The ALPAK System for Nonnumerical Algebra on a Digital Computer — I: Polynomials in Several Variables and Truncated Power Series with Polynomial Coefficients

By W. S. BROWN

(Manuscript received April 17, 1963)

This is the first of two papers on the ALPAK system for nonnumerical algebra on a digital computer. This paper is concerned with polynomials in several variables and truncated power series with polynomial coefficients. The second paper will discuss rational functions of several variables, truncated power series with rational-function coefficients, and systems of linear equations with rational-function coefficients. The ALPAK system has been programmed within the BE-SYS-4 monitor system on the IBM 7090 computer, but the language and concepts are machine independent.

The available polynomial arithmetic operations are add, subtract, multiply, divide (if divisible), substitute, differentiate, zero test, nonzero test, and equality test. The speed of the system is indicated by the rule of thumb that one man-hour equals one 7090-second. The available space in core is usually sufficient for approximately 8000 polynomial terms.

Section I of this paper consists of a nontechnical description of the system and a brief glimpse into the future. Section II discusses several specific problems to which the ALPAK system has been applied. These two parts do not presuppose any knowledge of computers or computer programming. Section III describes the use and the implementation of the algebraic operations relating to polynomials in several variables and truncated power series with polynomial coefficients. The reader of Section III is assumed to be acquainted with the elements of FAP (FORTRAN Assembly Program) programming, including the use of macros, as described in a series of IBM publications, the latest of which is IBM 7090-7094 Programming Systems MAP (Macro Assembly Program) Language (Form Number C28-6311).

TABLE OF CONTENTS

I. NONTECHNICAL DESCRIPTION

1.1 *Introduction*

Many theoreticians devote a substantial portion of their time to the routine manipulation of algebraic expressions. It has long been recognized that digital computers are capable in principle of easing this burden. The ALPAK system, which is described herein and in a subsequent paper and has been programmed for the IBM 7090 computer, represents a significant start toward the practical implementation of that capability. It performs a limited set of operations — add, subtract, multiply, divide, substitute, differentiate, zero test, nonzero test, and equality test — on a limited class of expressions: rational functions of several variables and truncated power series with rational-function coefficients. It can also solve (by Gaussian elimination) systems of linear equations with rational-function coefficients. This paper is concerned with polynomials in several variables and truncated power series with polynomial coefficients. The generalizations indicated above will be discussed in a separate paper by B. A. Tague, J. P. Hyde, and the present author.

The ALPAK system is not a "sophomore imitator" or "elementary mathematics system." There are many elementary mathematical operations (e.g., the proving of trigonometric identities) which are beyond its present capabilities. However, when faced with problems within its range of capability, its speed (one man-hour $\approx$ one 7090-second) and

power (the available space in core is usually sufficient for approximately 8000 polynomial terms) are impressive.

Neither is the ALPAK system a "symbol manipulation system," because it views a polynomial as an array of coefficients and exponents rather than as a string of numbers, variable names, operation symbols, parentheses, and the like. This is the key to speed and power. Polynomials are stored in a nearly optimal manner, and polynomial operations are reduced to their essentials.

We have been speaking of polynomials and rational functions without being specific about the possible coefficient rings. The coefficients may be integers or they may be elements of any other integral domain for which arithmetic and input-output facilities are available. All operations on coefficients are performed by a small set of macros (user-defined instructions which expand into one or more machine instructions). These are currently defined for integers, but the user may redefine them to suit his own needs. (Of course this requires reassembly of the ALPAK subroutines.) The use of floating-point coefficients is not in keeping with the spirit of symbolic computing and should be avoided if possible. The occurrence of roundoff error causes zero to be nonunique and gives rise to a host of difficult problems which the author has not attempted to solve. It is usually feasible and desirable to replace the nonrational numbers which occur in an expression by literal symbols. These can be treated by the ALPAK system as variables. The result will then involve no roundoff error, and the dependence on these symbols will be explicitly displayed.

To maximize speed and minimize space, the coefficients and exponents of a polynomial are stored in a contiguous block, and the exponents are packed as specified in a user-provided format statement. The names of the variables are kept in the format statement and are referred to as infrequently as possible. Storage allocation is automatic and dynamic, so that the programmer can refer to a polynomial by name without worrying about its size, structure, or location.

In Sections 1.2 and 1.3 we shall discuss the canonical form for polynomials and the implementation of the various polynomial-arithmetic operations. Section 1.4 contains a very brief preview of the rational-function operations and an even briefer mention of some of our hopes for the future.

1.2 *Choosing a Canonical Form*

In the ALPAK system *every* polynomial in storage is *always* kept in a unique canonical form, which we shall describe. Every subroutine, ex-

cept the input and output subroutines, assumes that its inputs are in canonical form and produces its outputs (if any) in canonical form. On input a polynomial is put into canonical form, and on output it is left in whatever form it is found. Barring trouble, this will always be canonical form.

It is important to recognize that the "best canonical form" for a given class of expressions need not be an approximation to what human beings would call the "simplest form." In fact, the two concepts are in some respects opposite. The simplest form may be defined roughly as "that form which requires the smallest number of symbols." On the other hand, an approximate definition of the best form is "that form into which the general expression of the class can most easily be put." This latter definition clearly favors canonical forms in which expressions are expanded over those in which they are collapsed, because the collapsing of expressions tends to be difficult, while their expansion tends to be easy. For example, in the case of polynomials in several variables we must choose between an "expanded form" in which each polynomial is represented as an ordered sum of terms and a "factored form" in which each polynomial is represented as an ordered product of irreducible factors. In general, the factored form is more compact, but we must reject it because the factoring algorithm† can be extremely time consuming, while the expansion of a factored polynomial into a sum of terms is always simple and fast.

Now a polynomial in n variables can be viewed as a finite n-dimensional array of coefficients. If a majority of them are zero, it is advantageous to represent the polynomial as a list of the nonzero ones together with their coordinate labels (i.e., their exponents). Otherwise, it is preferable to use the entire array. In many practical cases the number of variables and the maximum exponent sizes are all of the order of 10, so an array size as large as 10^{10} would not be unusual. However, it is difficult to imagine a practical case involving more than a few hundred (or conceivably a few thousand) nonzero terms. For generality we are therefore obliged to represent each polynomial as an ordered list of its nonzero terms. It is convenient to order the terms according to the magnitude of the first exponent, and to order those terms having the same first exponent according to the magnitude of the second, etc. The order of the variables is the order in which they appear in the format statement.

---

† See exercise 15 on page 82 of Ref. 1.

1.3 *Polynomial Arithmetic*

In this section we shall discuss the implementation of the various polynomial-arithmetic operations. Let us begin with a simple illustration of their use. Suppose polynomials A, B, C, and D are in storage, and C is thought to be a divisor of $A*B$. (The asterisk denotes multiplication.) To compute and print

$$F = \frac{A*B}{C} + D \tag{1}$$

we write†

```
POLMPY    F,A,B
POLDIV    F,F,C,NODIV
POLADD    F,F,D
POLPRT    F
```
(2)

The first line replaces F by $A*B$. The second replaces F by F/C; that is, by

$$\frac{A*B}{C} \tag{3}$$

This illustrates the fact that an output may overwrite an input. The third line replaces F by $F + D$; that is, by

$$\frac{A*B}{C} + D \tag{4}$$

which is the desired result. Finally, the fourth line causes this result to be printed on the output tape. If the division in the second line is unsuccessful, i.e., if C is not a divisor of $A*B$, control will be transferred to the location called NODIV.

A polynomial is represented on data cards as a sequence of coefficients and exponents, each coefficient being followed by its exponents. It is terminated by the appearance of a zero where a coefficient would other wise be expected. For example the polynomial

$$3x^2 + 2xyz - 5yz^2 \tag{5}$$

might appear as

† Note the similarity to the arithmetic orders of a three-address computer. The prefix "POL" stands for "polynomial."

$$\begin{array}{rl} 3 & 2,0,0 \\ 2 & 1,1,1 \\ -5 & 0,1,2 \\ 0 & \end{array} \tag{6}$$

We have chosen this type of representation primarily because of its appeal to the computer. However, for large polynomials it is also an unexpectedly appealing form for people. On several occasions we have observed geometrical patterns in the computer output which would not be apparent in a conventional human transcription.

The addition of two polynomials in canonical form is analogous to the ordered merging of two ordered subdecks of a deck of playing cards, except that the addition subroutine must also be on the lookout for combinations and cancellations.

The multiplication of a polynomial by a nonzero monomial does not disturb canonical form. When two polynomials are to be multiplied, the longer one is multiplied by each term of the shorter one, and each of these products is added to the sum of all the preceding ones.

The polynomial division subroutine is successful only when the dividend is exactly divisible by the divisor. However, it is programmed so that it can be used as a test for divisibility if that is desired. The divisor and dividend are treated as polynomials in one variable with coefficients in the ring of polynomials in all the remaining variables. Divisions in this ring can be handled by the division subroutine itself,† and the main task is carried out by the familiar process of "long division."

