Programming
Laboratory
Report

Proceedings of the 1968 Summer Institute on Symbolic Mathematical Computation

Robert G. Tobey, Editor

June 1969

Federal Systems Center

IBM

FSC69-0312

PROCEEDINGS OF THE 1968 SUMMER INSTITUTE ON SYMBOLIC MATHEMATICAL COMPUTATION

Robert G. Tobey, Editor

June 1969

IBM Boston Programming Center
545 Technology Square
Cambridge, Massachusetts 02139

PREFACE

The 1968 Summer Institute on Symbolic Mathematical Computation was held at IBM's Boston Programming Center from June 24 to August 16, 1968. This research institute was sponsored by the IBM Federal Systems Division through independent research and development funds and in conjunction with the Office of Naval Research contract N00014-68-C-0479 and the University of North Carolina through NASA university ~~grant 325-NAS-5~~. The purpose of the institute was to provide an intensive research environment with access to the PL/I FORMAC batch system and the Scope FORMAC interactive system. Its goals were to stimulate investigators to apply these two systems to real problems and to encourage research on the design and implementation of mathematical algorithms for symbolic mathematics.

The main features of the institute were the close proximity and varied disciplines of the participants, and the access to the PL/I FORMAC System. The institute featured formal and informal lectures on various aspects of FORMAC and other key developments in symbolic mathematics. Informal colloquiums and frequent coffee hours stimulated the flow of ideas and helped in the definition and pursuit of individual research projects .

The varied concerns expressed in the papers published here reflect the participants' wide range of interests and activities. The first section, "Basic Design Issues," begins with a discussion of the REDUCE system and the multifaceted problem of substitution by Anthony Hearn. Peter Marks' presentation of design and encoding tradeoffs in FORMAC follows.

The section "FORMAC and Applications" begins with a calculation of Hilbert matrices by Elizabeth Cuthill, then a paper by Robert Tobey and John Lipson, describing the Scope FORMAC language with elementary examples. Next, Stanley Gershwin discusses an application of FORMAC which did not work. Henry Feldman presents ideas concerning the use of FORMAC to perform finite field arithmetic and makes a plea for a more powerful polynomial manipulation capability within FORMAC. Kenneth Hartt presents his view of the symbolic numeric capabilities required to attack the large scale problems of theoretical physics. The use of FORMAC as an empirical aid in obtaining new results in optimal control theory is presented by Stanley Gershwin in the last article of this section.

The "Design and Analysis of Mathematical Algorithms" section begins with a paper in which Robert Risch presents a tutorial discussion of his algorithm for elementary function integration. Next, an analysis toward defining a formula manipulation subsystem for computing and manipulating asymptotic expansions is presented by John Halton and followed by George Collins' paper concerning computing time analysis for arithmetic and algebraic algorithms. In the last paper of the section, John Lipson develops an algorithm for the solution of linear equations with symbolic coefficients, performs timing analysis on the algorithm, and presents several applications.

The Proceedings concludes with a paper on significant problems in symbolic mathematics by Robert Tobey.

These Proceedings should not be viewed as complete within themselves. Rather they should be viewed as constituting a snapshot or instantaneous cross-section of the issues and probings which comprised research into symbolic mathematical computation in the year 1968. A complete picture emerges only when the many articles referenced in these Proceedings are also studied.

Participants in the Summer Institute

Resident Participants (in attendance eight weeks)

Dr. Elizabeth Cuthill, Naval Ship Research and Development Center
Professor John H. Halton, University of Wisconsin
Professor Kenneth Hartt, University of Rhode Island
Professor Don Mittleman, University of Notre Dame
Mr. Sami Al-Banna, graduate student, Columbia University
Mr. Henry A. Feldman, graduate student, Harvard University
Mr. Stanley B. Gershwin, graduate student, Harvard University
Mr. John D. Lipson, graduate student, Harvard University

Consulting Participants (present one week only)

Mr. Charles L. Baker, IBM Federal Systems Center
Professor George E. Collins, University of Wisconsin
Professor Anthony C. Hearn, Stanford University
Dr. Robert H. Risch, Systems Development Corporation

Casual Lecturers

Mr. Carl Engelman, MIT Lincoln Laboratory
Dr. James H. Griesmer, IBM Research Center
Dr. Fred G. Gustavson, IBM Research Center
Professor William Martin, MIT
Professor Joel Moses, MIT

Boston Programming Center Representatives

Mr. James J. Baker
Mr. Peter Marks
Mr. Jack B. Nance, Jr.
Dr. Robert G. Tobey

CONTENTS

Page

BASIC DESIGN ISSUES

FORMAC AND APPLICATIONS

DESIGN AND ANALYSIS OF MATHEMATICAL ALGORITHMS

Page

SUMMARY

BASIC DESIGN ISSUES

THE PROBLEM OF SUBSTITUTION

by

Anthony C. Hearn
Department of Physics
Stanford University
Stanford, California

Abstract

Minimizing intermediate expression swell is a central problem in symbolic mathematics system design. The strategy chosen for implementing substitution is frequently the key to keeping storage space requirements under control. This paper discusses the REDUCE system and its approach to substitution. The use of interactive substitution to reduce the complexity of program output is also covered.

This paper reports research sponsored by the Air Force Office of Scientific Research, Office of Aerospace Research, U.S. Air Force, under AFOSR Contract Number F44620-68-C-0075. Computer time was supported by the Stanford Artificial Intelligence Project under the sponsorship of the Advanced Research Project Agency of the Office of the Secretary of Defense (SD-183). The author is an Alfred P. Sloan Foundation Fellow.

THE PROBLEM OF SUBSTITUTION

by

Anthony C. Hearn

1. INTRODUCTION

One of the most significant features of programs designed for nonnumeric calculation is that the size of the expressions manipulated, and hence the amount of storage necessary, changes continually during the execution of the program. The user cannot always know ahead of time just how much output his program will produce or whether the calculation will fail because of inadequate computer memory. The key to controlling both the size of intermediate expressions and the complexity of output often lies in the manner in which substitutions for variables and expressions declared by the programmer are implemented by the system. In this paper, we discuss various methods developed to perform these substitutions in the author's own system, REDUCE.[1,2] The REDUCE system, like FORMAC,[9] is designed for general algebraic computations of interest to mathematicians, physicists, and engineers. Although the two systems share many capabilities, there are marked differences in the design of each. REDUCE was originally designed to handle the special problems of non-commutative and tensor algebra encountered in calculations in elementary particle physics scattering theory.[3] However, it was found that the techniques employed could be extended to handle many problems involving manipulation of large algebraic expressions by known algorithmic methods.

One major difference between FORMAC and REDUCE is that the former is largely a machine-coded system, while the latter was programmed entirely in LISP 1.5.[4] The big advantage of the LISP language is that it permits the development of a system which may be easily modified and which is also relatively machine-independent. Thus, the same program is operating at Stanford on two entirely different machines, an IBM System/360 Model.67 and a Digital Equipment Corporation PDP-6.

Section 2 of this paper uses the simplest type of substitution problem to introduce the REDUCE system and to discuss the general characteristics which permit the efficient coding of many types of substitutions. Section 3 presents the general problem of substitution in terms of the matching of expressions. Finally, the use of substitution to reduce the complexity of program output is discussed in section 4.

2. SIMPLE SUBSTITUTIONS

An assignment statement of the form

```
A = 2*B*C - 3*D**2 + COS(-X)*COS(Y)                    (2.1)
```

has an entirely different interpretation in a nonnumeric calculation than in a PL/I or FORTRAN program. In the latter cases, the right-hand side evaluates to a number which is then stored in a machine location reserved for A. In a non-numeric system, the evaluation of such an expression is more ambiguous. Here evaluation is usually referred to as simplification in the sense that the expression is reduced to a canonical form by rules built into the program or provided by the user.

There are as many philosophies concerning the meaning of simplification of expressions as there are systems. FORMAC, for instance, makes substitutions for variables with assigned values and performs several unambiguous reductions of the expression. In Equation (2.1), for example, COS(-X) would be replaced by COS(X). However, the basic form of the expression would remain the same, apart from conversion from an infix notation to an internal Polish prefix representation. In REDUCE, on the other hand, reduction of an expression to canonical form is more complicated. For one thing, this reduction always involves expansion of expressions, an operation under user control in FORMAC. The flexibility permitted by having expansion of expressions under user control is not present in REDUCE because many operations associated with high-energy physics calculations require the expression to be in a fully expanded form. Also, there is often a considerable gain in speed of calculation and decrease in storage requirements when such expansions are made in an early stage.

For the sake of simplicity in discussing the canonical form used let us begin by restricting ourselves to rational functions of polynomials in several variables. The simplification operation reduces such expressions to a canonical form consisting of a pair of standard forms which represent the numerator and denominator of the expression. In the standard form, which is similar to that described by Collins in reference 5, an expression in n variables $f(x_1,x_2 \ldots x_n)$ is written as a power series in a variable whose coefficients are functions of n-1 variables. Thus

$$f(x_1,x_2 \ldots x_n) = \sum_{i=0}^{m_1} f_i(x_2 \ldots x_n)x_1^i \qquad (2.2)$$

The polynomial coefficients are expanded in a similar manner, and the representation is continued until only integers remain. In Backus normal form, using the LISP dotted pair notation, the REDUCE standard form is

```
[standard form] : :  = () | [nonzero integer]
                       |( [standard term] .  [standard form] )   (2.3)
[standard term] : :  = ( [standard power] .  [standard form] )   (2.4)
[standard power] : : = ( [variable]  .  [nonzero
                                         positive integer] )     (2.5)
```

Thus, a standard term represents one term in the power series Equation (2.2), and a standard power represents a variable raised to a positive integer power. Comparison of Equation (2.2) with the equations that follow also shows that the dotted pair represents an implicit addition in Equation (2.3), multiplication in Equation (2.4), and exponentiation in Equation (2.5).

Since the same fixed power of a given variable appears many times in the expanded form of large expressions, considerable storage is saved by storing all standard powers uniquely on a single ordered list. Although no explicit attempt is made to store other subexpressions which occur several times, this often occurs automatically.

An ordering convention based on the machine location of the variables in core is used to decide the position of a variable in a standard form. Thus, two equal polynomials will have the same standard form.

If fractional powers of variables or expressions are encountered during reduction, a new variable is created to represent that power, and the user is informed to ensure that no fractional powers remain in the standard form. Likewise, real numbers are usually converted to the ratio of two integers, unless the user specifies floating point arithmetic.

An extension of the basic polynomial representation to include other operators is made in a straightforward manner. Each operator in the system has a simplification function associated with it. This function may transform its arguments in either of two ways. First, it may convert the expression completely into other operators in the system, leaving no functions of the particular operator for further manipulation. This is, in a sense, true of the simplification functions associated with the operators +, * and / , for example, because the standard form does not include these operators explicitly. It is also true of an operator such as the determinant operator DET , in which case the operator DET no longer appears after the relevant simplification function calculates the appropriate determinant. On the other hand, the simplification process may leave some residual functions of the relevant operator. For example, a residual expression COS(Y) will remain after simplifying Equation (2.1) unless a rule for the reduction of cosines into exponentials is introduced. These residual functions of an operator are termed kernels, and are stored uniquely like variables. Subsequently, the kernel is carried through the calculation as a variable unless transformations are introduced for the operator at a later stage. To include kernels in our standard form representation, we simply replace Equation (2.5) with

```
[standard power] : : = ([kernel] . [nonzero
                        positive integer])               (2.6)
```

and add

```
[kernel] : : = [variable] |
               ([operator] . [simplified list of arguments])      (2.7)
```

Often an assignment statement such as Equation (2.1) is intended in the sense of a "side relation" in that a substitution for A should be made if it occurs in expressions encountered later in the calculation. As the initial reduction of an expression to canonical form often involves considerable computation, it is obviously desirable to simplify it only when necessary, and then only once during a calculation. In such circumstances, no replacement or other simplification is necessary at the time the substitution is defined, and the expression may therefore be stored in quoted form rather than evaluated form. To indicate this, Equation (2.1) might best be written as

```
A = '2*B*C - 3*D**2 + COS(-X)*COS(Y)'                    (2.8)
```

In REDUCE, a quoted assignment is introduced by the instruction LET , as in

```
LET A = 2*B*C - 3*D**2 + COS(-X)*COS(Y)                  (2.9)
```

whereas an intended simplification is written

```
SIMPLIFY A = 2*B*C - 3*D**2 + COS(-X)*COS(Y)             (2.10)
```

When an expression to be simplified contains variables which were previously assigned either quoted or evaluated values, the speed of the calculation and, more important, the amount of storage used often depend crucially on just when the substitution for the relevant variables is made. There are many ways to make such substitutions. One is to substitute for variables as they are met during reduction to canonical form recognizing, as AUTSIM does in FORMAC, variables for which substitutions were already reduced to canonical form to avoid repetitious calculation. A second way is to make substitutions after reduction of the whole expression to canonical form.

Two simple examples will illustrate that neither method is better in all circumstances. With the substitution (2.1) already defined, consider the following assignments:

```
A1 = (A - 2*B*C + 3*D**2 - COS(-X)*COS(Y))**1000         (2.11)

B1 = A**1000 - A**1000                                    (2.12)
```

Although both A1 and B1 evaluate to zero, in the case of A, it is obviously better to substitute before raising the expression to the thousandth power and simplifying; in the case of B, the opposite is true. These are extreme cases, of course, but they illustrate what can happen if you are not careful. Both substitution mechanisms are implemented in REDUCE, and the decision as to whether the substitution of variables is made during or after reduction to standard forms is, to a limited extent, under user control. However, it has been found in practice that the system can often make a better decision than the average user in this regard.

We note in passing that a simplification of

```
C1 = (A - 2*B*C + 3*D**2 - COS(-X)*COS(Y))**1000
         + A**1000 - A**1000                                (2.13)
```

would involve catastrophic term growth regardless of which of the above methods of substitution is used. The user could avoid this by simplifying A1 and B1 as described above, then adding them to form C1 . However, a simplification step such as Equation (2.13) may occur in the middle of an extensive calculation without the user's knowledge. To ask the system to make such a simplification in a single step would require sophisticated heuristics far beyond the scope of present simplification systems to decide the optimal substitution method for each variable encountered.

As an extension of the simple variable substitutions discussed so far, REDUCE allows the user to define substitutions for powers of variables and expressions which reduce to kernels or powers of kernels. Again, these substitutions may be made either during reduction to canonical form or after and, because of the organization of the system, they are as efficient to implement as substitutions for variables. We illustrate this by discussing in detail the mechanism for substituting for kernels or kernel powers after reduction of an expression to canonical form. There are two ways this may be done. The first is to scan the expression and check whether each kernel has a substitution defined for it. If it does, at the first occurrence of this kernel check its substitution in canonical form for replacements by the same routine, store it in this new form, then continue the search to the end of the expression. The reconversion of the expression to canonical form can be made concurrent with the search procedure. This method is somewhat inefficient, however, as the same kernel often occurs many times in an expression. The second method recognizes this inefficiency and performs the substitutions in three passes, as we illustrate with a substitution for the kernel (COS X) in Figure 1. The lists of kernels in the system are searched, and changes in the appropriate list structure pointers are made if a substitution is required as shown in Figure 1b. Since kernels and kernel powers are stored uniquely, this change in list structure means that every occurrence of the substitution expression was changed in all expressions. Thus, the expression being simplified can be reconverted to standard forms by a second pass. The last pass, which is quite trivial, restores the original list structure of all substituted kernels without affecting the reconverted canonical form, as shown in Figure 1c.

In actual practice, REDUCE uses a combination of these methods. It is assumed that the expression is large and the substitutions are relatively small. To check whether a substitution expression contains terms which have substitutions themselves, the first method is used; the second method is used on the expression being simplified.

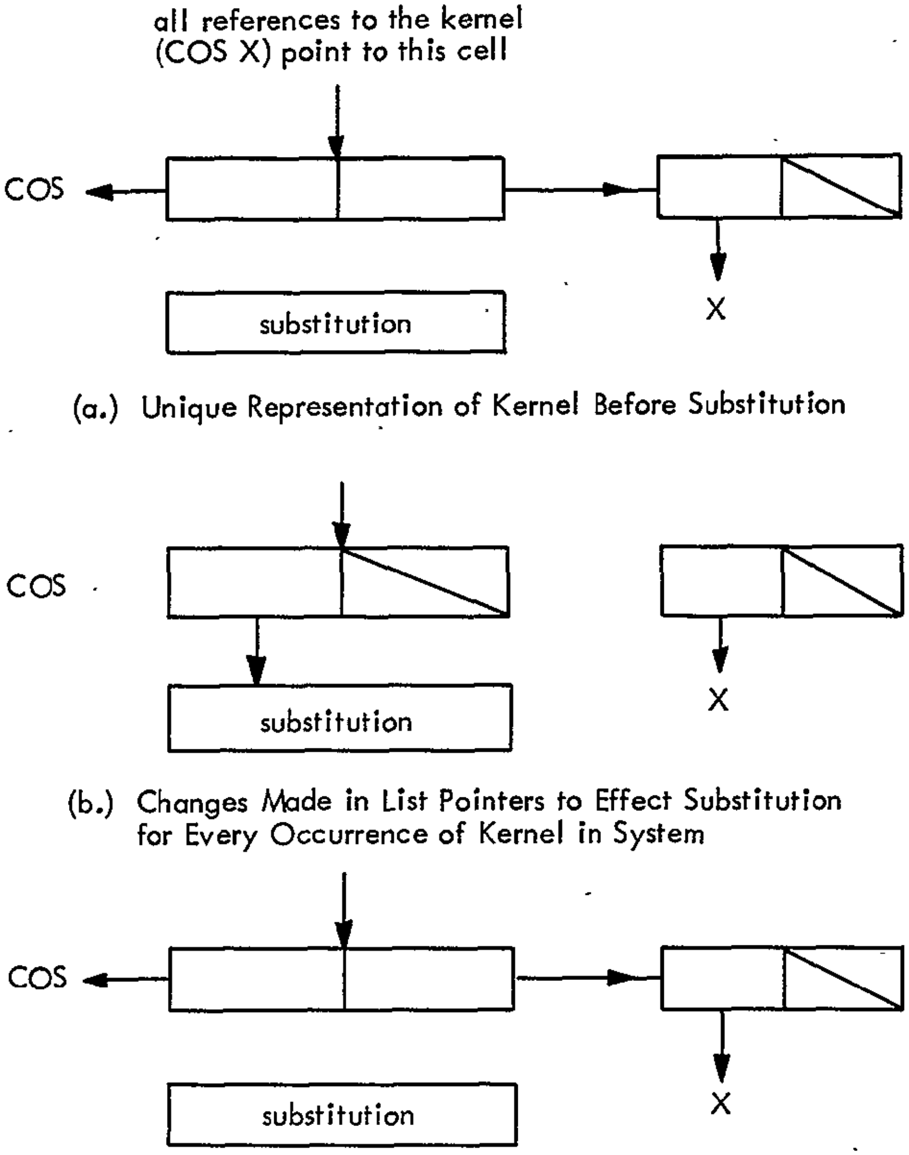
In substituting for powers of variables or kernels, a distinction must sometimes be made between substituting for that explicit power and generally substituting for that power whenever it occurs. For example, LET I**2 = -1 implies that I**3 = -I, I**4 = 1 , and so on. However, in integrating an expression by explicit substitution, a substitution

```
X**2 = Y**3/3
```

is not intended to apply to higher powers of X.

This latter type of substitution is really a matching operation, and is treated as such by the REDUCE system. Thus, a user would say

```
MATCH X**2 = Y**3/3                                    (2.14)
```



(a.) Unique Representation of Kernel Before Substitution

(b.) Changes Made in List Pointers to Effect Substitution for Every Occurrence of Kernel in System

(c.) Restoration of Kernel Representation After Reconversion of Expression to Canonical Form

Figure 1. A Substitution Mechanism for Kernels

to effect such a replacement. The general matching operation requires an altogether different programming technique than we have used so far. This operation is discussed in the next section.

3. MATCHING OF EXPRESSIONS

The substitutions considered so far have been rather limited in scope, as they involve only substitutions for variables and kernels. As we have seen, these are very efficient to implement because variables and kernels are stored uniquely in REDUCE. However, a more general type of substitution which requires extensive pattern matching within a given expression is often needed. Such substitutions cannot be as efficiently implemented as our earlier examples as much more searching is involved in their application.

The ideal system would allow for the replacement of any given expression f(a,b..x,y..) by another expression g(a,b..x,y..) where a,b,... stand for fixed subexpressions and x,y,... for arbitrary expressions. For example, in Equation (2.1) it might be convenient to replace COS(X)*COS(Y) by (COS(X+Y) + COS(X-Y))/2 . Presumably this type of replacement should apply whenever an arbitrary product of cosines is encountered, so that X and Y in the replacement rule should stand for any expression. Thus X and Y are free variables as far as the substitution rule is concerned. Similarly, if X is free, the rule

$$\text{SIN(X)**2} + \text{COS(X)**2} = 1 \tag{3.1}$$

should imply that $\sin^2(\cos(\log 2)+3) + \cos^2(\cos(\log 2)+3)$ is to be replaced by 1.

This general matching problem, which we mentioned in an earlier publication,[2] has been solved efficiently enough for use in large scale calculations, and as a result, most systems, including REDUCE, compromise at some point in the types of substitutions allowed. There is also a basic ambiguity associated with any substitution rule involving addition, such as Equation (3.1). For example, given this rule, should $2\cos^2(v) + \sin^2(v)$ be replaced by $1 + 2\cos^2(v)$, by $2 - \sin^2(v)$ or left unchanged? As we shall see in section 4, the choice made can often influence the compactness or symmetry and, hence, the intelligibility of the result.

Though REDUCE does not implement a general pattern matching algorithm, it does provide for substitutions for products of kernel forms or expressions which reduce to this form by means of the instruction MATCH.

The argument of MATCH is a list of equivalence expressions of the form

```
[kernel form] * [kernel form] ... * [kernel form]
= [expression]                                            (3.2)
```

where a kernel form is an expression which reduces to a kernel on simplification. Examples of the use of MATCH are

```
MATCH A**2*B = 3*C,
COS(X)*COS(Y) = (COS(X+Y)+COS(X-Y))/2                     (3.3)
```

In the second example, the fact that X and Y may stand for any expression is signified by the prior declaration

```
FREE X,Y.
```

The "matching" function which implements these substitutions is applied recursively to standard forms and has two arguments—the form and a list of substitution rules as given in Equation (3.2). Unless the form is an empty list or a number (in which case it is simply returned), the leading standard term is inspected. By Equation (2.4) this is a dotted pair of a standard power and another standard form. If the kernel in this power occurs in the left half of a substitution rule, two things are possible. If it is the only kernel in the left half, a complete "match" has been found, and the kernel is replaced by the right half of the rule. On the other hand, if other kernels remain in the left half, a new substitution rule is generated by moving the relevant kernel as a divisor to the right half of the substitution.

If no complete match has been found after all rules have been scanned for the kernel, the matching function is applied, with the additional substitution rules just generated, to the standard form which was paired with the kernel power in the leading term. This process continues until the whole expression has been scanned. If any match was successful during this scan, a second pass reconverts the expression to canonical form. In addition, the whole process must be repeated in case another valid match developed during the reconversion.

The algorithm must be modified somewhat to allow for replacement of an explicit kernel power, as required in Equation (2.14), or for the presence of free variables in a rule. Its efficiency can be increased, moreover, by exploiting the order key built into every standard form.

In spite of the limited nature of the types of substitutions allowed in REDUCE, it is surprising how useful a matching operation of the form defined in Equation (3.2) can be. This is especially true of problems involving analytic integration of multivariable expressions by table look-up which occur quite frequently in elementary particle physics.

4. SUBSTITUTIONS IN OUTPUT

Almost as catastrophic as the growth of expressions during a calculation can be the growth of output to the astonished user. This is not a trivial problem; the author knows several physicists and engineers who gave up calculations when confronted with 50 pages of output from a relatively simple problem in matrix manipulation. If any real progress is to be made in handling algebraic problems too tedious and complicated to be done by hand, a lot of research must be devoted to presenting output in a compact, intelligible form. One way to achieve compact output is to pick out the leading terms in the expression by order-of-magnitude arguments, but this method often conceals symmetries in the answer which can only be seen in the complete expression. Another method, developed by Baker,[7] involves recognizing common subexpressions within an expression and replacing them by a single variable, thus displaying the underlying fundamental or skeletal structure of the expression. In many cases, however, when such underlying structure exists, it is hidden by various relations between the variables occurring or by functional identities such as Equation (3.1).

The example in Figure 2 illustrates the problem of hidden underlying structures very well. Figure 2a shows a "raw" expression produced by the computer. As in many problems in physics and engineering, not all the variables appearing in the expressions are independent, and certain combinations have

```
(       M**4 *

(2 * PROP1 * PR * RS   -   2 * PROP1 * PR * RT   -   4 * PR**2 * RS
 -   4 * PR**2 * RT   -   4 * PR * RS**2   +   14 * PR * RS * RT   +
  2 * PR * RS * PROP2   -   4 * PR * RS * PS   -   4 * PR * RS * PT   -
   4 * PR * RS * QT   -   4 * PR * RS * QS   -   4 * PR * RS * QR   -
  1Ø * PR * RT**2   -   2 * PR * RT * PROP2   +   4 * PR * RT * PS   +
   4 * PR * RT * PT   +   4 * PR * RT * QT   +   4 * PR * RT * QS   -
  4 * PR * RT * QR   -   6 * RS**2 * RT   -   4 * RS * RT**2   -   6 *
 RS * RT * QR   -   6 * RT**3   +   6 * RT**2 * QR)


  +   M**2 *

(  -   PROP1 * PR * RS * RT   +   PROP1 * PR * RT**2   +   PROP1 * P
R * RT * PROP2   +   PROP1 * RS**2 * RT   +   2 * PROP1 * RS * RT**2
 -   2 * PROP1 * RS * RT * PT   +   PROP1 * RT**3   +   2 * PROP1 * RT
**2 * PS   +   6 * PR**2 * RT * QT   -   2 * PR**2 * RT * QS   +   4 *
 PR**2 * RT * QR   -   4 * PR * RS * RT * PROP2   +   4 * PR * RS * RT
 * PS   +   8 * PR * RS * RT * PT   +   4 * PR * RS * RT * QT   +   2
* PR * RS * RT * QS   -   4 * PR * RS * RT * QR   +   8 * PR * RS * PS
 * QT   +   8 * PR * RS * PS * QR   -   4 * PR * RT**3   +   2 * PR *
RT**2 * PROP2   +   4 * PR * RT**2 * PT   +   6 * PR * RT**2 * QT   -
  4 * PR * RT**2 * QS   +   PR * RT * PROP2**2   -   2 * PR * RT * PRO
P2 * PS   -   2 * PR * RT * PROP2 * PT   -   2 * PR * RT * PROP2 * QT
  -   2 * PR * RT * PROP2 * QS   -   8 * PR * RT * PS * QT   -   2 * P
R * RT * PS * QR   -   2 * PR * RT * PT * QR   +   2 * RS**2 * RT * QT
   +   4 * RS**2 * RT * QR   -   4 * RS * RT**3   -   2 * RS * RT**2 *
PROP2   +   4 * RS * RT**2 * PS   -   4 * RS * RT**2 * PT   +   6 * R
S * RT**2 * QT   -   2 * RS * RT**2 * QS   -   2 * RS * RT * PROP2 * P
T   +   RS * RT * PROP2 * QR   +   4 * RS * RT * PS * PT   +   2 * RS
* RT * PS * QR   +   4 * RS * RT * PT**2   +   4 * RS * RT * PT * QT
 +   4 * RS * RT * PT * QS   +   4 * RT**3 * PS   -   2 * RT**3 * QS
 +   4 * RT**3 * QR   +   2 * RT**2 * PROP2 * PS   -   RT**2 * PROP2 *
 QR   -   4 * RT**2 * PS**2   -   4 * RT**2 * PS * PT   -   4 * RT**2
* PS * QT   -   4 * RT**2 * PS * QS   +   4 * RT**2 * PS * QR   +   2
* RT**2 * PT * QR)


  -   2 * PROP1 * RS * RT**2 * PS   -   2 * PR**2 * RT * PROP2 * QT
 +   8 * PR * RS * RT**2 * QT   +   2 * PR * RS * RT * PROP2 * QT   -
   8 * PR * RS * RT * PS * QT   -   8 * PR * RS * RT * PS * QR   -   4
 * PR * RS * RT * PT * QT   -   4 * PR * RT**2 * PS * QT   +   4 * PR
* RT**2 * PS * QS   +   4 * PR * RT * PROP2 * PS * QT   +   2 * PR * R
T * PROP2 * PS * QR   +   4 * RS**2 * RT * PT * QT   -   4 * RS * RT**
2 * PS * QT   -   8 * RS * RT * PS * PT * QT   -   4 * RS * RT * PS *
PT * QR   +   8 * RT**2 * PS**2 * QT) /


(  -   4 * PROP1 * PR * RS * RT**2 * PROP3)
```

(a.) Expression Initially Produced by Computer

Figure 2. Example of Reducing the Size of Output Expressions by Substitution

```
PQ = M**2 - PROP1/2,

PR = QR + RT - RS,

PS = QS + RT - PROP1/2,

PT = QS - PR + RT,

QS = M**2 - PROP3/2,

QT = PS - QR - RT,

PROP2 = PROP1 - 2*RT + 2*RS
```

(b.) Relations Between Variables

```
((4*M**4 - (PROP1+PROP3)**2)*(- 2*M**2*QR - 4*QR*RT

        + 2*RT**2 - RT*(PROP1+PROP3)+(PR*PROP1+RS*PROP3)

        + 2*M**2*PR*RS/RT)

+ 4*M**2*QR*(PR + RS)*(2*M**2 + RT + (PROP1+PROP3))

+ 2*M**2*PR*RS*(2*QR - 6*RT - 3*(PROP1+PROP3))

+ 2*(QR - RT)*((PR*PROP1+RS*PROP3)*(M**2 - (PROP1+PROP3))

        + 2*QR*RT*(PROP1+PROP3))

+ 2*(QR**2 + RT**2)*(2*QR*RT - (PR*PROP1+RS*PROP3)

        + RT*(PROP1+PROP3)) + 6*M**2*RT**2*(PROP1+PROP3))

/ ( 4*PROP1*PROP3*RT*PR*RS)
```

(c.) Final Result Produced by Man and Machine

Figure 2

a more relevant physical interpretation than others. The relations between the variables are given in Figure 2b. It can be seen that only six of the 13 variables are independent. About five man-hours in front of a CRT display modifying expressions and checking within the computer that no errors were introduced by the hand modifications resulted in the expression in Figure 2c. Considerable reduction in the size of the expression was made by appropriate substitutions for the variables appearing in the answer. The explicit skeletal structure of this result could also be displayed by replacing the common subexpressions PROP1+PROP3 and PR*PROP1+RS*PROP3 by simple variables.

