 1/2*(- B - SQRT(B⊹2 - 4*A*C))/A

   PLEASE RENAME THOSE WHOSE PRESERVATION YOU WISH TO ENSURE.

3. If e is the equation: x⊹2 - 2*b*x + b⊹2 = 0, then "solve(e x)" yields:

   THIS EQUATION HAS A DOUBLE ROOT.
   X = B

We give a fourth example which, though simple, exhibits more of the meat of the program:

From the collection of the Computer History Museum (www.computerhistory.org)
4. If \( e \) is the equation: \( \frac{1}{(x^2 - 1)} = \frac{1}{(x - 1)} \), then “solve \((e x)\)” yields:

**THIS EQUATION HAS A SINGLE ROOT.**

\( x = 0 \)

There are two points here. First, the equation does not appear, at first glance, to be linear but the program analyzes it as such. Second, a naive attempt at solving the equation, e.g., inverting both sides, could yield what we feel is an extraneous root at \( x = 1 \). For an explanation of how the program avoids this trap, see the following section.

rename(x y)...

The expression, function, or equation that had the old name “x” obtains the new name “y.”

newname(A B)...

This command differs from all previous commands in that it affects, not the data, but the system commands themselves. It creates a new system command \( B \) whose effect is identical to \( A \) and which exists in addition to \( A \). For example, if the user tires of typing “differentiate,” he can give the command:

rename (differentiate d)

after which the command:

d (y x yprime)

will have exactly the same effect as

differentiate(y x yprime).

Before digging beneath the surface of MATHLAB, it might help clarify some of the preceding if we give a very short sample session possible today.

denote nil

e1 == r\*t^2 = s\*t,

\( s = x^2 \cdot y, \)

\( t = \log(w)/x, \)

, THANK YOU FOR THE EXPRESSIONS S T

AND THE EQUATION E1.

substitute ((s t) e1 e2)

(E2) R^t = X*Y*LOG(W)

denote nil

w = \sin(x\cdot t + y\cdot t),

, THANK YOU FOR THE EXPRESSION W.

substitute ((w) e2 e3)

(E3) R^t = X*Y*LOG(SIN(X\cdot t + Y\cdot t))

differentiate(e3 x e3prime)

(E3PRIME) 0 = Y*LOG(SIN(X\cdot t + Y\cdot t))

+ 2\*X\cdot t*Y*COS(X\cdot t + Y\cdot t)

/\SIN(X\cdot t + Y\cdot t)

denote nil

THANKS FOR NOTHING.

This last crack on the computer’s part is indicative of the fact that most of our commands have very heavy error protection. If the user makes a mistake in constructing an expression or an equation or tries to give an expression a name already assigned to an equation, etc., he will receive an instructive error return.

MATHLAB—BELOW THE SURFACE

LISP

The entire MATHLAB program is written in LISP. The system commands are all addressed to the LISP evalquote operator, e.g., the command “differentiate \((h t dh)\)” presents the evalquote operator with a couple consisting of the function “differentiate” and the list of arguments “(h t dh)”.

Numbers

The LISP system, written by R. Silver and P. Heckel for the IBM 7030 (STRETCH) at MITRE, contains only one type of number, namely, rational numbers, i.e., ordered pairs of integers. If any of the numbers, 12/5, 24/10, or 2.4, is typed in, it is converted to the rational 12/5.

In addition to rationals, MATHLAB possesses another type of number: the rational power of a rational number. It will, for example, compute the value of \((81/16)^{t(2/3)}\) to be \(9/4\cdot (3/2)^{t(2/3)}\).