The polynomial substitution subroutine works in the most straightforward possible way — substituting into one term at a time and preserving only the latest partial result. This procedure may involve substantial duplication of effort, but it uses a minimum of working space and a minimum of program, and in most practical cases the running time is reasonable.

The polynomial differentiation subroutine differentiates term by term with respect to a specified variable. It is perhaps worth remarking that this process does not upset the canonical ordering.

A truncated power series with polynomial coefficients can be treated as a polynomial, except that it is necessary to keep track of the order

† A subroutine which calls itself is called "recursive." At the innermost level it must, of course, operate by an independent mechanism. Collisions between the different levels are prevented by saving necessary information in a push-down list. It is perhaps worth remarking that every inductive algorithm can be programmed as a recursive subroutine. In the case of polynomial division the induction is on the number of variables, and the innermost level is simply coefficient division.

and to prevent the appearance of meaningless higher-order terms. The ALPAK system contains only two orders ("truncate" and "multiply and truncate") for dealing with truncated power series. These are sufficient for many applications, but much remains to be done.

1.4 *Rational Functions and the Future*

Every rational function can be represented as the quotient of two polynomials. The extension from polynomial operations to rational-function operations would be trivial except for the problem of removing all common factors from the numerator and denominator of each rational function. This has been accomplished by means of a generalized version of Euclid's greatest-common-divisor algorithm. However, we must caution the reader that Euclid's algorithm is extremely explosive, and the computer will not be able to handle rational functions with numerators and denominators of high degree in many variables until more sophisticated techniques are developed.

Aside from the difficulties mentioned above, the handling of truncated power series with rational-function coefficients and the solution by Gaussian elimination of systems of linear equations with rational-function coefficients are straightforward.

One of the primary problems encountered in the development of the ALPAK system is the problem of automatic dynamic storage allocation. Usually the inputs to a subroutine are polynomials of arbitrary size, and in general the required working space could not be predicted even if the sizes of the inputs were known. Therefore it is imperative to be able to obtain blocks of space as needed and to return idle space to the system. Our storage allocator provides these services in a manner suitable to our current needs, but it is not general or elegant. A general purpose storage allocation system including tracing and other service routines has been developed by Miss D. C. Leagus and the author, and will be described in a forthcoming paper. With this as a foundation, we hope to write a faster and more powerful version of the present ALPAK system, and perhaps to extend it into other areas of mathematics.

II. APPLICATIONS

2.1 *Introduction*

This section is devoted to a few general remarks about the usefulness of symbolic computing. The skeptic will protest that any symbolic calculation too long to be done with pencil and paper is not really worth

doing. This sentiment might be expressed in the form of the question, "Who wants to look at a polynomial ten pages long?" The objection is not without merit, but it is worth recalling that similar objections were once raised in connection with numerical calculations. Furthermore it is unmistakably clear that mathematical analyses arising in many different contexts involve substantial amounts of routine algebra which could be done faster and more reliably by a computer. What, then, are the types of problems to which symbolic computing facilities are likely to be applicable?

It often happens that a "straightforward calculation" whose end result is concise and understandable involves many tedious manipulations of lengthy expressions at intermediate stages. Sometimes the end result can also be reached by a shorter route, but the result itself (and the knowledge that it is indeed concise and understandable) may play a decisive role in the discovery of that route.

If the desired output of a calculation is numerical or graphical, it may nevertheless be advantageous (or even essential) to begin the calculation symbolically and allow a numerical program to take over only during the final stages. The problem of error analysis will not arise until these final stages are reached.

A third type of application arises when a simple calculation must be repeated many times with only minor variations, e.g., for all possible values of some set of indices.

Other types of applications may possibly occur to the reader. In the next five sections we shall discuss specific problems to which the ALPAK system has been applied.

2.2 *On the Zeros of Gaussian Noise*

Our first significant test problem arose in a study by D. Slepian[2] of the distribution of zeros of Gaussian noise. It was desired to find the leading term in the power series expansion of the determinant

$$\begin{vmatrix} \rho(ut - vt) & \rho(vt) & \rho(t - vt) & -\rho'(vt) & \rho'(t - vt) \\ \rho(ut) & 1 & \rho(t) & 0 & \rho'(t) \\ \rho(t - ut) & \rho(t) & 1 & -\rho'(t) & 0 \\ -\rho'(ut) & 0 & -\rho'(t) & 1 & -\rho''(t) \\ \rho'(t - ut) & \rho'(t) & 0 & -\rho''(t) & 1 \end{vmatrix} \tag{7}$$

where

$$\rho(t) = 1 - \frac{t^2}{2!} + \frac{at^3}{3!} + \frac{bt^4}{4!} + \frac{ct^5}{5!} + \frac{dt^6}{6!} + \frac{et^7}{7!} + \frac{ft^8}{8!} + \cdots. \tag{8}$$

The algebra is difficult not only because of the order of the determinant, but also because the leading term corresponds to an unexpectedly high power of t. In the general case, $a \neq 0$, the leading term is

$$\frac{a^3t^7}{9} v^2(1 - u)^2(3u - v - 2uv). \tag{9}$$

When $a = 0$ but $c \neq 0$, it is

$$\frac{2(1 - b)c^2t^{12}}{(5!)^2} v^2(1 - u)^2[2v^3(2u^3 - u^2 - 4u - 2) - 5uv^2(2u^2 - u - 4) + 5u^2(2u - 3)]. \tag{10}$$

Finally, when $a = 0$ and $c = 0$, it is

$$\frac{t^{16}}{144(4!)^2} (b^2 + d)(b^3 + d^2 + f + 2bd - bf)u^2v^2(1 - u)^2(1 - v)^2. \tag{11}$$

These results were obtained by a program written in the ALPAK language by Mrs. W. L. Mammel. Although approximately 2000 polynomial terms were in storage at the floodcrest of the computation, the computing time for all three cases was only 92 seconds.

2.3 *A Queueing System with Priorities*

Another interesting problem arose in a study by J. P. Runyon[3] of a queueing system in which a group of servers handles traffic from two sources, one of which is preferred over the other. It is desired to solve the functional difference equation

$$(\alpha - x)(\beta - \alpha)^{n-1}g_n(x) = \alpha(\beta - x)^n g_{n-1}(\alpha) - x(\beta - \alpha)^n g_{n-1}(x) \qquad n \geqq 1 \tag{12}$$

where $g_0(x) = 1$, and $0 < \alpha < \beta$. It follows by induction that for $n \geqq 1$, $g_n(x)$ is a polynomial of degree $(n - 1)$ in x, whose coefficients are polynomials in α and β. The value of $g_n(\alpha)$ is of particular interest. By the time this author was ready to attack the problem, Runyon had conjectured and J. A. Morrison[4] had proved that

$$g_n(\alpha) = \sum_{r=0}^{n-1} \binom{n-1}{r} \binom{n}{r} \frac{\beta^{n-r}\alpha^r}{n+1}. \tag{13}$$

Nevertheless, a short program was written to compute as many as possible of the polynomials $g_n(x)$ and the corresponding $g_n(\alpha)$. The program stopped after $87\frac{1}{2}$ seconds because of a coefficient overflow† during the

† The largest allowed coefficient is $2^{35} - 1$.

calculation of $g_{16}(x)$. The polynomial $g_{15}(x)$ has 197 terms and a maximum coefficient of several billion. If the program had been available sooner, it would have spared Runyon the necessity of calculating the first five of the $g_n(x)$ by hand.

2.4 *A Single-Server Queue with Feedback*

Another problem from queueing theory arose in a study by L. Takács[5] of a single-server queueing system with "feedback." The input is a Poisson process of density λ, the service times are determined by a distribution function with moments α_k, and after being served a customer rejoins the queue with probability p or departs with probability $q = 1 - p$.