The goal of simplification in this context is surely reduction of the size and/or symmetrization of the expression. There is something of an art involved in guessing the right substitutions, but it is obvious that the computer could be programmed to do a lot of this automatically. Although the author's progress in this area during the past year has not been outstanding, some success has been achieved by successive substitution for each relevant variable wherever it occurs in an expression, then checking to determine whether the substitution was successful in decreasing the number of terms in the expression. Because this method is painfully slow, a human and computer interactive combination remain economically more attractive at the moment.

This type of problem is analogous, in many ways, to theorem-proving on a computer, and it is probable that similar heuristics will have to be developed here before a successful solution can be found. There may be other algorithmic methods which could be used also. Engeli,[8] for example, suggested dividing the expression by any substitution equivalent to zero, thus keeping only the remainder for further manipulation. However, the author has found that this provides little reduction in expressions involving low powers of many variables, such as the example in Figure 1.

The problem of substitution, then, is one of the key problems to be considered in designing and using a simplification system for large expressions. Expressions must be kept as compact as possible during a calculation, and the output must be palatable and intelligible to the user if any major new discoveries are to result from nonnumerical mathematical calculations on a computer.

REFERENCES

1. A. C. Hearn, "REDUCE User's Manual," Institute of Theoretical Physics Stanford ITP-292, Stanford Artificial Intelligence Memo No. 50 (revised), Stanford University, Palo Alto, California, April 1968.

2. A. C. Hearn, "REDUCE, A User-Oriented Interactive System for Algebraic Simplification," Proceedings of the ACM Symposium on Interactive Systems for Experimental Applied Mathematics, held in Washington, D. C., August 1967 (to be published).

3. A. C. Hearn, "Computation of Algebraic Properties of Elementary Particle Reactions Using a Digital Computer, "*Communications of the Association for Computing Machinery*, Vol. 9, 1966, p. 573.

4. J. McCarthy, *et al.*, "LISP 1.5 Programmer's Manual," Computation Center and Research Lab of Electronics, MIT Press, Cambridge, Massachusetts, 1965.

5. G. E. Collins, "PM, A System for Polynomial Manipulation," *Communications of the Association for Computing Machinery*, Vol. 9, No. 8, August 1966, p. 578.

6. R. G. Tobey, R. J. Bobrow, and S. N. Zilles, "Automatic Simplification in FORMAC," AFIPS Conference Proceedings, Vol. 27, Part 1, Spartan Books, Washington, D. C., December 1965, p. 37.

7. R. G. Tobey, "Experience with FORMAC Algorithm Design," *Communications of the Association for Computing Machinery*, Vol. 9, 1966, p. 589.

8. M. Engeli, private communication.

9. R. G. Tobey, *et al.*, 'PL/I FORMAC Interpreter, Users Reference Manual," IBM Contributed Program Library, 360 D 03.3004, Hawthorne, New York, October 1967.

DESIGN AND DATA STRUCTURE: FORMAC ORGANIZATION IN RETROSPECT

by

Peter Marks
IBM Boston Programming Center
Cambridge, Massachusetts

Abstract

The interaction between FORMAC data organization and algorithm design is considered. Several organizational improvements are discussed.

The author is currently a lecturer in Computer Sciences at the University of Notre Dame, Notre Dame, Indiana.

PRECEDING PAGE BLANK NOT FILMED

DESIGN AND DATA STRUCTURE: FORMAC ORGANIZATION IN RETROSPECT

by

Peter Marks

1. FORMAC STRUCTURE AND FUNCTION

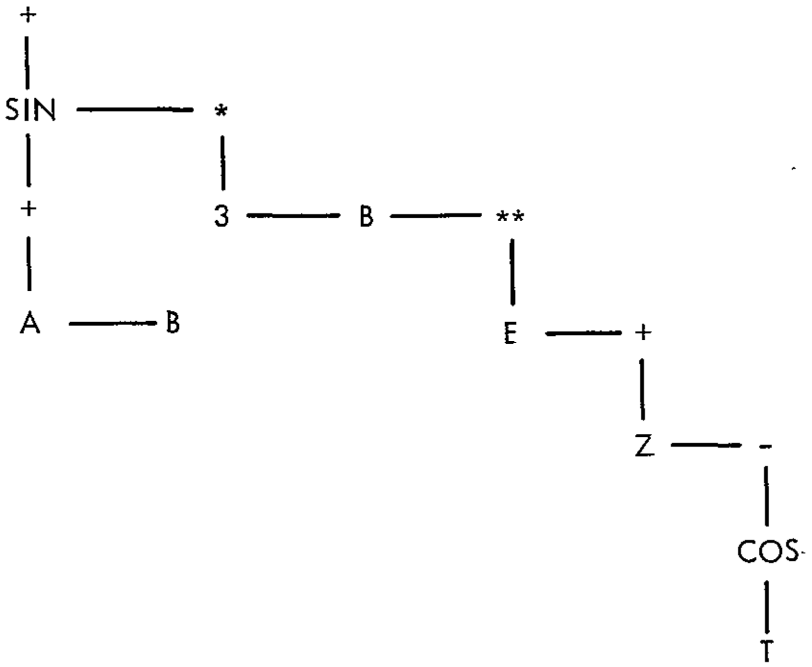
FORMAC[1, 2] effects calculations on explicitly defined analytic functions. The user describes the functions by expressions, and lists representing these expressions are manipulated by a library of subroutines. For example, the function

$$\sin(a + b) + 3be^{z-\cos(t)}$$

is represented by the expression

SIN(A + B) +3 * B * #E ** (Z - COS(T))

and finally by the list



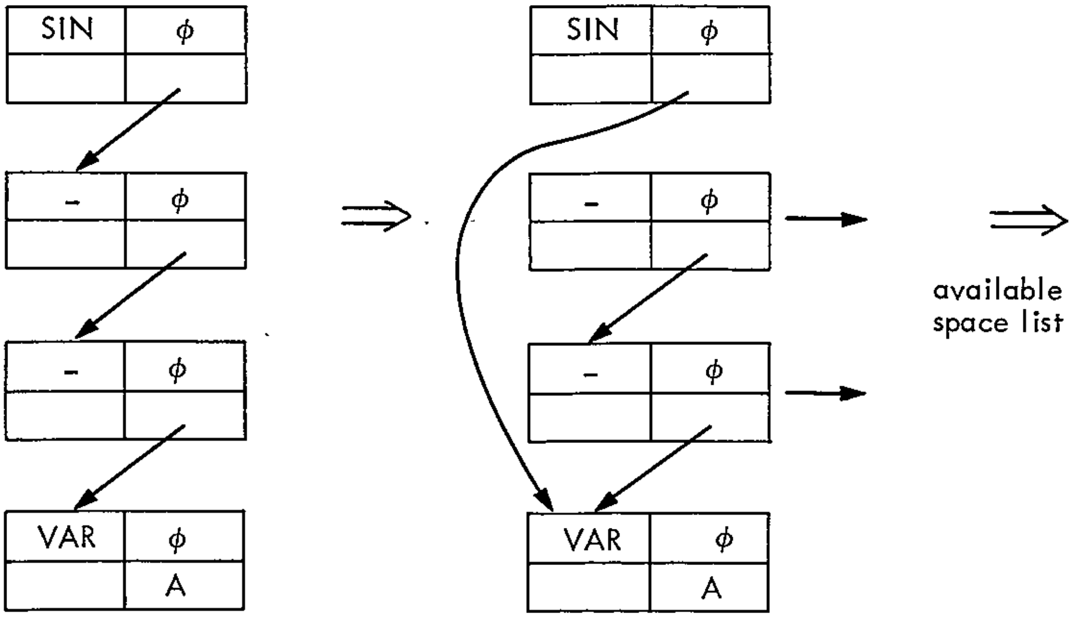
The binary branching structure of the tree is subordinated to a sequence of levels. Nonterminals are operators or functions linked downward to their operand/argument lists.

The FORMAC lists are built from 64-bit nodes:

OP	ACROSS
STAT	DOWN

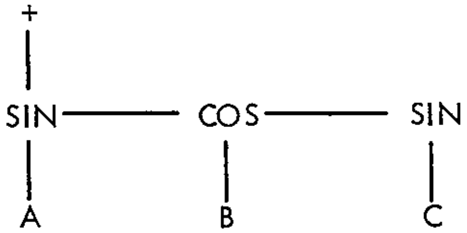
OP is an 8-bit field indicating the type of node (e.g., +, SIN, VARIABLE, CONSTANT). ACROSS is a 24-bit pointer to the next node on the same level. STAT is an 8-bit flag field containing temporary indicators for subroutine use and bits announcing the presence of various patterns at lower levels in the tree (e.g., of a product of sums). DOWN is another 24-bit pointer field. For operators, it points to the operand list; for variables, to the symbol table entry; for constants, to the value.

Some of the FORMAC list transformations act in place. These are the "implicit" simplification manipulations. For example,

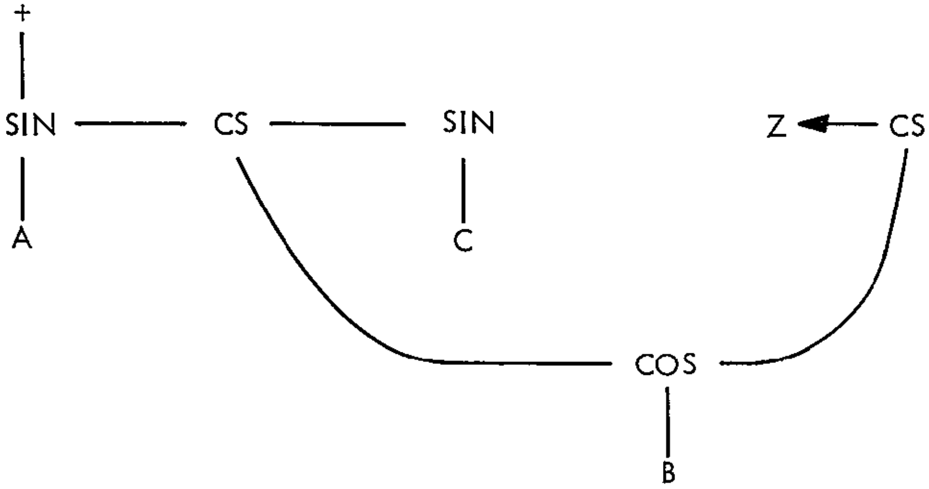


On the other hand, the routines directly callable by the user (e.g., EVAL, DERIV) all create a new list.

The major complication in the data structuring is the provision for multiple references to common subexpressions. Since a sublist appearing in two different trees would require two different ACROSS fields in the top node, a new node, called a CS, is inserted above the common subtree. For example, if the second summand in

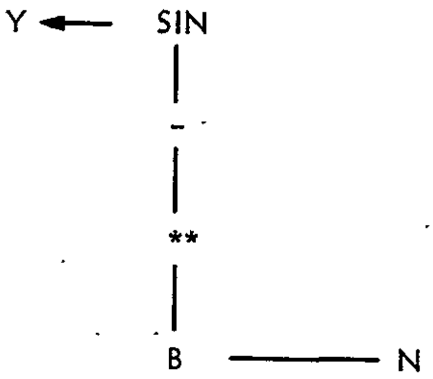


is copied and assigned to Z, the effect is

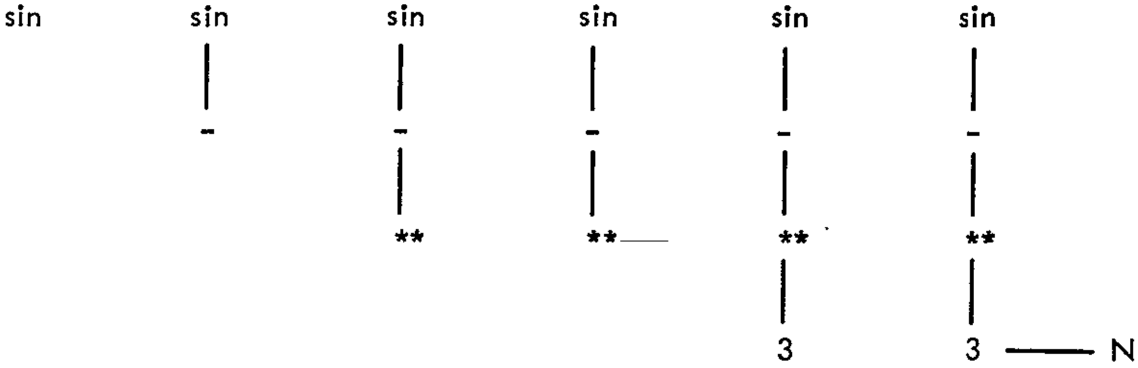


Multiple reference introduces the possibility of side effects: a change to one list may affect another list if the modification is made in a subtree common to both. Another difficulty comes from the attempt to preserve a maximum of common structure. This entails the insertion of CSs at the highest possible level. Thus, for example, an algorithm for replacement which creates

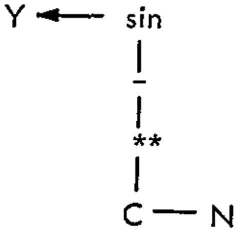
Z = EVAL (Y,B,3) from



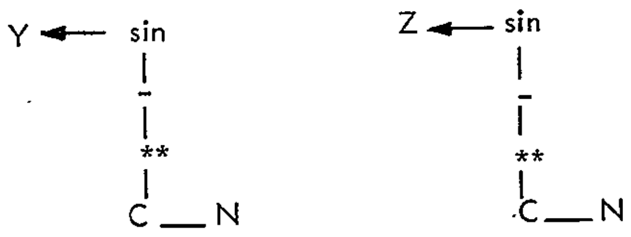
by successively building



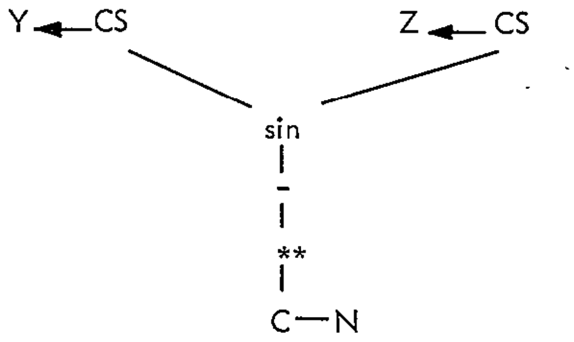
is not satisfactory, because in the case



where no replacement is to be made, this algorithm produces



rather than the desired

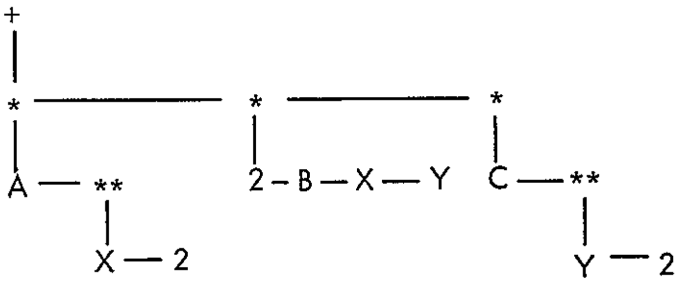


2. CHOICE OF A LIST STRUCTURE

To get a more complete picture of how to utilize successive "degrees of freedom" in the data organization, first consider the current structure. The input expressions themselves, in character string form, have one advantage: the expression

A * X ** 2 + 2 * B * X * Y + C * Y ** 2

requires 21 bytes, the tree



128 bytes. Actually, the string approach can be pushed quite far. For example, if a Polish string is used with + and * as delimited variary operators, then the character string

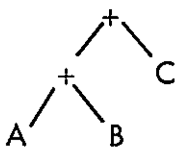
+ * A ** X 2) * 2 B X Y) * C ** X 2))

still requires fewer than 25 bytes, and is quite equivalent to a tree structure for operations producing new strings such as replacement and differentiation.

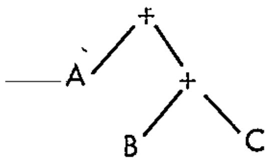
It is the simplification operations, like cancellation of terms, which force the change to a linked structure of uniform-sized elements; the storage allocation problem for rapidly fluctuating variable-sized strings is unmanageable.

The transition from a singly- to a doubly-linked structure again halves the packing density. But the second pointer allows multiple references to common substructures, and this usually more than compensates for the extra link space.

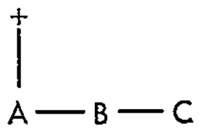
Finally, and here is the beginning of later considerations, a choice must be made between a binary tree organization and the list structure. On the surface, the binary tree is a natural choice, since +, *, and ** are all binary operators. But the list structure allows the distinction between



and



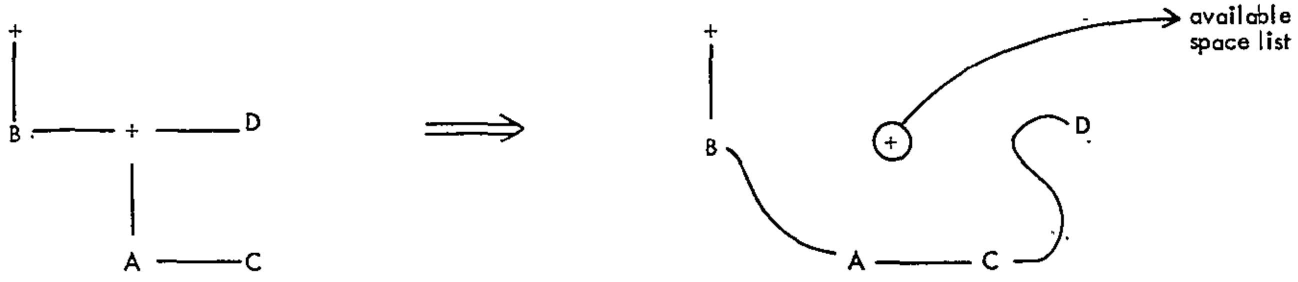
to be lost as



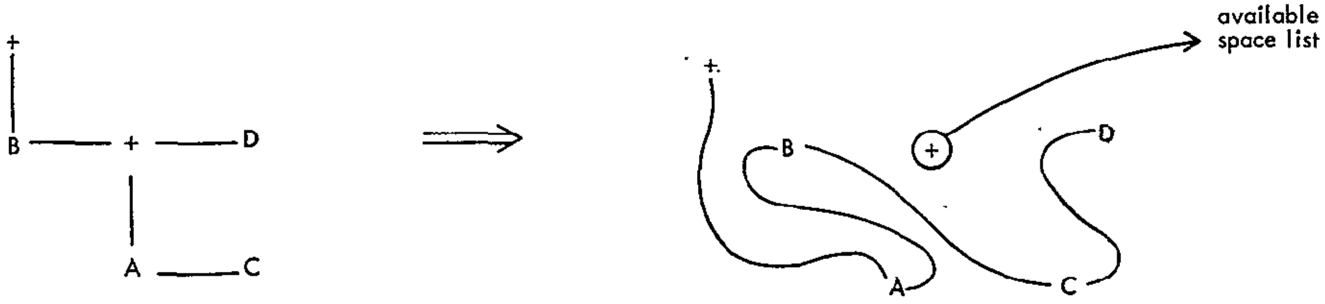
The associativity of addition makes + into a variary operator. More noteworthy, the associativity of addition can be explicitly encoded into the data structure.

3. EXPLOITING THE SORT

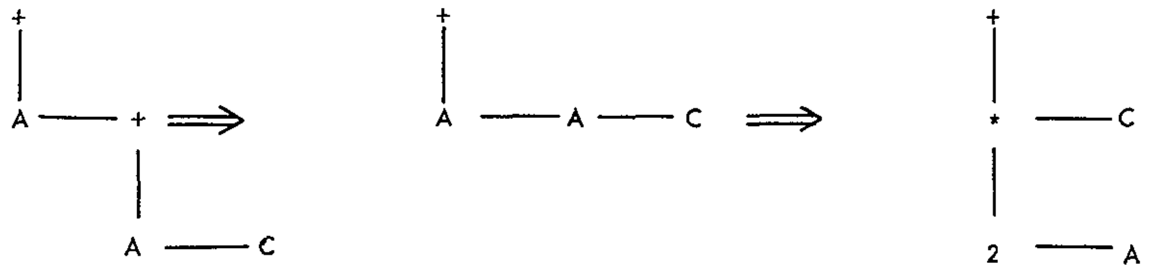
Like associativity, commutativity has an organization counterpart. Since commutativity is just the order-independence of summands or factors, it can be accounted for by sorting the lists. In fact, if the associativity is reflected by the transformation



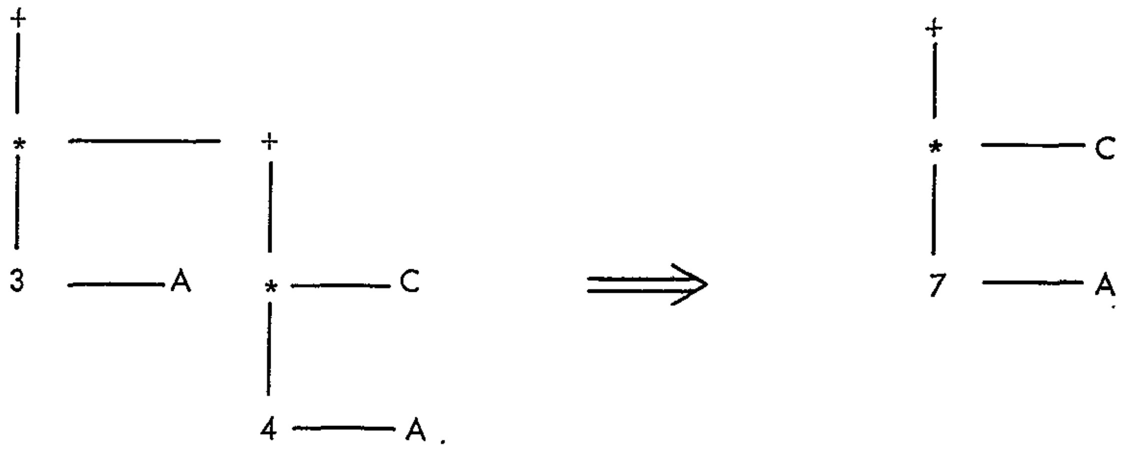
then a merging of the two summand lists according to some predetermined order will provide for commutatively equivalent sums:



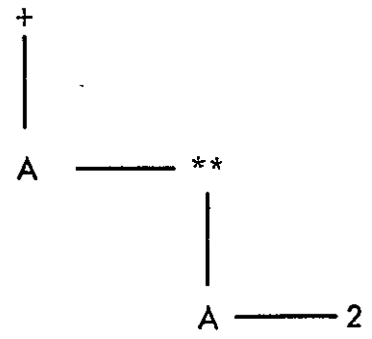
But since the commutativity is accounted for by any ordering principle, there is still freedom to try to determine the most useful one. An ordering is a subroutine which receives two lists as arguments and returns a "<," "=," or ">." The first thing to note is that more than one ordering is desirable (or, equivalently, that the order should be context-dependent). Consider what is appropriate if two summands compare "=." If they are identical, then the eventual transformation should be something like



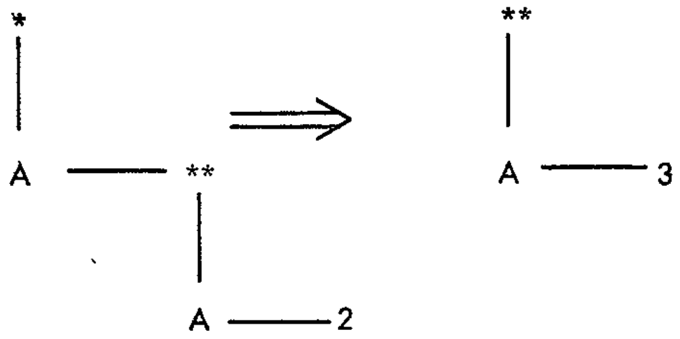
The merge can handle this directly once it has decided on equality of the A's. Then it could equally well handle



by comparing only the "non-combinable" parts of lists. However, what is combinable depends on the governing operator. Nothing can be done with

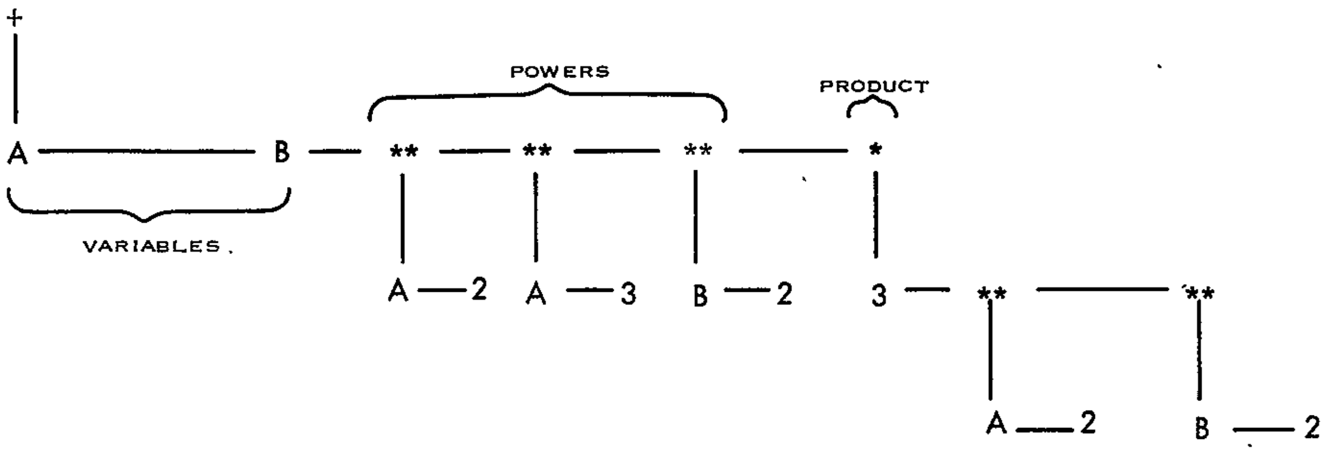


while

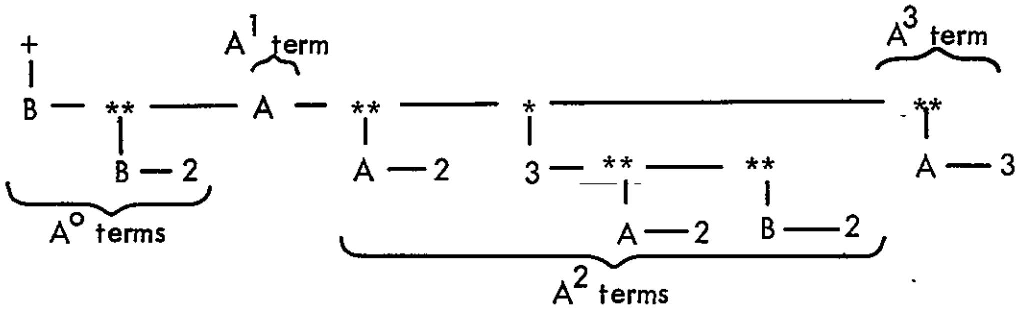


This criterion determines equality for the various orderings, but inequality has still to be pinned down. The easiest decision is to arbitrarily order the possible OP fields and stop on the first difference encountered in the scan. This is the present FORMAC device. It makes FORMAC's sort the fastest, but it provides the least data for the rest of the system to work with.

The first improvement over a first-difference sort is to force polynomial ordering on the lists. Thus, where an f - d sort might produce

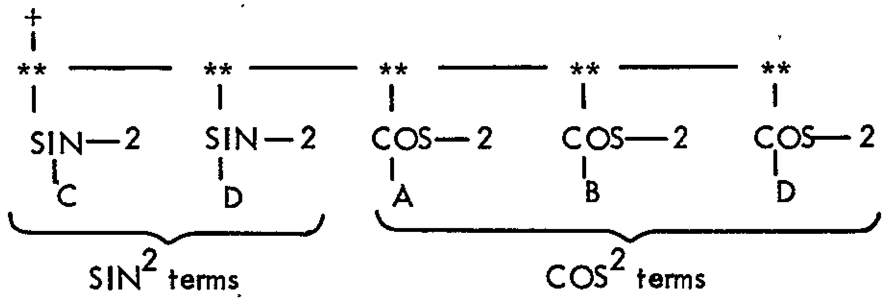


a polynomial ordering produces instead

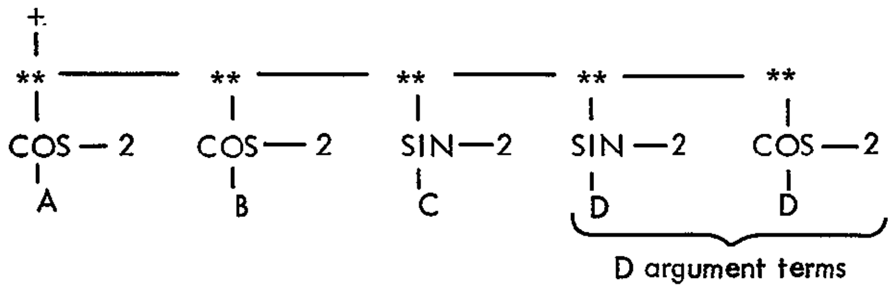


This ordering, done once, eliminates much repetition of effort in later applications of polynomial manipulation like addition, multiplication, division, factoring, and searching. In effect, not using a first-difference algorithm for sorting allows "first-difference" algorithms for later polynomial manipulations.

