A Preview of the Next IBM-PC version of muMATH

David R. Stoutemyer
Soft Warehouse, Inc.
P.O. Box 11174, Honolulu HI 96822 USA

ABSTRACT

A complete redesign of muMATH is nearing completion, and this paper previews some of the new capabilities, which include the following:

1. Rational arithmetic can be either exact or automatically rounded to any precision, thus providing a unified alternative to the usual combination of exact rational plus arbitrary-precision floating-point arithmetic.

2. There is function plot graphics.

3. Two-dimensional output of expressions is provided, including raised exponents and built-up fractions.

4. The algebraic capabilities are substantially improved:

   a) Simplification is more automatic and thorough, with options that are easier to use.

   b) The default normal form tends to preserve factors at all levels while automatically achieving significant simplification of rational expressions, radicals and elementary transcendental expressions.

   c) There are polynomial expansion, gcd and factoring algorithms with speeds that compare favorably with those of systems running on mainframes or Lisp machines.

1. INTRODUCTION

The intent of muMATH has always been to provide computer algebra on the most popular and inexpensive feasible personal computers using the most popular operating system. Thus, muMATH was originally designed for the Intel 8080 chip together with the CP/M-80 operating system, followed by the Apple II family. Subtracting memory space used by the operating system, these two families typically left at most 60 kilobytes of read-write random-access memory for muMATH together with the muSIMP interpreter in which muMATH is implemented.

When the larger address space microprocessors first appeared, it was unclear which if any would predominate and thus be most appropriate for the intended muMATH audience. An inexpensive S-100 dual processor board made the Intel 8088 processor family particularly attractive for development, so we implemented our software (including the separate muLISP) for that family. IBM's subsequent choice of that family turned out to make it the most popular one -- at least in the USA.

Subtracting space used by the operating system, the IBM-PC family and its many imitators typically provide a maximum amount of read-write random access memory ranging from 200 to 500 kilobytes. This is enough to accommodate many desirable capabilities that could not fit in 60 kilobytes. Also, experience implementing and using the original muMATH has suggested ideas for a much better system. Thus, I have been engaged in a complete redesign that should be ready for distribution late this year by the time this proceedings appears. Consequently, this paper previews some of the novel features of the new design rather than reviewing the currently distributed version.

2. FLOATING SLASH ARITHMETIC

muSIMP has always had exceptionally fast infinite precision integer arithmetic. Empirically, the seconds required to multiply two numbers each having d decimal digits is about $0.9\ 10^{-6}\ d^2$ asymptotically, and the time to divide their product by either of them is about 25% slower. As examples entailing a sequence of multiplications, interpreted factorial and "^" functions use 0.4 seconds to compute 300! and 0.5 seconds to compute 9999^300. These and all other muSIMP/muMATH times in this article are for the IBM-AT. Times for the IBM-PC and XT are slower by a factor of about 2.5.

In the new design, infinite-precision rational numbers are stored as an atom consisting of two integers, with the corresponding arithmetic routines also written in machine language. Empirically, the time required to determine the gcd of two integers varies from the time of dividing them up to about 5.6 times as long as to multiply them. Relatively prime integers take longest.

An arbitrary rounding level can optionally be specified for the rational arithmetic: If this level is set to a positive integer m, then whenever the lesser of the number of words in the numerator and denominator of a reduced fraction exceeds m, the number is "mediant" rounded to the closest reduced fraction that does not exceed this limitation. Rounding of a reduced fraction to slightly less than its given precision asymptotically requires about three times as long as determining the unit gcd of the numerator and denominator. Rounding to lesser precision requires approximately proportionally less time.

The user can independently specify a positive integer underflow level u, causing fractions with magnitude less than $2^{-16u} \approx 10^{-4.5u}$ to be rounded to zero. Thus, u=7 corresponds to the underflow magnitude of about 10^{-37} that is typical of many floating point implementations.

The rounding and underflow options permit the user to limit the number of significant digits and the magnitude of small non-zero fractions, thus providing an alternative to the usual arbitrary-precision floating-point arithmetic provided in most other modern computer algebra systems.