It is shown by Takács that the rth moment of the total time spent in the system is

$$\beta_r = (-1)^r \Phi_{r0} \tag{14}$$

where

$$\Phi_{ij} = \left[\left(\frac{\partial}{\partial s}\right)^i \left(\frac{\partial}{\partial t}\right)^j \Phi(s,t)\right]_{s=0,\, t=0} . \tag{15}$$

The function $\Phi(s,t)$ is implicitly defined by the equation

$$\Phi(s,t) = (q - \lambda\alpha_1)W(s,t) + p\psi(s + \lambda t)\Phi(s,\omega(s,t)) \tag{16}$$

where†

$$\begin{aligned} \psi(s) &= \sum_{i=0}^{\infty} \frac{(-1)^r \alpha_r s^r}{r!} \\ \omega(s,t) &= 1 - (1 - pt)\psi(s + \lambda t) \\ W(s,t) &= \psi(s + \lambda t) + S(s + \lambda t, \lambda\omega(s,t))T(\omega(s,t)) \end{aligned} \tag{17}$$

with

$$\begin{aligned} S(x,y) &= \frac{\psi(x) - \psi(y)}{x - y} \\ T(\omega) &= \frac{\lambda\omega(1 - \omega)}{1 - \omega - (1 - p\omega)\psi(\lambda\omega)} . \end{aligned} \tag{18}$$

This last pair of equations can be rewritten in the more useful form

† For convenience we have assumed that all of the service moments α_r are finite. However, for the calculation of β_r it is clearly sufficient to require only the finiteness of α_{r+1}.

$$S(x,y) = \sum_{r=0}^{\infty} \frac{(-1)^{r+1}\alpha_{r+1}}{(r+1)!} C_r(x,y)$$
$$T(\omega) = \frac{-\lambda(1-\omega)}{q - \lambda(1-p\omega)\varphi(\lambda\omega)} \tag{19}$$

where

$$C_r(x,y) = \frac{x^{r+1} - y^{r+1}}{x - y} = \sum_{k=0}^{r} x^k y^{r-k}$$
$$\varphi(x) = \frac{1 - \psi(x)}{x} = \sum_{r=0}^{\infty} \frac{(-1)^r \alpha_{r+1} x^r}{(r+1)!} . \tag{20}$$

It is now clear that

$$\psi(0) = \alpha_0 = 1$$
$$S(0,0) = -\alpha_1$$
$$T(0) = \frac{-\lambda}{q - \lambda\alpha_1} \tag{21}$$
$$W(0,0) = \frac{q}{q - \lambda\alpha_1}$$

so from (14)–(16)

$$\beta_0 = \Phi_{00} = \Phi(0,0) = 1 \tag{22}$$

as is required by the definition of the zeroth moment.

Now suppose all of the quantities Φ_{ij} for $i + j < r$, where r is some positive integer, have been calculated and are expressed as rational functions of λ and p (or q) and the service moments α_k. Then by differentiation of (16) we can obtain a system of $r + 1$ linear equations in the $r + 1$ unknowns, Φ_{ij} with $i + j = r$. These equations will also contain the quantities Φ_{ij} with $i + j < r$, which can be replaced by their known values. The solutions of this linear system will again be rational functions of λ and p (or q) and the service moments α_k. Theoretically, this procedure permits the calculation of arbitrarily many of the moments, but in practice the calculations are extremely lengthy.

The first moment can be calculated by hand, with the result

$$\beta_1 = \alpha_1 \left(\frac{1 - \lambda\alpha_1}{q - \lambda\alpha_1}\right) + \frac{\lambda\alpha_2}{2(q - \lambda\alpha_1)} . \tag{23}$$

The second moment was calculated with the aid of an IBM 7090 computer and the ALPAK system. The intermediate expressions are ex-

tremely lengthy, but the final result is the relatively compact expression

$$\beta_2 = \frac{(2qF - G)(q^2 - 2q)}{6(q - \lambda\alpha_1)^2[q^2 - q(\lambda\alpha_1 + 2) + \lambda\alpha_1]} \tag{24}$$

where

$$\begin{aligned} F &= 6\lambda\alpha_1^3 - 6\alpha_1^2 + 6\lambda\alpha_1\alpha_2 + 3\alpha_2 + \lambda\alpha_3 \\ G &= 12\lambda\alpha_1^3 - 12\alpha_1^2 - 6\lambda\alpha_1\alpha_2 + 2\lambda^2\alpha_1\alpha_3 - 3\lambda^2\alpha_2^2. \end{aligned} \tag{25}$$

For a more detailed discussion of this calculation, see the appendix in Ref. 5.

2.5 *The Triskelion Diagram*

The problem to be considered in this section arose in a study by D. B. Fairlie and the author[7,8] of the analyticity properties of the Feynman amplitudes corresponding to several simple vertex diagrams in quantum field theory. One of these is the triskelion diagram, which is shown in Fig. 1. Here the p's and q's are vectors in space-time, and

$$\begin{aligned} z_i &= p_i^2 \\ a_i &= q_i^2 \\ b_i &= (p_i - q_i)^2 \end{aligned} \tag{26}$$

for $i = 1, 2, 3$. The corresponding Feynman amplitude is the boundary value of an analytic function $H(a,b,z)$ of these nine variables, analytic

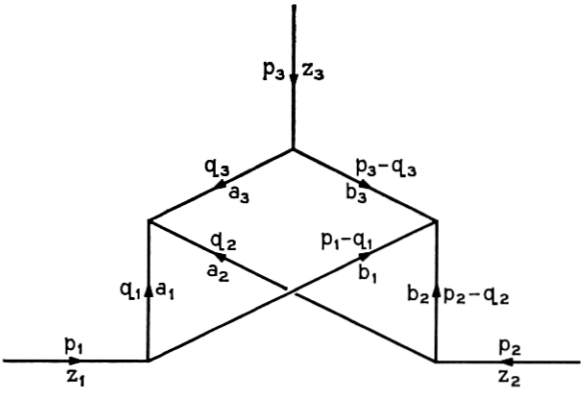

Fig. 1 — The triskelion diagram.

everywhere except on certain manifolds which can be obtained from lower-order "contracted" diagrams, and on the manifold

$$\Psi(a,b,z) \equiv 4D^2(4D + A^2) + 4AB^2(9D + 2A^2) - 27B^4 = 0 \quad (27)$$

where

$$\begin{aligned} A &= \tfrac{1}{8}\lambda(z + a + b) - \tfrac{1}{4}[\lambda(z) + \lambda(a) + \lambda(b)] \\ B &= -\tfrac{1}{4} \det (z,a,b) \\ D &= -\tfrac{1}{8} \sum_{i=1}^{3} \{z_i^2(a_jb_k + a_kb_j) + z_jz_k(a_iB_i + b_iA_i) \\ &\quad + z_i[2a_ib_jb_k + a_jb_kB_k + a_kb_jB_j + 2b_ia_ja_k + b_ja_kA_k + b_ka_jA_j]\}. \end{aligned} \quad (28)$$

Here (i,j,k) is a cyclic permutation of (1,2,3) and

$$\begin{aligned} A_i &= a_i - a_j - a_k \\ B_i &= b_i - b_j - b_k \\ \lambda(x) &= x_1^2 + x_2^2 + x_3^2 - 2x_1x_2 - 2x_1x_3 - 2x_2x_3 . \end{aligned} \quad (29)$$

It is shown in Ref. 8 that Ψ is a homogeneous twelfth-degree polynomial in its nine arguments, and is irreducible over the rationals. Furthermore, it is invariant under permutations of the indices, 1,2,3, permutation of the vectors, a,b,z, and transposition of the matrix of these vectors.

It is natural to ask whether the substitution of (28) and (29) into (27) yields a compact expression or an unwieldy monstrosity. A short program was written to perform the substitutions, but it stopped at an early stage because of insufficient space. However, the polynomial $\Psi(a_1,a_2,a_3;\ b_1,b_2,b_3;\ 0,0,z_3)$ was easily computed (in 50 seconds) and was found to have 2642 terms. Since $\Psi(a,b,z)$ contains all of these terms and many more, we can safely assume that (27) is the most useful way of writing it.

2.6 *Wave Propagation in Crystals*

The problem to be considered in this section arose in a study by R. N. Thurston[9] of wave propagation in crystals under pressure. It is of particular interest to investigate the effect of pressure on propagation velocity. For given temperature T, pressure p (hydrostatic or uniaxial), and propagation direction N (a unit vector), there are in general three modes of propagation, corresponding to three displacement directions which are mutually perpendicular if $p = 0$. In simple cases one of these modes is longitudinal and the other two are transverse. For a given mode,

let $V(p,T)$ be the propagation velocity and let ρ_0 be the crystal density at $p = 0$. Then define

$$S' \equiv \rho_0 \frac{\partial}{\partial p} [V^2(p,T)]_{p=0} . \tag{30}$$

It can be shown that

$$S' = U_p U_q D_{pq} \tag{31}$$

(summation convention understood), where U is a unit vector in the direction of particle displacement in the given mode, and where

$$D_{pq} = N_j N_k F_{st} [\delta_{pq} C_{jkst}{}^{T} + 2\delta_{qs} C_{pjtk}{}^{S} + 2N_s N_t C_{pjqk}{}^{S} + C_{pjqkst}] . \tag{32}$$

The C's are elastic constants at zero pressure. The six-index C array has 3^6 entries of which at most 56 are distinct, while each four-index C array has 3^4 entries of which at most 21 are distinct. F_{st} is a symmetric matrix whose entries are rational functions of these elastic constants (and of the direction of pressure in the uniaxial case). Our task is to perform the indicated summations in special cases to get explicit expressions for S'.