Even with the polynomial ordering, there is still freedom left. As far as polynomial structure is concerned, no more can be said about SIN(X), SIN(Y), and COS(X) than that they are different monomials. Figuratively, the "horizontal" ordering is fixed but the "vertical" ordering is open. Again, f - d is a possibility; but again, it is not the best choice. Consider, for example, the problem of replacing SIN^2(X) + COS^2(X) by 1 for all expressions X. An f - d ordering giving, say



requires the arguments of the two SIN^2 occurences to be compared to all COS^2 occurences; the "bottom-to-top" ordering

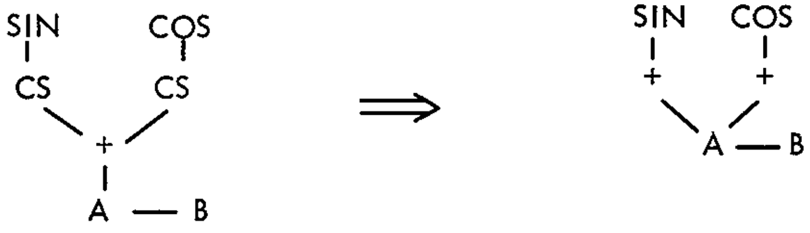


will always require, at most, one argument comparison for each SIN^2. (This is related to a parsing strategy where a bottom-up scan precedes a top-down scan so that the information the latter tentatively seeks is more readily available.)

4. ANOTHER PASS AT THE STRUCTURE

Once the preferred processing techniques are established, the data structure can (at least in an essay) be "fine-tuned" to facilitate the processing. For example, it was clear early (though not early enough) in the PL/I - FORMAC implementation that " — " should not be a distinct operator, but rather a bit in each node. Then the sort would not have to do any extra work to correctly place " - A": it need not notice the " - " unless a possibility for combination arises.

The CS node is another example of undesirable obtrusiveness, since extra work must be done to make believe it is not there. Here the improvement is to make the multiple references to the operand lists rather than to the operator. This way, individual operators can be used instead of CSs to carry the contextual ACROSS pointers. Thus



However, this again requires a bit position in each node, since common lists must be recognizable to control side effects.

Gradually, of course, changes become less clear-cut. Bringing exponent values into the corresponding nodes is desirable in the same way as bringing the sign in. But here space considerations become more delicate because more than one bit is involved in each node. In fact, since allowing for symbolic exponents requires room for a pointer, node size will be increased by 50 percent. The saving achieved when a " ** " does appear can be canceled by the extra space used in nodes not raised to a power; in fact, at least three times as many nodes must have exponents as not to save space. Though this ratio is not unusual in practice, it is not achieved for, say, univariate polynomials.

The case for the triple pointer node is even more convincing because the exponent of a product can be distributed over the factors at no additional cost in space, and then the exponent field in a " * " can be used for a numeric coefficient. Because of these advantages, the newer form is always superior — for polynomial calculations at least. However, ingenuity is beginning to replace naturalness.

5. A PLACE FOR SYSTEMS MICROPROGRAMMING

The single point which most aggravates the FORMAC space problem on System/360 is the enormous pointer size. Although it provides great flexibility to monitor designers, it is very wasteful to user-level systems running in a partition or on a moderate-sized machine. Consider that 2^{16} of the FORMAC 8-byte nodes requires 512K so, in almost all cases, eight bits of the 24-bit pointer fields are really unused.

This is an ideal application for microprogramming since a 16-bit relative pointer would have to be shifted and relocated against a base register for every nodal reference.

More generally, microprogram accelerators for systems have usually failed because the functions put in control storage do not use a significant portion of total system time. There is, of course, good reason for this: any bookkeeping operation which used up a lot of time would indicate poor system design. In particular, serious packing is often avoided. The unpacking is just not there to be speeded up. Thus, using microprogramming to pack an already fast data processing organization may well be a more fruitful approach than acceleration.

6. SUMMARY

In a programming system like FORMAC, with a heavy "computational" use of list structures, the question of data organization is a paramount one. The accessibility of pertinent data at a given time is a critical parameter of efficiency; but the commitment to lists introduces the possibility of great flexibility in the design of data representations. In this paper the ways in which tailoring the flexibility to the problem at hand may facilitate a more efficient solution were discussed.

REFERENCES

1. R. G. Tobey, et al., 'PL/I FORMAC Interpreter, User's Reference Manual," IBM Contributed Program Library, 360D 03.3.004, Hawthorne, New York, October 1967.

2. J. Baker, P. Marks, and R. Tobey, 'PL/I FORMAC Course Notes," IBM Federal Systems Division, February 1968.

FORMAC AND APPLICATIONS

CALCULATION OF TABLES OF INVERSES AND DETERMINANTS OF FINITE SEGMENTS OF THE HILBERT MATRIX

by

Elizabeth Cuthill
Applied Mathematics Laboratory
Naval Ship Research and Development Center
Carderock, Maryland

Abstract

In this paper, tables of inverses and determinants of finite segments of the Hilbert matrices from order 2 to order 37 are calculated using variable precision rational arithmetic. The evaluation of the determinants is carried out to order 62.

PRECEDING PAGE BLANK NOT FILMED

CALCULATION OF TABLES OF INVERSES AND DETERMINANTS OF FINITE SEGMENTS OF THE HILBERT MATRIX

by

E. Cuthill

1. INTRODUCTION

Tables of determinants and inverses of the matrix H_n with $(i, j)^{th}$ element

$$\frac{1}{i + j - 1} \qquad i, j = 1, 2, \ldots, n \qquad (1)$$

were calculated for $n = 2$ to 37 inclusive.* These matrices, which augment those given in references 1 and 2, arise in least squares fitting of polynomials. They are useful in estimating the mean value function of certain stochastic processes and they are frequently used in testing computer subroutines for inversion of matrices.

2. INVERSION OF H_n

The method of computation we used to obtain the inverse of H_n is based on that given by A. R. Collar,[3] who showed that the $(i, j)^{th}$ element of the inverse of H_n is given by

$$H_n^{ij} = \frac{F_n(i)\, F_n(j)}{i + j + 1} \qquad (2)$$

where

$$F_n(k) = \frac{(-1)^k\, (n+k-1)!}{[(k-1)!]^2\, (n-k)!} \;. \qquad (3)$$

*Tables to order 37 have been deposited in the Unpublished Mathematical Tables Repository of Mathematics of Computation (formerly MTAC). Tables for n = 2 to 20 inclusive are published in reference 7.

To calculate the functions $F_n(k)$ we used the recurrence relations:

$$F_n(k) = \frac{n+k-1}{n-k} F_{n-1}(k) \qquad \text{for } k = 1,2,\ldots,n-1 \tag{4}$$

and

$$F_n(n) = \frac{-(2n-1)F_n(n-1)}{(n-1)^2} \tag{5}$$

When these relations are used, $F_{n-1}(k)$ for $k = 1,2,\ldots,n-1$ must be available. This requirement applies if we start with n=1 and

$$F_1(1) = -1 \tag{6}$$

and then calculate inverses for successive values of n. However, to start with n = N, we used the recurrence relation

$$F_N(k+1) = -\left[\frac{N^2 - k^2}{k^2}\right] F_N(k) \qquad \text{for } k = 1,2,\ldots,N-1 \tag{7}$$

with

$$F_N(1) = -N. \tag{8}$$

A quick derivation of the inverse of H_n is given in reference 1. There Cramer's rule is used to write the $(i,j)^{th}$ element of the inverse in the form

$$H_n^{ij} = \frac{(-1)^{i+j}\,\Delta_n^{ij}}{\Delta_n} \tag{9}$$

where Δ_n^{ij} is the minor of the $(i,j)^{th}$ element of H_n and Δ_n is the determinant of H_n. Then the following theorem due to A. Cauchy (see references 4 and 5) is applied directly to the evaluation of the determinants Δ_n^{ij} and Δ_n :

Given 2n numbers $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ such that $a_i + b_j \neq 0$ for $i, j = 1,2,\ldots,n$, the determinant of the matrix with $(i,j)^{th}$ element

$$\frac{1}{a_i + b_j} \qquad i,j = 1,2,\ldots,n$$

is given by

$$\frac{\prod_{j>k}^{1,2,\ldots,n} (a_j - a_k)(b_j - b_k)}{\prod_{j,k}^{1,2,\ldots,n} (a_j + b_k)} \tag{10}$$

3. THE DETERMINANT OF H_n

The determinant of H_n is given by (10) on substituting

$$\left.\begin{aligned} a_r &= r \\ b_r &= r-1 \end{aligned}\right\} \text{ for } r = 1,2,\ldots,n \tag{11}$$

which yields directly

$$\det(H_n) = (1!\,2!\,\ldots(n-1)!)^2 \div \left(\frac{n!}{0!}\,\frac{(n+1)!}{1!}\,\ldots\frac{(2n-1)!}{(n-1)!}\right)$$

$$= \frac{(1!\,2!\,\ldots(n-1)!)^4}{1!\,2!\,\ldots(2n-1)!} \quad . \tag{12}$$

To calculate det (H_n) we use the form of (2), noting that in matrix notation

$$H_n^{-1} = F_n H_n F_n$$

where F_n is the diagonal matrix whose i^{th} diagonal element is $F_n(i)$. In view of this

$$(\det(H_n))^{-1} = (\det(F_n))^2 (\det(H_n))$$

from which

$$\det(H_n) = |\det(F_n)| = \prod_{i=1}^{n} |F_n(i)|. \tag{13}$$

4. CALCULATIONS

The determinants and inverses of H_n to order 20 are displayed in Appendix A of reference 7. Since H_n is symmetric, i.e., $H_n^{ij} = H_n^{ji}$, elements of the inverse were tabulated only for $j = 1,2,\ldots,i$ for each i, $i = 1,2,\ldots,n$. The calculations were performed using PL/I-FORMAC[6] on a System/360 Model 50. Major use was made of the PL/I-FORMAC facility for variable precision rational arithmetic. The computer program which was used is displayed in Appendix B of reference 7. The program is set up to calculate determinants and inverses of H_n for n ranging from a specified lower limit ML to a specified upper limit MU. Equations (7) and (8) are used to calculate F_{ML}, and then equations (4) and (5) are used to calculate F_n for the successive values of n. In all cases, Equation (2) is then used to calculate the elements of the inverse; Equation (13) to calculate the determinant.

The calculations were performed in the five following computer runs:

RUN	ML	MU	(Approximate) RUNNING TIME
1	2	21*	10 min.
2	21	28*	9 min.
3	28	30	5 min.
4	30	37*	15 min.
5	37	38*	5 min.

Calculations for the starred values of n were incomplete. However, except for n = 38, calculations for those values of n were obtained in another run. Note that check calculations were made at n = 21, 28, 30, and 37 in that elements of the inverses for these values of n were calculated two ways: using equations (7), (8), and (2); and using equations (5), (6), and (2).

Values of the determinants of H_n were also calculated independently for n =2 to 62, using Equation (12) directly. In reference 7, Appendix C gives the results of these calculations and Appendix D contains the program used for the calculations. For n = 2 to 37, these calculations were then checked against the determinant values tabulated in Appendix A of reference 7. These determinant values were calculated using Equation (13), and they were based on the F_n values used in calculating the elements of the inverses. No discrepancies between the calculations were found.

5. SUMMARY

The calculation of inverses and determinants of finite segments of Hilbert matrices using PL/I-FORMAC variable precision rational arithmetic proved to be a straightforward operation.

PRECEDING PAGE BLANK NOT FILMED

Appendix

FORMAC PROGRAM AND SAMPLE OUTPUT

PRECEDING PAGE BLANK NOT FILMED

FORMAC PROGRAM AND SAMPLE OUTPUT

For reference, the determinant and inverse of the matrix H_{20} are displayed in this appendix, along with the FORMAC Program which generated them.

In the FORMAC program, the ML and MU parameters are set to the starting value and final value of n. The program uses

a. Equations (7), (8), and (2) to calculate the inverse of H_n for n = ML, then

b. Equations (4), (5), and (2) to calculate the inverses of H_n recursively for n = ML + 1, ML + 2, ..., MU and, as a check,

c. Equations (7), (8), and (2) to recalculate H_n for n = MU.

```
INVERSE OF SEGMENT OF HILBERT MATRIX OF ORDER          20
S(1,1) = 400
------------
S(1,2) =  - 79800
-----------------
S(2,2) = 21226800
-----------------
S(1,3) = 5266800
----------------
S(2,3) =  - 1576089900
----------------------
S(3,3) = 124826320080
---------------------
S(1,4) =  - 171609900
---------------------
S(2,4) = 54777880080
--------------------
S(3,4) =  - 4519175106600
-------------------------
S(4,4) = 168285473017200
------------------------
S(1,5) = 3294910080
-------------------
S(2,5) =  - 1095557601600
-------------------------
S(3,5) = 92965887907200
-----------------------
S(4,5) =  - 3533994933361200
----------------------------
S(5,5) = 75391891911705600
--------------------------
S(1,6) =  - 41186376000
-----------------------
S(2,6) = 14085740592000
-----------------------
S(3,6) =  - 1220177278782000
----------------------------
S(4,6) = 47119932444816000
--------------------------
S(5,6) =  - 1017790540808025600
-------------------------------
S(6,6) = 13878961920109440000
-----------------------------
S(1,7) = 356948592000
---------------------
S(2,7) =  - 124619677182000
---------------------------
S(3,7) = 10966531592016000
--------------------------
S(4,7) =  - 428791385247825600
------------------------------
S(5,7) = 9355448405407104000
----------------------------
S(6,7) =  - 128637415574347680000
---------------------------------
S(7,7) = 1200615878693911680000
-------------------------------
S(1,8) =  - 2237302782000
-------------------------
S(2,8) = 793496720016000

-------------------------
S(3,8) =  - 70700557753425600
-----------------------------
S(4,8) = 2792314957736304000
----------------------------
S(5,8) =  - 61430929070198689000
--------------------------------
S(6,8) = 850582094818135680000
------------------------------
S(7,8) =  - 7986020779125829440000
----------------------------------
S(8,8) = 53392253209012688256000
--------------------------------
S(1,9) = 10440746316000
-----------------------
S(2,9) =  - 3749272002075600
----------------------------
S(3,9) = 337434480186804000
---------------------------
S(4,9) =  - 13438015734105963000
--------------------------------
S(5,9) = 297703733196347488000
------------------------------
S(6,9) =  - 4146587712238411440000
----------------------------------
S(7,9) = 39131501817716564256000
--------------------------------
S(8,9) =  - 262789996263109325010000
------------------------------------
S(9,9) = 1298491746241246076520000
----------------------------------
S(1,10) =  - 37006645275600
---------------------------
S(2,10) = 13423319513604000
---------------------------
S(3,10) =  - 1218166245859563000
--------------------------------
S(4,10) = 48851589962162988000
------------------------------
S(5,10) =  - 1088692576299632304000
-----------------------------------
S(6,10) = 15241696068194852256000
--------------------------------
S(7,10) =  - 144479577313097037010000
-------------------------------------
S(8,10) = 974059652089259255520000
---------------------------------
S(9,10) =  - 4829712441609243878620000
--------------------------------------
S(10,10) = 18019628875711681573360000
-------------------------------------
S(1,11) = 100927214388000
-------------------------
S(2,11) =  - 36914128662411000
------------------------------
S(3,11) = 3373383450072636000
----------------------------
S(4,11) =  - 136086572037454039000
----------------------------------
S(5,11) = 3048339213638970451200

-------------------------------
S(6,11) =  - 42867270191798021970000
------------------------------------
S(7,11) = 407939512413450457440000
---------------------------------
S(8,11) =  - 2759835680919567890640000
--------------------------------------
S(9,11) = 13726551149836798192920000
-----------------------------------
S(10,11) =  - 51355942295778292498326000
----------------------------------------
S(11,11) = 146731263702223692852360000
--------------------------------------
S(1,12) =  - 213323430411000
----------------------------
S(2,12) = 78568660369836000
--------------------------
S(3,12) =  - 7222704706855638000
--------------------------------
S(4,12) = 292867300483909351200
------------------------------
S(5,12) =  - 6589514260887960402000
-----------------------------------
S(6,12) = 930284366243006174400000
---------------------------------
S(7,12) =  - 988364144801191699640000
-------------------------------------
S(8,12) = 6028662379412383621920000
----------------------------------
S(9,12) =  - 30067953572431763314326000
---------------------------------------
S(10,12) = 112776921688485978803880000
-------------------------------------
S(11,12) =  - 322952093926118939302020000
-----------------------------------------
S(12,12) = 712281693323218846365720000
-------------------------------------
S(1,13) = 350069219136000
-------------------------
S(2,13) =  - 129700645689889000
-------------------------------
S(3,13) = 11984339661745651200
-----------------------------
S(4,13) =  - 498112167473182252000
----------------------------------
S(5,13) = 11025592488805999104000
--------------------------------
S(6,13) =  - 156195893591418320640000
-------------------------------------
S(7,13) = 1496192243875691281920000
----------------------------------
S(8,13) =  - 10181740892252025572576000
---------------------------------------
S(9,13) = 50908704461260128362880000
-----------------------------------
S(10,13) =  - 191379018622885297364160000
-----------------------------------------
S(11,13) = 549174575178714331566720000
-------------------------------------
S(12,13) =  - 1213516958995113590104560000
```

```
------------------------------------------
S(13,13) = 2071068943351660527111782400
------------------------------------------
S(1,14) =  - 444318624288000
------------------------------------------
S(2,14) = 165464255684851200
------------------------------------------
S(3,14) =  - 15357151230750252000
------------------------------------------
S(4,14) = 627936850324010304000
------------------------------------------
S(5,14) =  - 14233235274010900224000
------------------------------------------
S(6,14) = 202261764420154897920000
------------------------------------------
S(7,14) =  - 1942836614902487880576000
------------------------------------------
S(8,14) = 132543372561986068922880000
------------------------------------------
S(9,14) =  - 66422303749750640852160000
------------------------------------------
S(10,14) = 250215634844739665861760000
------------------------------------------
S(11,14) =  - 719369950178626539352560000
------------------------------------------
S(12,14) = 1592364028560688696595782400
------------------------------------------
S(13,14) =  - 2721989792411433669565440000
------------------------------------------
S(14,14) = 3582789983174023804385280000
------------------------------------------
S(1,15) = 431623806451200
------------------------------------------
S(2,15) =  - 161454280100652000
------------------------------------------
S(3,15) = 15043739981143104000
------------------------------------------
S(4,15) =  - 617257652189248224000
------------------------------------------
S(5,15) = 14034489776092380672000
------------------------------------------
S(6,15) =  - 199991479309316424576000
------------------------------------------
S(7,15) = 1925943874830454458390000
------------------------------------------
S(8,15) =  - 13169271311145899644160000
------------------------------------------
S(9,15) = 66127617397710930821760000
------------------------------------------
S(10,15) =  - 249577329653809207530480000
------------------------------------------
S(11,15) = 719782709402970517637782400
------------------------------------------
S(12,15) =  - 1593613921832714355013440000
------------------------------------------
S(13,15) = 27281672104865881364961280000
------------------------------------------
S(14,15) =  - 3595846797981509313876480000
------------------------------------------
S(15,15) = 3613560329006048768624640000
```

```
------------------------------------------
S(1,16) =  - 314725692204000
------------------------------------------
S(2,16) = 118188754060608000
------------------------------------------
S(3,16) =  - 11050648504666848000
------------------------------------------
S(4,16) = 454821427928919744000
------------------------------------------
S(5,16) =  - 10369928556779370163200
------------------------------------------
S(6,16) = 148141836525419573760000
------------------------------------------
S(7,16) =  - 1429793179798367704320000
------------------------------------------
S(8,16) = 9796684058920137899520000
------------------------------------------
S(9,16) =  - 49289566671441943806960000
------------------------------------------
S(10,16) = 186351072808177541622758400
------------------------------------------
S(11,16) =  - 537551171562050600834880000
------------------------------------------
S(12,16) = 1193570920878823097170560000
------------------------------------------
S(13,16) =  - 2046121578649411723720960000
------------------------------------------
S(14,16) = 2700329396185347990497280000
------------------------------------------
S(15,16) =  - 2716862025141584914928896000
------------------------------------------
S(16,16) = 2044949911396891796183040000
------------------------------------------
S(1,17) = 166619484108000
------------------------------------------
S(2,17) =  - 62787775594698000
------------------------------------------
S(3,17) = 5888832426829044000
------------------------------------------
S(4,17) =  - 2430452900549516498200
------------------------------------------
S(5,17) = 5555318869703234016000
------------------------------------------
S(6,17) =  - 79542065634387214320000
------------------------------------------
S(7,17) = 769290995452382623520000
------------------------------------------
S(8,17) =  - 5281025000511636836460000
------------------------------------------
S(9,17) = 26616366002578649655758400
------------------------------------------
S(10,17) =  - 100790844667884497656540000
------------------------------------------
S(11,17) = 291173551262777408785560000
------------------------------------------
S(12,17) =  - 647405655744540206724210000
------------------------------------------
S(13,17) = 1111255684956145647193280000
------------------------------------------
S(14,17) =  - 1468304109175782969832896000
```

```
------------------------------------------
S(15,17) = 1478936989492394959739520000
------------------------------------------
S(16,17) =  ¬ 1114337939999478146748180000
------------------------------------------
S(17,17) = 607820694545169898276280000
------------------------------------------
S(1,18) =  - 60440401098000
------------------------------------------
S(2,18) = 22846471615044000
------------------------------------------
S(3,18) =  - 2148710655394888200
------------------------------------------
S(4,18) = 88904324471894316000
------------------------------------------
S(5,18) =  - 2036717251537942512000
------------------------------------------
S(6,18) = 292224664913370479520000
------------------------------------------
S(7,18) =  - 283160273442983396460000
------------------------------------------
S(8,18) = 1947218076313218066758400
------------------------------------------
S(9,18) =  - 9829706635234995048540000
------------------------------------------
S(10,18) = 37278274729144512543480000
------------------------------------------
S(11,18) =  - 107840723323596625577210000
------------------------------------------
S(12,18) = 240083137818163818458280000
------------------------------------------
S(13,18) =  - 412587318324548191720896000
------------------------------------------
S(14,18) = 545758754722107075515520000
------------------------------------------
S(15,18) =  - 550283540316104136728340000
------------------------------------------
S(16,18) = 415028663403391672145280000
------------------------------------------
S(17,18) =  - 226587340130160526888140000
------------------------------------------
S(18,18) = 84541831107387625158264000
------------------------------------------
S(1,19) = 13431200244000
------------------------------------------
S(2,19) =  - 5091096452488200
------------------------------------------
S(3,19) = 480017665520316000
------------------------------------------
S(4,19) =  - 19906187129228862000
------------------------------------------
S(5,19) = 456976817575340832000
------------------------------------------
S(6,19) =  - 6569041752645524460000
------------------------------------------
S(7,19) = 63763498612345890758400
------------------------------------------
S(8,19) =  - 439187362891157921040000
------------------------------------------
S(9,19) = 2220336112394187267480000
```

```
-----------------------------------------
S(10,19) =  - 8431990712544592122930000
-----------------------------------------
S(11,19) = 2442369723633605994228000O
-----------------------------------------
S(12,19) =  - 5443860450115566418539600O
-----------------------------------------
S(13,19) = 936578141955366265555200000
-----------------------------------------
S(14,19) =  - 1240169392481948562803400000
-----------------------------------------
S(15,19) = 1251673746772133614406400000
-----------------------------------------
S(16,19) =  - 944890965700532238326400000
-----------------------------------------
S(17,19) = 516315420543505115942640000
-----------------------------------------
S(18,19) =  - 192799443369042423629100000
-----------------------------------------
S(19,19) = 44002275363505177765200000
```

```
-----------------------------------------
S(1,20) =  - 1378465288200
-----------------------------------------
S(2,20) = 523816809516000
-----------------------------------------
S(3,20) =  - 49500688499262000
-----------------------------------------
S(4,20) = 2057028610969332000
-----------------------------------------
S(5,20) =  - 47311658052294636000
-----------------------------------------
S(6,20) = 681287875953042758400
-----------------------------------------
S(7,20) =  - 6623632127321249040000
-----------------------------------------
S(8,20) = 45689544061930248480000
-----------------------------------------
S(9,20) =  - 231303316813521882930000
-----------------------------------------
S(10,20) = 879523723192157283240000
```

```
-----------------------------------------
S(11,20) =  - 2550618797257256121396000
-----------------------------------------
S(12,20) = 5691463431896356634520000
-----------------------------------------
S(13,20) =  - 9801964799377058648340000
-----------------------------------------
S(14,20) = 12991953343553024480640000
-----------------------------------------
S(15,20) =  - 13124524296038259424320000
-----------------------------------------
S(16,20) = 9916307245895573787264000
-----------------------------------------
S(17,20) =  - 5422980525099141914910000
-----------------------------------------
S(18,20) = 2026580957476495940520000
-----------------------------------------
S(19,20) =  - 462861087880434258020000
-----------------------------------------
S(20,20) = 48722219250572027160000
```

```
DETERMINANT OF SEGMENT OF HILBERT MATRIX OF ORDER                        20
H = 1/2377454716768534509091644243427616440175419837753486493033185331234419759310644585187585766816573773440565759
--------------------------------------------------------------------------------------------------------------------
867265558971765638419710793303386582324149811241023554489166154717809635257779783680000000000000000000000000000000000000
--------------------------------------------------------------------------------------------------------------------
```

REFERENCES

1. R. Savage and E. Lukacs, "Tables of Inverses of Finite Segments of the Hilbert Matrix," NBS Applied Mathematics Series, No. 39, Contributions to the Solution of Systems of Linear Equations and the Determination of Eigenvalues, 1954, pp. 107-108.

2. R.B. Smith, "Table of Inverses of Two Ill-Conditioned Matrices," Reviewed in Mathematical Tables and Other Aids to Computation, 11, 1957, p. 216 (deposited in the UMT file of MTAC).

3. A.R. Collar, "On the Reciprocation of Certain Matrices, Royal Society Edinburgh, *Proceedings*, Vol. 59, 1939, pp. 195-206.

4. A. Cauchy, *Oeuvres Completes*, 2^e series XII, p. 177.

5. G. Polya and G. Szego, *Aufgaben and Lehrsatze aus der Analysis 2*, J. Springer, Berlin, 1954, p. 98.

6. R. Tobey, *et. al.*, "PL/I-FORMAC Interpreter, Users Reference Manual," IBM Contributed Program Library, 360D 03.3.004, Hawthorne, N. Y., October 1967.

7. E. Cuthill, "Tables of Inverses and Determinants of Finite Segments of the Hilbert Matrix to Order 20," Naval Ship Research and Development Center, Applied Mathematics Laboratory Technical Note, AML-52-68, Washington, D.C., December 1968.

THE SCOPE FORMAC LANGUAGE

by

Robert G. Tobey
IBM Boston Programming Center
Cambridge, Massachusetts

and

John D. Lipson
Graduate School of Arts and Sciences
Harvard University
Cambridge, Massachusetts

Abstract

The Scope FORMAC System is an experimental online interactive symbolic prototype implemented on the IBM 2250 graphic display unit. The language for this system is described in this paper. The language gives the user access to the FORMAC capability in an interactive environment. Sample programs and output are presented.

This paper reports research sponsored by NASA under contract number NAS 12-87. The senior author, Robert G. Tobey, is currently with the Applied Mathematics Division of Argonne National Laboratory, Argonne, Illinois.

THE SCOPE FORMAC LANGUAGE

by

R. G. Tobey and J. D. Lipson

1. INTRODUCTION

The Scope FORMAC experimental system is an online interactive prototype implemented on the IBM 2250 graphic display unit. With this prototype one can perform symbolic mathematical computations in an interactive, time-sliced environment. One can use the system as a symbolic desk calculator, executing only a few statements at a time, or one can compose, debug, and execute complicated algorithms (which may or may not interact with the user). The scope FORMAC prototype lends itself to both short, one-shot calculations and the user's definition of a personalized interactive calculator with functions tailored to the class of problems he wishes to solve.

Scope FORMAC runs in a standard OS/360 environment, either in a partition under MFT or as a task under MVT. The system can only qualify as a prototype; this is indicated by the fact that the FORMAC object-time routines which perform the symbolic calculation have been carried over intact from the PL/I-FORMAC batch system.[1] In a production system, the design of these routines would need to be significantly modified to accommodate interrupt and control functions necessary to an interactive environment.

This paper presents a concise description of the Scope FORMAC Language. A more detailed description of the Scope FORMAC system with applications is being prepared for publication under NASA contract NAS 12-87.

2. LANGUAGE OVERVIEW

The Scope FORMAC language is a simple language, with strong similarities to PL/I. It was designed to provide easy access to the FORMAC symbolic capability in an online, interactive environment. Prior to presenting the language, some general remarks are in order.

The OS/360 batch FORMAC system contains two kinds of variables—PL/I and FORMAC—and facilities for conversion from one kind to the other. Scope FORMAC contains only one kind of variable, namely, FORMAC; consequently the rules for naming variables and forming expressions are those of FORMAC, as described in the "PL/I-FORMAC Interpreter User's Reference Manual." [2]

Like PL/I, any Scope FORMAC statement may be prefixed by a label for reference. The rules for naming labels are the same as those for naming FORMAC variables, and the name of a variable or label may coincide with neither the name of any other variable or label nor with the names of the Scope FORMAC keywords (DO, END, FOR, GET, IF, PUT, SET, TO).

Scope programs may be written in a relatively free form, i.e., statements are separated by at least one blank (the use of a semicolon as a statement delimiter is optional), and more than one statement may appear on a single line. Comments consisting of arbitrary character strings enclosed by quotes (") may be used freely in the program for documentation purposes; they are ignored at execution time (except when they appear as an argument of the output statement PUT, as described below). Character strings may appear without enclosing quotes, provided that they do not fall within the range of executable statements of a program.

3. LANGUAGE STATEMENTS

The statements of the language are:

1. Assignment Statement
2. GET and PUT statements
3. TO statement
4. DO statement
5. IF statement
6. FOR statement
7. SET statement

In describing the format of the above statements, we use "var" to stand for an arbitrary FORMAC variable, "expr" to denote an arbitrary FORMAC expression, and "stat" to stand for an arbitrary Scope FORMAC statement.

1. The Assignment Statement

Format: var = expr

Result: The value of expr is assigned to var.

Examples: 1. Y = DERIV(F,X, 2)+SIN(THETA)+7.6
R = 3/7

Note that any FORMAC assignment statement (i.e., any statement that could appear within the scope of a "LET" in batch FORMAC) is permissible.

2. The GET and PUT Statements

(i) GET

Format: GET var1, var2, ...

Result: For each variable in the list, a request for input is made and accepted when given.

Upon encountering a statement of the form GET var1, var2, the system causes the message

```
*****************

*PLEASE SPECIFY*
*var1          *

******************
```

to appear on the scope face. The user may then type in any executable statements. In this case, he would presumably want to assign a value to var1. To do this, he must type in

var1 = expr

where expr is the desired expression var1 is to be. Typing in expr alone is an error.