The arithmetic described here is essentially a conservative arbitrary precision variant of the **floating slash** arithmetic described by Matula and Kornerup [1985], for which:

1. The **expected** relative rounding error is about 0.5 10^-9 m, which is about nine significant decimal digits per precision level m. This is more space efficient than floating point for numbers having magnitude close to 1. However:
2. The relative gaps between numbers is much less uniform than for floating point: The **worst case** relative rounding error is half as many significant digits as the expected level -- about 4.5m. This would be alarming, except that:
3. Only about 10^{-n} of the roundings lose as much or more than n significant digits out of the expected 9m, down to the worst case of losing 4.5m digits. For example, only about 1/10 of the roundings will lose one or more digits and only about 1/10000 of the roundings will lose four or more digits. Thus for a given number of operations, as the requested precision increases, the chance of losing a significant **percentage** of the digits becomes negligible.

This arithmetic is particularly good for computer algebra because:

1. It requires no additional data type and very little additional code over that required for the infinite precision rational arithmetic anyway.
2. It provides a natural transition from exact to approximate arithmetic without any need for a burdensome arsenal of type declarations, type conversion functions, and type coersion rules.
3. The relative gaps between successive representable numbers are largest for simple fractions and smallest for fractions having large denominators. Thus when an exact result is rational and exactly representable within the chosen precision and underflow levels, but intermediate growth or irrational functions make it impractical or impossible to use exact rational arithmetic, then mediant rounding often yields the exact rational result despite the approximate intermediate results. As examples, with precision m = 1 and underflow level u = 7:

$$\frac{(5/1029)^{1/3} - (2/1029)^{1/3}}{(7*20^{1/3} - 19)^{1/6}} \quad \text{--->} \quad \frac{4}{7},$$

$$\frac{\text{pi}}{42\ \text{atan}\ (1/8) + 14\ \text{atan}\ (1/57) + 7\ \text{atan}\ (1/239)} \quad \text{--->} \quad \frac{1}{7}.$$

In the case of approximate real data such as computing statistics for experimental data, the bias toward simple fractions is not worth the associated nonuniform precision. Moreover, when used at precision levels of m=1 and m=2, this **arbitrary** precision rational arithmetic cannot compete in speed with floating point of fixed single and fixed double precision hardware. However, beyond about 14 digits, approximate arithmetic is almost invariably associated with theoretical calculations on exact mathematical inputs, and exact outputs would be preferred if possible -- at least if they are concise. In this context, the approximate arithmetic is regarded as a necessary evil rather than an efficiency ploy that is justified by consistency with experimental error in the input data. Consequently, the enhanced capturing of exact rational results due to less uniform gaps between representable numbers rational numbers is a benefit rather than a liability for typical computer algebra applications.

Approximate computation of fractional powers, pi, exponentials, trigonometric functions and their inverses is programmed in muSIMP,

using variants of the usual argument reduction and iterative or series techniques described by Fateman [1976] and Sasaki [1979]. Asymptotically, the computing time grows quadratically with precision. Table 1 below gives various timings for m = 11 and m = 22, together with corresponding timings for MACSYMA using Franz LISP on a DEC VAX 780. The rows have **not** been mislabeled -- The PC-AT' s implementation is generally much faster on such examples. The entries were computed left to right so that the values of e, pi and ln 2 were not recomputed when needed for argument reduction. The values of n, d and r listed in the headings are n = 123456789, d = 987654321, and r = n/d,

requested digits	e	pi	ln 2	exp r	ln r	sin r	atan r	r^{99}	$d^{2/3}$	$d^{1/n}$
PC-AT: 99:	0.3	0.4	0.9	0.7	0.7	0.9	0.9	0.8	0.5	1.5
VAX780	8.0	3.0	30.0	6.7	10.3	6.3	7.6	0.2	41.0	24.0
PC-AT: 198:	1.0	1.2	3.6	1.6	1.3	2.5	3.1	1.6	1.4	5.8
VAX780	29.0	4.0	94.0	19.0	31.0	18.0	29.0	0.3	5.0	81.0

Table 1: Computing times in seconds, for floating slash

Table 2 shows the **actual** numbers of significant digits in the above examples for the PC-AT, determined by comparison with more precise results. As illustrated there, the expected accuracy is actually about 9.5 significant digits per precision level m:

requested digits	e	pi	ln 2	exp r	ln r	sin r	atan r	r^{99}	$d^{2/3}$	$d^{1/n}$
99	104	106	104	104	105	104	104	104	106	105
198	212	211	209	211	209	210	209	211	207	211