The complete analysis for the case of cubic crystals is given in Ref. 9. A program has been written by J. P. Hyde (using the ALPAK system) to evaluate S' and serve as a check for this analysis. In the cubic case, the six-index C array has only six distinct nonzero elements, which are abbreviated as C_{111}, C_{112}, C_{144}, C_{166}, C_{123}, and C_{456}. The four-index C^T array has only three distinct nonzero elements, abbreviated as $C_{11}{}^{T}$, $C_{12}{}^{T}$, and C_{44}, and the four-index C^S array has only three distinct nonzero elements, abbreviated as $C_{11}{}^{S}$, $C_{12}{}^{S}$, and C_{44}. Note that C_{44} appears in both arrays. The results for the case of hydrostatic pressure and wave propagation along (1,1,0) are as follows: For longitudinal displacement along (1,1,0)

$$S' = 2C_{11}{}^{S} + 2C_{12}{}^{S} + 4C_{44} + \tfrac{1}{2}C_{111} + 2C_{112} + C_{144} + 2C_{166} + \tfrac{1}{2}C_{123} . \tag{33}$$

For transverse displacement along (1,−1,0)

$$S' = 2C_{11}{}^{S} - 2C_{12}{}^{S} + \tfrac{1}{2}C_{111} - \tfrac{1}{2}C_{123} . \tag{34}$$

And for transverse displacement along (0,0,1)

$$S' = 4C_{44} + C_{144} + 2C_{166} . \tag{35}$$

The computing time to obtain these results was approximately 20 seconds.

A modified version of this program would make possible the corresponding calculations for crystals of lower symmetry, including quartz.

III. USERS' MANUAL

3.1 *Introduction*

This section consists of a brief outline of Section III and a discussion of several basic concepts. The polynomial input-output and arithmetic operations are discussed in Sections 3.2 and 3.3, respectively. Section 3.4 consists of a brief introduction to the theory of truncated power series and a description of the orders for dealing with them. In Section 3.5 the rules for writing main programs (including those governing the use of POLBEG and VARTYP) are described, and two sample programs are presented. Loading instructions for assembly and/or run are given in Section 3.6. Finally, the dumping facilities and diagnostics are described in Section 3.7, and hints for debugging are given in Section 3.8.

3.1.1 *A Polynomial in Core*

A nonconstant polynomial† in core consists of a pointer, a heading, a data block, and a format statement (see Fig. 2). The pointer is a single word containing the heading address. The heading is a three-word block containing the data address, the format address, and the number of terms. The data block contains the terms, stored consecutively in a manner determined by the format statement. The format statement contains the names of the variables and the maximum exponent size in bits associated with each. The name of a polynomial is ordinarily used for the symbolic address of its pointer, and the name of a format statement for its symbolic address.

3.1.2 *Format Compatibility*

A format statement is usually shared by many polynomials. In fact two polynomials cannot be added, subtracted, multiplied, or divided unless they share the same format statement.

3.1.3 *More Than One Pointer to a Heading*

If two or more polynomials are equal, their pointers may point to a common heading. This is especially convenient when arrays of polynomials with many equal elements must be dealt with, but the user must keep in mind that if one of the polynomials is changed the others will be changed in the same way.

† A constant polynomial has only a pointer and a heading. Its value is kept in the heading (see Section 3.2.7), and no format is needed.

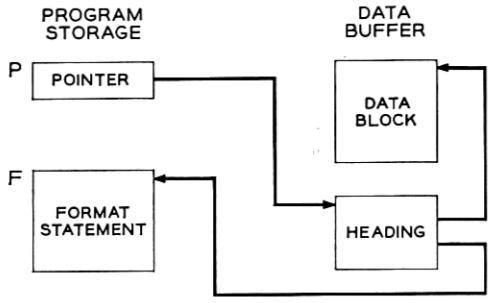

Fig. 2 — A polynomial P with format F.

3.1.4 *Storage Allocation*

Space for headings and data blocks is provided by the storage allocator. Headings are never moved, but the storage allocator is free to move data blocks as necessary.

Space for pointers and format statements must be provided by the user. Each pointer must be a full word, but only its address field is used. This must initially contain zero and will be filled in by the system. The prefix, tag, and decrement fields will be cleared. When a polynomial is read or computed its pointer is tested. If the address field of the pointer contains zero, a heading is created and the pointer is filled in with the heading address. Otherwise it is assumed that the pointer contains the address of a heading which can be overwritten. The data block (if any) previously attached to that heading is left "headless" and thereby becomes "garbage."

3.1.5 *Macros and Subroutines*

The polynomial portion of the ALPAK system consists of a macro deck and two subroutine packages, ALPAK1 and ALPAK2. ALPAK1 consists of input, output, and service subroutines, while ALPAK2 contains the operating subroutines. Together the two packages occupy less than 5000_{10} words of memory. Most of the macros expand into calling sequences for subroutines of the same name. For example the macro

$$\text{POLADD} \quad \text{R,P,Q} \tag{36}$$

which is represented by the equation

$$R = P + Q \tag{37}$$

("replace R by $P + Q$"), expands to

```
TSX     POLADD,4
PZE     R
PZE     P
PZE     Q
```
(38)

Here P, Q, and R are the symbolic addresses of pointers. When POLADD is executed, the P and Q pointers must contain the addresses of polynomial headings. The address field of the R pointer may contain the address of a heading to be overwritten or it may contain zero. In the latter case, a new heading will be created by the storage allocator and the R pointer will be filled in with its address. In either case, a data block for the sum of the polynomials P and Q will be obtained from the storage allocator and attached to the R heading, and the sum will be computed therein.

3.1.6 *Indexing*

This method of communication gives us a natural way of handling indexed arrays of polynomials. For example the set of polynomials

$$R_i = P_i + Q_i\,; \qquad i = 1, \cdots, n \tag{39}$$

can be computed by writing

```
POLADD    (R,1)(P,1)(Q,1)
```
(40)

inside a suitable loop (see Section 3.5), where index register 1 corresponds to the index, i. The expansion of this macro is simply

```
TSX     POLADD,4
PZE     R,1
PZE     P,1
PZE     Q,1
```
(41)

Clearly, index register 4 cannot be used for this type of indexing, because it has been reserved for the subroutine linkage.

3.2 *Input-Output*

3.2.1 *Summary* (*See Descriptions Section* 3.2.2)

	POLRDF	F	read format	(a)
F	POLCVF	(X,15,Y,21,Z,36)	convert format	(b)
	POLRDD	P,F	read data	(c)
	POLCVD	P,F,H	convert data	(d)
	POLCLR	P	clear	(e)
	POLSTZ	P	store zero	(f)
	POLSTI	P	store identity	(g)
	POLSTC	P,C	store constant	(h)
	POLSTV	P,X,F	store variable	(i)
	POLPRT	P,CC,(NAME)	print	(j)
	POLPCH	P,(NAME)	punch	(k)
	POLPRP	P,CC,(NAME)	print and punch	(l)
	POLRDP	P,F,CC,(NAME)	read and print	(m)
	POLCVP	P,F,H,CC,(NAME)	convert and print	(n)

(42)

C = constant (symbolic address of constant)
CC = control character for printer
F = format (symbolic address of/for format statement)
H = Hollerith data (symbolic address of data)
NAME = alternative name for polynomial (not exceeding 21 characters)
P = polynomial (symbolic address of pointer)
X = variable (specified in the manner indicated by the last previous VARTYP declaration — see Section 3.5.2).

3.2.2 *Descriptions* (*See Also Sections* 3.2.3–3.2.8)

(a) POLRDF F

Read a polynomial format statement from cards into a block starting at location F. The length of this block must be at least $(2 + 2v + e)$ words where v is the number of variables and e is the number of exponent words per term.

(b) F POLCVF (X,15,Y,21,Z,36)

Assemble the parenthesized polynomial format statement and assign the symbol F to its first location. (F is a location-field argument of the macro.)

(c) POLRDD P,F

Read the polynomial P from cards according to the format F and put P into canonical form. Here, P is the address of a "pointer" for the polynomial, and F is the address of a format statement.

(d) POLCVD P,F,H

Same as POLRDD except that the data is to be found in core in a block of not more than 12 words of binary-coded information (BCI) starting at location H.

(e) POLCLR P

Clear the polynomial P.

(f) POLSTZ P

Set P equal to zero.

(g) POLSTI P

Set P equal to one.

(h) POLSTC P,C

Set P equal to the constant C.

(i) POLSTV P,X,F

Set P equal to the variable X using the format F.

(j) POLPRT P,CC,(NAME)

Print the polynomial P using CC for the control character for the first line of print and NAME (not more than 21 characters of BCI) for the name. If NAME is not provided P will be used for the name, and if CC is not provided a minus (triple space) will be used for the control character.

(k) POLPCH P,(NAME)

Punch the polynomial P on cards using NAME (not more than 21 characters of BCI) for the name. If NAME is not provided, P will be used for the name.

(l) POLPRP P,CC,(NAME)

Same as POLPRT followed by POLPCH.

(m) POLRDP P,F,CC,(NAME)

Same as POLRDD followed by POLPRT.