(ii) PUT

Format: PUT $\text{var}_1, \text{var}_2, \ldots, \text{var}_n$

Result: The values of $var_1, \ldots, var_n$ are displayed in order. Also, comments which may appear in the output list are displayed.

Example: PUT "SOLUTION FOR A AND B", A, B results in the scope display.

```
SOLUTION FOR A AND B

A = (value of A)

B = (value of B)
```

3. The TO Statement:

Format: TO label

Result: Control is transferred to the indicated labeled statement.

Example:

```
TO L
 .
 .
 .
L:A = COS(X)
```

4. The DO Statement:

Format: DO $stat_1$ $stat_2$ ... $stat_n$ END

Result: The statements $stat_1, \ldots, stat_n$ are executed. Syntactically, "DO ... END" is considered as a single statement and may appear wherever CLINK expects a statement (see 5 and 6 below).

Example: DO A = B*C PUT A TO L END

Another important use of the DO is to invoke remote (out-of-line) code.

Format: DO name

Result: Control is transferred to the statement labeled "name." Execution proceeds until the END corresponding to the DO of "DO name" (see note below) is encountered, causing transfer of control to the statement following "DO name." Thus, "DO name" acts as though the statement labeled "name" immediately follows the DO.

This use of the DO roughly corresponds to the procedure capability of PL/I, and the END serves as a return statement.

Note: To match up DO-END pairs, think of DO and END as (statement) parentheses with the DO acting as a left parenthesis and the END acting as a right parenthesis. Then DO-END pairs are determined in the same manner as matching parentheses in expressions, as illustrated below:

```
DO
   DO
      DO...END
   END
END
```

It is permissible to transfer out of a DO-END, but it is not permissible to transfer into a DO-END without executing the DO itself.

```
Example:  DO SOLVE
              .
              .
              .
   SOLVE:  A = B
             .
             .
             .
           DO...END
             .
             .
             .
           END
```

Note that the END corresponding to the DO of DO SOLVE is the last END.

It is also permissible to have more than one name following a DO

DO name_1, $\text{name}_2, \ldots, \text{name}_n$

and this is equivalent to

DO name_1 DO $\text{name}_2 \ldots$ DO name_n .

5. The IF Statement:

Format: IF $expr_1$ relation $expr_2$ stat where the relation is any of <, < =, =, >, > = or ¬ (not) followed by any one of <, = , >. The right-hand expression, $expr_2$, must be a variable or, if it is an algebraic expression, it must be enclosed in parentheses. Otherwise the condition will not be evaluated correctly.

Result: If the condition "$expr_1$ relation $expr_2$" holds, then stat is executed; otherwise, stat is skipped. When $expr_1$ or $expr_2$ evaluates to nonnumeric expressions, then it is considered equal if and only if it is identical (see the IDENT function on page 33 of the "FORMAC Reference Manual"). For completeness, the following table gives the result of "$expr_1$ relative $expr_2$" when $expr_1$ or $expr_2$ evaluates to nonnumeric expressions.

relation	$expr_1$ and $expr_2$ identical	$expr_1$ and $expr_2$ not identical
=, < =, > =	yes	no
¬ =	no	yes
<, >	no	no
¬<, ¬>	yes	yes

Note: Any number of conditions (i.e., $expr_1$ relation $expr_2$ triplets separated by commas) may follow the IF, in which case the statement stat is executed if and only if all the conditions hold.

Examples: 1.
```
IF X> 0 A = SQRT(X)
```
2.
```
IF 0<X, X<=10, R=SIN(A)
 DO Y = A+2/3
           ARG=X DO PROC
END
```

6. The FOR Statement:

Format: 1. FOR var = expr stat

2. FOR var = $expr_1$ ($expr_2$) $expr_3$ stat where parentheses may not appear in $expr_1$ or $expr_2$.

Result: 1. The statement stat is executed with var = expr.

2. The statement stat is executed for each value of var, with expr_1 the initial value, expr_2 the increment value, and expr_3 the final value. It is equivalent to the following configuration:

```
   var=expr1
L: stat
   var=var+expr2
   IF var < = expr3 TO L
```

Examples: 1. FOR ARG=DERIV(F,X) DO PROC

2. FOR I=1 (1) 100

```
DO A(I)=SQRT(X(I)) PUT X(I) END
```

Note: More than one variable may be set following the FOR by having more than one list of the forms 1 and 2 above. For example,

```
FOR I=1(1)3,  J=1(2)N,X=ALPHA
```

is equivalent to the nested FORs below.

```
FOR I=1(1)3
FOR J=1(2)N
FOR X=ALPHA
```

7. The SET Statement:

Format: SET option_1, option_2, . . . , option_n where option_1, . . . , option_n may be any of the FORMAC options described on pages 39 to 41 of the "FORMAC Reference Manual." In addition, the option STOP stops execution. The options TRANS, INT, EXPAND, EDIT, PROPER, and PRINT may be abbreviated T, I, E, D, R, and P, and prefixed by an N for NOTRANS, NOINT, etc.

Result: Equivalent to the OPSET statement of batch FORMAC, as described on pages 39 to 41 of the "FORMAC Reference Manual." In particular, the default options are as described there.

4. SAMPLE PROGRAMS

Examples from the "FORMAC Reference Manual"

For comparative purposes, the two sample programs presented on pages 12 to 15 of the "FORMAC Reference Manual" are rewritten below, in the scope FORMAC language.

First Program.

```
"LEGENDRE POLYNOMIAL PROGRAM"
SET E
"GENERATE LEGENDRE POLYNOMIALS BY METHOD 1"
FOR N=0(1)10
    P(N)=DERIV((X**2-1)**N, X, N)/(2**N*FAC(N))
"GENERATE LEGENDRE POLYNOMIALS BY METHOD 2"
Q(O)=1 Q(2)=X
FOR N=2(1)10
    Q(N) = (2*N-1)/N*X*Q(N-1)-(N-1)/N*Q(N-2)
"CHECK THAT P(N) =Q(N) AND PRINT OUT RESULTS"
PUT "LEGENDRE POLYNOMIALS"
FOR N=0(1)10 DO
    IF P(N)=Q(N) DO PUT P(N) TO F END PUT "ERROR"
F:END
```

Second Program.

```
"INDUCTION PROGRAM"
   SET E
   SUMSQ=N*(N+1)*(2*N+1)/6
"BASIS"
   S=EVAL(SUMSQ, N, 1)
   IF S¬=1 DO PUT "NOT TRUE" SET STOP END.
"INDUCTION STEP"
   S=EVAL(SUMSQ, N, N+1)
   IF S= (SUMSQ+(N+1)**2)
      DO PUT "PROOF BY INDUCTION SUCCESSFUL" TO F END
   PUT "NOT TRUE"
F:SET STOP
```

The output produced by these programs is identical to that shown on pages 14 and 15 of the "FORMAC Reference Manual."

Taylor Series Solution

The final example is indicative of the power and utility of an interactive system for symbolic mathematics in solving problems of an interesting and substantial nature. Specifically, this example illustrates the use of the FORMAC Scope System in computing the Taylor series solution to an arbitrary first order ordinary differential equation.

Given the first order ordinary differential equation

$$y' = f(x,y) \tag{1}$$

with specified initial conditions (x_o, y_o), we wish to compute the p^{th} order Taylor series approximation to the solution y(x) of Equation 1. Thus, we have

$$
\begin{aligned}
y(x) &\approx T(x;x_o,y_o,p) \\
&= y_o + \frac{y_o'}{1!}(x-x_o) + \frac{y_o''}{2!}(x-x_o)^2 + \dots + \frac{y_o^{(p)}}{p!}(x-x_o)^p \\
&= y_o + \frac{f(x_o,y_o)}{1!}(x-x_o) + \frac{f'(x_o,y_o)}{2!}(x-x_o)^2 + \dots + \frac{f^{(p-1)}(x_o,y_o)(x-x_o)^p}{p!} \qquad (2)
\end{aligned}
$$

where the (total) derivatives of f with respect to x in (2) are given recursively by

$$f^{(o)} = f \qquad (3a)$$

$$f^{(m)} = f_x^{(m-1)} + f_y^{(m-1)} f \quad (m = 1, 2, \dots, p-1). \qquad (3b)$$

Concerning the utility of the Taylor series method in solving the ODE, the comments of M. V. Wilkes on page 55 of his book "Numerical Analysis"[2] are appropriate. He states: "The (Taylor Series) method has the very great advantage, compared with finite difference methods, that derivates are not plagued by rounding errors in the way that differences are," and later he states: "Application of the Taylor series method is at present limited by the necessity for the programmer himself to derive the formulae ((3) above) for the derivatives. Advances in programming languages for symbol manipulation should enable this load to be taken from the programmer and put on the machine."

TAYLOR (see following listing) requests as input F, the right-hand side of (1), the initial conditions X0, and Y0 and P the order of approximation. TAYLOR then computes analytically the derivatives of (3b) and finally computes the Taylor series solution T according to (2).

The user may then use T in order to advance the solution. For example, in order to compute and print out values of T = T (X) for X = 0(.1) 1, the user may type

```
        FOR XVAL = 0(.1) 1

           DO TVAL = EVAL(T, X, XVAL) PRINT TVAL END
```

At any point, the user may decide because of convergence considerations to determine the Taylor series solution about a new point, say x_o'. He need only invoke TAYLOR, specifying the new initial conditions $(x_o', T(x_o'))$ in order to obtain the new expansion. Thus, the analytic continuation of the solution function may be carried out under online interactive control by the user, who can change any or all of the parameters and the routine itself during the course of a "seance" with TAYLOR.

The Scope FORMAC routine TAYLOR follows (note its brevity).

```
TAYLOR: GET F, XO, YO, P PUT F, XO, YO, P

        F(0)=F SET E

"COMPUTE REQUIRED DERIVATIVES OF F"

        FOR I=1(1)(P-1)

            DO F(I)=DERIV(F(I-1), X)+DERIV(F(I-1), Y)*F

            PUT F(I) END

"COMPUTE TAYLOR SERIES SOLUTION TO GIVEN ODE"

        T=YO

        FOR I=1(1)P

            T=T+EVAL(F(I-1), X, XO, Y, YO)*(X-XO)**I/FAC(I)

        PUT T

        END
```

When one types DO TAYLOR, the response is a request for F, X0, Y0, P. If one then types

```
F  = X**2 + Y**2
X0 = 0
Y0 = 0
P  = 7
```

TAYLOR outputs the analytic expressions for $f^{(m)}$ $(m=1,2,\ldots,6)$, followed by

$$T = 1/3\ X^3 \times 1/63\ X^7 \quad .$$

Compare this example with page 99 of the "PL/I-FORMAC User's Reference Manual.

REFERENCES

1. R.G. Tobey, et.al., "PL/I-FORMAC Interpreter, User's Reference Manual," IBM Program Information Department, 360D 03.3.004, Hawthorne, New York, October 1967.

2. M.V. Wilkes, A Short Introduction to Numerical Analysis, Cambridge University Press, 1966.

This paper is dedicated to the late Clinton J. Carter, who played a key role in both the design and implementation of the Scope FORMAC Language.

AN ATTEMPT TO SOLVE DIFFERENTIAL EQUATIONS SYMBOLICALLY

by

Stanley B. Gershwin
Graduate School of Arts and Sciences
Harvard University
Cambridge, Massachusetts

Abstract

A set of experimental programs was written in the interactive FORMAC language to implement Picard iteration in solving systems of ordinary differential equations. The reasons why this method is apparently impractical are discussed, and possible remedies are suggested.

AN ATTEMPT TO SOLVE DIFFERENTIAL EQUATIONS SYMBOLICALLY

by

Stanley B. Gershwin

1. PICARD ITERATION

Consider the initial value problem

$$y' = f(x,y) \tag{1}$$

$$y(x_o) = y^o \tag{2}$$

The process of Picard iteration attempts to solve (1) and (2) by choosing an initial guess $y_o(x)$ and solving

$$y_{m+1}' = f(x,y_m(x)) \tag{3}$$

with initial condition (2). This is equivalent to integrating

$$y_{m+1} = y^o + \int_{x_o}^{x} f(t,y_m(t))\ dt. \tag{4}$$

Picard showed that (4) converges to the solution to (1) and (2) under suitable conditions.[1, 2]

A set of programs was written to implement (4) symbolically, using the experimental Scope FORMAC system. Because symbolic integration is a difficult operation, the choice of f(x,y) and initial guess $y_o(x)$ must be restricted. To perform the integration in (4) for $m=0$, $f(t,y_o(t))$ must be symbolically integrable, its integral must also be integrable, and so on. Further, because the FORMAC system does not provide symbolic integration subroutines, an integration routine had to be written. Thus, a simple class of functions was considered.

It is easy to show that

$$\int x^n e^{cx} \, dx = \frac{n!\,e^{cx}}{c} \sum_{i=0}^{n} (-c)^{i-n} \frac{x^i}{i!} \tag{5}$$

where n is a positive integer. Then the integral in (4) may be performed using (5) if

$$f(x,y_m(x)) = \sum_{n=0}^{N} x^n \left(A_n + \sum_{k=0}^{k_n} b_n \, e^{C_{nk}x}\right) \tag{6}$$

Equation (6) can be guaranteed to hold for all m if the following restrictions are made:

$$f(x,y) = \sum_{n=0}^{M} y^n \left(f_n(x) + \sum_{k=0}^{L} g_{nk}(x) e^{h_{nk}x}\right) \tag{7}$$

where $f_n(x)$ and $g_{nk}(x)$ are polynomials in x , and $y_o(x)$ is of the same form as (6).

An obvious candidate for an error indicator is

$$\epsilon_m = \int_{x_o}^{x_f} (y_m'(t) - f(t,y_m(t)))^2 \, dt \tag{8}$$

According to (3), this may be written

$$\epsilon_m = \int_{x_o}^{x_f} (f(t,y_{m-1}(t)) - f(t,y_m(t)))^2 \, dt \tag{9}$$

Equation (9) was used to calculate error because all quantities in the integrand had to be calculated for other reasons and because (6) implies that the integrand of (9) is integrable using (5). In (9), x_f is the upper limit of the interval $[x_o,x_f]$ on which the solution to (1) and (2) is desired.

2. DIFFICULTIES ENCOUNTERED

There are some important difficulties in applying the method of the previous section to initial value problems. The first is that Picard's proof that (4) converges does not hold for arbitrary x_f , but only for sufficiently small $x_f - x_o$.

Under some circumstances, however, the results of iterating (4) may be applied on a wider interval than is specified in Picard's proof. For instance, if neither f(x,y) nor the initial guess $y_o(x)$ have exponentials in x, then each iteration $y_m(x)$ is a polynomial in x. One can show that the coefficients of the low powers of x agree with the corresponding coefficients where the Taylor expansion is applied to the solution to (1) and (2). One can also show that $y_m(x)$ approaches the Taylor series as $m \to \infty$. Therefore $y_m(x)$ must be a good approximation to the solution (for sufficiently large m) in any interval which contains x_o but does not contain any singularities of the solution.

Picard's interval is much more restrictive than this. For example, if the problem is

$$y' = y^3 \qquad (10)$$

$$y(0) = 1 \qquad (11)$$

then Picard's interval is $|x| \le 4/27$, and the result of iterating with (4) is a Taylor series starting

$$1 + x + \frac{3}{2} x^2 + \frac{5}{2} x^3 + \frac{35}{8} x^4 + \ldots \qquad (12)$$

The exact solution to (10) and (11) is

$$y(x) = \frac{1}{\sqrt{1-2x}} \qquad (13)$$

of which (12) is the Taylor expansion. Since (13) has only one singularity which occurs at $x = 1/2$, (12) is equal to (13) in the wider interval $|x| < 1/2$.

The second important difficulty in using (4) to solve (1) and (2) is that expressions grow very rapidly in size. In most cases attempted, expressions grew so rapidly that all the core allocated to the FORMAC system was used and execution ceased. Also, as expression size increases, the time necessary to manipulate the expressions grows.

For example, consider (1) and (2) with (7) satisfied. Equation (7) may be written

$$f(x,y) = \sum_{n=0}^{M} y^n A_n(x) . \qquad (14)$$

Assume an initial guess of $y_o(x) = y^o$. Then

$$y_1' = \sum_{n=0}^{M} (y^o)^n A_n(x) . \qquad (15)$$

To simplify the problem and to get a lower bound on the growth of expression sizes, assume that $A_n(x)$ is a constant for each n . In this case, $y_1' = C$ and $y_1 = y^o + Cx$ (if, for simplicity, $x_o = 0$). Then

$$y_2' = \sum_{n=0}^{M} (y^o + Cx)^n A_n . \qquad (16)$$

When (16) is expanded, it contains $M + 1$ terms, and so y_2 contains $M + 2$ terms. In general, if y_m contains $k + 1$ terms, i.e., if y_m is a k^{th} degree polynomial, then y_{m+1}' contains $Mk + 1$ and y_{m+1} contains $Mk + 2$ terms. Therefore, the number of terms in y_m is roughly proportional to M^m.

Of course, the situation is much worse if A_n is a function of x in (7); the time spent expanding expressions depends on the presence and form of each $A_n(x)$. If, instead of a first order system, a higher order system is analyzed for expression growth, the problem requires significantly more time and space.

3. PRACTICAL PROBLEMS REQUIRING FURTHER STUDY

The solution of systems of ordinary differential equations using (4) might be practical if the questions listed below were answered.

a. Under what more general conditions does iteration of (4) converge to the solution of (1) and (2)? Is it possible to find an a priori bound on the error on a larger interval than Picard specifies?

b. Does some class C of functions other than polynomials with exponential coefficients (as herein) exist such that it is suitable for use with (4)? C must have the following property: a class D of 2-variable functions must exist such that if $f(x,y) \epsilon D$ and $y_m \epsilon C$, then $y_{m+1} \epsilon C$. Also, if $y(X) \epsilon C$, then there must exist an algorithm to integrate $f(x,y(X))$ symbolically. Finally, the rate of growth in the size of polynomial expressions should be decreased.

c. Can a good first guess $y_o(X)$ be chosen just on the basis of (1) and (2)? "Good" in this case refers not only to accuracy, but also to formal appearance, size of expression, and sizes of later expressions $y_1, y_2, \ldots$ which are consequences of y_o.

d. Can a relatively quick and systematic method be developed to delete from expressions terms which contribute little (numerically) but which take up space and time? Or, is it possible to calculate the numerical impact of a given term on later iterations, so that terms whose future impact is negligible may be deleted before they require processing time, rather than after?

4. CONCLUSION

Picard's iteration method (4) was found to be impractical in the implementation described herein. If some difficulties can be overcome—the extension of the interval of convergence and the reduction of the size of expressions—a practical implementation may be possible.

REFERENCES

1. E. L. Ince, Ordinary Differential Equations, Dover, 1956.

2. E. A. Coddington, An Introduction to Ordinary Differential Equations, Prentice-Hall, 1961.

SOME SYMBOLIC COMPUTATIONS IN FINITE FIELDS

by

Henry A. Feldman
Graduate School of Arts and Sciences
Harvard University
Cambridge, Massachusetts

Abstract

This paper contains bits of the elementary theory of finite fields and a report on some symbolic computations suggested by that theory. These computations, carried out in FORMAC, chiefly involved modular polynomial arithmetic.

The issue of generality versus special-purpose efficiency in a symbol manipulation language arises in the course of a discussion on the suitability of FORMAC for this project.

SOME SYMBOLIC COMPUTATIONS IN FINITE FIELDS*

by

Henry A. Feldman

1. INTRODUCTION

Finite fields are intricate algebraic structures. Each can have millions of elements, or only a handful, or as few as two; but in all cases these elements can be combined and manipulated in most of the ways rational or real numbers can, without leaving the field. One can add, subtract, multiply, divide, solve simultaneous linear equations, invert matrices, extract certain roots—all the time having at one's disposal only a finite number of elements.

What do these elements look like, and how can one do arithmetic with them? The basic theory of finite fields shows that all elements can be represented in terms of one element A; in fact, each element can be written as a polynomial in A, with each coefficient of the polynomial reduced to its remainder on division by a certain prime integer p. (From now on, "polynomial" will mean a polynomial with its coefficients so reduced.) Arithmetic among these expressions is carried out just like ordinary polynomial arithmetic, except that the polynomials too must be reduced to their remainders on division by a certain modulus polynomial f(A). Finite field arithmetic is exactly like the arithmetic of residues of f(A); in more precise mathematical terms, each finite field is isomorphic to the field of polynomial residues modulo some f(A).

If f(A) is chosen properly, the powers of A also represent the whole field. Such an f(A) is called a primitive polynomial.

Since the elements of a finite field can be represented by powers and polynomials, it should be clear that computer symbol-manipulation language is a convenient tool for performing computations in finite fields.

Below are described four FORMAC programs developed to work with finite fields. The first program takes a specified size for the finite field and finds what the modulus polynomial f(A) must be. It picks f(A) to be primitive and monic.

The second program takes the primitive polynomial f(A) and generates a table showing the correspondence between powers of A and polynomials in A. This table can be used for addition and multiplication in the field.

A third program solves simultaneous linear equations with coefficients taken from the finite field. A fourth finds all irreducible polynomials satisfied by elements of the field. All of these programs make heavy use of polynomial arithmetic: adding, multiplying, dividing, and taking greatest common divisors.

This paper presents small amounts of finite field theory, descriptions of how each of the four programs works and how it was implemented, the problems that arose, and how and where the programs might be improved.

2. PRIMITIVE POLYNOMIALS AND ARITHMETIC TABLES

What the Programs Do

The first two programs work hand in hand: the first finds the primitive polynomial f(A), and the other uses it to make a table of corresponding powers and polynomials in A.

A bit of theory will illuminate the way to find a primitive polynomial. Say the field is to have p^n elements. (All finite fields have prime-power cardinality.) Elementary theory (see Chapter 5, Sections 36 and 37 of reference 1) tells us that the polynomial

$$c(x) = \prod_{jk=p^n-1} (x^k - 1)^{\mu(j)}$$

called the cyclotomic polynomial, has irreducible factors of degree n, each of which is a satisfactory choice for the modulus polynomial. Here μ is the Möbius function, which is defined thus: $\mu(j) = 0$ if j is divisible by a square greater than 1; $\mu(j) = (-1)^k$ if j is the product of k distinct primes; and $\mu(1) = 1$. The product of all primitive polynomials for the field of p^n elements is, in fact, c(x).

The preceding discussion shows that the procedure which follows finds f(A).

1. Given p and n, form c(x) using the above formula.
2. Find an irreducible factor f(x) of c(x).
3. Reduce all powers $A, A^2, A^3, \ldots, A^{p^n-1}$ to their remainders on division by f(A). This gives a list, a one-to-one correspondence between powers of A and polynomials in A, which can be used to construct addition and multiplication tables for the finite field.

An Example

For instance, to find a primitive polynomial for the field of 16 elements (p = 2, n = 4) one forms

$$c(x) = \prod_{jk=15} (x^k - 1)^{\mu(j)}$$

$$= \frac{(x^{15} - 1)(x - 1)}{(x^5 - 1)(x^3 - 1)}$$

$$= x^8 + x^7 + x^5 + x^4 + x^3 + x + 1.$$

This polynomial factors into $(x^4 - x - 1)(x^4 - x^3 - 1)$. (Remember that all coefficients are being reduced modulo 2.) The choice of $x^4 - x - 1$ for f(x) results in Table 1, a representation of the field.

Table 1

Two representations of the field of 16 elements. On the left are the powers of A; on the right are their remainders on division by $A^4 - A - 1$.

0	$=$	0
A	$=$	A
A^2	$=$	A^2
A^3	$=$	A^3
A^4	$=$	$A + 1$
A^5	$=$	$A^2 + A$
A^6	$=$	$A^3 + A$
A^7	$=$	$A^3 + A + 1$
A^8	$=$	$A^2 + 1$
A^9	$=$	$A^3 + A$
A^{10}	$=$	$A^2 + A + 1$
A^{11}	$=$	$A^3 + A^2 + A$
A^{12}	$=$	$A^3 + A^2 + A + 1$
A^{13}	$=$	$A^3 + A^2 + 1$
A^{14}	$=$	$A^3 + 1$
A^{15}	$=$	1

Finding the Cyclotomic Polynomial

Computing

$$c(x) = \prod_{jk=p^n-1} (x^k - 1)^{\mu(j)}$$

requires a list of factors j of p^n-1. Any j's which are divisible by a square can be ignored, since they have $\mu(j) = 0$ and make no contribution to c(x). So if the prime factors of p^n-1 are $p_0, p_1, p_2, \ldots$, then the j's and their μ's are as shown in Table 2.

The program for finding c(x) generates the lists in Table 2 by the recursive formulas

$$j_{2^h+i} = p_h j_i \qquad h = 0, 1, 2, \ldots$$

$$\mu(j_{2^h+i}) = -\mu(j_i). \qquad i = 0, 1, 2, \ldots, 2^h$$

The program then finds the polynomials

$$c_1(x) = \prod_{\substack{jk=p^n-1 \\ \mu(j) =1}} (x^k - 1)$$

and

$$c_2(x) = \prod_{\substack{jk=p^n-1 \\ \mu(j)=-1}} (x^k - 1)$$

Table 2

Systematic listing of the squarefree factors of a number whose prime factors are p_0, p_1, p_2, . . . , along with their values for the Möbius function. Note: the first 2^m elements are repeated with an added factor as the second 2^m, whatever m is.

j_0	=	1	$\mu(j_0)$	=	1
j_1	=	p_0	$\mu(j_1)$	=	-1
j_2	=	p_1	$\mu(j_2)$	=	-1
j_3	=	p_1p_0	$\mu(j_3)$	=	1
j_4	=	p_2	$\mu(j_4)$	=	-1
j_5	=	p_2p_0	$\mu(j_5)$	=	1
j_6	=	p_2p_1	$\mu(j_6)$	=	1
j_7	=	$p_2p_1p_0$	$\mu(j_7)$	=	-1
j_8	=	p_3	$\mu(j_8)$	=	-1
j_9	=	p_3p_0	$\mu(j_9)$	=	1
j_{10}	=	p_3p_1	$\mu(j_{10})$	=	1
j_{11}	=	$p_3p_1p_0$	$\mu(j_{11})$	=	-1
j_{12}	=	p_3p_2	$\mu(j_{12})$	=	1
j_{13}	=	$p_3p_2p_0$	$\mu(j_{13})$	=	-1
j_{14}	=	$p_3p_2p_1$	$\mu(j_{14})$	=	-1
j_{15}	=	$p_3p_2p_1p_0$	$\mu(j_{15})$	=	1

Since $c(x) = c_1(x)/c_2(x)$, the program can now perform ordinary polynomial division to obtain c(x).

The polynomial division algorithm is particularly easy to program for $p = 2$, since all coefficients in this computation turn out to be +1 or -1. The only inconvenience is that c(x) usually has a great many terms, and these must be added to the quotient one by one as the algorithm proceeds. This labor would not have been necessary had there been available a polynomial manipulation language with polynomial division performed by an efficient, built-in function, as opposed to a general formula manipulation language such as FORMAC, which handles a broader set of algebraic expressions and has few special tools for working with polynomials.

Factoring c(x) with Berlekamp's Algorithm

This section contains the theory of our solution for step 2 of the subsection entitled "What the Programs Do," while the next two sections deal with implementation.

Step 2 involves extracting a primitive factor from the cyclotomic polynomial. Fortunately we know that each irreducible factor of c(x) is of degree n and is a primitive polynomial. So all we need to do is find an n^{th} degree factor of c(x).

We accomplish this task with Berlekamp's factorization algorithm. (See reference 2, in particular the final example on page 5a.) In its most general form, Berlekamp's algorithm can be applied to any polynomial with coefficients reduced modulo p. However, it can be vastly simplified for the limited purpose of factoring the polynomial $x^m - 1$ when m is relatively prime to p, as when one factors $x^{p^n-1}-1$. Since c(x) always divides $x^{p^n-1}-1$ (this is easily proven with an argument about the order of roots of c(x)), applying Berlekamp's algorithm to c(x) is just like finishing off the factorization of $x^{p^n-1}-1$, and the simplified algorithm works perfectly on c(x).

The following three steps constitute the algorithm in its simplified form.

a. Form the polynomials

$$K_i(x) = x^i + x^{ip} + x^{ip^2} + x^{ip^3} + \ldots, i = 1, 2, 3, \ldots$$

with all exponents reduced modulo $p^n - 1$ and the polynomial truncated where it begins to repeat. For instance, if $p = 2$, $n = 4$, then

$$K_3(x) = x^3 + x^6 + x^{12} + x^9.$$

It should be clear that there are only finitely many distinct $K_i(x)$.

b. Compute the polynomials

$$G_i(x) = \text{greatest common divisor} \left[c(x), K_{m_1}(x) + i\right],$$

$$i = 0, 1, 2, \ldots, p - 1, \text{ and } m_1 \text{ arbitrary.}$$

Berlekamp proves that

$$c(x) = G_0(x)\, G_1(x)\, G_2(x) \ldots G_{p-1}(x)$$

is a factorization, not necessarily complete, of $c(x)$.

c. To refine the factorization, split each $G_i(x)$ by computing the polynomials

$$G_{ij}(x) = \text{g.c.d.} \left[G_i(x), K_{m_2}(x) + j\right], \; i, j = 0, 1, 2, \ldots, p - 1,$$

$$m_2 \neq m_1.$$

Each polynomial $G_i(x)$ is split into

$$G_i(x) = G_{i0}(x)\, G_{i1}(x)\, G_{i2}(x) \ldots G_{i,p-1}(x)$$

yielding the refined factorization

$$c(x) = G_{00}(x)\, G_{01}(x)\, G_{10}(x)\, G_{11}(x)\, G_{02}(x) \ldots G_{p-1,p-1}(x).$$

To continue refining, compute the polynomials

$$G_{ijk}(x) = \text{g.c.d}\left[G_{ij}(x), K_{m_3}(x) + j\right], \quad i, j, k = 0, 1, 2, \ldots, p-1$$
$$m_3 \neq m_2, m_3 \neq m_1$$

and so forth. Berlekamp proves that by the time the $K_i(x)$ are exhausted, the factorization of c(x) is complete.

The author has no idea how many steps are generally needed to complete the factorization; this and other questions about the algorithm are discussed below in the subsection entitled "What It Would Help to Know About Berlekamp's Algorithm."