Table 2: Achieved significant digits for floating slash

3. FUNCTION PLOTTING

Graphics protocols were extremely varied for the 8-bit CP/M-80 machines, but the market share of the IBM-PC and its imitators makes their graphics a de facto standard for the vast majority of machines based on the 8088 processor family. This together with the floating slash arithmetic has made it feasible to provide 200 row by 320 or 640 column raster function plotting in the forthcoming muMATH. So far I have implemented only plotting an explicit function of the form y(x). For the sake of those who do not have graphics cards, I have implemented a version that uses the half-height solid rectanges of the (de facto standard) IBM-PC extended character set to yield a 50-row by 80 column "character-plot" resolution on the 25-line by 80 column screen. I plan to implement also parameter plots of the form [x(t), y(t)] and wire-frame perspective plots for functions of two variables having the form z(x,y), with hidden line removal.

4. POLYNOMIAL EXPANSION

A particularly crucial system design decision is the choice of polynomial representation or representations. This choice has a profound effect on not only the time and space efficiency, but also the class of representable expressions, the appearance of the output, and the suitability for higher-level algorithms such as integration or Grobner bases. For the sake of simplicity in design and use, it would be nice if a single representation was uniformly best or at least reasonably acceptable in these regards. However, given the luxury of sufficient program space, it appears worthwhile to use more than one representation. The challenge is to make them work together in an automatic or semiautomatic manner that is natural and easy to use.

The **f and g series** provide one classic real-life test for polynomial expansion, differentiation and infinite precision arithmetic: f_n and g_n are expressions for coefficients in a power series used for for elliptic path prediction in celestial mechanics. Defined by mutual recurrence relations, f_n and g_n are sparse trivariate polynomials. The degrees, the number of terms, and the sparsity grow with n, while the integer coefficient magnitudes grow even more dramatically with n. Barton and Fitch [1972] give comparative timings for mainframe algebra systems. In the muSIMP environment, distributed form with packed exponents turned out to be particularly fast and space efficient for this problem. In the semilog plot of Figure 3,

the asterisks indicate the times for this representation, whereas each isolated letter indicates a time for a different mainframe system:

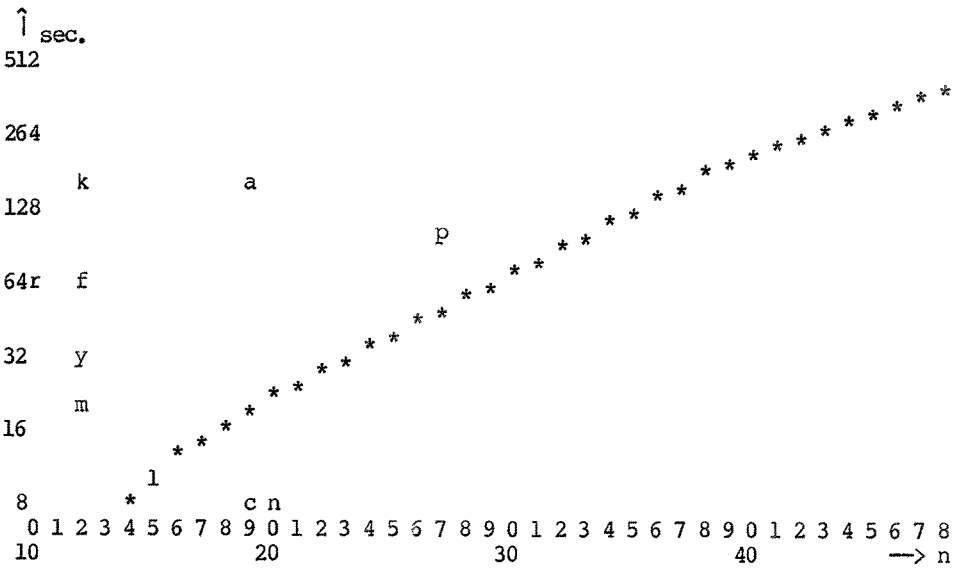

Figure 1: Cummulative times for f & g series through order n

There was insufficient memory for me to do n = 49, but it was not stated which of the mainframe data represented their highest achievable order. It is important to note for all of the comparisons in this paper that the various mainframes varied in power, that the systems generally reside on more powerful mainframes now, and that some of the implementations have since been made more efficient. The only point that I wish to make with these comparisons is that with appropriate algorithms and data structures, the muSIMP IBM-PC environment is suited for serious large-scale symbol crunching.