(n) POLCVP P,F,H,CC,(NAME)

Same as POLCVD followed by POLPRT.

3.2.3 *Polynomial on Cards*

A polynomial is represented on data cards as a sequence of coefficients and exponents separated by blanks and/or commas, each coefficient being followed by its exponents. It is terminated by the appearance of a zero where a coefficient would otherwise be expected. It is customary to use one card for each term and one as an end card. For example, the polynomial

$$3x^2 + 2xyz - 5yz^2 \tag{43}$$

is usually represented as

```
 3     2,0,0
 2     1,1,1
-5     0,1,2
 0
```
(44)

or

```
 3     2 0 0
 2     1 1 1
-5     0 1 2
 0
```
(45)

However, it is equally correct to put more than one term on a card

```
3,2,0,0     2,1,1,1
-5,0,1,2    0
```
(46)

or to use more than one card for a term

```
 3     2
       0,0
 2     1
       1,1
-5     0
       1,2
 0
```
(47)

If two commas are adjacent or separated only by blanks, a zero is understood. Similarly if the first (last) character on a card is a comma, a preceding (succeeding) zero is understood. Thus (43) can be represented as

$$\begin{array}{rl} 3 & 2,, \\ 2 & 1,1,1 \\ -5 & 0,1,2, \end{array} \tag{48}$$

or

$$3,2,,,2,1,1,1,-5,,1,2, \tag{49}$$

If identifying comments are desired, they may be printed on the last card, after the blank or comma which terminates the conversion of the final zero, and/or in columns 73–80 of any card.

The data is read from cards, converted, packed into the data buffer, and put into canonical form by the subroutine POLRDD (read data). The manner of packing is determined by a format statement which must be read first. If the polynomial has k variables, the first number in the data sequence is interpreted as a coefficient and the next k numbers are interpreted as exponents. This process is repeated until a zero appears in the position of a coefficient. The reading is then terminated, and the subroutine POLCFM (canonical form) is called to put the polynomial into canonical form.

3.2.4 *Format Statements*

Before discussing the operation of POLCFM it will be necessary to consider in detail the format statements and the representation of polynomials in core. A format statement on card(s) is an alternating sequence of variable names and field widths, starting in column 1 and separated by commas. Each field width must be a positive integer not greater than 36. It is the maximum exponent size in bits of the corresponding variable. Each variable name must be a string of not more than six characters (usually a FAP symbol) containing neither blanks nor commas. It is legal to skip to the next card after any comma, and this makes it possible to use as many continuation cards as necessary. The format statement is terminated by a blank immediately following a field width. Each field width specifies the number of bits to be reserved in each term for the exponent of the corresponding variable, and thereby determines the maximum allowable exponent for that variable. As an example, the format statement

$$X,15,Y,21,Z,36 \tag{50}$$

specifies three variables, X, Y and Z, with field widths of 15, 21 and 36 respectively. This means that the maximum exponent sizes are $2^{15} - 1$,

$2^{21} - 1$ and $2^{36} - 1$ respectively. The sum of the field widths must be an integral multiple of 36, and each smaller multiple (if any) must be included among the partial sums. The card(s) is (are) read by the subroutine POLRDF (read format), which stores the format statement in a block *provided by the user*. POLRDF also counts v, the number of variables, computes e, the number of exponent words per term (the sum of the field widths divided by 36), and constructs a mask for use in exponent addition (see Section 3.3). The mask is a block of e words partitioned into v bit fields as indicated by the format statement with a one at the right end of each bit field. These items are stored as part of the format statement, whose length is $2 + 2v + e$ words. For example the internal format statement (in octal) corresponding to (50) is

000000000002	2 exponent words per term	
000000000003	3 variables	
676060606060	X	
000000000017	15	
706060606060	Y	(51)
000000000025	21	
716060606060	Z	
000000000044	36	
000010000001	MASK	
000000000001		

3.2.5 *Polynomial in Core*

A polynomial term is stored in two or more consecutive locations in a manner determined by the format statement. The coefficient is placed in the first word and the exponents are packed into the remaining words, allowing the specified number of bits for each. For example, the term

$$5x^2y^7z^3 \tag{52}$$

in the format (50) has the octal representation

000000000005	5	
000020000007	2,7	(53)
000000000003	3	

A nonconstant polynomial in core consists of a pointer, a heading, a data block, and a format statement as explained in Section 3.1 (see Fig. 2). The data block contains the terms as in (53) stored consecutively.

3.2.6 *Canonical Form*

We are now prepared to discuss the canonical form subroutine, POLCFM. Its task is to put any given polynomial, stored in the manner described above, into canonical form. More precisely, it must order the terms according to their exponent sets, combining terms with equal exponent sets and discarding any resulting zeros. The terms are to be arranged in increasing order of the first exponent, and terms having the same first exponent are to be arranged in increasing order of the second, etc. If there is only one exponent word per term, this means that the terms can be ordered according to the magnitude of that word treated as an unsigned 36-bit integer. Otherwise they must be ordered according to the magnitude of the first exponent word and subordered according to the magnitude of the second, etc. No working space is required. The ordering is done first, with the aid of the system sort, FAPSTL, and the combinations and cancellations, if any, are then performed. Finally if the result is a constant, it is stored according to the "heading convention" which we shall now describe.

3.2.7 *Heading Convention*

As we mentioned in Section 3.1, each nonconstant polynomial has a fixed heading of three words containing the data address, the format address, and the number of terms, respectively. Since constant polynomials can usually profit from special treatment and in any case the zero polynomial requires it, we have devised a special representation for constants. The first word of the heading contains the code number 5, which cannot possibly be a legal data address, and the second contains the value of the constant. Such a heading has no associated data block, and its third word is never consulted. The code number zero signifies an idle heading, and the numbers one to four are reserved for rational functions.

The macro POLCLR (clear) stores zero in the first word of the heading, thereby marking it as idle and destroying the attached data block (if any). The macros POLSTZ (store zero), POLSTI (store identity), and POLSTC (store constant) store 5 in the first word of the heading and the specified constant in the second word.

3.2.8 *Output*

There is one output subroutine with three entry points — POLPRT (print), POLPCH (punch), and POLPRP (print and punch). Each

term of a polynomial is printed (punched) on a single line (card), except that continuation lines (cards) will be used if necessary. All the coefficients are right adjusted to column 22, so that they form a column in the output. The exponents form one or more additional columns headed by the corresponding variable names. In each line the coefficient is separated from the first exponent by two blanks, and the exponents are separated from each other by single blanks. Therefore the exponent columns are not always straight. In printed output the first line contains the name of the polynomial (or any comment not more than 21 characters long) starting in column 2, and the names of the variables (separated by single blanks) starting in column 25. In punched output the first card contains the name or comment, the next card(s) is (are) a complete format statement, the ensuing cards contain the data, and finally an end card including the name is appended.

As an example, suppose the polynomial (43) is in core (in canonical form), and its name (i.e., the symbolic address of its pointer) is P. If P is then printed, the output will be

$$\begin{array}{rrrrr} \text{P} & & \text{X} & \text{Y} & \text{Z} \\ & -5 & 0 & 1 & 2 \\ & 2 & 1 & 1 & 1 \\ & 3 & 2 & 0 & 0 \end{array} \tag{54}$$

If it is punched, the output will be

$$\begin{array}{llrrrr} \text{P} & & & & & \\ \text{X,12,Y,12,Z,12} & & & & & \\ & & -5 & 0 & 1 & 2 \\ & & 2 & 1 & 1 & 1 \\ & & 3 & 2 & 0 & 0 \\ 0 & \text{END} & \text{P} & & & \end{array} \tag{55}$$

where each line represents one card. The second card is a valid format statement, and the last one is a valid END card. A polynomial in many variables may require more than one line (card) for the list of variables (format statement) and/or more than one line (card) per term.

3.3 *Polynomial Arithmetic*

3.3.1 *Summary (See Descriptions Below)*

(i) Basic Operations

POLADD	R,P,Q	$R = P + Q$	add	(a)
POLSUB	R,P,Q	$R = P - Q$	subtract	(b)

POLMPY	R,P,Q	$R = P*Q$	multiply	(e)
POLDIV	R,P,Q,NODIV	$R = P/Q$	divide (if divisible)	(d)
POLSST	G,F(LISTP) (LISTV)	$G = F$(LISTV = LISTP)	substitute	(e)
POLDIF	Q,P,X	$Q = \partial P/\partial X$	differentiate	(f)
POLZET	P	skip *iff* $P = 0$	zero test	(g)
POLNZT	P	skip *iff* $P \neq 0$	nonzero test	(h)
POLEQT	P,Q	skip *iff* $P = Q$	equality test	(i)
POLDUP	Q,P	$Q = P$	duplicate	(j)
POLCHS	P	$P = -P$	change sign	(k)

(*ii*) *Alternatives for Added Convenience and/or Efficiency*

POLSMP	Q,C,P	$Q = C*P$	scalar multiply	(l)
POLSMO	C,P	$P = C*P$	scalar multiply and overwrite	(m)
POLOMP	Q,M,P	$Q = M?P$	one-term multiply	(n)
POLOMO	M,P	$P = M*P$	one-term multiply and overwrite	(o)
POLSAD	Q,C,P	$Q = C + P$	scalar add	(p)
POLSAO	C,P	$P = C + P$	scalar add and overwrite	(q)
POLADO	P,Q	$P = P + Q$	add and overwrite	(r)
POLDFO	P,X	$P = \partial P/\partial X$	differentiate and overwrite	(s)

(*iii*) *Explanation of Symbols*

F,G,P,Q,R = polynomials (symbolic addresses of pointers)
C = scalar (symbolic address of scalar)
M = monomial (symbolic address of pointer)
X = variable (specified in the manner indicated by the last previous VARTYP declaration — see Section 3.5.2)
LISTP = list of polynomials
LISTV = list of variables.