In searching for a primitive polynomial there is no point in finding all the factors of c(x); only one is needed. So in applying Berlekamp's algorithm to the search for a primitive polynomial, a lot of trouble is saved if the algorithm is modified as shown below.

a'. Split c(x) into

$$c(x) = G_0(x)\, G_1(x)\, G_2(x) \ldots G_{p-1}(x).$$

Select only one $G_i(x)$, nth degree or higher, and split it until some $G_{ij}(x)$ of degree n or higher appears.

b'. In turn, split $G_{ij}(x)$, and continue until some $G_{ijk...m}$ of exactly nth degree appears. This is the primitive polynomial.

Difficulties in Programming Berlekamp's Algorithm in FORMAC

This section discusses how each step of Berlekamp's algorithm is implemented in a factorization program.

a. Generating

$$K_i(x) = x^i + x^{ip} + x^{ip^2} + x^{ip^3} + \ldots$$

is straightforward and simple in FORMAC. These polynomials can be formed as they are needed in (b) and (c) below.

b. and c. To compute

$$G_i(x) = \text{g.c.d.}\left[c(x), K_{m_1}(x) + i\right]$$

the factorization program employs Euclid's algorithm in a straightforward way. Collin's methods[3] are better; but Euclid's algorithm, in its simplicity, brought out clearly major difficulties in FORMAC's handling of polynomials.

Euclid's algorithm involves repeated polynomial divisions, each of which requires that a zero test be performed on many partial remainders as they are generated. To test whether a new partial remainder is zero, one must first reduce its (newly computed) coefficients modulo p. To reduce a coefficient modulo p, one must extract it with the COEFF routine, convert it to a PL/I constant, apply the MOD function, and replace it in the polynomial as a FORMAC constant.

The upshot is that this extraction, reduction, and replacement must be performed for each coefficient of each partial remainder of each division in each performance of Euclid's algorithm for each refinement of the factorization of c(x). The time consumed can be prohibitive, even for small values of p^n.

What It Would Help to Know About Berlekamp's Algorithm

There are two apparent approaches to streamlining the factorization process. First Berlekamp's algorithm should be investigated mathematically, with the specific aim of reducing the number of polynomial greatest common divisors (g.c.d.'s) one must compute. Is there a way to pick m_1 and i so that an n^{th} degree $G_i(x)$ is found on the first try? Failing such a spectacular finding, are there at least optimal choices for i and m? Is there a quick way to estimate the degree of a polynomial g.c.d. before computing it, so as to pick the smallest, most worthwhile $G_i(x)$ to compute? Is there a quick way to tell whether a polynomial g.c.d. will be 1 before computing it, so as to avoid wasting a lot of time computing $G_i(x) = 1$? (Collins[3] gave a method for quick determination of a g.c.d. of unity for polynomials with ordinary integer coefficients. This method may be adaptable to the present case: the author did not have time to investigate.) Answers to any of these or similar questions could make the search for a primitive polynomial much speedier.

Some Suggestions Regarding FORMAC

The author made no attempt to answer the preceding mathematical questions, but has concentrated instead on a second line of attack: programming efficiently in FORMAC. However, no amount of cleverness has proved sufficient to avoid the laborious, term-by-term reduction of coefficients. This problem is symptomatic of a general problem in handling polynomials in FORMAC. The author suspects that the labor of extracting terms and putting them back in place after some trivial operation is one of the greatest and most common inefficiencies forced on the programmer in the current version of FORMAC.

The finite field computations described in this paper would be much easier to program if FORMAC contained built-in functions like the ones listed in Table 3. These functions are designed to eliminate common term-by-term operations on polynomials. Besides generally enriching the FORMAC library, they would particularly permit more rapid manipulation of simple, 1-variable polynomials.

Of course FORMAC is a general-purpose formula manipulation language, and it can be argued that a language of such broad scope and simplicity should not be overloaded with the special functions one expects in a special-purpose polynomial manipulation system. Still, since high-level languages are written principally to provide the programmer with convenient macros for very common tasks, such as polynomial manipulation, it seems a shame to saddle the FORMAC programmer, for whom the possibilities of invention are great, with needless tedium in writing his code.

Making Tables

This section continues with a program for step 3 of the subsection entitled "What the Programs Do." Once f(x) is found, FORMAC easily generates arithmetic tables. To list the powers of A paired with their residues modulo f(A), the program simply sets up a list A and puts in A(I) the remainder of A^I when divided by f(A).

Table 3

Some suggestions for polynomial manipulation functions in FORMAC. In each case the function call implies that A and B are to be regarded as polynomials in the FORMAC expression E. N is a FORMAC integer-valued expression.

TRUNC (A, E, N)	A with all terms above E^N dropped
TUMULT (A, B, E, N)	Product of A and B, ignoring all terms in product above E^N
MOD (A, E, N)	A with any numerical coefficients reduced modulo N
ABS (A, E)	A with any numerical coefficients replaced by their absolute values
PRIM (A, E)	A divided by g.c.d. of numerical coefficients

The program avoids polynomial division by doing the whole process recursively: given A(I), the program multiplies it by A, reduces it modulo f(A) by replacing any occurrence of A^n by A^n - f(A), reduces the coefficients modulo p, and places the result in A(I+1).

Table 1 is essentially a conversion table from powers of A to polynomials in A. To construct a conversion table in the opposite direction, one can use the fact that each residue of f(A) gives a different number between 0 and p^n when p is substituted for A. A list, matching the number to which each polynomial evaluates with the power to which it corresponds, acts as a key. Table 4 should make the method clear; it is the inverse of Table 1.

With these tables, it is simple to multiply and add, going from powers to polynomials or vice versa. It is most convenient to convert to polynomials before adding and to powers before multiplying.

3. SIMULTANEOUS LINEAR EQUATIONS

Finite field arithmetic at its messiest is exhibited by a program for the solution of simultaneous linear equations by Gaussian elimination, with coefficients and answers to be drawn from the field of 16 elements. This program reads the equations from cards as PL/I character strings which are then converted to FORMAC expressions. On the cards finite field elements may be expressed as either powers of A or polynomials in A.

After input the program converts the coefficients to powers of A, using Table 4, and represents the equations by a PL/I matrix containing the exponents of the powers of A. (A special element represents zero coefficients, since zero is no power of A.)

To perform arithmetic on the coefficients in the course of Gaussian elimination, the program extracts them from the PL/I array, refers to appropriate entries in the powers-to-polynomials table (Table 1), performs the arithmetic in FORMAC, converts back to powers of A, and replaces the answer in the array.

The programming of Gaussian elimination and back-substitution is straightforward. Sample input and output are reproduced in Table 5.

Table 4

Conversion table for polynomials to powers of A for the field of 16 elements. Given any polynomial in the first column, one substitutes p for A and obtains the number beside it in the second column. The corresponding power is directly across in the third column (see Table 1).

Polynomial	Number it Evaluates To with A = p	Power it Corresponds to
1	1	15
A	2	1
A + 1	3	4
A^2	4	2
$A^2 + 1$	5	8
$A^2 + A$	6	5
$A^2 + A + 1$	7	10
A^3	8	3
$A^3 + 1$	9	14
$A^3 + A$	10	9
$A^3 + A + 1$	11	7
$A^3 + A^2$	12	6
$A^3 + A^2 + 1$	13	13
$A^3 + A^2 + A$	14	11
$A^3 + A^2 + A + 1$	15	12

The programming of Gaussian elimination and back-substitution is straightforward. Sample input and output are reproduced in Table 5.

Table 5

Sample input and output for solution of simultaneous linear equations over the field of 16 elements.

$$\underline{0 = X + A^{3} X - 17A^{4} Y + A^{19} S + 3}$$

$$\underline{0 = A^{5} Q + A^{8} X + A^{3} Z}$$

$$\underline{0 = Q + X + Y + Z + S}$$

$$\underline{0 = R + A Q + A X + A Y + A Z + A S}$$

$$\underline{0 = A^{13} Q + A^{6} Y}$$

$$\underline{0 = A Q + A^{10} Y + A^{7} S}$$

$$\underline{Y = A^{10}}$$

$$\underline{X = A^{12}}$$

$$\underline{S = A}$$

$$\underline{Z = A}$$

$$\underline{Q = A^{3}}$$

$$\underline{R = 0}$$

4. MINIMAL POLYNOMIALS

Basic theory shows that all p^n elements of a finite field are roots of $(x^{p^n}-x)$. They are also roots of smaller polynomials, factors of $(x^{p^n}-x)$. It is of interest to know, for each element of the field, the smallest polynomial it satisfies. This polynomial is called the minimal polynomial of that particular element. This section describes a technique for finding all minimal polynomials of elements of a finite field.

It is simple to show that all minimal polynomials are irreducible, and that their product is $(x^{p^n}-x)$. So it is only necessary to factor $(x^{p^n}-x)$ to find the various minimal polynomials.

The first step is easy: $(x^{p^n}-x) = (x)(x^{p^n-1}-1)$. The factorization of $(x^{p^n}-x)$ can be finished with Berlekamp's algorithm. This time, of course, the program must keep track of all $G_{ijk\ldots m}(x)$ at each step, except for those equal to 1. This is easily accomplished by storing in one FORMAC location the undistributed product of all $G_{ijk\ldots m}(x)$ found so far.

A priori information, which can be used to guide the program in factoring $(x^{p^n-1}-1)$, is available on the size and number of minimal polynomials. Various theorems show that, for each divisor m of n, there are

$$\frac{1}{m} \sum_{jk=m} p^k_\mu (j)$$

m^{th} degree minimal polynomials, and that these are the only minimal polynomials. So, for example, if the program comes to a $G_{ij}(x)$ with a degree not dividing n, this factor is not a minimal polynomial and must be factored further; if this factor's degree does divide m and fewer than

$$\frac{1}{m} \sum_{jk=m} p^k_\mu (j)$$

m^{th} degree factors have been found so far, it may be a minimal polynomial and may not have to be factored further.

One can also use the fact that there are a total of

$$\sum_{m|n} \frac{1}{m} \sum_{jk=m} p^k \mu(j)$$

minimal polynomials to determine when to terminate the factorization.

Lack of time prevented the author from implementing this routine. As it is simply a longer version of the primitive polynomial program of section 2, the remarks of the three subsections of section 2 which discuss Berlekamp's algorithm apply all the more strongly to this routine.

5. SUMMARY

The project described here shows that finite field computation lends itself very well to computer symbol-manipulation language, to the extent that FORMAC programs can make arithmetic tables and even solve simultaneous linear equations in a conceptually straightforward manner.

On the other hand, the author found FORMAC awkward in programming and sluggish in performing extensive polynomial manipulations. Three routes for improvement are evident: one might

a. Study and streamline the factorization algorithms;

b. Use a special-purpose polynomial manipulation system; or

c. Introduce special-purpose capabilities into FORMAC.

REFERENCES

1. B. L. Van der Waerden, Modern Algebra, Frederick Ungar Publishing Company, 1949.

2. E. R. Berlekamp, "On the Factorization of Polynomials Over Finite Fields," Bell Telephone Laboratories, Murray Hill, New Jersey.

*3. C. E. Collins, "Computing Time Analyses for Some Arithmetic and Algebraic Algorithms," these Proceedings.

*This article appears in these Proceedings.

SYMBOLIC-NUMERIC EIGENVALUE PROBLEMS IN QUANTUM MECHANICS

by

Kenneth Hartt
Department of Physics
University of Rhode Island
Kingston, Rhode Island

Abstract

The problem of symbolic numeric conversion (SINCON) is discussed in the context of solving quantum mechanical eigenvalue problems with FORMAC. The eventual use of SINCON in a problem eliminates the need for containing the entire structure of each FORMAC expression in a single equality. Consequently, special coding techniques which reduce core storage requirements (called "reductions") are possible. Using an example involving SINCON for illustration, the paper shows methods of performing reduction by explicit FORMAC coding. It is argued that a new version of FORMAC in which automatic reduction techniques are incorporated is needed. Two alternative reduction devices are discussed.

SYMBOLIC-NUMERIC EIGENVALUE PROBLEMS IN QUANTUM MECHANICS

by

Kenneth Hartt

1. INTRODUCTION

Because a large class of physics and engineering problems involves lengthy symbolic derivations of formulas followed by numerical evaluation, a totally programmed solution requires a symbolic-numeric conversion (SINCON). This paper discusses the application of the present SINCON capabilities of FORMAC to quantum mechanical eigenvalue problems. Based on our recognition that expressions destined for numerical evaluation need not have their entire structure shown explicitly in a single equality, this paper suggests, by illustration, some extensions of FORMAC for improving SINCON.

Wherever the Rayleigh-Ritz method is applicable to an eigenvalue problem, the task of determining eigenfunctions and eigenvalues is partly reducible to optimizing a set of parameters. Using a simple example of parameter optimization, we show how SINCON allows use of special coding techniques, which we call "reduction," that reduce core storage requirements.

All users of symbol manipulation systems must perform reduction. Many do so by analysis prior to coding. A more efficient method for symbolic differentiation was developed by Eisenpress and Bomberault, who coded a procedure which transforms lengthy FORMAC expressions into outputs consisting of sequences of impressively short PL/I statements.[1] We propose incorporating a reduction device which could be similar to theirs but which is automatically invoked by FORMAC when needed. The concluding section of this paper discusses some suggestions concerning external specifications of optional reduction devices, as well as some unresolved issues.

2. SOLVING A QUANTUM MECHANICS PROBLEM IN FORMAC

In quantum mechanics one tries to solve a Schroedinger equation for a system of some number N of interacting particles $H\psi = E\psi$. Let the particle coordinate vectors be $\underline{x}_j$. Assume the interaction is pairwise through a potential energy operator $V(\underline{x}_i,\underline{x}_j) = V_{ij}$. Then

$$H = -\sum_{i=1}^{N} \frac{\hbar^2}{2M_i} \nabla_i^2 + \sum_{i<j} V_{ij}$$

and $\psi = \psi(\underline{x})$ is square-integrable over 3N-dimensional space (where the collection $\underline{x}_1,\underline{x}_2,\ldots,\underline{x}_N$) is represented by $\underline{x}$) . E is an eigenvalue, interpreted as a possible energy of the physical system. E must be negative for square-integrable solutions. Here discussion is restricted to finding the smallest eigenvalue E_o and the ground state ψ_o , such that $H\psi_o = E_o\psi_o$.

The two-body problem $(N = 2)$ is easily solved numerically, assuming any of a very general class of interactions for V_{12}.[2] This is not so for $N \geq 3$. Therefore, extensive effort in theoretical physics has been devoted to finding analytic approximations. One approach to this problem has been the familiar variational method, which is justified by the Rayleigh-Ritz principle.[3] Given a trial solution $\psi_t(\underline{x},\underline{A})$, where $\underline{A} = (A_1,A_2,\ldots,A_M)$ is a parameter vector, this procedure involves optimizing A by minimizing the quotient $E_t = (\psi_t,H\psi_t) / (\psi_t, \psi_t)$. A disadvantage of this approach is that E_t is only the approximate eigenvalue; the minimum E_t must still be obtained through a numeric iteration procedure which uses the functional forms of E_t and derivatives of E_t obtained symbolically. Another problem is that the inner products $(\psi_t,H\psi_t)$ and (ψ_t, ψ_t) are linear combinations of integrals whose derivation in terms of standard integrals is often a lengthy task. Frequently the forms of the standard integrals are well known, and the computational problem consists of lengthy algebraic and analytic procedures. Because of these problems an efficient SINCON is needed.

Provided that sufficient reduction is achieved, using FORMAC we should frequently be able to avoid the use of elaborate minimization algorithms, such as VARMINT.[4] VARMINT requires as input a function F and its gradient vector ∇F in the parameter space where F is being minimized. The VARMINT algorithm proceeds by making successive estimates of the inverse of the matrix $G_{ij} = \frac{\partial^2 F}{\partial A_i \partial A_j}$ (Hessian matrix) and advancing $\underline{A}$ by increments $(\underline{A} \rightarrow \underline{A}_M = \underline{A} + \Delta \underline{A})$ until a convergence criterion is satisfied. An increment vector $\Delta \underline{A}$ is uniquely defined by postulating that $F(\underline{A})$ is in the neighborhood of a minimum and that a second-order Taylor series expansion at $\underline{A}_M$ of F is exact.

In cases in which ψ_t contains several parameters nonlinearly (as in arguments of exponentials), it is frequently impractical to compute the input expression ∇E_t analytically by hand calculation. Rather, one should be able to employ a system like FORMAC. If, in addition, G_{ij} is known analytically, the estimation of G_{ij}^{-1} in VARMINT is not needed and a simple algorithm to minimize E_t might be coded directly in FORMAC, possibly with the inversion of G_{ij} performed in PL/I.

Tobey's description of some of the major scientific efforts involving FORMAC[5] reveals applications in optimization similar to those described above. Here we present a simple example to illustrate SINCON and to show what might be done to enable FORMAC to reduce expressions such as G_{ij} sufficiently without a great deal of coding by the user.

3. REDUCTION IN FORMAC

Our example is the following problem: determine the first increment ΔB from $B = 3$ towards the minimum of the expression $E = ((B^2 - 1)^2 + 4) / (B - 2)^2$, using the method of steepest descent. Assume that E corresponds to the trial energy eigenvalue to be minimized, B to the parameter, $(B^2 - 1)^2 / (B - 2)^2$ to the potential-energy, and $4/(B - 2)^2$ to the kinetic energy. One must compute the first and second derivatives of E from which $\Delta B = -E'/E''$. A straightforward PL/I FORMAC procedure, MIN1, performs the calculation

```
                MIN1:  LET(
        I.1     FNC(F) = $(1)**2;
        I.2     N = F(B**2 - 1.) + 4.;
        I.3     D = F(B - 2.);
        I.4     E = N/D;
        I.5     EP = DERIV(E,B);
        I.6     EPP = DERIV(EP,B);
        I.7     NEP = EVAL(EP,B,3.);
        I.8     NEPP = EVAL(EPP,B,3.);
        I.9     DELTA = - NEP/NEPP); .
```

The numeric calculation could have been done in PL/I after I.8 instead of FORMAC by the PL/I statements NEP = ARITH(NEP); NEPP = ARITH(NEPP); DELTA = - NEP/NEPP; .

Straightforward coding of a nuclear physics problem whose solution closely parallels MIN1 creates a size problem, traceable to the fact that the symbolic part of MIN1 combines and manipulates expressions on two levels, E and F. The user can reduce core requirements somewhat by performing a sufficiently extensive precoding analysis. Subexpressions for E, EP, EPP can be derived separately and FORMAC can be used to compute the values of F, F', and F'' and to make substitutions into the subexpressions. However, in MIN1 the structural form of F is needlessly repeated in the expressions for E, EP, and EPP, creating large structures that must be manipulated as whole entities and increasing the likelihood that freelist will be exceeded. The structure of second derivatives can easily grow to a prohibitive size. The precoding analysis required to save core grows as the complexity of the expressions manipulated increases until a point is reached at which such analysis becomes a formidable problem.

An alternative method of coding our example is MIN2, which uses function variables as shown below.

```
MIN2: LET(

II.1  N = G.(B) + 4.;

II.2  D = H.(B);

II.3  E = N/D;

II.4  EP = DERIV(E,B);

II.5  EPP = DERIV(EP,B);

II.6  FNC(F) = $(1)**2;

II.7  NEP = EVAL(EP,G.($(1)),F($(1)**2 - 1.),H.($(1)),

           F($(1) - 2.),B,3.);

II.8  NEPP = EVAL(EPP,G.($(1)),F($(1)**2 - 1.),H.($(1)),

             F($(1) - 2.),B,3.);

II.9  DELTA = - NEP/NEPP);
```

Notationally it appears that MIN2 reduces the core requirements, since the functions in II.8 and II.9 have numeric arguments. However, the interpretation of the numerical arguments as numbers and the subsequent reduction by the internal automatic simplification procedure does not occur until after the functional substitutions are made. Hence the phenomenon of intermediate swell,[6] i.e., the growth of storage requirements for expressions above the final sizes in an executed FORMAC statement, becomes important.

If the user is willing to code a rather elaborate series of substitutions, MIN2 can be modified so an eventual saving of core results for cases where F becomes increasingly complex. One replaces the function variables and their derivatives by subscripted atomic variables D(J, L) . The arguments of the function variables are assigned to other subscripted variables X(I) . Then the D(J, L) are assigned values of the numerically evaluated function F and its derivatives. This procedure,. which we give as MIN3, can be generalized to several functions and variables, in which case PL/I loops perform the substitutions with additional indexes.

```
          MIN3:  LET(
     III.1    N = G(1).(B) + 4.;
     III.2    D = G(2).(B);
     III.3    E = N/D;
     III.4    X(1) = B**2 - 1.;
     III.5    X(2) = B - 2.;
     III.6    E(1) = DERIV(E,B);
     III.7    E(2) = DERIV(E(1),B);
     III.8    FNC(F) = $(1)**2);
     III.9    SUB: DO I = 1 TO 2; DO J = 1 TO 2;
     III.10   LET(I = "I"; J = "J";
     III.11   E(I) = REPLACE(E(I),G(J).(B),D(J,1)));
     III.12   DO L = 1 TO 2; LET(L = "L";
     III.13   E(I) = REPLACE(E(I),DRV(G(J).(B),$(1),L),D(J,L+1)));
     III.14   END SUB;
```

The final expressions are compact and are suitable patterns for substitutions of numerical values of F .

```
III.15    NUM:  DO I = 1 TO 2; DO J = 1 TO 2;
III.16    LET(I = "I"; J = "J"; E(I) = EVAL(E(I),D(J,1),
            EVAL(F(X(J)),B,3.),B,3.));
III.17    DO L = 1 TO 2; LET(L = "L"; E(I) = EVAL(E(I),D(J,L+1),
            EVAL(DERIV(F(X(J),B,L),B,3.))); END NUM;
III.18    LET(DELTA = - E(1)/E(2));
```

The substitution method of MIN3 requires careful precoding analysis, but less than the modifications already discussed in connection with MIN1.

Table 1 compares execution times and core sizes of the three different programs. Core sizes are roughly estimated as follows. All programs are run with the printout option, which causes a printing of FORMAC expressions obtained from the execution of all statements. The total length of the printout from each procedure is an approximate relative measure of the accumulated amount of core required. Two important qualifications need to be made here. First, if SAVE is employed the maximum amount possible, then the size of the largest collection of expressions handled simultaneously in any FORMAC statement is a more appropriate measure of required core. As suggested by the last column of Table 1, this tends to offset the large core size shown for MIN3. Second, no measure of intermediate swell is provided. Although the comparisons in Table 1 should not be accepted literally, they reveal the overhead in time and space resulting from performing the substitutions of MIN3.

We wish to emphasize that MIN3 has a high potential efficiency relative to MIN2 that is not shown in Table 1; as the function F becomes lengthier, all times for MIN2 grow faster than those for MIN3, and intermediate swell in MIN2 grows excessively.

Table 1

COMPARISON OF EXECUTION TIMES AND PRINT-OUT LENGTHS

Code	CPU time (seconds)	CPU time (ratios)	Summed* expression lengths	Summed expression length ratios	Maximum* expression length
MIN1	13	1.6	8.4	2.5	4.3
MIN2	8	1.0	3.4	1.0	1.7
MIN3	16	2.0	25.0	7.4	2.1

* lengths are in units of 12-inch lines.

4. A SUGGESTED SINCON CAPABILITY

If numerical evaluations are to be made in lengthy symbolic mathematics codes, an option to automatically split an expression into its subfunctional parts is desirable. We refer to the class of such options as Automatic Splitting-Subfunctional Parts (ASSP). The functional configuration of ASSP will not cause an explicit definitional substitution at each occurrence of a function, but it will generate a function list and an argument list for each FORMAC expression that contains functions. Therefore a statement containing a sequence of nested functions, e.g., F(G(H(X))), which can quickly grow to a prohibitive size when the explicit forms of F, G, and H are given, remains compact. Since the ASSP option is designed to save space,. an internal automatic SAVE is desirable. A technical question requiring study is the extent to which the function and argument lists of different expressions should overlap; i.e., whether these lists should have local or global scope.

Another question is whether ASSP should coexist with conventionally formed expressions in core or be exclusive. Although an exclusive ASSP might be simpler to implement, it might be difficult to use for studying the algebraic structures of expressions that have been only partially evaluated numerically.

For example, a symbolic-numeric expression might equal zero to within the numerical precision employed, but a demonstration of this property could require a recombination from the ASSP-generated structures. The external specifications of a coexistent ASSP could be extremely simple. One could, for example, reserve XX for prefixes of ASSP-structured names, with the possibility of causing a recombination with such a statement as LET(A = XXA); . An alternative solution to determine whether an expression equals zero would be to have a unique form for each expression.

Principal advantages of ASSP include the saving of space resulting from decreased repetition of common subexpressions and functional forms, reduced intermediate swell, and improved coding efficiency. A disadvantage might be the necessity for analyzing the propagation of numerical error. As can be seen in the example of section 3, the completion of a parameter optimization task involves SINCON. The specific manner in which this is done can lead to different types of numerical error. However, the sequences of short expressions envisaged as the output of an ASSP transformation should be an especially suitable form for processing in PL/I, thereby alleviating the problem of numerical error analysis in the design of a new FORMAC. (A user would manually check the PL/I code prior to evaluation.)

An alternative device to facilitate SINCON is a procedure that would cause a function variable association with an evaluated function (FAEF).

FAEF(G(1).(B)) = F(X(1));

would be a FORMAC statement, providing the code automatically for the substitutions given in MIN3. A possible way for FAEF to work would be for all occurrences of G(1).(B) and its derivatives encountered after execution of the above statement to be replaced by the evaluated F and its derivatives; the evaluation would be performed with the values of X(1) and B that were current when FAEF was executed. FAEF would provide a limited form of automatic back-substitution.

FAEF does not go as far as ASSP because the internal formats need not be changed from those presently employed in FORMAC, and the problem of efficiently generating the expressions used for F(X) still must be solved. For the same reason, no new problem would arise in inspecting algebraic structures, such as would arise in ASSP.

Although ASSP appears to have considerably greater potential as a reduction device, either ASSP or FAEF could facilitate the solution of a large class of problems. In a nuclear three-body problem, for instance, several nonlinear parameters must be optimized in minimizing E_t.[7] Because a symbolic system is inherently slower than a numerical system, ASSP appears to be superior in cases in which a large number of numerical iterations is required. This is because ASSP should be easier to adapt for outputs to a PL/I numeric routine. In contrast, FAEF brings the symbolic-numeric interface deeper into the logic of FORMAC and it should therefore be more effective in making FORMAC itself a more efficient system.

A by-product of ASSP would be improved comparisons of nested expressions. Automatic duplication of the results of expression-matching codes such as SHRINK, developed by J. B. Baker and reported in reference 6, should be facile.

It seems worthwhile to study implementation of ASSP and FAEF in FORMAC. Because of the interpretive nature of FORMAC, such automatic but optional procedures might easily be incorporated into the existing FORMAC system.[8, 9] Hopefully, such implementation will not be unreasonably complex and it will not require too much overhead of space and time.

Bringing the additional class of problems within reach of FORMAC through implementing these ideas would make the FORMAC system of even greater importance in the physical sciences and engineering.

REFERENCES

1. Harry Eisenpress and Abel Bomberault, "Efficient symbolic differentiation using PL/I-FORMAC," IBM TR 320-2956, 1968.

2. L. Lovich and S. Rosati, "Direct numerical integration of the two-nucleon Schroedinger equation with tensor forces," Physics Review, 140, 4B, 22 November 1965, pp. B877-B882.

3. K. Gottfried, Quantum mechanics, W.A. Benjamin, Inc., New York, 1966.

4. W. C. Davidon, "Variable metric method for minimization," ANL-5990 (Rev. 2), Argonne National Laboratory, Argonne, Illinois, 1966 (unpublished).

5. R. G. Tobey, "Eliminating monotonous mathematics with FORMAC," Communications of the Association for Computing Machinery Vol. 9, No. 2, October 1966, pp. 742-751.

6. R. G. Tobey, "Experience with FORMAC algorithm design," Communications of the Association for Computing Machinery Vol. 9, No. 8, August 1966, pp. 589-597.

7. L. M. Delves and J. M. Blatt, "Binding energy of the triton," Nuclear Physics A98, October 1967, pp. 503-526.

8. R. G. Tobey, et al., "PL/I FORMAC Interpreter, User's Reference Manual," IBM Contributed Program Library, 360D 03.3.004, Hawthorne, New York, October 1967.

9. R. G. Tobey, private communication.

THE USE OF COMPUTER-AIDED SYMBOLIC MATHEMATICS TO EXPLORE THE HIGHER DERIVATIVES OF BELLMAN'S EQUATION

by

Stanley B. Gershwin
Graduate School of Arts and Sciences
Harvard University
Cambridge, Massachusetts

Abstract

The FORMAC system is used to solve the following problem in optimal control theory: how do the third and higher space derivatives of the optimal value function behave along the optimal trajectory?

Bellman's equation is analyzed by taking derivatives of all orders. It is found that, unlike the second derivative which satisfies a nonlinear equation, the third and higher derivatives of the optimal value function satisfy linear differential equations along the optimal trajectory. Analysis of those equations shows that if certain conditions are satisfied, their solutions are unique and bounded. Consequently, all partial derivatives of the optimal feedback control function are unique and bounded.

A general algorithm is proposed for numerically solving optimal control problems using the higher derivatives. The differential equations for the third and fourth derivatives are displayed.

THE USE OF COMPUTER-AIDED SYMBOLIC MATHEMATICS TO EXPLORE THE HIGHER DERIVATIVES OF BELLMAN'S EQUATION

by

Stanley B. Gershwin

1. INTRODUCTION

An optimal control problem is a problem of the following form. Consider the differential equation

$$\dot{x} = f(x, u, t) \tag{1}$$

with initial conditions

$$x(t_o) = x_o. \tag{2}$$

Then if u(t) is a known function of time, x(t) depends on $u(\tau)$, $t_o \leq \tau \leq t$. Define

$$J(x_o, t_o) = \int_{t_o}^{t_f} L\,(x(t),\, u(t),\, t)\, dt + \phi\,(x(t_f)). \tag{3}$$

If $u(\tau)$, $t_o \leq \tau \leq t_f$ is known, $J(x_o, t_o)$ is known. Find the function $u(\tau)$, $t_o \leq \tau \leq t_f$ which minimizes $J(x_o, t_o)$. The minimum will be called $V(x_o, t_o)$.