The Y_{2n} polynomials provide another classic sparse polynomial test. In contrast to the f and g series, the number of terms increases more rapidly and the number of variables also increases with n. Consequently, although the coefficients also grow rapidly with n, the arithmetic speed is much less relevant than for the f and g series. Y_{2n} are defined by recurrence relations that arise from a series solution to a certain second order differential equation, as described by Campbell [1972]. Despite the extreme sparsity of these polynomials, the **dense recursive** representation described by Stoutemyer [1984a] turned out to be particularly fast and space efficient. For this problem, the variety of solution methods caused as much spread as differences in machines and systems. I used solution method 3

described by Bourne [1972]. The entire issue of the journal containing that paper is devoted to a comparison of mainframe solutions by various methods, as summarized by the semilog plot of Figure 2:

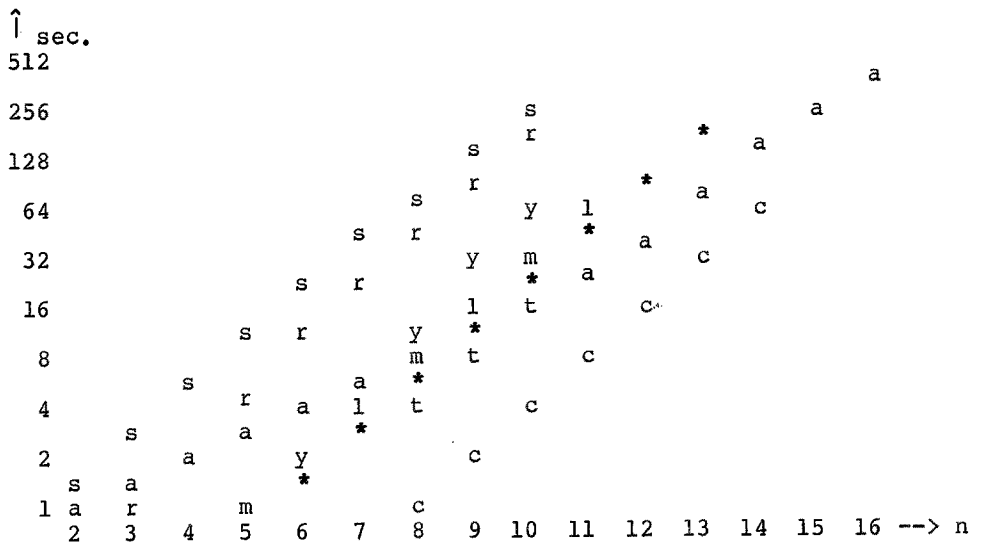

Figure 2: Incremental computing times for Y_{2n}

There was insufficient memory for me to do n = 14. The problem challenge was to compute through n = 10, but it was not always stated which of those that went further went as far as possible.

5. POLYNOMIAL FACTORING

Using a sparse recursive representation, Mike Lucks, now at IBM Thomas J. Watson Research Center, has implemented a polynomial factoring over the integers. I leave it to him to publish the techniques, but to whet your appetite, Table 3 shows a comparison with the DEC KL10 MACSYMA timings published by Wang [1978]:

eg:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
KL10	1	1	1	5	6	3	1	8	5	9	6	0.3	1	28	1
PC-AT	5	2	3	16	11	19	4	77	29	22	32	0.4	5	13	9

Table 3: Polynomial factoring times, in seconds

6. POLYNOMIAL GREATEST COMMON DIVISORS

Stoutemyer [1985b] describes improved methods of computing polynomial gcds using remainder sequences and the same sparse recursive representation used for the above factoring examples. Using an asterisk to indicate space exhaustion, Table 4 shows a comparison with the times published by Zippel [1979] for various algorithms implemented in MACSYMA on a DEC KL10.

mach.	algm.	v = 1	2	3	4	5	6	7	8	9	10
KL10	EZ	0.04	0.3	0.4	1	3	*	*	*	*	*
"	Red.	0.05	0.2	0.5	2	*	*	*	*	*	*
"	Mod.	0.05	0.3	0.9	8	65	484	2409	*	*	*
"	EEZ	0.04	0.4	0.5	1	2	2	2	*	*	*
"	SprsMod	0.04	0.2	0.4	1	2	3	4	5	4	8
PC-AT	New PRS	0.32	1.8	5	12	15	35	57	73	96	116

Table 4: Polynomial gcd times, in seconds

Although the Sparse Modular algorithm is the clear winner for this type of example, the remarkable fact is that despite its simplicity, the new PRS algorithm is the only other one that completes all of the examples without running out of space.