3.3.2 *Descriptions*

(a) POLADD R,P,Q

P and Q are assumed to be in canonical form. The addition is analogous to the ordered merging of two ordered subdecks of a deck of playing cards, except that POLADD must also perform combinations and cancellations. Suppose P has n terms and Q has m terms. Then a block long enough for $n + m$ terms is reserved for R if space permits. Otherwise all the remaining space is reserved for R, and the subroutine proceeds in the hope that combinations and/or cancellations will compensate for the deficiency. If space runs out, the job will be dumped. The first (next) term of R is found by comparing the exponent sets of the first (next) term of P and the first (next) term of Q. If these differ, the first (next) term of R is the first (next) term of P or of Q, depending on

which comes earlier in the canonical ordering. If they are the same, the first (next) term of R is the sum of the first (next) terms of P and Q, unless the sum is zero. In that case the first (next) terms of P and Q cancel, making no contribution to R.

(b) POLSUB R,P,Q

This uses POLCHS (twice) and POLADD. If P and Q have the same heading, it uses POLSTZ instead.

(c) POLMPY R,P,Q

POLMPY multiplies the longer of the polynomials P and Q by each term of the shorter using POLOMP and accumulates these products using POLADD or POLAOE. The latter is a slightly modified version of POLADO, not normally available to the outside world. Its mnemonic is "Add, Overwrite the first argument, and Erase the second."

Suppose P has m terms and Q has n terms with $m \leqq n$. Let P_i be the ith term of P, let $T_i = P_iQ$ be the ith partial product, and let $S_i = \sum_{j=1}^{i} T_j$ be the ith partial sum.

If there is enough space for $(nm + n)$ terms, then the "leapfrog method," a fast method involving no data moving (see Fig. 3), is employed. Imagine the space partitioned into $m + 1$ blocks, each n terms long. The first partial product, T_1, is placed in the mth block and the second, T_2, in the $(m + 1)$st block. POLADD is then directed to add these, starting the sum S_2 at the beginning of the $(m - 1)$st block. This partial sum overwrites a portion (perhaps all) of the mth block as explained in the discussion of POLADO. The next partial product T_3 is then placed in the $(m + 1)$st block, and the next partial sum S_3 is started at the beginning of the $(m - 2)$nd block, overwriting a portion (perhaps all) of S_2. This process is repeated, each partial sum overwriting a portion (perhaps all) of the preceding one, until the final result S_m appears starting at the beginning of the first block.

If there is not enough space for this procedure, then the slower "compact method" (see Fig. 4) is used, requiring only enough space for the final result (or the longest partial sum) and n additional terms. The latest partial sum always starts at the top of the available space. The next partial product is placed immediately below it, and both are then moved down leaving a gap n terms long above the partial sum. The partial product is then added to the partial sum by POLAOE to produce a new partial sum, starting at the top of the available space and overwriting a portion (perhaps all) of the previous partial sum. This process

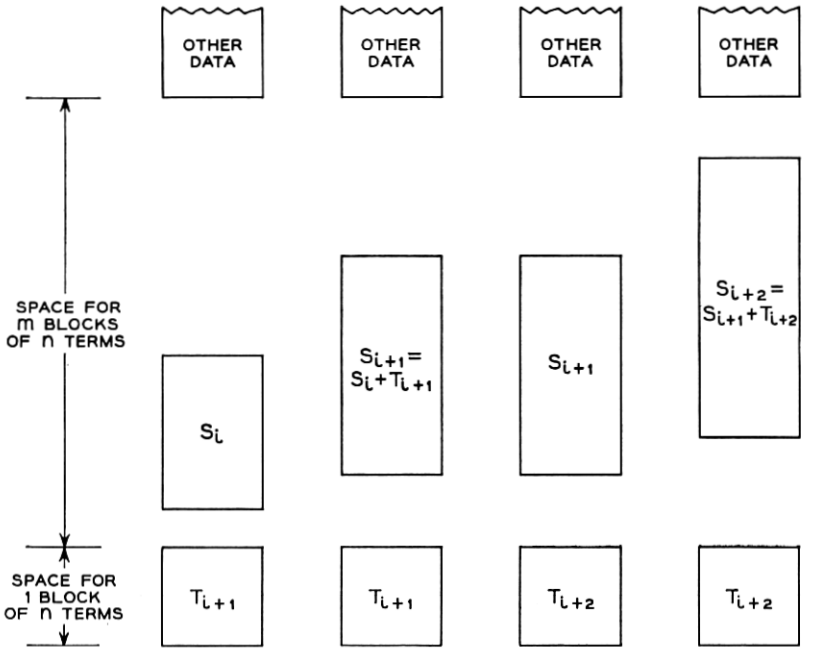

FIG. 3 — Successive steps in multiplication by the "leapfrog method."

is repeated until the final result is achieved or the available space exhausted.

(d) POLDIV R,P,Q,NODIV

The dividend P and the divisor Q are treated as polynomials in one variable (the first variable that at least one of them depends on) with coefficients in the ring of polynomials in all the remaining variables (if any). Divisions in this ring can be handled by calling POLDIV itself,† and the main task is carried out by the familiar process of "long division." The fourth argument, NODIV, is an address to which control will be transferred if Q does not divide P. If the fourth argument is omitted, the macro will supply ENDJOB in its place.

(e) POLSST G,F(LISTP)(LISTV)

† A subroutine which calls itself is called recursive. At the innermost level it must, of course, operate by an independent mechanism. Collisions between the different levels are prevented by saving necessary information in a push-down list. It is perhaps worth noting that every inductive algorithm can be programmed as a recursive subroutine. In the case of polynomial division the induction is on the number of variables, and the innermost level is simply coefficient division.

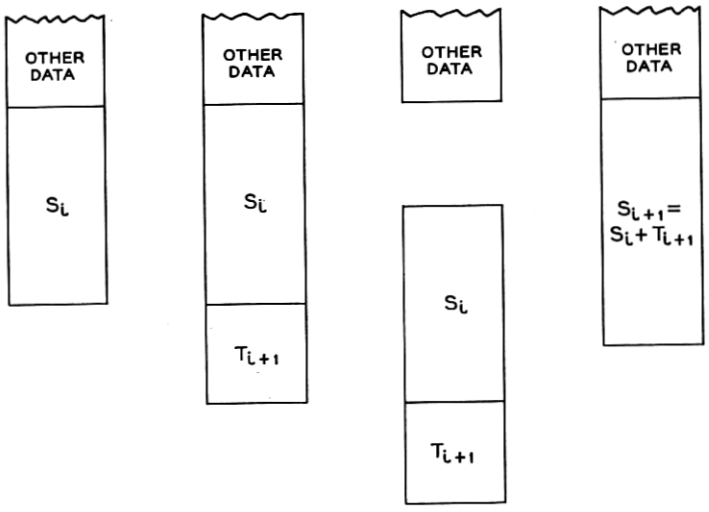

Fig. 4 — Successive steps in multiplication by the "compact method."

Here LISTP is a list of polynomials in a common format† which must include all the variables of F not being replaced, and LISTV is a list of the variables of F which are to be replaced by the polynomials in LISTP. For example if F depends on $X1, \cdots, X10$ and we wish to replace $X3$ and $X4$ by P and Q respectively, we write

POLSST G,F(P,Q)(X3,X4)

The variables in LISTV must be specified in the manner indicated by the last previous VARTYP declaration. If LISTV is not provided, it is understood to be the list of all the variables in the format of F.

POLSST works in the most straightforward possible way — substituting into one term at a time and preserving only the latest partial result. This procedure may involve substantial duplication of effort, but it uses a minimum of working space and a minimum of program, and in most practical cases the running time is reasonable.

(f) POLDIF Q,P,X

P is duplicated using POLDUP, and the copy is then differentiated with respect to X using POLDFO.

(g) POLZET P

† If all the polynomials in LISTP are constants (which have a universal format — see Section 3.2.7), then the format of F is used.

The next instruction is skipped if and only if $P = 0$.

