CAPABILITIES OF THE MUMATH-79 COMPUTER ALGEBRA SYSTEM FOR THE INTEL-8080 MICROPROCESSOR

Albert D. Rich
and
David R. Stoutemyer
Electrical Engineering Department
University of Hawaii at Manoa
Honolulu, Hawaii 96822

ABSTRACT

This paper describes the capabilities of a microcomputer algebra system intended for educational and personal use. Currently implemented for INTEL-8080 based microcomputers, the system offers a broad range of facilities from indefinite-precision rational arithmetic through symbolic integration, symbolic summation, matrix algebra, and solution of a nonlinear algebraic equation. The talk will include a filmed demonstration, and informal live demonstrations will be given afterward.

1. INTRODUCTION

Computer algebra has been a successful though little-known research tool for some years now. Many researchers who could benefit from computer-algebra are ignorant of its existence or only vaguely aware of its existence. This fact is perhaps most attributable to the almost total lack of exposure in the educational system, even at the graduate level. Even the few schools in the world where an internationally recognized computer-algebra research effort is underway each tend to have at most 1 or 2 faculty members and perhaps 3 to 4 graduate students specializing in the area.

Concerned about this clear neglect of a valuable resource, we set about to infuse computer-algebra awareness and experience throughout the math-science curriculum, starting at the undergraduate level and working downward. We quickly discovered that although the current computer-algebra systems are highly suitable for funded research, these large systems are too expensive for use by masses of undergraduate and pre-university students.

For this reason we developed a system intended primarily for educational use, using one of the smallest existing microprocessors. To our pleasant surprise the breadth, speed, and expression capacity of the system significantly exceeds our initial expectations, with space still left for implementing additional facilities.

Although the system was motivated by educational needs, we now believe that, analogous to hand-held numeric calculators, the system is also useful in a research role, for performing or checking small to medium-size analytical derivations.

2. BUILT-IN FACILITIES

muMATH is organized into an incremental hierarchy of packages so that users can save space by loading only the packages they need. Here is a brief summary of what the basic packages provide:

2.1 An Operating System

An operating system is needed to provide interactive console I/O. If a user wishes to save programs or expressions as text on an external storage medium or to load such files, then the operating system must also perform batch I/O for sequential files. So far, we have versions of muMATH which interface to the popular CP/M^{tm}, IMSAI IMDOS^{tm}, and Cromemco CDOS^{tm} operating systems.

2.2 muSIMP-77^{tm}

muSIMP-77 is a variant of muLISP-77^{tm}, both developed by Albert Rich. muSIMP provides bignum arithmetic, dotted-pair data structures, data-base property-list primitives, automatic garbage-collection storage management, an extendable parser-deparser, general structured control constructs, and a sweetened LISP-like evaluation mechanism which are directly suitable for bootstrapping a wide variety of computer-algebra system designs. muSIMP also provides commands for opening and closing input or output files so that input or output can occur with a storage medium instead of a terminal.

2.3 Algebra Package

The algebra package provides algebraic simplification for the operators "+", "-", "*", "/" and "↑". The internal representation is Cambridge prefix, and the simplifier is of the liberal type. Simplifications which a user may or may not desire are under his control via the settings of several option variables. For example, the distribution of factors in a numerator over a sum in a numerator is controlled by a variable named NUMRNUMR. When this variable is a positive multiple of 2 then integer factors are distributed over sums. When NUMRNUMR is a positive multiple of 3 then other non-sum factors are distributed over sums. When the variable is a positive multiple of 5, then sum factors are distributed over sums. Thus, the user can independently specify the desired expansions by assigning NUMRNUMR the appropriate multiple of one, two, or three of these primes. Conversely, when NUMRNUMR is a negative multiple of 2 or 3, numeric or non-numeric content are respectively factored from SUMS. None of the above transformations are done when NUMRNUMR is 0.

A variable named DENRDENR similarly controls the distribution or factoring of denominator factors with respect to sums in denominators. Moreover, a variable named DENRNUMR similarly controls the distribution or factoring of denominator factors with respect to sums in numerators: Positive values request the desired types of denominator factors to be distributed over the terms of a numerator, whereas negative factors request the desired type of common denominator.

To complete this suite of control variables, there is one named NUMRDENR which similarly controls the distribution or factoring of numerator factors with respect to sums in denominators: Positive settings request a sort of continued-fraction expansion whereas negative settings request a selective denesting of nested rational expressions.

Multinomial expansion is separately controlled by a variable named PWREXPD: A positive multiple of 2 causes such expansion in numerators, whereas a positive multiple of 3 causes such expansion in denominators.

The distribution or "factoring" of a base over an exponent which is a sum, $u\uparrow(v+w)\rightleftarrows u\uparrow v*u\uparrow w$, is selectively controlled by a variable named BASEDIST. Again, multiples of 2, 3, and 5 correspond to integers, non-sums, and sums for u. Finally, the distribution or "factoring" of an exponent over a base which is a product, $(v*w)\uparrow u\rightleftarrows v\uparrow u*w\uparrow u$ is similarly controlled by a variable named EXPTDIST.