In this paper, only the case where x(t) and u(t) are scalars is considered.

It has been shown[1] that the minimizing $u(\tau)$ may be obtained from the solution of the following equations simultaneously with Equation (1).

$$\dot{\lambda} = - L_x - \lambda f_x \tag{4}$$

$$0 = L_u + \lambda f_u \tag{5}$$

where

$$\lambda(t_f) = \phi_x(x(t_f)) \tag{6}$$

and thus $V(x_o, t_o)$ may be found by integrating Equation (3). Equations (4) and (5) are the Euler-Lagrange equations.

Alternatively, it has been shown[1] that $V(x,t)$ and $u(x,t)$ satisfy the Hamilton-Jacobi-Bellman Equation

$$- \frac{\partial V}{\partial t} = \min_{u} \; [L\,(x,\, u,\, t) + \frac{\partial V}{\partial x}\, f(x,\, u,\, t)] \tag{7}$$

with boundary condition

$$V(x(t_f)) = \phi\,(x(t_f)). \tag{8}$$

This approach is known as dynamic programming.

Differential dynamic programming (see references 2 and 3) seeks to use (7) to improve a nominal, nonoptimal trajectory, if the nominal is known to be sufficiently close to the optimal trajectory.

It is also of interest to solve the related discrete problem: analogously to equations (1) and (2), $x(t_i)$ is determined as a function of $u(t_j)$, $o \leq j \leq i$ from

$$x(t_{i+1}) = f(x(t_i),\, u(t_i),\, t_i) \qquad \text{for } i = 0, \ldots, N - 1 \tag{9}$$

$$x(t_o) = x_o \tag{10}$$

and

$$J(x_o, t_o) = \sum_{i=0}^{N-1} L(x(t_i), u(t_i), t_i) + \phi (x(t_N)). \qquad (11)$$

The minimum of $J(x_0, t_0)$ with respect to $u(t_0), \ldots, u(t_{N-1})$ is $V(x_o, t_o)$.

Equations analogous to (4) and (5) are found in reference 1, and equations analogous to (7) and (8) are found in reference 4. The latter reference also contains a differential dynamic programming treatment for the discrete problem, similar to that for the continuous problem in (2) and (3).

It is of interest to find perturbation feedback laws. If an optimal trajectory passing through a given point is known, how is the optimal trajectory passing through a neighboring point related to it? Stated more precisely, if $x(t) = \tilde{x} + \delta x$, the optimal control at time t will be $u(t) = u^o + \delta u$, where u^o is the optimal control if $x(t) = \tilde{x}$. Find δu as a function of δx. In particular, if this function can be expressed as a Taylor series, find the derivatives

$$\frac{\partial u}{\partial x}_{\text{optimal}}, \quad \frac{\partial^2 u}{\partial x^2}_{\text{optimal}}, \quad \ldots .$$

Significant computational and theoretical results have been obtained by performing Taylor expansions to first and second order on (4) and (5) in reference 1, on (7) in references 2 and 3, and on equations (9), (10), and (11) in reference 4. The aim of this paper is to generalize some of those results by performing higher order expansions on (7).

The second order analysis of (7) shows that the derivatives V_x and V_{xx} satisfy certain differential equations along optimal trajectories. The equation for V_{xx} is nonlinear, and we must show that V_{xx} is finite to guarantee that a given trajectory is truly optimal and unique. The succeeding sections show that the higher derivatives V_{xxx}, V_{xxxx}, etc., all satisfy linear differential equations along the optimal trajectory and thus if V_{xx} is finite, all higher derivatives are also finite. Then all derivatives $\frac{\partial u}{\partial x}_{\text{optimal}}$, are finite (and known if the derivatives of V are known).

The FORMAC system was used to generate the lower-order equations (third and fourth), from which the general analysis of this paper was performed. Similar experimentation was applied to the discrete problem (9), (10), (11). The resulting expressions were considerably more complicated and the analysis was not completed.

In the rest of the paper, the following notation will be used:

$$A = A(\bar{x}, u^*, t); \bar{A} = A(\bar{x}, \bar{u}, t)$$

where A is some function evaluated along a nominal trajectory

$$A_x = \frac{\partial}{\partial x} A(x, u^*, t) \Big|_{x = \bar{x}}$$

($\bar{x}$, $\bar{u}$, u* will be defined below.)

As in the literature, the following exception is made. The derivatives of H

$$H(x, u, t) = L(x, u, t) + \frac{\partial V}{\partial x}(x, t)\, f(x, u, t)$$

are taken with $\frac{\partial V}{\partial x}$ held constant. Thus

$$H_x = \frac{\partial}{\partial x} H(x, u^*, t) = L_x(x, u^*, t) + V_x(x, t)\, f_x(x, u^*, t).$$

Define $\mathcal{H} = H$; but when differentiating $\mathcal{H}$ with respect to x, allow for the variation of V_x. Thus

$$\mathcal{H}_x = H_x + V_{xx} f.$$

$$\mathcal{H}_{xx} = H_{xx} + 2V_{xx} f_x + V_{xxx} f.$$

2. DIFFERENTIAL DYNAMIC PROGRAMMING

As described in references 2, 3, and (in a discrete form) 4, differential dynamic programming has been implemented as described below.

Equation (1) is integrated using the nominal control history $\bar{u}(\tau)$ (which is given and is nonoptimal) to produce the initial nominal trajectory $\bar{x}(t)$. Then Equation (7) is integrated backwards along the nominal trajectory, using an approximation to $\frac{\partial V}{\partial x}$. The value of u satisfying (7) (called u*) is used in calculating a new nominal trajectory, and the cycle repeats.

To get the approximation to $\frac{\partial V}{\partial x}$ and to see how to use u* to obtain a new nominal trajectory, the following analysis is performed. Assuming $\bar{x}$ and u* are known, the optimal trajectory is defined by

$$x(t) = \bar{x}(t) + \delta x(t) \tag{12}$$

$$u(t) = u^*(t) + \delta u(t).$$

If (7) is expanded in a Taylor series about x and u*, the result is

$$-\frac{\partial}{\partial t}\ [V(\bar{x},\ t) + \frac{\partial V}{\partial x}\ (\bar{x},\ t)\ \ \delta x + \dots] \tag{13}$$

$$= \min_{u}\ [L(\bar{x},\ u^*,\ t)\ + \frac{\partial L}{\partial x}\ \delta x + \frac{\partial L}{\partial u}\ \delta u + \dots$$

$$+ \left(\frac{\partial V}{\partial x}\ (\bar{x},\ t)\ + \frac{\partial^2 V}{\partial x^2}\ \delta x + \dots\right)\ (f\ (\bar{x},\ u^*,\ t) + f_x\ \delta x + f_u \delta u + \dots)].$$

The minimization is performed, and u is found as a function of x. Under proper conditions, this may be expressed as a Taylor series

$$u(t) = u^*(t) + \delta u\ (t) \tag{14}$$

$$= u^*(t) + u_x \delta x + \frac{1}{2}\ u_{xx} \delta x^2 + \dots\ .$$

Clearly the coefficients in (14) will involve $\frac{\partial V}{\partial x}$, $\frac{\partial^2 V}{\partial x^2}$, etc.

If the expansion (Equation (14)) for δu is inserted into Equation (13), both sides of (13) are Taylor series in δx, and thus they are termwise equal. Insolating terms in Equation (13) from which δx is absent, one obtains a prediction of the improved value of V:

$$-\frac{\partial}{\partial t} V(\bar{x}, t) = L(\bar{x}, u^*, t) + \frac{\partial V}{\partial x}(\bar{x}, t)\, f(\bar{x}, u^*, t) \quad . \tag{15}$$

Or since

$$\frac{dA}{dt} = \frac{\partial A}{\partial t} + \frac{\partial A}{\partial x}\dot{x} = \frac{\partial A}{\partial t} + \frac{\partial A}{\partial x}\bar{f} \tag{16}$$

along the current nominal trajectory, where

$$\bar{f} = f(\bar{x}, \bar{u}, t)$$

then

$$-\frac{d}{dt} V(\bar{x}, t) = L + \frac{\partial V}{\partial x}(f - \bar{f}) \; . \tag{17}$$

Similarly, the terms in Equation (13) containing δx to the first power form the equation for $\frac{\partial V}{\partial x}$:

$$-\frac{d}{dt}\frac{\partial V}{\partial x} = \frac{\partial L}{\partial x} + \frac{\partial V}{\partial x}\frac{\partial f}{\partial x} + \frac{\partial^2 V}{\partial x^2}(f - \bar{f}) + \left(\frac{\partial L}{\partial u} + \frac{\partial V}{\partial x}\frac{\partial f}{\partial u}\right) u_x \; . \tag{18}$$

In Equation (18), the last term may be dropped because u* was found to minimize $H = L(\bar{x}, u^*, t) + \frac{\partial V}{\partial x}(\bar{x}, t)\, f(\bar{x}, u^*, t)$

and thus

$$H_u = \frac{\partial}{\partial u} H = L_u + \frac{\partial V}{\partial x} f_u = 0 \; . \tag{19}$$

Likewise, equating coefficients of δx^2 in Equation (13) yields

$$-\frac{d}{dt}\frac{\partial^2 V}{\partial x^2} - L_{xx} + \frac{\partial V}{\partial x} f_{xx} + 2f_x \frac{\partial^2 V}{\partial x^2}$$

$$+ u_x (L_{ux} + V_x f_{ux} + f_u V_{xx}) + V_{xxx}(f - \bar{f}). \tag{20}$$

Clearly there is no conceptual difference between finding Equation (20) and finding similar equations for higher order derivatives. However, the computational complexity is considerable. The third order equation was found by McReynolds[5]. In addition, T.E. Bullock informed D.H. Jacobson in a private communication that he computed equations up to the seventeenth order for a specific scalar problem. He noticed, in that case, that all equations of an order exceeding two were linear.

To find the coefficients in (14) (u_x and u_{xx}) it should be noticed that δu in Equation (12) is chosen to preserve

$$H'_u(\bar{x} + \delta x, u^* + \delta u, t) = 0 \tag{21}$$

or

$$L_u(\bar{x} + \delta x, u^* + \delta u, t) + \frac{\partial V}{\partial x}(\bar{x} + \delta x, t)$$

$$f_u(\bar{x} + o x, u^* + \delta u, t) = 0. \tag{22}$$

Thus,

$$\frac{\partial}{\partial x} \mathcal{H}_u \delta x + \frac{\partial}{\partial u} \mathcal{H}_u \delta u + \ldots = 0 \tag{23}$$

or

$$u_x = \lim_{\delta_x \to o} \frac{\delta u}{\delta x} = -\frac{\frac{\partial}{\partial x}\mathcal{H}u}{\frac{\partial}{\partial u}\mathcal{H}_u} \tag{24}$$

assuming $\frac{\partial}{\partial u} \mathcal{H}_u$ is nonzero.

This may be written

$$u_x = - (H_{uu})^{-1} (L_{ux} + V_x f_{ux} + f_u V_{xx}) \tag{25}$$

or

$$u_x = - (H_{uu})^{-1} (H_{ux} + f_u V_{xx}) \text{ .}$$

3. HIGHER ORDER EQUATIONS

Form of the Equations

The Hamilton-Jacobi-Bellman Equation may be written

$$0 = V_t + \mathcal{H} \text{ .} \tag{26}$$

Then the equations for V_x (17), V_{xx} (20), and all higher derivatives may be written

$$0 = (\frac{\partial}{\partial x})^n V_t + (\frac{\partial}{\partial x})^n \mathcal{H} \tag{27}$$

where the derivatives with respect to x are taken preserving

$$0 = H_u = \mathcal{H}_u \text{ .} \tag{28}$$

But,

$$(\frac{\partial}{\partial x}) \Bigg|_{H_u = 0} = \frac{\partial}{\partial x} \Bigg|_{u = \text{constant}} + u_x \frac{\partial}{\partial u} \Bigg|_{x = \text{constant}} \tag{29}$$

where u_x is given by Equation (25).

Thus the equation for V_x (15) may be written

$$0 = V_{xt} + \mathcal{H}_x \tag{30}$$

because

$$\frac{\partial}{\partial x} \Bigg|_{H_u = o} \mathcal{H} = \mathcal{H}_x + u_x \mathcal{H}_u = \mathcal{H}_x \tag{31}$$

or

$$0 = V_{xt} + H_x + V_{xx} f \, .$$

The equation for V_{xx} (20) is

$$O = V_{xxt} + \left(\frac{\partial}{\partial_x} + u_x \frac{\partial}{\partial u}\right)\left(H_x + V_{xx} f\right) \tag{32}$$

or

$$0 = V_{xxt} + H_{xx} + 2V_{xx} f_x + V_{xxx} f + u_x \left(H_{ux} + V_{xx} f_u\right) \, . \tag{32'}$$

This is quadratic in V_{xx} because of the form of u_x.

When the transformation (16) is applied to (31) and (32'), the latter equations become ordinary differential equations in V_x and V_{xx} along the nominal trajectory. Equation (31) is a linear differential equation, and (32) is a nonlinear Ricatti equation (because u_x is given by (25)).

It will now be shown that the higher order equations which are generated from (27) using transformation (16) and using expressions for u_x and u_{xx} derived from Equation (25) are all linear ordinary differential equations. Thus, Equation (27) is a linear equation for all values of n except n = 2, in which case it is a Ricatti equation.

Equation (32) may be written

$$0 = V_{xxt} + \mathcal{H}_{xx} + \mathcal{H}_{xu} u_x \, . \tag{33}$$

The equivalent of Equation (27) may be obtained by applying (29) to (33) n - 2 times.

$$0 = V_{x^n t} + \left(\frac{\partial}{\partial x}\right)^{n-2} H_{xx} + \left(\frac{\partial}{\partial x}\right)^{n-2} \left(H_{xu} u_x\right) \tag{34}$$

where the derivative is taken along $\mathcal{H}_u = 0$.

When Equation (34) is expanded, all derivatives of u with respect to x up to $u_{x^{n-1}}$ are present. The latter is present only in the term

$$\mathcal{H}_{xu} \; u_{x^{n-1}} \; .$$

From Equation (25) it is apparent that $u_{x^{k-1}}$ contains all derivatives of V up to V_{x^k}. The latter appears only linearly, as

$$- \; H_{uu}^{-1} \quad f_u V_{x^k} \; .$$

Thus, on expanding Equation (34), the only contribution of V_{x^n} due to $u_{x^{n-1}}$ is

$$- \; \mathcal{H}_{xu} \; H_{uu}^{-1} \; f_u V_{x^n} \; .$$

In general, the last term of Equation (34) will expand out as a sum of terms of the form

$$\mathcal{H}_{x^i u} \; u_{x^{n-i}} \tag{35}$$

for $i = 1, \ldots, n-1$.

The highest derivative of V that appears in $\mathcal{H}_{x^i u} = (\frac{\partial}{\partial x})^{\,i} \, (L_u + \frac{\partial V}{\partial x} f_u)$ is $V_{x^{i+1}}$. Thus for $i \neq 1$ ($i = 1$ was covered above) the only occurrence of V_{x^n} is due to $\mathcal{H}_{x^{n-1}u} \; u_x$, and the term containing V_{x^n} is

$$u_x f_u V_{x^n} = - \mathcal{H}_{xu} (H_{uu})^{-1} f_u V_{x^n} .$$

Therefore, the last term in (34), $(\frac{\partial}{\delta x})^{n-2} (H_{xu} u_x)$, contributes V_{x^n} in the form

$$2 u_x f_u \; V_{x^n}$$

and contributes no higher derivatives of V.

Finally, $\mathcal{H}_{xx} = L_{xx} + V_{xxx} f + 2 V_{xx} f_x + V_x f_{xx}$. The terms involving $V_{x^{n+1}}$ and V_{x^n} in $(\frac{\partial}{\delta x})^{n-2} \mathcal{H}_{xx}$ are

$$V_{x^{n+1}} f + (n-2) \; V_{x^n} \; (f_x + u_x f_u) + 2 \, V_{x^n} f_x$$

$$= V_{x^{n+1}} \; f + n \, V_{x^n} f_x + (n-2) \; u_x f_u V_{x^n}$$

and no higher derivatives of V appear. Therefore, the terms in V_{x^n}, $V_{x^{n+1}}$, and $V_{x^n t}$ in Equation (34) are

$$V_{x^n t} + V_{x^{n+1}} \; f + n \; (f_x + u_x f_u) \; V_{x^n} . \tag{36}$$

All other terms in (34) involve derivatives of V of lower order. Thus (34) may be written

$$\frac{dV_{x^n}}{dt} + n \, (f_x + u_x f_u) \; V_{x^n} = G_n(t) - V_{x^{n+1}} \; (f - \bar{f}) \tag{37}$$

by means of (16) where the expression $G_n(t)$ is formed from derivatives of L, f, and V, and contains no derivative of V of higher order than $V_{x^{n-1}}$.

In $G_n(t)$, derivatives of L and f are with respect to x and u, and derivatives of V are only with respect to x.

The only quantity that appears in a denominator of a fraction in $G_n(t)$ is H_{uu} and its powers. This situation exists because the form of u_x implies that u_{xx} must be written as a quantity divided by $(H_{uu})^2$ and in general, u_{x^k} is a quantity divided by $(H_{uu})^k$. Since $G_n(t)$ is a sum of products of powers of u_{x^k}, it will contain inverses of powers of H_{uu}. Clearly from the form of (34), $G_n(t)$ will have no other expression as a denominator.

In Equation (37), all quantities are evaluated at $x = \bar{x}$ and $u = u^*$, except $\bar{f}$, which is evaluated at $x = \bar{x}$ and $u = \bar{u}$.

The boundary conditions for equations (17), (18), (20), and (37) are given at $t = t_f$ by Equation (38).

$$V_{x^n}(x(t_f), t_f) = \phi_{x^n}(x(t_f)) \tag{38}$$

for $n = 0, 1, 2, \ldots.$

Higher Order Algorithms

Computationally Equation (37) may be used to generalize the first and second order differential dynamic programming algorithms of references 2 and 3. Briefly, such an algorithm would be executed as follows.

1. Guess $u\ (t)$.
2. Integrate Equation (1) with $u = \bar{u}\ (t)$ forward from $t = t_o$ to $t = t_f$ to find $\bar{x}$. Use the boundary condition in (2).
3. Integrate (15), (20), and (37) for $n = 3, \ldots, N$ along $x = \bar{x}$ and $u = u^*$ from $t = t_f$ to $t = t_o$, using the boundary conditions (38). (In the N^{th} equation, the $V_{x^{n+1}}(f - \bar{f})$ term is ignored.) To find

u^*, minimize $L(\bar{x}, u, t) + V_x(\bar{x}, t)\, f(\bar{x}, u, t)$ with respect to u.

4. Calculate $u_x, u_{xx}, \ldots, u_{x^{N-1}}$. (The formulas for these are found by differentiating Equation (24). Exactly N-1 derivatives are possible because N derivatives of V are known.)

5. Integrate Equation (1) with $u = u^* + u_x \delta x + \frac{1}{2} u_{xx} \delta x^2 + \ldots + \frac{1}{(N-1)!} u_{x^{N-1}} \delta x^{N-1}$ where $\delta x = x - \bar{x}$ and x is the solution to (1) with the boundary condition (2). Thus $\delta x(t_o) = 0$, so that $u(t_o) = u^*(t_o)$.

6. Go to step 3.

Important details have been left off this description of the algorithm, such as the fact that δx must be limited in size to keep the approximate expansion

$$V(x + \delta x, t) = V(x, t) + V_x(x, t)\, \delta x + \ldots + \frac{1}{N!} V_{x^N}(x, t)\, \delta x^N \tag{39}$$

accurate, i.e., (39) must be valid within some tolerance. (This is described more carefully in references 2, 3, and 4, along with all other relevant details of the algorithm, including the process by which δx is limited.)

Thus it is advantageous to use large values of N because large values of δx will approximately satisfy Equation (39) and because, for a given value of δx, the approximation in step 5

$$u = u^* + u_x \delta x + \ldots + \frac{1}{(N-1)!} u_{x^{N-1}} \delta x^{N-1} \tag{40}$$

will be more exact than it would be for smaller values of N. Since larger values of N allow the use of larger values of δx, convergence to a given accuracy may be possible with fewer iterations.

However, there are serious disadvantages to using large values of N. N versions of Equation (37) must be integrated, and N-1 derivatives $u_x, \ldots, u_{x^{N-1}}$ must be calculated (at each time step). These difficulties are compounded immensely when (as is usually the case) x is a vector. If x has M components, (37) represents M^n equations for each value of n. (There are symmetries, but the number of independent equations is on the order of M^n).

A Sufficiency Proof

The theoretical significance of Equation (37) appears on the optimal trajectory, where $u^* = u = \bar{u}$, and thus $f = \bar{f}$. Then (37) becomes

$$\frac{dV_{x^n}}{dt} + n\,(f_x + u_x f_u)\; V_{x^n} = G_n(t)\;. \tag{41}$$

Also, equations (18) and (20) become

$$\frac{dV_x}{dt} + f_x V_x = -\,L_x \tag{42}$$

$$\frac{dV_{xx}}{dt} + (2f_x + u_x f_u)\; V_{xx} = -\,L_{xx} - f_{xx} V_x$$

$$- u_x\,(L_{ux} + f_{ux}\,V_x). \tag{43}$$

These equations are coupled in one direction. If we knew the optimal trajectory, (42) could be solved (with the boundary condition (38), n = 1). Then Equation (43) would be solved, using (38) for n = 2. Only then could (41) be solved for n = 3 and then for n = 4, etc.

The solution to Equation (41) with boundary condition (38) is

$$V_{x^n}(x(t), t) = \Phi_{x^n}(x(t_f)) \quad e^{n \int_t^{t_f} \alpha(t') dt'} - \int_t^{t_f} G_n(\tau)\, e^{n \int_t^{\tau} \alpha(t') dt'} d\tau \qquad (44)$$

for $n = 3, 4, \ldots$

where $\alpha(t) = f_x(x(t), u(t), t) + u_x(t) f_u(x(t), u(t), t)$ and u_x is given by (25).

Thus a solution to Equation (41) exists, and it is finite for $n \geq 3$ if

a. $\alpha(t) = f_x + u_x f_u$ is defined and continuous on $[t_o, t_f]$, and if

b. $G_n(\tau) = e^{n \int_t^{\tau} \alpha(t') dt'}$ is defined and continuous on $t_o \leq t \leq \tau \leq t_f$.

Conditions a and b are both satisfied in V_{xx}, H_{uu} and sufficient derivatives L_{xu}, f_{xu} and L_{xuu} are defined and continuous on $[t_o, t_f]$.

Since u_{x^k}, is a sum of expressions divided by powers of H_{uu} and since each of these expressions is a polynomial in $V_x, V_{xx}, \ldots, V_{x^{k-1}}$, if V_{xx} and H_{uu}^{-1} are defined and continuous, u_{x^k}, $k = 1, 2, \ldots$ exists. Therefore perturbation feedback laws and neighboring extermals exist.

The values of $u_x, u_{xx}, \ldots$ are known; they are given by Equation (25) and its derivatives. If V_{xx} and H_{uu}^{-1} are assumed finite at every point along the trajectory, then u_x, and $u_{xx}, \ldots$ are all finite at every point.

A paper extending these theoretical results to the vector case has been accepted for publication.[9]

4. USE OF THE FORMAC SYSTEM

The FORMAC [6] batch and scope [7] systems were helpful in arriving at the theoretical results of the previous section. The FORMAC system did not do the proof of the previous section, nor was it able to find the general linear form of Equation (37). Instead, the author used the system to obtain the form of 3rd, 4th, and higher order equations. A pattern was recognized, and it was found that the pattern could be generalized. This kind of mathematical experimentation is discouraging to undertake without benefit of a mechanical device because, "...the manipulations seem quite formidable."[5] FORMAC made an empirical approach to this problem practical.

The following are the derivatives of the Hamilton-Jacobi-Bellman Equation as produced by the FORMAC system.

$$-\frac{\partial}{\partial t} V_x = V_{xx} f + V_x f_x + L_x \tag{45}$$

$$\begin{aligned} -\frac{\partial}{\partial t} V_{xx} &= V_{xxx} f + 2V_{xx} f_x + V_x u_x f_{xu} + V_x f_{xx} \\ &+ V_{xx} u_x f_u + L_{xu} u_x + L_{xx} \end{aligned} \tag{46}$$

$$\begin{aligned} -\frac{\partial}{\partial t} V_{xxx} &= V_{x^4} f + L_{xuu} u_x^2 + 3\ V_{xxx} f_x \\ &+ 4V_{xx} u_x f_{ux} + V_x u_{xx} f_{ux} + V_x u_x^2 f_{xuu} \\ &+ 3V_{xx} f_{xx} + 2V_x u_x f_{xxu} + V_x f_{xxx} \\ &+ 2V_{xxx} u_x f_u + V_{xx} u_{xx} f_u + V_{xx} u_x^2 f_{uu} \\ &+ 2L_{xxu} u_x + L_{xu} u_{xx} + L_{xxx} \end{aligned} \tag{47}$$

$$- \frac{\partial}{\partial t} V_{x^4} = \tag{48}$$

$$V_{x^5} f + 3L_{x^2u^2} u_x^2 + L_{xu^3} u_x^3 + 4V_{x^4} f_x$$

$$+ 9\, V_{x^3} u_x f_{xu} + 6\, V_{xx} u_{xx} f_{xu} + V_x u_{x^3} f_{xu}$$

$$+ 3V_x u_{xx} u_x f_{xu^2} + 6V_{xx} u_x^2 f_{xuu}$$

$$+ 6V_{x^3} f_{xx} + 9V_{xx} u_x f_{xxu}$$

$$+ 3\, V_x u_{xx} f_{x^2u} + 3\, V_x u_x^2 f_{x^2u^2} + 4\, V_{xx} f_{x^3}$$

$$+ 3\, V_x u_x f_{x^3u} + V_x f_{x^4} + 3\, V_{x^4} u_x f_u + 3\, V_{x^3} u_{xx} f_u$$

$$+ V_{xx} u_{x^3} f_u + 3\, V_{x^3} u_x^2 f_{uu} + 3\, V_{xx} u_{xx} u_x f_{uu}$$

$$+ V_{x^2} u_x^2 f_{u^3} + 3\, L_{xu^2} u_{xx} u_x + 3\, L_{x^3u} u_x + f_{xu3} V_x u_x^3$$

$$+ 3\, L_{xxu} u_{xx} + L_{xu} V_{x^3} + L_{x^4}$$

In addition to equations (45) through (48), the fifth, sixth, and seventh derivatives of Equation (7) and the first seven derivatives of Equation (19) were also produced. Space limitations preclude displaying those expressions here.

5. SUMMARY

The FORMAC system is used to obtain new results in a problem in optimal control theory.

The differentiation of Equation (27) was first performed using the FORMAC system to obtain the third and fourth order equations. It was perceived that nonlinear terms did not arise, and this fact was generalized. Clearly this is an important application of a symbol-manipulation system: supplying specific examples of phenomena for the mathematician to generalize.

There are other problems in optimal control theory to which a symbol manipulation system may be applied. One is the discrete time problem in (9), (10), and (11), for which results like those in this paper may be obtainable. Another is the symbolic solution of the Euler-Lagrange equations (4), (5), and (6). An attack was made on this problem (see these Proceedings, reference 10), but the approach appears to be impractical.

REFERENCES

1. A. E. Bryson, Jr. and Y. C. Ho, Optimization, Estimation, and Control, Blaisdell, 1969, to be published.

2. D. H. Jacobson, "Differential Dynamic Programming Methods for Determining Optimal Control of Non-Linear Systems," Ph.D. Thesis, University of London, October 1967.

3. D. H. Jacobson, "New Second-Order and First-Order Algorithms for Determining Optimal Control: A Differential Dynamic Programming Approach," Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts, February 1968.

4. S. B. Gershwin and D. H. Jacobson, "A Discrete Time Differential Dynamic Programming Algorithm with Application to Optimal Orbit Transfer," Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts, August 1968.

5. S. R. McReynolds, "Higher Order Optimal Feedback Schemes," unpublished memorandum, September 1964.

6. R. Tobey, et al., "PL/I - FORMAC Interpreter, User's Reference Manual," IBM Contributed Program Library 360D 03.3.004, Hawthorne, N.Y., October 1967.

*7. R. G. Tobey and J. D. Lipson, "The Scope FORMAC Language," these Proceedings.

8. E. L. Ince, Ordinary Differential Equations, Dover, 1956.

9. S. B. Gershwin, "On the Higher Derivatives of Bellman's Equation," Journal of Mathematical Analysis and Applications (to appear).

*10. S. B. Gershwin, "An Attempt to Solve Differential Equations Symbolically," these Proceedings.

*These articles appear in these Proceedings.

DESIGN AND ANALYSIS OF MATHEMATICAL ALGORITHMS

PRECEDING PAGE BLANK NOT FILMED

SYMBOLIC INTEGRATION OF ELEMENTARY FUNCTIONS

by

Robert H. Risch
IBM Research Center
Yorktown Heights, New York

Abstract

The problem implied by the phrase "symbolic integration of elementary functions" is difficult to formulate. This paper presents the issues that arise in attempting such a formulation, the formulation proposed by the author, and the work he has done toward its solution. Actual examples are worked, employing an algorithm for the integration of functions built up using logarithms, exponentials, and rational operations.

The paper was prepared while the author was employed by the System Development Corporation, Santa Monica, California.

SYMBOLIC INTEGRATION OF ELEMENTARY FUNCTIONS

by

Robert H. Risch

1. INTRODUCTION

In elementary calculus courses students are asked to find the indefinite integrals of "elementary functions" like $\int \frac{dx}{x+1}$ or $\int \sin^2 x\, dx$. The teacher usually states without proof that there is no elementary expression in closed form for integrals like $\int \frac{\sin x}{x} dx$ or $\int \frac{dx}{(1-x^2)\,(1-k^2x^2)}$. Elementary is usually defined as those functions that can be built upon from the rationals and an independent variable using only exponential, logarithmic, trigonometric, and inverse trigonometric operations.

Joseph Liouville[2, 3] did the first significant work on the problem of integration in finite terms during the years 1833-1841. (See also references 4, 5, and 7 - 11 for work on integration.) Among other things, he showed that when the integral of an elementary function is elementary it must be of a certain definite form, and that the two integrals above are not integrable in finite terms.

During the remainder of the nineteenth century, French and Russian mathematicians did some work on the problem of determining when an algebraic function has an elementary antiderivative.