(h) POLNZT P

The next instruction is skipped if and only if $P \neq 0$.

(i) POLEQT P,Q

The polynomials P and Q are considered to be equal if and only if they have the same format address, the same number of terms, and identical data blocks.

(j) POLDUP Q,P

Q is replaced by a copy of P.

(k) POLCHS P

The signs of all the coefficients of P are reversed.

(l) POLSMP Q,C,P

P is duplicated using POLDUP, and the copy is then multiplied by C using POLSMO.

(m) POLSMO C,P

Each coefficient of the polynomial P is multiplied by the scalar C.

(n) POLOMP Q,M,P

P is duplicated using POLDUP, and the copy is then multiplied by M using POLOMO.

(o) POLOMO M,P

Each term of the polynomial P is replaced by its product with the monomial M. To multiply two monomials, it is necessary to multiply their coefficients and add their exponents. In the case of integer coefficients, the coefficient multiplication macro

```
CMP     Z,X,Y
```

expands to

```
LDQ     X
MPY     Y
TZE     *+2
REM1
STQ     Z
```

where REM1 is the REMARK macro (see Section 3.7) for coefficient overflow. The exponent addition macro

```
EAD     Z,X,Y
```

adds the exponents one word at a time even though several exponents may be packed into each word. To check for overflow, EAD uses the appropriate word from the mask in the format statement. Suppose all the exponents are packed into a single word. Then the mask is a word containing a one in the low-bit position of each exponent block and zeros elsewhere. Now EAD expands to

```
CAL     X
ACL     Y
SLW     Z
ERA     X
ERA     Y
ANA     MASK
TZE     *+2
REM2
```

where REM2 is the REMARK macro (see Section 3.7) for exponent overflow. The first three lines compute the sum correctly, provided no overflows occur. After line 5 the low-bit positions in the AC should be zero, since ERA is the same as addition without carry. After line 6 the entire AC should therefore be zero. If it is not, control will pass to REM2 and the AC will contain a one-bit immediately to the left† of each exponent block which has overflowed.

```
(p)     POLSAD  Q,C,P
```

P is duplicated using POLDUP, and C is then added to the copy using POLSAO.

```
(q)     POLSAO  C,P
```

The scalar C is added (or appended) to the polynomial P.

```
(r)     POLADO  P,Q
```

Since P is to be replaced by the sum $P + Q$, it is not necessary to have space for both P and the sum. Instead it is possible to open a gap the size of Q above P, and then to use that gap together with the block occupied by P as a block for the sum. It is easy to see that no term of P can be overwritten by a term of the sum before making its contribution.

† Here we think of the AC as a circular register. An overflow in the leftmost exponent block will leave a one-bit at the right end of the AC.

(s) POLDFO P,X

Each coefficient of P is multiplied by the corresponding exponent of X. If the exponent is zero the term is deleted. Otherwise the exponent is reduced by one.

3.4 *Truncated Power Series*

Let x represent the k-tuple of variables $(x_1, \cdots, x_k)$. A *formal power series in* x is an expression of the form

$$A(x) = \sum_{i_1,\cdots,i_k=0}^{\infty} a_{i_1\cdots i_k} x_1^{i_1} \cdots x_k^{i_k} \tag{56}$$

where the a's are elements of any integral domain. The sum $i = i_1 + \cdots + i_k$ of the exponents in any individual term will be called the *order* of the term. Letting $a_i(x)$ be the (finite) polynomial consisting of all the terms of order i, we have

$$\begin{aligned} A(x) &= \sum_{i=0}^{\infty} A_i(x) \\ A_i(x) &= \sum_{\substack{i_1,\cdots,i_k \geqq 0 \\ i_1+\cdots+i_k=i}} a_{i_1\cdots i_k} x_1^{i_1} \cdots x_k^{i_k}. \end{aligned} \tag{57}$$

A *truncated power series of order* p is a formal power series from which all terms of order higher than p have been dropped. We shall restrict our attention to the case in which the a's are polynomials in a set of variables $y_1, \cdots, y_l$ $(l \geqq 0)$ not including any of the x's. The *sum* of two truncated power series

$$\begin{aligned} A(x) &= \sum_{i=p}^{p} A_i(x); \qquad A_{p'}(x) \neq 0 \\ B(x) &= \sum_{j=q}^{q} B_j(x); \qquad B_{q'}(x) \neq 0 \end{aligned} \tag{58}$$

is their polynomial sum truncated to order

$$\min(p,q) \tag{59}$$

while their *product* is their polynomial product truncated to order

$$\min(p + q', q + p'). \tag{60}$$

The ALPAK system contains two macros for dealing with truncated power series. These are POLTRC (truncate) and POLMPT (multiply and truncate). Addition can be handled with POLTRC and POLADD.

Each truncated power series must be stored as a polynomial in a format whose first k variables are the x's and whose remaining variables, if any, are the y's. The command

$$\text{POLTRC} \qquad \text{P,ORD,K} \tag{61}$$

causes P to be truncated to order ORD. That is, all terms of order greater than ORD are deleted. The command

$$\text{POLMPT} \qquad \text{R,ORDR,P,ORDP,Q,ORDQ,K} \tag{62}$$

is represented by the equation

$$R = P*Q \tag{63}$$

where P and Q are truncated power series. K is the address of the number of power series variables [i.e., the x's of (56) and (57)], ORDP and ORDQ are the addresses of the orders of P and Q respectively, and ORDR is an address for the order of R, which is to be computed by the rule (60).

If it is desired to multiply a truncated power series by a polynomial, the latter should be thought of as a truncated power series of order infinity. It is required that all finite orders be less than 2^{35}, and any number greater than or equal to 2^{35} is treated as infinity. Thus if P is a truncated power series of order 4 in 3 variables and we wish to multiply it by the polynomial Q, we write

$$\text{POLMPT} \qquad \text{R,ORDR,P,=4,Q,=-1,=3} \tag{64}$$

where the order -1 of Q will be interpreted† as $2^{35} + 1$, which is equivalent to infinity.

3.5 *The Main Program*

3.5.1 *POLBEG*

Every main program starts with the macro POLBEG (begin). At assembly time, this reserves a block of storage for the "data buffer" and at execution time it initializes the storage allocator. The command

$$\text{POLBEG} \qquad \text{N} \tag{65}$$

† In the IBM 7090 computer some operations interpret a word as a signed 35-bit integer and others interpret it as an unsigned 36-bit integer. If a negative integer is examined by one of the latter, the sign bit is assumed to represent a contribution of 2^{35} to the magnitude of the number.

(where N is an integer — not the address of an integer) reserves an N-word block in the "remote program," while the command

```
POLBEG    N,COMMON                    (66)
```

reserves an N-word block in "common storage." If COMMON is used, the space occupied by the loader at loading time can be a part of the data buffer at execution time. Therefore the size of the data buffer can be somewhat larger. However, no other program using COMMON can be loaded at the same time without careful use of ORIGIN cards.

3.5.2 *VARTYP*

Every program which uses POLSTV, POLSST, POLDIF, or POLDFO must contain at least one VARTYP declaration. The command

```
VARTYP    T                           (67)
```

indicates that all subsequent references to variables (prior to the next VARTYP declaration if any) are of type T, which may be any of the following

NAM	(name)	
NUM	(number)	(68)
NAM*	(address of name)	
NUM*	(address of number)	

The variables in a format statement are numbered according to the order of their appearance.

For example, if we wish to differentiate the polynomial P with respect to the variable X, we use NAM and write

```
POLDIF    Q,P,X                       (69)
```

To differentiate P with respect to the third variable we use NUM and write

```
POLDIF    Q,P,3                       (70)
```

To differentiate P with respect to the variable whose name is at location LX, we use NAM* and write

```
POLDIF    Q,P,LX                      (71)
```

Finally, to differentiate P with respect to the variable whose number is at location K, we use NUM* and write

```
POLDIF    Q,P,K                       (72)
```

Typically, NAM is used in main programs and NUM* in subroutines, since the main programmer usually knows the names of the variables while the subroutine programmer usually knows nothing about the format.

3.5.3 *Sample Programs*

The following program computes $R = P + \partial Q/\partial Y$.

```
        POLBEG   10000
        VARTYP   NAM
FMT     POLCVF   (X,12,Y,12,Z,12)
        POLRDP   P,FMT
        POLRDP   Q,FMT
        POLDIF   DQDY,Q,Y
        POLADD   R,P,DQDY                    (73)
        POLPRT   R,-,(R = P + DQ/DY)
        TRA      ENDJOB
P       PZE
Q       PZE
R       PZE
        END
```

A slightly more complicated example illustrates the use of indexing. To compute

$$R_i = P_i + Q_i\,; \qquad i = 1, \cdots, 10 \tag{74}$$

we write

```
        POLBEG   10000,COMMON
        POLDRF   FMT
        AXT      10,1
RD1     POLRDP   (P,1),FMT
        TIX      RD1,1,1
        AXT      10,1
RD2     POLRDP   (Q,1),FMT
        TIX      RD2,1,1
        AXT      10,1
ADD     POLADD   (R,1),(P,1),(Q,1)
        TIX      ADD,1,1
        AXT      10,1                        (75)
```

```
PRT     POLPRT    (R,1),-,(R(I)=P(I)+Q(I))
        TIX       PRT,1,1
        TRA       ENDJOB
FMT     BSS       20
P       BES       10
Q       BES       10
R       BES       10
        END
```

The storage section of a main program must contain a block for each format statement read by POLRDF and a pointer (whose address field initially contains zero) for each polynomial. For further discussion of these rules see Section 3.2.