Other simplifications provided by the algebra package include

1. The use of integer factorization together with Newton's method to simplify fractional powers of numbers.
2. The reduction of powers of i.
3. The reduction of complex exponentials.

muSIMP provides an upward-compatible auto-quote variant of the LISP 1-level EVAL function, which serves directly as the algebraic simplifier. Thus, we did not waste precious space implementing an algebraic evaluator in terms of a LISP evaluator, and the body of the driver loop is essentially DEPARSE(EVAL(PARSE(...))). EVAL is also available to the user, who may wish to use it occasionally to resimplify a result under the influence of intervening assignments to option variables or to indeterminates within the result.

To resimplify under the influence of a local substitution, the user can use the function named EVSUB, which reevaluates a copy of its first argument wherein every syntactic instance of its second argument is replaced by its third argument. The second argument can be any expression, including a sum or product, and substitution takes place only if the internal representation of the second argument is identical to some part of the internal representation of the first argument.

2.4 Logarithmic Package

The base of the natural logarithms is represented as #E, so the logarithmic package implements the rule #E↑LN(u)→u, for all u. Inversely, provided the variable named PBRCH is TRUE, the package performs the transformations LN(1)→0, LN(#E)→1, and LN(#E↑u)→u for all u. More drastic transformations are controlled by a variable named LNEXPD. When this variable is a positive multiple of 2, then logarithms of powers are expanded. When this variable is a positive multiple of 3, then logarithms of products are expanded. Negative multiples of 2 or 3 cause the opposite transformations.

2.5 Trigonometry Package

The trigonometry package provides simplifications for the sine, cosine, and arctangent functions, with π represented as #PI. Automatic simplifications include exploitation of symmetry and exact evaluation for angles which are multiples of $\pi/6$. More drastic transformations are controlled by a variable named TRIGEXPD: A positive multiple of 2 requests multiple-angle expansion, whereas a positive multiple of 3 requests angle-sum expansion. Negative multiples of 2 or 3 request the corresponding inverse transformation, which together provide a Poisson-series capability.

2.6 Calculus Package

The calculus package provides differentiation and integration. The latter uses linearity of the integral operator together with derivatives-divides rules for all of the built-in functions.

2.7 Summation Package

The summation package provides a function

$$\text{SIGMA}(u_j, j, m, n) = \sum_{j=m}^{n} u_j \ .$$

SIGMA uses iteration when m and n are numeric. Otherwise, SIGMA uses a variety of elementary techniques, including linearity of the summation operator, telescoping sums, reduction formulas, and a few known sums. Thus, for example, this function is able to determine a closed-form representation for

$$\sum_{k=1}^{n} (2^{3k-c} + ak^3) \ .$$

The package includes an analogous function named PROD for products.

2.8 Matrix Package

Row vectors are delimited by square brackets, and column vectors are delimited by curly brackets. A matrix is a row of columns or a column of rows. A partitioned vector is a row of rows or a column of columns. The elements of vectors or matrices can be arbitrary symbolic expressions, including vectors or matrices nested to any level. For element-wise operations or for the inner product of a row vector with a column vector, the shorter of the two vectors is regarded as having implicit trailing zeros. Thus, matrices can be triangular or "ragged" in other ways.

Besides matrix multiplication and inversion, the package provides a matrix "division" operator, \, such that $A\backslash B$ is equivalent to but more efficient than $A^{-1} \cdot B$ when A^{-1} exists. Moreover, $A\backslash B$ yields a result containing introduced extra parameters when A^{-1} does not exist but A is consistent with B.

2.9 Equation Package

The equation package provides parsing and simplification for the operator "=". Thus, equations are regarded as legitimate expressions, distinct from assignments or pattern-match rules. Equations can be assigned, added, multiplied, squared, etc. Moreover, there is a function named SOLVE which uses the factorization settings of the control variables together with known inverses of powers, quadratics, etc., in order to determine the exact solutions of equations such as $x\ln^4(x-a)+6x=5x\ln^2(x-a)$.

2.10 Programming Package

Although we discuss it last, the programming package is loaded immediately after muSIMP so that all of the math algorithms are written in the same high-level syntax available to the user for his programs. Thus, the user need not master two languages in order to understand or fully utilize all of the underlying algorithms. Also, we expect that many users will find that muSIMP together with the programming package provide an ideal base from which to implement algebra systems based on entirely different internal representations, or to implement entirely different non-algebraic applications.