Internal Representation of Mathematical Expressions

The internal representation of any mathematical expression is a well-formed LISP S-Expression in a prefix notation. If the user types:

denote nil

\( v = t^2 + \sin(pi\cdot t), \)

then there is stored on the property list of the atom “v” the property EXPRESSION followed by the S-Expression:

\((PLUS (EXPT T 2) (SIN (TIMES PI T)))).\)

Should the user then type:

differentiate(v t dv)

there is stored, upon completion of the differentiation and simplification, on the property list of the atom “dv” a pointer to the S-Expression:

\((PLUS(TIMES 2 T) (TIMES PI (COS (TIMES PI T))))).\)
This is translated back into the original infix notation and the typewriter prints:

\[ DV = 2T + \pi \cos(\pi T) \]

Equations are stored similarly to expressions, except there are two pointers, one to the S-Expression representing the left side of the equation and one to the right. Functions are stored by having the indicator FUNCTION point to the listed pair consisting of the list of dummy variables and the value of the function represented by the corresponding S-Expression.

Besides the obvious need for well-formed LISP expressions, there are two reasons for our choice of this internal representation of mathematical expressions. First, this representation has become fairly standard and this allows us to exchange programs with other workers. Second, the prefix notation turns out to be well suited to our applications. Consider the differentiation we just discussed. Probably the easiest and fastest thing for LISP to tell us about any list is the first item on it: in this case, PLUS. But this is precisely the first thing our differentiation program would want to know so as to invoke the rule that the derivative of a sum is the sum of the derivatives. Both the input (infix→prefix) and the output (prefix→infix) translation programs are written in LISP, the former employing the character-reading functions.

Other internal representations of expressions also occur, e.g., polynomials as lists of coefficients and rational functions as dotted pairs of lists of coefficients. All such alternative representations have translation programs connecting them in both directions with the standard prefix representation.

Simplification and Differentiation

The internal programs for simplification and differentiation have been borrowed from the Stanford Simplify Program. They have been modified in two ways. Simplify has been enlarged to handle a family of simplifications typified by the transformation: \((\text{EXPT (MINUS X) 4/3}) \rightarrow (\text{EXPT X 4/3})\), i.e., \((-x)^{4/3} \rightarrow x^{4/3}\). This simplification was impossible in the original Stanford system because \(4/3\) could only be represented by an approximating decimal, such as 1.3333333 and nothing can be done with \((-x)^{1.333333}\).

Differentiation has been modified so as to look to the typewriter for the derivatives of new functions. If necessary, the program will demonstrate, by example, the correct format for teaching it derivatives.

Integration

The program for the integration of rational functions with numerical coefficients was written by M. Manove at MITRE in the summer of 1964. It is based on a theorem of Hardy that states that the integral of such a function is of the form \(R_2 + \int R_1\) where \(R_1\) and \(R_2\) are rational and \(R_1\) has only simple poles. The program always finds \(R_1\) and does the best it can with \(\int R_2\), that best depending on its ability to factor the denominator of \(R_2\). It is sufficient to consider \(R_1\) monic with integral coefficients. To factor \(R_2\), “integrate” first calls a simple program written by the current author which, after finding all rational (hence, integral) roots and factoring them out, will, if left with a quartic, factor it into the product of quadratics with integral coefficients (should such a factorization exist). In addition to this factorization program, “integrate” memorizes the factors of the denominator of the original problem (if that denominator is presented in factored form) and uses these as trial divisors of \(R_2\).

“Integrate” is powerful enough to have found several errors in published tables of integrals.

Solve

“Solve” first brings the equation it has to solve over on one side. It then combines the various terms into a single rational expression with one numerator and one denominator, employing greatest common divisor routines (for polynomials with symbolic coefficients) to eliminate common factors from numerators and denominators. The roots of the original equation are then the roots of the numerator of the constructed rational function. If that numerator is quadratic, its roots are found by the quadratic formula. The vanishing of the discriminant is the test for a double root. We are, of course, dependent on the simplification programs here and within the g.c.d. routines to tell us when complicated S-expressions are equivalent to zero.

CURRENT WORK—September 1, 1965

Current work involves new representations of polynomials and rational functions in several variables with applications to simplification, the factorization of polynomials in several variables, and the integration of rational functions with symbolic coefficients.

In addition, we are working on programs for the display of mathematical expressions on scopes and the adaptation of MATHLAB to the AN/FSQ-32 computer at the Systems Development Corporation, Santa Monica.
REFERENCES


From the collection of the Computer History Museum (www.computerhistory.org)