7. RECURSIVELY FACTORED FORM

Brown [1974] and Hall [1974] make a compelling case for a default normal partially factored representation. They used a top-level product of fully-distributed polynomials. However, recursive forms afford even greater opportunities for partial factorization, because coefficients may then remain factored even when a polynomial must be expanded at the top level. I have implemented such a recursive partially factored form, but using LISP prefix rather than the implicit operator dense and sparse representations mentioned in sections 4 through 6 above. The implementation was significantly more complicated than for expanded forms, but the benefits are enticing.

Brown and Hall argue that the main benefit is for rational functions. However, there can be substantial benefits even for purely polynomial problems: As test cases, I computed the determinant of many families of matrix examples, using both the ordinary top-down

minor expansion and the bottom-up minor expansion described by Horowitz and Sahni [1975] and by Gentleman and Johnson [1976]. For polynomial entries, neither of these methods cause polynomial divisions or gcds. The test cases were taken from the computer algebra literature and from Muir [1960]. A surprising percentage of the determinants have nontrivial factorizations even at the top level, and the partially factored form delivered a surprising percentage of full or at least partial factorizations: Of 29 examples having nonmonomial determinants, 23 had nontrivial factorizations. Of these, the partially factored form delivered 8 complete factorizations and 3 partial factorizations. The 8 complete factorizations occurred for the following families:

$$\begin{vmatrix} 1 & a & a^2 & \dots \\ 1 & b & b^2 & \dots \\ 1 & c & b^2 & \dots \\ & \dots & & \end{vmatrix} = (a-b)(a-c)\dots(b-c)\dots$$

(Vandermonde)

$$\begin{vmatrix} 1 & -1 & 0 & & \dots & 0 \\ x & h & -1 & 0 & \dots & 0 \\ x^2 & hx & h & -1 & 0 \dots & 0 \\ x^3 & hx^2 & hx & h & -1 & 0 \dots 0 \\ & & & \dots & & \\ x^n & hx^n & hx^{n-1} & & \dots & h \end{vmatrix} = (x+h)^n$$

(Muir [1960, p. 700])

$$\begin{vmatrix} x & a & b & c & \dots & 1 \\ a & x & b & c & \dots & 1 \\ a & b & x & c & \dots & 1 \\ a & b & c & x & \dots & 1 \\ & & \dots & & & \\ a & b & c & \dots & & 1 \end{vmatrix} = (x-a)(x-b)(x-c)\dots$$

(Muir [1960, p. 63])

$$\begin{vmatrix} a & a & a & \dots & a \\ a & b & b & \dots & b \\ a & b & c & \dots & c \\ a & b & c & \dots & d \\ & & \dots & & \\ a & b & c & d & \dots \end{vmatrix} = a\,(b-a)(c-b)(c-a)(d-c)(d-b)(d-a)\dots$$

(Muir [1960, p. 46])

$$\begin{vmatrix} 1 & 1 & 1 & 1 & 1 \\ -x & 0 & a & b & c \\ -x & -a & 0 & d & e \\ -x & -b & -d & 0 & f \\ -x & -c & -e & -f & 0 \end{vmatrix}_{n \text{ by } n} = \begin{cases} (\ \ldots\ )^2 & \text{when } n \text{ is odd, or} \\ x\,(\ \ldots\ )^2 & \text{when } n \text{ is even.} \end{cases}$$

(Muir [1960, p. 398)

$$\begin{vmatrix} A & 0 \\ -I & B \end{vmatrix} = |A|\,|B|, \quad \text{with A \& B general square submatrices,}$$

(Muir [1960, p. 152)

$$\begin{vmatrix} A & B \\ C & D \end{vmatrix} = (a_1 d_1 - c_1 b_1)(a_2 d_2 - c_2 d_2) \ldots (a_n d_n - c_n d_n),$$

where A, B, C & D are general diagonal matrices having diagonal elements a_1, a_2 ... a_n etc. (Muir [1960, p. 174])

$$\begin{vmatrix} P & Q & 0 & \ldots & 0 \\ 0 & P & Q & 0 & \ldots & 0 \\ 0 & 0 & P & Q & 0 & \ldots & 0 \\ & & & \ldots & & & \\ 0 & & \ldots & 0 & P & Q \\ 0 & & & \ldots & 0 & P \end{vmatrix}_{2^m \text{ by } 2^m} = [(c+b-a)c + (b-a)b - a^2]^{2\hat{}m},$$

where $P = \begin{bmatrix} c+b & c+a \\ b+a & c+b \end{bmatrix}$, $Q = \begin{bmatrix} 0 & 0 \\ c+a & 0 \end{bmatrix}$,