The user can extend the built-in facilities by providing function definitions, simplification rules, or operator definitions. For example, here is a definition of a function which uses straightforward differentiation and substitution to generate an Nth degree Taylor series expansion of expression EXPN with respect to indeterminate X about point A:

```
FUNCTION TAYLOR(EXPN,X,A,N,
          J,C,ANS),
    J ← ANS ← 0,
    C ← 1,
    LOOP
        ANS ← ANS + C*EVSUB(EXPN,X,A),
        WHEN J=N, ANS EXIT,
        EXPN ← DIF(EXPN,X),
        J ← J+1,
        C ← C*(X-A)/J,
        ENDLOOP
    ENDFUN;
```

An example of the use of this function is TAYLOR(#E↑X,X,0,5) which takes 4 seconds to evaluate on an 8080. As illustrated by the function definition

1. Unused parameters are available for use as local variables, and parameters are generally of call-by-value type.
2. A function body consists of a sequence of expressions separated by commas, and the returned value is that of the last expression evaluated when the definition is applied to specific arguments.
3. A value is returned by every command and control construct, including assignments and loops.
4. A <u>conditional exit</u> consists of the <u>matchfix</u> operator named WHEN, followed by one or more expressions separated by commas, followed by the matching delimiter named EXIT. The value of a conditional exit is that of the last expression evaluated therein when the conditional exit is evaluated. If the first expression evaluates to FALSE, then the exit fails and evaluation proceeds sequentially to any successive expressions following the conditional exit. For a successful exit, when evaluation first reaches an EXIT delimiter, it proceeds to the point following the next ENDFUN, ENDLOOP, or ENDBLOCK delimiter.

Not illustrated here is the <u>block</u> control construct, which consists of the matchfix operator named BLOCK, followed by a conditional exit, followed by zero or more arbitrary expressions, followed by the matching delimiter named ENDBLOCK. The value of a block is that of the last expression evaluated therein when the block is evaluated. Since a block can have any number of conditional exits interspersed among other expressions, it provides a structured generalization of the case-statement of some other languages, which includes the IF-THEN-ELSE construct as a special instance. Similarly, since a loop expression can contain any number of conditional exits anywhere in the loop, we have a structured loop construct which contains the WHILE, REPEAT, and half-loop constructs of some other languages as special cases.

Although function definitions bear a resemblance to their counterparts in traditional programming languages, the generalizations are inspired by a desire to support and encourage structured programming in either a Von Neumann or an applicative style, with a few simple control constructs and with trivial parsing from the external to internal representations of these constructs.

muMATH actually implements most of its transformations via a sort of "poor man's" pattern matcher, using property lists. Our original intent merely was to increase modularity by permitting simplification rules to be distributed among appropriate packages rather than collected into monolithic function definitions. However, users also can employ either or both techniques to implement their applications.

The programming package uses an incremental Pratt parser, which the user easily can extend to help implement his applications: Every operator can have a left and/or a right binding power. An operand between two operators is acquired by the operator having greater binding power toward the operand. There are generous gaps between the binding powers of the built-in operators, so it is easy for a user to insert new operators. For example, 170 is a rather high binding power. Consequently, to establish SIN as a prefix operator so that we can omit parentheses from around suitable arguments of SIN, we can enter the command PROPERTY SIN, PREFIX, 170.

Besides mathematical operators, parser extensions can include programming constructs, so that the user can tailor the programming syntax to suit his personal tastes.

3. PERFORMANCE

How complicated can expressions be before muMATH-79 exhausts the available storage space or the user's patience? The answer depends strongly upon the particular expressions, the particular transformations, and the amount of space available for storing expressions. The answer also depends strongly upon individual patience: If a computer would otherwise go unused, an overnight or weekend computation may be acceptable; but for us, one minute is the acceptable order-of-magnitude time limit for an interactive computation.

To give some indications of interactive performance: in one minute on an 8080 running at 2 megahertz with 48 kilobytes, the system can expand 290!, $(1+x)^{20}$, $\sin(16x)$, $(x_1+x_2+\ldots+x_9)^2$, or $\sin(x_1+x_2+\ldots+x_5)$.

4. HISTORY AND DISTRIBUTION

Albert Rich developed muLISP-77 for his IMSAI 8080 computer, and David Stoutemyer prevailed upon him to make a variant more suitable for computer algebra implementation. In 1978, Albert Rich joined an NSF-sponsored research project,* and together we implemented the algebra system in this LISP variant called muSIMP-77. The source listing of the algebra system, written in muSIMP-77, is public domain, and it has been submitted for publication. That listing may be freely copied, adapted, and sold, provided it is not called muMATH if altered whatsoever. (We would, of course, appreciate a reference to this article for any system derived from muMATH.) In contrast, muLISP-77 and muSIMP-77 are copyrighted.

We expect the demand for these software systems to be too large to fill on an informal basis, using university or NSF resources. Consequently, we established a company named The Soft Warehouse to handle the distribution, maintenance, and further development of the systems. In keeping with the educational objectives of this software, the distribution charges will be set very low. The names muLISP, muSIMP, and muMATH are trademarks of The Soft Warehouse.

The address of The Soft Warehouse is P.O. Box 11174, Honolulu, Hawaii 96828.

*Grant MCS7802234