In 1913 the Russian mathematician D. D. Mordoukhay-Boltovskoy[4] discussed integrals of functions built up using exponentials, logarithms, and the four rational operations. Although he claimed that a general method exists for determining whether such a function has an elementary indefinite integral and finding the integral if it does, he made only a vague attempt to explicitly give such an algorithm.

Since the 1931 work of Gödel, mathematicians have realized that integration in finite terms is a decision problem, and that it should be examined in order to see whether it is decidable or undecidable. According to this modern point of view, one must define elementary function precisely, and one must explain the phrase "given an elementary function." The way of function is usually given is a symbol on a piece of paper, but the function that the symbol represents must be uniquely specified.

This paper presents the issues that arise when one tries to precisely formulate this problem. The formulation and the work done by this author toward its solution are discussed. Examples are worked to illustrate the algorithm developed for integrating elementary functions that are built up using exponentials, logarithms, and the rational operations, provided that exponentials and logarithms cannot be replaced by adjoining constants and performing algebraic operations. The techniques a computer will have to invoke for the algorithm to be implemented are mentioned.

2. THE PROBLEM OF FORMULATION

In order to precisely state the decision problem, a countable set of symbols to represent our functions must be given, and the functions corresponding to the individual symbols should be specified. Care must be exercised here. For example, our elementary functions might contain, as a subclass, a set of numbers S for which the problem of telling whether one is identically zero is undecidable (the computable "reals" is such a class). Then the integration problem for the set of functions $\{ae^{x^2} : a \epsilon S\}$ is undecidable since $\int ae^{x^2} dx$ is elementary iff $a = 0$.

In an unpublished thesis, [6] Daniel Richardson constructed a class of real functions, which he calls elementary, in which each function is either identically 0 on the real line or identically 1 on some interval. Given such a function, whether the function is of the first or second type is undecidable. $\{fe^{x^2} : f \epsilon$ Richardson's class$\}$ gives a solution for the integration problem similar to that in the preceding paragraph. The result depends heavily on the fact that $|A|$ (or $\log |A|$) is allowed as an elementary operation. With the absolute value function, the minimum of two functions can be defined and, thus, distinct analytic functions can be spliced together.

These artificial examples do not give any real insight into the question of why a function that is known not to be the zero function has or does not have an elementary indefinite integral, and they seem to have little connection with the classical problem considered by Liouville.

The formulation given here is motivated by the three considerations that follow.

a. One must be able to determine when a function of the class being considered is equal to zero (or equivalently when two are equal).

   It is important to know how to do this so one does not try to integrate a symbol that does not represent a function such as $\log \left(\frac{1}{e^{\log x}-x}\right)$. One must avoid considering expressions for constants like $e^{\sin 1/2-e^{e}} -\log(3-e^{6})$ since there is no known method for determining when two of these constants are equal. (See reference 1 for an account of problems of this type.)

b. It is easier to deal with complex rather than real functions.

   If $i = \sqrt{-1}$ is allowed as a constant, the only transcendental operations needed are exp and log because sin, $\tan^{-1}$, etc., can be written in terms of those two; e.g., $\tan^{-1} z = \frac{1}{2i} \log \left(\frac{i-z}{i+z}\right)$. If one starts with a given function which has an expression involving the trigonometric functions and real constants, and then converts to complex form and finds an elementary indefinite integral in complex form, one can convert back to real expression involving the functions one started with. (See reference 8 for details.)

c. An unambiguous situation is necessary when dealing with the branches of multiple-valued functions.

   For example, $\int e^{z^2} (\log e^{z} - z)\, dz$ is elementary iff the branch of the logarithm chosen is $\log e^{z} = z$. Generally, for an arbitrary symbol representing an elementary function, it is an undecidable problem to tell whether there is a choice of the logarithms involved that makes the integral elementary. (See Prop. 2.2 of the revised version of reference 7.) Here this situation will be avoided by choosing the symbols so that no matter what branch of the logarithms or algebraic functions involved is chosen, the question of whether or not the integral is elementary depends only on the symbol, as in $\int \log\sqrt{x}\, dx$.

Let K be a field of constants over which one can perform algebraic operations including the factoring of polynomials. Let us consider fields of elementary functions of the form $K(z, \theta_1, \ldots, \theta_n)$ where z is the identity function and each θ_i is either algebraic over $K(z, \theta_1, \ldots, \theta_{i-1})$ or an exponential or logarithm of an element of $K(z, \theta_1, \ldots, \theta_{i-1})$. If θ_i is an exponential or logarithm, it is transcendental over any field that can be obtained by adding constants to $K(z, \theta_1, \ldots, \theta_{i-1})$. Then θ_i is called a monomial over $K(z, \theta_1, \ldots, \theta_{i-1})$, and $K(z, \theta_1, \ldots, \theta_n)$ is regular elementary over $K(z)$. For example, in $Q(z, e^z, \log e^z)$ where Q = rational numbers, e^z is a monomial over $Q(z)$. However $\log e^z$ is not a monomial over $Q(z, e^z)$ since it is in $Q(z, 2\pi i)$.

Similarly, $e^{\frac{z+1}{2}}$ is not a monomial over $Q(z, e^z)$ since it is the square root of an element of $Q(z, e^z, e)$.

The sequence of symbols $\epsilon = \langle 1_\epsilon, z_\epsilon, \exp_\epsilon z_\epsilon \rangle$ represents the generators of the field $Q(z, e^z)$. The subscript ϵ emphasizes that we are dealing with symbols rather than the functions they represent. The terms built up from these symbols using the four binary operation symbols $+_\epsilon, \cdot_\epsilon, -_\epsilon, \div_\epsilon$ serve to designate each element of the field. ϵ is called an elementary field description (efd). $Q(z, e^z)$ is called a model of ϵ. If an efd has a model which is regular elementary over $K(z)$, it is called regular. The efd given above is regular, but $\langle 1_\epsilon, z_\epsilon, \exp_\epsilon z, \log_\epsilon \exp_\epsilon z_\epsilon \rangle$ is not regular.

Regular efd's have some properties which make them particularly valuable in finding solutions to the types of problems discussed in this paper. Since the models of a regular efd ϵ are isomorphic, problems with branches of the logarithm like those described previously do not occur. Since bases for the field determined by the efd can be constructed, one can determine when two elements of the field are equal.

Any elementary function lies in a model of some regular efd. (The treatment of a nonregular efd is discussed in reference 12.) The problem of integration in finite terms involves:

a. Deciding for a given efd whether it is regular.

b. Deciding whether a given symbol, built up from a regular efd, represents a function with an elementary antiderivative, and finding the antiderivative if it does.

3. THE ALGORITHM

In reference 7 the author gave an algorithm for completing the two steps above for efd's corresponding to fields $K(z, \theta_1, \ldots, \theta_n)$ where each θ_i is an exponential or logarithm of an element of $K(z, \theta_1, \ldots, \theta_{i-1})$. In reference 9 the author reduced the case in which θ_i may be algebraic over $K(z, \theta_1, \ldots, \theta_{i-1})$ to a problem in algebraic function theory.

The algorithm in reference 7 extends the well known partial fraction algorithm of a rational function (i.e., $K(z)$) integration to functions in pure monomial extensions of $K(z)$. The starting point of this extension is the classical theorem of Liouville which tells us (in our version) that for a differential field $\mathcal{F}$ (i.e., a field closed under differentiation, like those discussed above) with an algebraically closed constant field L, if an $f \in \mathcal{F}$ has an elementary indefinite integral, then $\int f = v_o + \sum_j c_j \log v_j$ where v_o, v_j are in $\mathcal{F}$, and the c_j are in L.

To apply the theorem to functions f in pure monomial extensions $\mathcal{F} = K(z, \theta_1, \ldots, \theta_n)$, one sets $\mathcal{F} = \mathcal{D}(\theta)$ where $\theta = \theta_n$. The monomial θ is treated as an indeterminate over $\mathcal{D}$, and a partial fraction decomposition of f is made (which is a rational function of θ with coefficients in $\mathcal{D}$) over $\mathcal{D}$.

Using Liouville's theorem to find the schemata for the functions involved in the integral, one gets something like

$$f = \left\{\begin{matrix} A_k\theta^k + \dots + A_o \\ + \dfrac{A_{1,k_1}}{p_1^{k_1}} + \dots + \dfrac{A_{1,1}}{p_1} \\ \cdot \\ \cdot \\ \cdot \\ + \dfrac{A_{s,k_s}}{p_s^{k_s}} + \dots + \dfrac{A_{s,1}}{p_s} \end{matrix}\right\} = \left\{\begin{matrix} B_{k+1}\theta^{k+1} + \dots + B_o + \Sigma_j d_j \log D_j \\ + \dfrac{B_{1,k_1-1}}{p_1^{k_1-1}} + \dots + \dfrac{B_{1,1}}{p_1} + \displaystyle\int \dfrac{B_{1,0}}{p_1} \\ \cdot \\ \cdot \\ \cdot \\ + \dfrac{B_{s,k_s-1}}{p_s^{k_s-1}} + \dots + \dfrac{B_{s,1}}{p_s} + \displaystyle\int \dfrac{B_{s,0}}{p_s} \end{matrix}\right\}$$

where the p_i's are monic irreducible polynomials in θ, the A_j and B_j are in $\mathcal{D}$, and the $A_{i,j}$ and $B_{i,j}$ are in $\mathcal{D}[\theta]$. The above schema corresponds to the case $\theta = \log f$. (See pages 33, 38, and 39 of reference 7 for further details.)

The right-hand side of the schema shown above is differentiated. By equating corresponding parts we obtain the conditions on the B's required for the existence of an elementary antiderivative for f. These conditions immediately lead to the problems of:

a. Telling whether an element of $\mathcal{D}$ has an elementary indefinite integral

b. Telling whether a first-order linear differential equation with coefficients in $\mathcal{D}$ has a solution in $\mathcal{D}$ ($\mathcal{D} = \overline{K}(z, \theta_1, \dots, \theta_{n-1})$.

These problems are reduced to similar problems over $\overline{K}(z, \theta_1, \dots, \theta_{n-2})$, then they finally reduced to the problem of determining whether a set of linear equations with coefficients in K has a solution in K.

4. COMPUTATIONAL ASPECTS

In this section we illustrate the algorithm, but first we shall mention a useful theorem that will appear in a forthcoming paper[12] which enables one to easily solve the problem of determining whether a given efd is regular. This theorem provides a monomial test. Let $\mathcal{D} = K(z, \theta_1, \ldots, \theta_n)$ be a pure monomial extension of $K(z)$. Let $\log f_i$, $i=1,\ldots,r$, and $\exp g_i$, $i=1,\ldots,s$, (where $r+s=n$) be respectively the logarithmic and exponential monomials among $\theta_1, \ldots, \theta_n$. For f, g in $\mathcal{D}$, $\log f$, and $\exp g$ are not monomials over $\mathcal{D}$ iff:

a. $f = c \prod_{i=1}^{r} f_i^{k_i} \prod_{i=1}^{s} (\exp g_i)^{m_i}$ where c is a constant and k_i and m_i are rationals.

b. $g = d + \sum_{i=1}^{r} p_i \log f_i + \sum_{i=1}^{s} q_i g_i$ where d is a constant and p_i and q_i are rationals.

It is apparent from this theorem that $\log z$ is a monomial over $K(z)$. Since there are no f_i or g_i here, if $\log z$ were not a monomial z would equal a constant, which is a contradiction. Likewise, e^{z^2} is clearly a monomial over $K(z, \log z)$. Thus the efd $<1_\epsilon, z_\epsilon, \log_\epsilon z_\epsilon, \exp_\epsilon z_\epsilon \cdot_\epsilon z_\epsilon>$ is regular. (The same is true for $<1_\epsilon, z_\epsilon, \exp_\epsilon z_\epsilon \cdot_\epsilon z_\epsilon, \log_\epsilon z>$.)

One can consider integrating functions represented by symbols that are built up from it. In order to integrate elements of $\mathcal{D} = K(z, \theta_1, \ldots, \theta_n)$, one must be able to perform the four rational operations on elements of $\mathcal{D}$. One must also be able to find the partial fraction decomposition of elements of $\mathcal{D}$ over $\overline{K}(z, \theta_1, \ldots, \theta_{n-1})$ where $\overline{K}$ is the algebraic closure of K. This implies that one must be able to factor the polynomials in θ_n over $\overline{K}(z, \theta_1, \ldots, \theta_{n-1})$. For example, in $\mathcal{D} = Q(z, \log z)$,

$$\frac{4z^2}{\log z - 2z^2} = \frac{\sqrt{2}z}{\log z - \sqrt{2}z} + \frac{\sqrt{2}z}{\log z + \sqrt{2}z}$$

(See page 31 of reference 7 for a discussion of how to do this.) It is generally extremely difficult to carry out this factoring in practice, even on a computer.

Let us look at some functions in $Q(z, e^{z^2}, \log z)$ (the field corresponding to the efd $<1_\epsilon, z_\epsilon, \exp_\epsilon z_\epsilon \cdot_\epsilon z_\epsilon, \log_\epsilon z_\epsilon>$). Let us examine

$$\int \left[2ze^{z^2} \log z + \frac{e^{z^2}}{z} + \frac{\log z - 2}{[(\log z)^2 + z]^2} + \frac{\frac{2}{z} \log z + \frac{1}{z} + 1}{(\log z)^2 + z} \right] dz$$

One must be able to set up a schema which indicates the form of the integral. Here one knows that if the integral is elementary it is of the form

$$B_2(\log z)^2 + B_1 \log z + B_0 + \Sigma d_i \log D_i + \frac{B_{1,1}}{(\log z)^2 + z} + c_1 \log [(\log z)^2 + z]$$

where B_0, B_1, B_2 are in $Q(z, e^{z^2})$, $B_{1,1}$ is in $Q(z, e^{z^2})$ [log z], and c_1 and d_i are constants.

Differentiate and get

$$B_2' (\log z)^2 + \left(\frac{2}{z} B_2 + B'_1\right) \log z + \frac{1}{z} B_1 + B_0' + \Sigma d_i \frac{D_i'}{D_i}$$

$$+ \frac{B_{1,1}\left(\frac{2}{z} \log z + 1\right)}{[(\log z)^2 + z]^2} + \frac{B_{1,1}'}{(\log z)^2 + z} + c_1 \frac{\frac{2}{z} \log z + 1}{(\log z)^2 + z}$$

Now equating the corresponding parts of the two expressions for the integrand one gets

$$B_2' = 0 \quad \text{so} \quad B_2 \text{ is a constant and } \frac{2}{z} B_2 + B_1' = 2ze^{z^2}$$

Thus, $B_1 = \int 2\, ze^{z^2}\, dz - 2B_2 \log z$.

Here one has to integrate an element of $\mathcal{D} = Q(z, e^{z^2})$. We assume that this can be done using induction. Thus $B_1 = e^{z^2} + b_1 - 2B_2 \log z$ where b_1 is a constant. $B_2 = 0$ since B_1 must be in $Q(z, e^{z^2})$.

Thus,

$$B_1 = e^{z^2} + b_1 . \quad \frac{e^{z^2}}{z} + \frac{b_1}{z} + B_0' + (\Sigma c_i \log D_i)' = \frac{e^{z^2}}{z}$$

$$B_0 + \Sigma d_i \log D_i + b_1 \log z = \int 0 = \text{constant},$$

so b_1 can be taken to be 0.

$$-B_{1,1}\left(\frac{2}{z} \log z + 1\right) \equiv \log z - 2 \mod ((\log z)^2 + z)$$

By the Euclidean algorithm the equation $A\left[\frac{2}{z} \log z + 1\right] + B\ [(\log z)^2 + z]$ $= \log z - 2$ can be solved for A and B in $Q[z, \log z]$, degree $A<2$, and degree $B = 0$. Thus, one must be able to find gcd's of polynomials in several variables (the monomials are the variables here).

In this case, $A = \log z$, $B = -\frac{2}{z}$. Thus $B_{1,1} = -\log z$.

Substituting this into $\frac{\log z - 2}{[(\log z)^2 + z]^2} + \frac{\frac{2}{z}\log z + \frac{1}{z} + 1}{(\log z)^2 + z} = \left[\frac{B_{1,1}}{(\log z)^2 + z}\right.$ $\left. + c_1 \log [(\log z)^2 + z]\right]'$, after canceling one obtains $c_1 \left(\frac{2}{z} \log z + 1\right) = \frac{2}{z} \log z + 1$. Therefore, $c_1 = 1$. These calculations show that the integral is $e^{z^2} \log z$

$$- \frac{\log z}{(\log z)^2 + z} + \log [(\log z)^2 + z].$$

If, instead of $<1_\epsilon, z_\epsilon, \exp z_\epsilon \cdot {}_\epsilon z_\epsilon, \log_\epsilon z_\epsilon>$, $<1_\epsilon, z_\epsilon, \log_\epsilon z_\epsilon, \exp_\epsilon z_\epsilon \cdot {}_\epsilon z_\epsilon>$, was chosen as the efd, the integrand would be written as

$$e^{z^2} \left(2z \log z + \frac{1}{z}\right) + \frac{\log z - 2}{[(\log z)^2 + z]^2} + \frac{\frac{2}{z} \log z + \frac{1}{z} + 1}{(\log z)^2 + z}$$

Here the schema for the solution runs $B_1 e^{z^2} + B_0 + \Sigma d_i \log D_i$

where B_0, B_1, D_i are in $Q(z, \log z)$ and the d_i are constants. For B_1 one obtains $B_1 + 2z\,B_1 = 2z \log z + \frac{1}{z}$. One is to determine if this differential equation has a solution $Q(z, \log z)$. By a discussion similar to that in the next example (or by inspection) we see that $B_1 = \log z$. The rest of this problem is worked in a manner similar to the preceding.

It is not clear whether a permutation of the monomials in an efd can lead to a simpler computation. In certain cases the choice of an efd is important. For example, it would be better to obtain $e^{z^2/2}$ as an element of the field $Q(z, e^{z^2/2})$ than the field $Q(z, e^{z^2}, \sqrt{e^{z^2}})$.

We next examine, in the field $Q(z, \log z, e^{z^2})$,

$$\int e^{z^2}\left[\left(\frac{-1}{(z+1)^2} - \frac{2}{z+1} + 2\right)\log z - \frac{1}{z+1} + \frac{1}{z} - 2z^3 + 3z^2 + 2\right]dz$$

This integral must be for the form Ae^{z^2} where $A_{\epsilon Q(z, \log z)}$

$$(*) \qquad A' + 2zA = \left[\frac{-1}{(z+1)^2} - \frac{2}{z+1} + 2\right]\log z - \frac{1}{z+1} + \frac{1}{z} - 2z^3 + 3z^2 + 2$$

A must be in $Q(z)\,[\log z]$ since any denominator would remain after differentiation. Thus,

$$A = A_k \log^k z + A_{k-1} \log^{k-1} z + \dots$$

$$A' = A_k' \log^k z + (A_{k-1}' + k\frac{A_k}{z}) \log^{k-1} z + \dots$$

In equation (*) the terms of highest degree in log z must cancel, so one gets $k = 1$ or $k > 1$ and $A_k' + 2z\,A_k = 0$. From the latter one gets $A = c\,e^{-z^2}$ which is impossible since e^{z^2} is a monomial over $Q(z)$.

Thus, $k = 1$

$$A_1' + 2zA_1 = - \frac{1}{(z-1)^2} - \frac{2}{z+1} + 2$$

$$A_1' + [2(z+1)-2]\ A_1 = - \frac{1}{(z+1)^2} - \frac{2}{z+1} + 2$$

one obtains $A_1 = \frac{a}{z+1}$ (see page 36 of reference 8).

$$A_1' = - \frac{a}{(z+1)^2} - \frac{2a}{z+1} + 2a = \frac{-1}{(z+1)^2} - \frac{2}{z+1} + 2$$

$$- a -(z+1)\ 2a + (z+1)^2\ 2a = -1 - 2(z+1) + 2(z+1)^2 .$$

Thus, $a = 1$ and $A_1 = \frac{1}{z+1}$

$$A_0' + \frac{A_1}{z} + 2zA_0 = - \frac{1}{z+1} + \frac{1}{z} - 2z^3 + 3z^2 + 2$$

$$A_0' + 2zA_0 = - 2z^3 + 3z^2 + 2$$

Let $A_0 = b_k z^k + \ldots$

$$A_0' = kb_k\ z^{k-1} + \ldots$$

$$k + 1 = 3,$$

$$k = 2.$$

$$2b_2 z + b_1 + 2z^3\ b_2 + 2z^2\ b_1 + 2zb_0 = - 2z^3 + 3z^2 + 2.$$

It is necessary to be able to solve systems of linear equations with constant coefficients.

$$\left.\begin{aligned} 2b_2 &= -2 \\ 2b_1 &= 3 \\ 2b_2 + 2b_0 &= 0 \\ b_1 &= 2 \end{aligned}\right\}$$

This system cannot be solved, so the integral is not elementary.

5. SUMMARY

The formulation of the classical problem of integration in finite terms was discussed. The concept of a monomial was introduced, and it was indicated how one goes about integrating elements of pure monomial extensions $K(z, \theta_1, \ldots, \theta_n)$ of $K(z)$, where K is a field of constants and z an indeterminate. The algorithm requires one to have a facility for performing operations on the elements of a field of rational functions of several variables $\mathfrak{D} = K(x_1, \ldots, x_m)$, viz.,

a. Performing the four rational operations on $\mathfrak{D}$

b. Factoring elements of $K[x_1, \ldots, x_m]$ over $\overline{K}$ where $\overline{K}$ is the algebraic closure of K

c. Finding the partial fraction decomposition of elements of $\mathfrak{D}$ over $\overline{K}(x_1, \ldots, x_m)$

d. Computing gcd's of pairs of elements of $K[x_1, \ldots, x_m]$

e. Solving simultaneous sets of linear equations with coefficients in K.

REFERENCES

1. S. Lang, *Introduction to Transcendental Numbers*, Addison-Wesley, 1966.
2. J. Liouville, "Sur la determination des integrales dont la valeur est algebrique," *Paris Ecole Polytechnique Journal*, 14, 1833, pp. 124-193.
3. J. Liouville, "Memoire sir les transcendantes elliptiques de premiere et de seconde espece, considerees comme fonctiones de leur amplitude, "*Paris Ecole Polytechnique Journal*, 14, 1833, pp. 57-83.
4. D.D. Mordoukhay-Boltovskoy, "On the Integration of transcendental functions," Warsaw Universitet, Izvietiia, nos. 6-9, 1913 (Russian).
5. A. Ostrowski, "Sur l'integrabilite elementare de quelques classes d'expressions," *Commentarii Mathematique Helvetici*, 28, 1946, pp. 283-308.
6. D. Richardson, "Some Unsolvable Problems Involving Functions of a Real Variable," Doctoral dissertation, University of Bristol, England, 1966.
7. R. H. Risch, "The problem of integration in finite terms," SDC document SP-2801/000/00, 23 March 1967 (To appear in revised form in the Transactions of the American Mathematical Society in May 1969).
8. R.H. Risch, "On real elementary functions," SDC document SP/2801/001/00, 22 May 1967.
9. R. H. Risch, "On the integration of elementary functions which are built up using algebraic operations," SDC document SP-2801/002/00, 22 June 1968.
10. J.F. Ritt, *Integration in finite terms, Liouville's theory of elementary methods*, Columbia University Press, 1948.
11. M. Rosenlicht, "Liouville's theorem on functions with elementary integrals," *Pacific Journal of Mathematics*, 24, 1968, pp. 153-161.
12. R.H. Risch, "Further Results on Elementary Functions" (to appear).

ASYMPTOTICS FOR FORMULA MANIPULATION

by

John H. Halton
Computer Sciences Department
The University of Wisconsin
Madison, Wisconsin

Abstract

This paper presents some proposals for the implementation, in a computer formula-manipulation system, of a subprogram for computing and manipulating asymptotic expansions. The first part (sections 1 to 4) reviews the relevant mathematical theory. The second part (sections 5 to 6) discusses implementation of the computations involved in an asympotic package given a limit-subprogram. Notions of commensurate functions (section 3) and of natural asymptotic expansions (section 5) are proposed. Algorithms are given for forming linear combinations, products, and arbitrary real powers of given asymptotic expansions (the explicit expression for a real power of an asymptotic series had to be derived: this apparently new result is given in Lemma 15). Possible extensions and difficulties are considered throughout the discussion.

PRECEDING PAGE BLANK NOT FILMED

ASYMPTOTICS FOR FORMULA MANIPULATION

by

John H. Halton

1. INTRODUCTION

This paper presents some proposals for implementing, in a computer formula manipulation system, a subprogram for computing and manipulating asymptotic expansions. These proposals are viewed as merely a beginning; sufficient (it would seem) to immediately construct only the most primitive and limited kind of asymptotic package in an existing formula-manipulation system (such as PL/I-FORMAC); but containing, despite many unresolved questions and difficulties, the seeds of a much more powerful and broad-ranging facility. It is in the hope of stimulating further research and development of the ideas adumbrated here that this paper is published.

Sections 2 to 4 review the relevant (more elementary) parts of the theory of asymptotic series. Section 5 deals with the detailed implementation of theoretical results in a series of programmed algorithms (expressed in a suitable, ad hoc, simple programming language).

Though the concepts described below can be extended to more general situations, we limited ourselves to real-valued functions defined on either finite or infinite intervals, or countable unions of intervals, on the real line. Such functions include most of those upon which computations are performed.

Frequently a function is defined in an implicit manner, and it cannot be given in closed form in terms of simple functions. For example, the function may be defined by an integral or a differential equation. It may be, too, that the behavior of the function is not readily perceived from an examination of a table of values, and that a convergent power series is not available (or would require the evaluation of an excessive number of terms) in the region under consideration, as is the case when this region contains a singularity of the function. Even when these difficulties are absent, it still may be more laborious or complicated than we find acceptable to examine the behavior of the function in these explicit terms. Such situations occur very frequently in pure mathematics (analysis and theory of differential and integral equations), numerical analysis, all branches of theoretical physics, statistics, and engineering—indeed whenever what is broadly termed "advanced calculus" is used.

In such cases, we are often interested in finding a relatively simple and well-understood "easy" function φ, whose behavior, in the neighborhood of a certain point α, is very similar to the behavior of the given "difficult" function f. (The point α must be a limit-point of the domains of definition of φ and of f, and we include $\alpha = \pm\infty$ if the domains of φ and f are unbounded.) When the relation between $f(x)$ and $\varphi(x)$, for x in some neighborhood of α, is one of approximate equality, we say that φ is an *approximation* to f near α. When the relation is defined in the manner which we are about to discuss, we say that φ is an *asymptotic approximation* or an *asymptotic form*, for f near α, or just that φ is *asymptotic* to f as $x \to \alpha$.

The study of asymptotic relations is a major branch of mathematical analysis (see references 1, 2, 3 and 4); but here we consider only the elementary aspects of the subject which are relevant to the construction of an asymptotic package to be included in a formula-manipulation system. Both theorems and algorithms will be given for the most straightforward operations.

2. THE ORDER-SYMBOLS

Let $\mathfrak{R}$ denote the real line, with its usual (order) topology and (Lebesgue) measure, induced by the ordering $x < y$ and the distance $|x - y|$ between the points corresponding to real numbers x and y. Let the equation

$$\mathfrak{D} = \mathfrak{D}(f, \varphi) \qquad (1)$$

indicate that the set $\mathfrak{D} \subseteq \mathfrak{R}$ is the intersection of the domains of definition of the functions f and φ, which take their values in $\mathfrak{R}$. (We shall later consider larger collections of functions, and we shall always let $\mathfrak{D}$ denote the intersection of all their domains, this being indicated by an equation such as $\mathfrak{D} = \mathfrak{D}(f, \varphi_1, \varphi_2, \cdots, \varphi_k)$ only when necessary for clarity.) Let $\bar{\mathfrak{R}}$ denote the extended real line ($\mathfrak{R}$ compactified by adjoining to it the two points $\pm \infty$), let $\bar{\mathfrak{D}}$ be the closure of $\mathfrak{D}$ in $\bar{\mathfrak{R}}$, and let $\alpha \in \bar{\mathfrak{D}}$. If we can find a neighborhood $\mathfrak{N}_\alpha$ of α in $\mathfrak{D}$ and a constant $A > 0$, such that

$$|f(x)| \leq A\,|\varphi(x)| \quad \text{for all } x \in \mathfrak{N}_\alpha, \tag{2}$$

then we shall write

$$f = \underline{O}(\varphi)_\alpha , \tag{3}$$

the subscript α being omitted when it is well understood. An equivalent notation, also often seen, is

$$f(x) = \underline{O}[\varphi(x)] \text{ as } x \to \alpha . \tag{4}$$

In reference 2 de Bruijn points out that this notation is easily liable to abuse. The = sign is used very unconventionally, rather as if one wrote $x = L(y)$ instead of $x < y$. In both cases, one must either treat $= \underline{O}$ (or $= L$) as a single symbol or interpret $\underline{O}(\varphi)_\alpha$ as denoting a class of functions—all f satisfying (2) for some $\mathfrak{N}_\alpha$ and A (and similarly L(y) as a class of real numbers (all $x < y$, for the given Y)). In the latter interpretation, it would be more correct to write $f \in \underline{O}(\varphi)_\alpha$ or $x \in L(y)$. The classical notation can be made fairly clear, if one interprets $O(\varphi)_\alpha$ as denoting a generic member of the class of functions which satisfy (2) for some $\mathfrak{N}_\alpha$ and A, with the understanding that $\underline{O}(\varphi)_\alpha$ does not necessarily denote the same member of the class on different occurrences of the symbol, even in the same equation.

If we define the function ρ on $\mathfrak{D}$ to $\overline{\mathfrak{R}}$ by

$$\rho\,(x) = \left\{ \begin{array}{ll} |\,f(x)/\varphi(x)\,| & \text{if } \varphi(x) \neq 0 \\ 0 & \text{if } f(x) = \varphi(x) = 0 \\ \infty & \text{if } f(x) \neq 0 \text{ and } \varphi(x) = 0 \end{array} \right\}, \tag{5}$$

then we see that (3) or (4) holds when $\rho(x)$ is bounded (away from ± ∞) on some neighborhood $\mathfrak{N}_\alpha$ of α. In particular, this will be the case if $f(x)/\varphi(x)$ tends to a finite limit as $x \to \alpha$.