3.6 *Loading Instructions*

The polynomial portion of the ALPAK system consists of a macro deck and two subroutine packages, ALPAK1 and ALPAK2. Most of the macros expand into calling sequences for subroutines of the same name, but a few call one or more differently named subroutines and a few others call no subroutines at all. The macro deck is available as a symbolic deck or as a CRUNCH deck with no END card crunched in. ALPAK1 and ALPAK2 are available as binary decks and also as symbolic decks or CRUNCH decks. In their present form these decks can only be used within the BE-SYS-4 monitor system on an IBM 7090 computer.

The following example illustrates the arrangement of decks and control cards for a typical ALPAK assembly:

```
JOB
FAP
UNLIST
MACROS (CRUNCH deck with no end card crunched in)          (76)
LIST
MAIN PROGRAM (Symbolic deck with END card)
```

The UNLIST and LIST cards are normally included in order to suppress the printing of eleven pages of macro definitions. This is a FAP assembly and may be embellished in any way that conforms to the rules of FAP.

The next example shows a typical arrangement of decks and control cards for assembly and run:

JOB
FAP
UNLIST
MACROS (CRUNCH deck with no END card crunched in)
LIST
MAIN PROGRAM (Symbolic deck with END card)
LOAD BATCH (77)
ALPAK1 (Binary deck, preceded by LOAD card and followed by binary transfer card)
ALPAK2 (Binary deck, preceded by LOAD card and followed by binary transfer card)
TRA
DATA

Our final example illustrates a run with a previously assembled main program:

JOB
MAIN PROGRAM (Binary deck preceded by LOAD card and followed by binary transfer card) (78)
ALPAK1 (Binary deck preceded by LOAD card and followed by binary transfer card)
ALPAK2 (Binary deck preceded by LOAD card and followed by binary transfer card)
TRA
DATA

3.7 *Diagnostics*

The ALPAK diagnostic mechanism recognizes the following ten types of failure:

1. COEFFICIENT OVERFLOW. No coefficient or scalar can have magnitude greater than $2^{35} - 1$.
2. EXPONENT OVERFLOW. No exponent can be greater than $2^B - 1$, where B is the corresponding field width (in bits).
3. INSUFFICIENT SPACE. The reporting subroutine was unable to obtain needed space from the storage allocator.
4. ILLEGAL SUBROUTINE ARGUMENT. One of the inputs to the reporting subroutine failed some simple test.
5. INCOMPATIBLE FORMATS. See *Format Compatibility* in Section 3.1.2.
6. INTERNAL INCONSISTENCY. There may be a bug in the reporting subroutine.

7. POLBEG NOT CALLED. Every main program must begin with the macro POLBEG (see Section 3.5.1).
8. ILLEGAL FORMAT CARD. See *Format Statements* in Section 3.2.4.
9. END OF FILE. All the data cards have been read.
10. INPUT READING ERROR. An unrecoverable parity check failure has occurred on input.

Whenever a failure is detected, control is transferred to the REMARK subroutine, which performs the following functions: First it takes a hollerith snapshot of two locations containing the words "REMARK SNAP" in BCD. The purpose of this is to provide a console dump at the time of the failure. It then prints the location of the failure, the type of failure, and the subroutine nesting list. Finally it transfers control to the DUMP section (if any) of the first subroutine on the nesting list, whose function is to print the inputs and perhaps a partial result.

As an example, suppose the multiplication

```
POLMPY      C,A,B                    (79)
```

fails because of insufficient space. This might result in the output

```
LOCATION 1703
POLDUP REPORTS
INSUFFICIENT SPACE

SUBROUTINE NESTING LIST
NAMES AND CALLING LOCATIONS
    POLMPY 00174, POLOMP 03412, POLDUP 03152

FINAL DUMPS FROM POLMPY

R = P*Q. PS = PARTIAL SUM.

P          X Y Z
       1   0 1 0
       1   1 0 0

Q          X Y Z
       1   0 0 2
       1   0 2 0
       1   2 0 0
```

```
PS        X Y Z
      1   0 1 2
      1   0 3 0
      1   2 1 0
```

and the snapshot

```
                2175    SNAP    H,2410,2411
AC...MQ...SI...EK...SW...SL...OVF...TM...
IR1...IR2...IR4...
                2140    "REMARK"    "SNAP."
```

which will be the last snap prior to post mortems. The output indicates that POLMPY (multiply) was called from location 174 in the main program, POLOMP (one-term multiply) was called from location 3412 in POLMPY, POLDUP (duplicate) was called from location 3152 in POLOMP, and the space shortage was discovered at location 1703 in POLDUP. Furthermore, POLMPY was attempting to compute $R = P*Q$ where

$$\begin{aligned} P &= X + Y \\ Q &= X^2 + Y^2 + Z^2 \end{aligned} \tag{80}$$

and had obtained the partial result

$$PS = X^2Y + Y^3 + YZ^2 = Y(X^2 + Y^2 + Z^2) \tag{81}$$

which is the product of Q and the first term in the canonical ordering of P. Note that P, Q and R in the output are dummy names, which in this case correspond to A, B and C in the user's program [see (79)].

The subroutine nesting list is maintained automatically by the ENTER and EXIT macros, which are used in all but the lowest level subroutines. If a failure is detected in one of these unentered subroutines, its name will appear along with the location of the failure but not on the nesting list.

3.8 *Debugging*

The normal method of debugging an ALPAK program is to run it and see what happens. Most programming errors and all overflows will be located and identified by the diagnostic mechanism, which is described in the preceding section. If difficulties persist, POLPRT (print) orders can be inserted (by reassembly) into the main program, or even into one or more of the ALPAK subroutines. Each POLPRT order is essentially a symbolic snapshot. If an error is detected by POLPRT, a suitable

remark (see below) is printed but the flow of control is not affected (unless it depends on the AC, the MQ, or XR4). The following remarks are available:

1. EMPTY POINTER. The pointer contains ± 0.
2. NO DATA. The number of terms is $+0$.
3. GARBAGE. Either the heading is idle (see Section 3.2.7), the data address is outside the data buffer, the number of terms is $\leqq -0$, or the number of exponent words per term is $\leqq 0$.
4. ILLEGAL FORMAT. Since POLRDF and POLCVF do not accept illegal format statements, this remark implies that the format address is wrong or the format statement has been overwritten.
5. DATA OVERFLOW. The data block begins in the data buffer but ends beyond it.

If all else fails, ordinary snaps and/or post mortems can be taken in the usual manner. However, a snap of the data buffer is unusually difficult to comprehend and should be taken only in desperation.

IV. ACKNOWLEDGMENTS

The author is indebted to M. D. McIlroy for many valuable discussions throughout the course of this work, and to B. A. Tague and J. P. Hyde for major contributions to the planning and programming. He also wishes to thank S. P. Morgan and all of the above for their critical reading of the manuscript and their many suggestions for improvements in the presentation.

REFERENCES

1. Birkhoff, G., and MacLane, S., *A Survey of Modern Algebra*, rev. ed., MacMillan Company, New York, 1953.
2. Slepian, D., On the Zeros of Gaussian Noise, Ch. 6 of Proceedings of Symposium on Time Series Analysis, ed. Rosenblatt, M., John Wiley and Sons, Inc., New York, 1963.
3. Runyon, J. P., unpublished work.
4. Morrison, J. A., unpublished work.
5. Takács, L., A Single-Server Queue with Feedback, B.S.T.J., **42**, March, 1963, p. 503.
6. Riordan, J., *Stochastic Service Systems*, John Wiley and Sons, Inc., New York, 1962, p. 50.
7. Brown, W. S., and Fairlie, D. B., Analyticity Properties of the Momentum-Vertex Function, J. Math. Phys., **3**, March–April, 1962, pp. 221–235.
8. Brown, W. S., and Fairlie, D. B., Some Examples from Perturbation Theory, Ch. 14 of *On the Analytical Properties of the Vertex Function with Mass Spectral Conditions*, by Brown, W. S., Princeton University Report, 1961, pp. 251–275.
9. Thurston, R. N., Wave Propagation in Fluids and Normal Solids, Ch. 1 of *Physical Acoustics*, ed. Mason, W. P., Academic Press, New York, 1963.