(Wang [1977])

The polynomial arithmetic can optionally check for factors revealed only by divide checks. Only the Vandermonde example required the divide check option, the others requiring no more than checks for similar factors.

Unfortunately, for problems that have little or no partial factorization (such as the f & g series or Y_{2n} polynomials), the LISP prefix recursive factored form can be an order of magnitude slower than the implicit operator representations used for the examples of sections 4 through 6. In contrast, ALTRAN's implicit-operator factored form appears to cause only a minor speed degradation for polynomials that end up fully expanded. Consequently I am currently engaged in converting the factored representation to implicit operators.

8. SUMMARY

The next version of muMATH will be a full-spectrum tool for scientific computation, combining some traditional and novel graphics, numeric, and symbolic capabilities.

9. REFERENCES

Barton, D. and Fitch, J.P. [1972]: "Application of Algebraic Manipulation Programs in Physics", **Reports on Progress in Physics 35**, pp. 235ff.

Bourne, S.R. [1972]: **ACM SIGSAM Bulletin 24**, pp. 8-11.

Brown, W.S. [1974]: "On Computing with Factored Rational Expressions", **ACM SIGSAM Bulletin 31**, August, pp. 26-34.

Campbell, J.A. [1972]: "Problem 2 -- The Y_{2n} Functions, **ACM SIGSAM Bulletin 22**, pp. 8-9.

Fateman, R.J. [1976]: "The MACSYMA 'Big-Floating-Point' Arithmetic System", **Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation**, editor R.D. Jenks, pp. 209-213.

Gentleman, W.M. and Johnson, S.C., [1976]: "Analysis of Algorithms, A Case Study: Determinants of Matrices With Polynomial Entries", **ACM Transactions on Mathematical Software 2**, No. 3, September, pp. 232-241.

Hall, A.D. [1974]: "Factored Rational Expressions in ALTRAN", **ACM SIGSAM Bulletion 31**, August, pp. 35-45.

Horowitz, E. [1975]: "On Computing the Exact Determinant of Matrices with Polynomial Entries", **Journal of the ACM 22**, No 1, January, pp. 38-50.

Matula, D.W. and Kornerup, P. [1985]: "Finite Precision Rational Arithmetic: Slash Number Systems", **IEEE Transactions on Computers C-34**, No. 1, January 1985, pp. 3-18.

Muir, S.T. [1960]: **A Treatise on the Theory of Determinants**, Dover Press, N.Y.

Sasaki, T. [1979]: "An Arbitrary Precision Real Arithmetic Package in REDUCE", in **Symbolic & Algebraic Computation**, Lecture Notes in Computer Science # 72, editor E.W. Ng, Springer Verlag, Berlin. pp. 358-368.

Stoutemyer, D.R. [1984a]: "Which Polynomial Representation is Best?", **Proceedings of the 1984 MACSYMA Users' Conference**, editor E. Golden, General Electric, Schenectady, N.Y. USA, pp. 221-243.

Stoutemyer, D.R. [1984b]: "Polynomial Remainder Sequence Greatest Common Divisors Revisited", proceedings of **RSYMSAC, The Second International Symposium on Symbolic and Algebraic Computation by Computers**, RIKEN Institute of Physical and Chemical Research, Wako-shi, Saitama, 351-01 Japan, pp. 1-1 through 1-12.

Wang, P.S. [1978]: "An Improved Multivariable Polynomial Factorising Algorithm", **Math. Comp. 32**, pp. 1215-1231.

Wang, P.S. [1979]: "Matrix Computations in MACSYMA", **Proceedings of the 1977 MACSYMA User's Conference**, NASA CP 2012, pp. 435-446.

Zippel, R. [1979]: "Probabilistic Algorithms for Sparse Polynomials", **Symbolic & Algebraic Computation**, editor E.W. Ng, Lecture Notes in Computer Science 72, Springer-Verlag, pp. 227-239.