An equivalent notation to (3) is

$$\varphi = \Omega(f)_\alpha\,. \tag{6}$$

Here however, we are saying that φ is a member of a class of functions which satisfy (2) for some $\mathfrak{N}_\alpha$ and A, for the given function f and point α. If we define σ on $\mathfrak{D}$ to $\overline{\mathfrak{R}}$ (essentially as $^1/_\rho$) by

$$\sigma(x) = \left\{ \begin{array}{ll} |\,\varphi(x)/f(x)\,| & \text{if } f(x) \neq 0 \\ \infty & \text{if } \varphi(x) = f(x) = 0 \\ \infty & \text{if } \varphi(x) \neq 0 \text{ and } f(x) = 0 \end{array} \right\} \tag{7}$$

then we have (6) whenever $\sigma(x)$ is bounded away from 0 on some neighborhood $\mathfrak{N}_\alpha$ of α. In particular, this will be the case if $\varphi(x)/f(x)$ tends to a nonzero (possibly infinite) limit as $x \to \alpha$.

Similarly, if for the given f, φ, and $\alpha \in \overline{\mathfrak{D}}$,

$$\rho(x) \to 0 \quad \text{as} \quad x \to \alpha \quad \text{in} \quad \mathfrak{D}, \tag{8}$$

we write

$$f = \underline{o}\,(\varphi)_\alpha, \tag{9}$$

omitting the α whenever safe. Equivalently, we have

$$\sigma(x) \to \infty \quad \text{as} \quad x \to \alpha \quad \text{in} \quad \mathfrak{D}, \tag{10}$$

and write

$$\varphi = \omega(f)_\alpha \ . \tag{11}$$

A number of simple lemmas can now be obtained without much difficulty. Since they often apply similarly to all four symbols $\underline{O}$, Ω, $\underline{o}$, and ω, we can abbreviate them by using Υ to denote any of these symbols.

Lemma 1

If $p' > 0$, then

$$f = \Upsilon(\varphi)_\alpha \Longrightarrow |f|^p = \Upsilon(|\varphi|^p)_\alpha \ . \tag{12}$$

Lemma 2

If $c_1, c_2, \cdots c_k$ are real constants and we have

$$f_i = \Upsilon(\varphi_i)_\alpha \quad \text{for } i = 1, 2, \cdots, k \tag{13}$$

with Υ denoting the same symbol for all i, then

$$\Sigma_{i=1}^k |c_i f_i| = \Upsilon(\Sigma_{i=1}^k |c_i \varphi_i|)_\alpha \ . \tag{14}$$

The modulus bars can be removed from the left of (14) (but not the right) if Υ is $\underline{O}$ or $\underline{o}$, and from the right (but not the left) if Υ is Ω or ω.

The relation

$$f(x,y) = \Upsilon[\varphi(x,y)] \quad \text{as } x \to \alpha \text{ in } \mathscr{D} \tag{15}$$

in which the functions are defined for y in some set $\mathcal{K}$, is said to hold <u>uniformly</u> in $\mathcal{K}$ if we can find a single neighborhood $\mathfrak{N}_\alpha$ of α in $\mathscr{D}$ and a single constant $A > 0$, such that (2) will hold for every y in $\mathcal{K}$. For example, (13) can be put in the form (15), with $\mathcal{K} = \{1, 2, \cdots, k\}$. For any finite $\mathcal{K}$, the relation is always uniform, of course.

Lemma 3

If $\mathcal{K}$ is a set on which we define a measure μ and $c(y)$ is a real-valued measurable function on $\mathcal{K}$, and if (15) holds uniformly in $\mathcal{K}$ for measurable functions f and φ, then

$$\int_{\mathcal{K}} |c(y)f(x,y)|\,d\mu(y) = \Upsilon\left(\int_{\mathcal{K}} |c(y)\,\varphi(x,y)|\,d\mu(y)\right) \tag{16}$$

as $x \to \alpha$ in $\mathfrak{D}$. In particular, if relations (13) extend to all positive integers i uniformly, then

$$\Sigma_{i=1}^{\infty} |c_i f_i| = \Upsilon(\Sigma_{i=1}^{\infty} |c_i\varphi_i|)_\alpha\,. \tag{17}$$

(The rules for removing modulus bars apply to (16) and (17) as they did to (14) in Lemma 2.)

Lemma 4

If (13) holds, then

$$\Pi_{i=1}^{k} f_i = \Upsilon(\Pi_{i=1}^{k} \varphi_i)_\alpha\,. \tag{18}$$

Lemma 5

Let f and φ be Lebesgue-measurable in some interval $[\lambda, \alpha] \subseteq \mathfrak{D}$. Then

$$f = \Upsilon(\varphi)_\alpha \Rightarrow \int_x^\alpha |f(t)|\,dt = \Upsilon\left(\int_x^\alpha |\varphi(t)|\,dt\right) \text{ as } x \to \alpha\,. \tag{19}$$

(Removal of modulus bars applies as in lemmas 2 and 3.)

We mentioned earlier that, for given φ and α, a preferable notation to (3) might be $f \in \underline{O}(\varphi)_\alpha$, with $\underline{O}(\varphi)_\alpha$ denoting the class of functions satisfying (2) for some $\mathfrak{N}\alpha$ and A. In this paper a statement of the form $\Upsilon_1(\varphi_1)_\alpha = \Upsilon_2(\varphi_2)_\alpha$ means that a member of the first class is also a member of the second class, so a better notation might be $\Upsilon_1(\varphi_1)_\alpha \subseteq \Upsilon_2(\varphi_2)_\alpha$. This should be borne in mind when the following three lemmas are considered. Again, since the proofs are completely straightforward, they are omitted.

Lemma 6

$$\Upsilon[\Upsilon(\varphi)] = \Upsilon(\varphi) . \tag{20}$$

This is to be interpreted as meaning that, if $f = \Upsilon(\psi)$ and $\psi = \Upsilon(\varphi)$, then $f = \Upsilon(\varphi)$. Similar interpretations apply to the results below.

$$\left.\begin{array}{l} \underline{O}[\underline{o}(\varphi)] = \underline{o}(\varphi), \quad \underline{o}[\underline{O}(\varphi)] = \underline{o}(\varphi), \\ \Omega[\omega(\varphi)] = \omega(\varphi), \quad \omega[\Omega(\varphi)] = \omega(\varphi) . \end{array}\right\} \tag{21}$$

Lemma 7

$$\Upsilon(\varphi)\ \Upsilon(\psi) = \Upsilon(\varphi\psi) . \tag{22}$$

This is to be interpreted as meaning that, if $f = \Upsilon(\varphi)$ and $g = \Upsilon(\psi)$, then $fg = \Upsilon(\varphi\psi)$.

$$\underline{O}(\varphi)\underline{o}(\psi) = \underline{o}(\varphi\psi), \quad \Omega(\varphi)\omega(\psi) = \omega(\varphi\psi) . \tag{23}$$

$$\underline{O}(\varphi) + \underline{o}(\varphi) = \underline{O}(\varphi), \quad \Omega(\varphi) + \omega(\varphi) = \omega(\varphi) . \tag{24}$$

Lemma 8

$$\underline{o}(\varphi) = \underline{O}(\varphi), \quad \omega(\varphi) = \Omega(\varphi) . \tag{25}$$

$$\varphi = \underline{O}(\varphi), \quad \varphi = \Omega(\varphi) . \tag{26}$$

$$f\underline{O}(\varphi) = \underline{O}(f\varphi), \quad f\Omega(\varphi) = \Omega(f\varphi) . \tag{27}$$

$$\left.\begin{array}{l} 1/\underline{O}(\varphi) = \Omega(1/\varphi), \quad 1/\underline{o}(\varphi) = \omega(1/\varphi), \\ 1/\Omega(\varphi) = \underline{O}(1/\varphi), \quad 1/\omega(\varphi) = \underline{o}(1/\varphi) . \end{array}\right\} \tag{28}$$

If C is a finite nonzero constant, then

$$C\ \Upsilon(\varphi) = \Upsilon(\varphi) . \tag{29}$$

3. ASYMPTOTIC EQUIVALENCE

If f and g satisfy both

$$\left.\begin{array}{l} f = \underline{O}(g)_{\alpha} \text{ and } g = \underline{O}(f)_{\alpha}, \\ \text{or equivalently,} \\ f = \underline{O}(g)_{\alpha} \text{ and } f = \Omega(g)_{\alpha} \end{array}\right\} \qquad (30)$$

i.e., if we can find a neighborhood $\mathcal{N}_{\alpha}$ of α in $\mathcal{D}$ and two constants $A > B > 0$, such that

$$B\,|g(x)| \le f(x) \le A\,|g(x)| \quad \text{for all } x \in \mathcal{N}_{\alpha} \qquad (31)$$

then we write

$$f \asymp g \text{ at } \alpha \qquad (32)$$

and say that f and g are commensurate at α. It is clear that $\asymp$ has the three properties of an equivalence relation [$(\forall f)\ f \asymp f$, $(\forall f,g)\ f \asymp g \Rightarrow g \asymp f$, and $(\forall f,g,h)\ f \asymp g$ and $g \asymp h \Rightarrow f \asymp h$.] It thus splits the class of all functions (or any subclass) into equivalence classes, such that two functions are commensurate if and only if they belong to the same equivalence class. In particular, we note that $f \asymp g$ at α if $f(x)/g(x)$ tends to a finite nonzero limit as $x \to \alpha$.

More strongly, if

$$f(x)/g(x) \to 1 \text{ as } x \to \alpha \text{ in } \mathcal{D}, \qquad (33)$$

then we write

$$f \sim g \text{ at } \alpha \qquad (34)$$

and say that f and g are asymptotic (or asymptotically equivalent). This is an equivalence relation and yields equivalence classes of functions. Note that the asymptotic equivalence classes are distributed as subsets of the commensurate equivalence classes.

We see that (34) is equivalent to

$$f = g[1 + \underline{o}(1)_\alpha] \text{ or } f = g + \underline{o}(g)_\alpha . \tag{35}$$

We note also that if k is a constant, $f \sim k$ at α is equivalent to $f(x) \to k$ as $x \to \alpha$. Similarly, we see that $\underline{O}(k)_\alpha$ is the class of all functions finitely bounded in some neighborhood of α, and that $\underline{o}(k)$ is the class of all functions which (The choice of the value of k in these last two cases is arbitrary and irrelevant, by (29), so long as $k \neq 0$; and $k = 1$ is usually used.)

Given a sequence of functions $\varphi_0, \varphi_1, \varphi_2, \cdots$, with $\mathfrak{D} = \mathfrak{D}(\varphi_0, \varphi_1, \varphi_2, \cdots)$, and some $\alpha \in \overline{\mathfrak{D}}$, if

$$\varphi_k = \underline{o}(\varphi_{k-1})_\alpha \text{ for } k = 1, 2, 3, \cdots , \tag{36}$$

we say that they form an <u>asymptotic sequence</u> for $x \to \alpha$ in $\mathfrak{D}$. If the relations (36) are uniform in k, we have a <u>uniformly asymptotic sequence</u>.

A very important example of an asymptotic sequence for $x \to \alpha$ in $\mathfrak{D}$, with α finite, is

$$\varphi_k(x) = (x-\alpha)^k \quad (k = 0, 1, 2, \cdots) . \tag{37}$$

It is clear that this sequence is uniformly asymptotic, since every $\varphi_k(x)/\varphi_{k-1}(x) = x-\alpha$. For $x \to \infty$, the appropriate sequence corresponding to (37) is

$$\varphi_k(x) = x^{-k} \quad (k = 0, 1, 2, \cdots) . \tag{38}$$

Lemma 9

Let $[\varphi_k]_{k=0}^{\infty}$ be an asymptotic sequence. Then: (i) any subsequence $[\varphi_{k_r}]_{r=0}^{\infty}$ $(k_0 < k_1 < k_2 < \cdots)$ is asymptotic; (ii) for any $p > 0$, $[|\varphi_k|^p]_{k=0}^{\infty}$ is any asymptotic sequence; (iii) if $\varphi_k \asymp \psi_k$ for $k = 0, 1, 2, \cdots$, then $[\psi_k]_{k=0}^{\infty}$ is an asymptotic sequence; and (iv) if $[\varphi_k]_{k=0}^{\infty}$ and $[\psi_k]_{k=0}^{\infty}$ are asymptotic sequences, so are $[|\varphi_k| + |\psi_k|]_{k=0}^{\infty}$, $[\varphi_k\psi_k]_{k=0}^{\infty}$, and $[f\varphi_k]_{k=0}^{\infty}$, where f is any function.

The proof of (i) follows from (20); that of (ii) from Lemma 1, that of (iii) from (21), and that of (iv) from Lemma 2, Lemma 4, and the definition (8) with (5). Two asymptotic sequences related as in (iii) are called equivalent.

Lemma 10

(i) If $[\varphi_k(x,y)]_{k=0}^{\infty}$ is asymptotic, as $x \to \alpha$ in $\mathfrak{D}$, uniformly in $y \in \mathcal{K}$, if all φ_k are measurable relative to a measure μ on $\mathcal{K}$, and if φ_0 is integrable on $\mathcal{K}$, then $[\int_{\mathcal{K}} |\varphi_k(x,y)| \, d\mu(y)]_{k=0}^{\infty}$ is an asymptotic sequence. (ii) If $[\varphi_k(x)]_{k=0}^{\infty}$ is asymptotic, if all the φ_k are measurable, and φ_0 is integrable in some neighborhood of α, relative to the Lebesgue measure on $\mathfrak{R}$, then $[\int_x^{\alpha} |\varphi_k(t)| \, dt]_{k=0}^{\infty}$ is an asymptotic sequence for $x \to \alpha$ in $\mathfrak{D}$.

The proof of (i) is a consequence of Lemma 3; that of (ii) follows from Lemma 5.

4. ASYMPTOTIC SERIES

We now turn to a concept due, in its present form, to Poincaré[5] and Stieltjes.[6] Let $[\varphi_k(x)]_{k=0}^{\infty}$ be an asymptotic sequence for $x \to \alpha$ in $\mathfrak{D}(\varphi_0, \varphi_1, \varphi_2, \cdots) \subseteq \mathfrak{R}$. Consider the formal series

$$S = \Sigma_{k=0}^{\infty} a_k \varphi_k , \tag{39}$$

at present simply a composite symbol without necessary mathematical meaning; though its partial sums

$$S_m = \Sigma_{k=0}^{m-1} a_k \varphi_k \tag{40}$$

are well defined functions. If, for a function f defined on a set $\mathcal{F} \subseteq \mathcal{D}$ such that $\alpha \in \bar{\mathcal{F}}$, we know that

$$f - S_{m+1} = \underline{o}(\varphi_m)_\alpha \text{ for } m = 0,1,2,3,\cdots; \tag{41}$$

then we write

$$\left.\begin{array}{ll} & f \approx S \\ \text{or} & \\ & f(x) \approx S(x) \text{ as } x \to \alpha \text{ in } \mathcal{F}, \end{array}\right\} \tag{42}$$

and we say that S is an asymptotic series (or asymptotic expansion) for f at α . It follows immediately from (41) that

$$f - S_m = a_m \varphi_m + \underline{o}(\varphi_m)_\alpha \tag{43}$$

for $m = 0,1,2,\cdots$ (where we put $S_0 = 0$) . This yields Lemma 11.

Lemma 11

If S is an asymptotic series for f at α, then for $m = 0,1,2,\cdots$,

$$a_m = \lim_{\substack{x \to \alpha \\ (\text{in } \mathcal{F})}} \left[\frac{f(x) - S_m(x)}{\varphi_m(x)}\right]; \text{—} \qquad (44)^{\dagger}$$

and so S is the unique asymptotic series for f at α, in terms of the given asymptotic sequence $[\varphi_k]_{k=0}^{\infty}$.

Note that if S is an asymptotic series for f and if $\varphi_0 = \underline{O}(1)$, then (41) implies the weaker property

$$f - S_{m+1} \to 0 \text{ as } x \to \alpha \tag{45}$$

[†] Equations of particular significance to the algorithms presented in section 5 will be marked in this manner.

for each value of m in the set $\{0,1,2,3,\cdots\}$. By contrast, if S were a convergent series for f, then we would have

$$f - S_{m+1} \to 0 \quad \text{as} \quad m\to\infty$$

for each value of x in the region of convergence.

We observe that the converse of the uniqueness assertion of Lemma 11 is false: a formal series (39) does not uniquely determine a function to which it is asymptotic. If $f \approx S$ and $g \approx S$, then we only require that

$$f - g = \underline{o}(\varphi_m)_\alpha \quad \text{for} \quad m = 0,1,2,\cdots . \tag{46}$$

For example, if we use the sequence (38) as $x\to\infty$, $f - g$ could be like e^{-x}.

Further, given an asymptotic sequence $[\varphi_k]_{k=0}^{\infty}$ for $x\to\alpha$ in $\mathfrak{D}$, we can find a function f, defined on $\mathfrak{D}$, for which no asymptotic series exists. For example, if the sequence is (38) for $x\to\infty$, the function sin x has no asymptotic series since, formally, by (44), we would have $a_0 = \lim_{x\to\infty}\sin x$, but no such limit exists.

The simple examples (37) and (38) can be slightly generalized by putting

$$\left.\begin{aligned} &\varphi_k(x)/\varphi_{k-1}(x) = \lambda(x) \quad (k = 1,2,3,\cdots); \\ \text{where}\quad & \\ &\lambda(x)\to 0 \quad \text{as} \quad x\to\alpha \quad \text{in} \quad \mathfrak{D} . \end{aligned}\right\} \tag{47}^{\dagger}$$

In that case,

$$\varphi_k = \varphi_0\lambda^k ; \tag{48}^{\dagger}$$

and

$$S = \varphi_0\Sigma_{k=0}^{\infty}a_k\lambda^k \tag{49}$$

will be called an <u>asymptotic power series</u>.

Lemma 12

(i) If $f_i(x) \approx \Sigma_{j=0}^{\infty} a_{ij}\varphi_j(x)$ for $i = 1,2,\cdots, k$ as $x\to\alpha$ in $\mathcal{F}$, where $[\varphi_j]_{j=0}^{\infty}$ is asymptotic as $x\to\alpha$ in $\mathcal{F}$, then for any real numbers $c_1, c_2, \cdots, c_k$,

$$\Sigma_{i=1}^{k} c_i f_i(x) \approx \Sigma_{j=0}^{\infty}(\Sigma_{i=1}^{k} c_i a_{ij})\varphi_j(x) \ . \tag{50}^{\dagger}$$

(ii) If $f_i(x) \approx \Sigma_{j=0}^{\infty} a_{ij}\varphi_j(x)$ uniformly in i $(i = 1,2,3,\cdots)$ as $x\to\alpha$ in $\mathcal{F}$, and if $\Sigma_{i=1}^{\infty} c_i$ is absolutely convergent, while $\Sigma_{i=1}^{\infty} c_i a_{ij}$ converges for $j = 0,1,2,\cdots$, then $\Sigma_{i=1}^{\infty} c_i f_i(x)$ converges for all x in some neighborhood of α, and

$$\Sigma_{i=1}^{\infty} c_i f_i(x) \approx \Sigma_{j=0}^{\infty}(\Sigma_{i=1}^{\infty} c_i a_{ij})\varphi_j(x) \ . \tag{51}^{\dagger}$$

(iii) If $f(x,y) \approx \Sigma_{j=0}^{\infty} a_j(y)\varphi_j(x)$ uniformly in $y \in \mathcal{K}$, if the $f(x,y)$ (for each x in some neighborhood of α) and $a_j(y)$ (for each j) are measurable relative to a measure μ on $\mathcal{K}$, if $c(y)$ is integrable on $\mathcal{K}$, and if the integrals $\int_{\mathcal{K}} c(y)a_j(y)d\mu(y)$ exist (for each j), then the integral $\int_{\mathcal{K}} c(y)f(x,y)d\mu(y)$ exists for each x in some neighborhood of α, and

$$\int_{\mathcal{K}} c(y)f(x,y)d\mu(y) \approx \Sigma_{j=0}^{\infty}\left(\int_{\mathcal{K}} c(y)a_j(y)d\mu(y)\right)\varphi_j(x) \ . \tag{52}$$

Parts (i) and (ii) of this lemma are special cases of part (iii), with $\mathcal{K} = \{1,2,\cdots,k\}$ and $\mathcal{K} = \{1,2,3,\cdots\}$, respectively. The three parts correspond respectively to (14), (17), and (16), with $\Upsilon = \underline{o}$ and modulus bars removed from the left. To prove (iii), we note that, since $f \approx \Sigma_{j=0}^{\infty} a_j\varphi_j$ uniformly in y as $x\to\alpha$, by the relations (36) and (41) we know that, for all $m = 0,1,2,\cdots$,

$$f(x,y) - \Sigma_{j=0}^{m} a_j(y)\varphi_j(x) = \underline{o}[\varphi_m(x)] \tag{53}$$

uniformly in y. Thus by Lemma 3, we go from (15) (in the form of (53)) to (16), with $\Upsilon = \underline{o}$, to get

$$\int_{\mathcal{K}} c(y)\{\, f(x,y) - \Sigma_{j=0}^{m} a_j(y)\varphi_j(x)\}\, d\mu(y) = \underline{o}[\varphi_m(x)\int_{\mathcal{K}} |\, c(y)\,| d\mu(y)]$$
$$= \underline{o}[\varphi_m(x)]\ , \tag{54}$$

by (29) and (21), since c(y) is μ-integrable. As we noted in the proof of Lemma 3, the integral on the left of (54) exists for some neighborhood $\mathfrak{N}_\alpha$ of α in $\mathcal{F}$, and in that neighborhood

$$\int_{\mathcal{K}} c(y)\{f(x,y) - \Sigma_{j=0}^{m} a_j(y)\varphi_j(x)\}\, d\mu(y) =$$
$$\int_{\mathcal{K}} c(y)f(x,y)d\mu(y) - \Sigma_{j=0}^{m}(\int_{\mathcal{K}} c(y)a_i(y)d\mu(y))\varphi_j(x) \tag{55}$$

since the integrals $\int_{\mathcal{K}} c(y)a_j(y)d\mu(y)$ are supposed to exist for each j. This shows that $\int_{\mathcal{K}} c(y)f(x,y)d\mu(y)$ also exists for $x \in \mathfrak{N}_\alpha$. Finally, when we combine (54) and (55) to obtain the relation (41) corresponding to (52), we complete the proof of part (iii) of our lemma.

Lemma 13

If $f_i \approx \Sigma_{j=0}^{\infty} a_{ij}\varphi_j$ for $i = 1, 2, \cdots k$ as $x \to \alpha$ in $\mathcal{F}$, where $[\varphi_j]_{j=0}^{\infty}$ is asymptotic as $x \to \alpha$ in $\mathcal{F}$, and if the set of functions $\Pi_{i=1}^{k}\varphi_{j_i}$, with each j_i ranging through $0, 1, 2, \cdots$, can be rearranged into an asymptotic sequence $[\psi_t]_{t=0}^{\infty}$ (i.e., for each $[j_i, j_2, \cdots j_k]$ we have a corresponding $t = g(j_1, j_2, \cdots, j_k)$), then

$$\Pi_{i=1}^{k} f_i \approx \Sigma_{t=0}^{\infty} p_t\psi_t \tag{56}^{\dagger}$$

where p_t is the sum of all products $a_{1j_1}a_{2j_2}\cdots a_{kj_k}$ for which $t = g(j_1, j_2, \cdots, j_k)$. In particular, if we are dealing with a set of asymptotic power series which satisfy (48) and (49), then

$$g(j_1, j_2, \cdots, j_k) = \Sigma_{i=1}^{k} j_i \qquad (57)^{\dagger}$$

and

$$\psi_t = \varphi_0{}^{k}\lambda^{t} = \varphi_0^{k-1}\varphi_t \,. \qquad (58)^{\dagger}$$

To prove the lemma, we first note that, if $\Pi_{i=1}^{k}\varphi_{j_i} = \psi_t$, then any product with only one index j_i changed to j_i' will be a $\psi_{t'}$, with $t' > t$ if $j_i' > j_i$, and $t' < t$ if $j_i' < j_i$, by (36). Further, if the products $\Pi_{i=1}^{k}\varphi_{j_i}$ can be ordered in an asymptotic sequence, then products $\Pi_{i=1}^{h}\varphi_{j_i}$, for any $h < k$, can also be ordered in an asymptotic sequence $[\psi_t^{(h)}]_{t=0}^{\infty}$ (since the ordering will be the same as that of $\Pi_{i=1}^{k}\varphi_{j_i}$ with $j_i = 0$ for $h < i \leq k$). Suppose that the relation (56) holds with k replaced by $h - 1$ $(h \leq k)$ (this is trivially true for $h = 2$), and consider $\Pi_{i=1}^{h} f_i$. We know by (41) that

$$\left.\begin{aligned} \Pi_{i=1}^{h-1} f_i &= \Sigma_{t=0}^{m} p_t \psi_t^{(h-1)} + \underline{o}(\psi_m^{(h-1)}) \\ \text{and} \qquad f_h &= \Sigma_{j=0}^{n} a_{hj}\varphi_j + \underline{o}(\varphi_n) \,. \end{aligned}\right\} \qquad (59)$$

Hence

$$\Pi_{i=1}^{h} f_i = \Sigma_{t=0}^{m}\Sigma_{j=0}^{n} p_t a_{hj}\psi_t^{(h-1)}\varphi_j + \underline{o}(\varphi_0\psi_m^{(h-1)}) + O(\psi_0^{(h-1)}\varphi_n)$$
$$+ \underline{o}(\psi_m^{(h-1)}\varphi_n)$$

by (23). Now we know that $\varphi_0\psi_m^{(h-1)} = \psi_r^{(h)}$ and $\psi_0^{(h-1)}\varphi_n = \psi_s^{(h)}$ for some r and s, depending on m and n. The double sum contains a finite number $w \le (m+1)(n+1)$ of distinct $\psi_t^{(h)}$. Thus, for any $v \le \min\{r,s,w\}$, we have

$$\Pi_{i=1}^{h} f_i = \Sigma_{u=0}^{v} q_u \psi_u^{(h)} + \underline{o}(\psi_v^{(h)}) , \tag{60}$$

where the coefficients q_u are obtained by summing $p_t a_{hj}$ for all values of t and j for which $\psi_t^{(h-1)}\varphi_j = \psi_u^{(h)}$. Equation (60) is of the form (41) for $\Pi_{i=1}^{h} f_i$. Finally, we note that the double sum contains the (m+1) terms $\psi_t^{(h-1)}\varphi_0$ for which the corresponding $\psi_u^{(h)}$ are distinct and have $u < r$. Thus $r > m$ and $w > m$; and, similarly, $s > n$ and $w > n$. It follows that we can take any $v \le \min\{m,n\}$ in (60), so that (60) holds for all values of v (as is shown by taking m and n large enough). This proves, by induction, that (56) holds for k factors.

In the case of asymptotic power series, the relations (57) and (58) follow immediately.

The sequence

$$\varphi_k(x) = x^{-k}\sin 2^k x \quad (k = 0,1,2,\cdots) \tag{61}$$

has $\varphi_k(x)/\varphi_{k-1}(x) = 2x^{-1}\cos 2^{k-1}x \to 0$ as $x\to\infty$, so that it is uniformly asymptotic. However, we see that

$$[\varphi_k(x)]^2/\varphi_{k-1}(x)\varphi_{k+1}(x) = \cos 2^{k-1}x/\cos 2^k x ; \tag{62}$$

and so this ratio oscillates infinitely, and infinitely often as $x\to\infty$ (since $\cos 2^{k-1}x$ vanishes when $x = (4n+2)(\pi/2^{k+1})$, and $\cos 2^k x$ vanishes when $x = (4n\pm 1)(\pi/2^{k+1})$, for all values of the integer n). This shows that the products φ_k^2 and $\varphi_{k-1}\varphi_{k+1}$ cannot be ordered by relation (36). Thus the condition of Lemma 13 is not necessarily satisfied.

Lemma 15

If $[\varphi_j/\varphi_0]_{j=0}^{\infty}$ is a totally multiplicative asymptotic sequence for ~~$x\to$~~ $\mathcal{D}$, and if (64) holds as $x\to\alpha$ in $\mathcal{F}\subseteq\mathcal{D}$, then for any real r,

$$f^r \approx a_0{}^r\,\varphi_0{}^r\,\Sigma_{t=0}^{\infty} p_t\psi_t, \tag*{(65)†}$$

where p_t is the sum of all terms

$$\frac{r(r-1)\cdots(r-i_1-i_2-\cdots-i_k+1)}{i_1!\,i_2!\cdots i_k!}\left(\frac{a_1}{a_0}\right)^{i_1}\left(\frac{a_2}{a_0}\right)^{i_2}\cdots\left(\frac{a_k}{a_0}\right)^{i_k}, \tag*{(66)†}$$

with each $i_j \geq 0$ for $1 \leq j < k$, but $i_k > 0$, and any $k = 0,1,2,\cdots$ (when $k = 0$, $t = 0$, $p_0 = 1$, and $\psi_0 = 1$) for which $g_{i_1+i_2+\cdots+i_k}$ (1, i_1 times; 2, i_2 times; $\cdots$, k, i_k times) $= t$. Each p_t is the sum of a finite number of terms.

To prove this lemma, we first note that, by (41), to have (65) we must show that, for $n = 0,1,2,\cdots$,

$$f^r = a_0{}^r\,\varphi_0{}^r\left\{\Sigma_{t=0}^{n} p_t\psi_t + \underline{o}(\psi_n)\right\}. \tag{67}$$

By the general binomial theorem, since (64) implies (41),

$$\left.\begin{aligned} f^r &= a_0{}^r\,\varphi_0{}^r\,\Sigma_{h=0}^{\infty}\frac{r(r-1)\cdots(r-h+1)}{h!}\,\Phi_m{}^h, \\ \text{where}\quad \Phi_m &= \Sigma_{j=1}^{m}\left(\frac{a_j\varphi_j}{a_0\varphi_0}\right) + \underline{o}\left(\frac{\varphi_m}{\varphi_0}\right). \end{aligned}\right\} \tag{68}$$

By the conditions of this lemma, $\Phi_m \asymp \varphi_1/\varphi_0$ and, for any n, a power s will exist, such that $\Phi_m{}^h = \underline{o}(\psi_n)$ whenever $h > s$, so that for any $\epsilon > 0$, there is a neighborhood $\mathcal{O}_\alpha(\epsilon)$ of α in $\mathcal{F}$, such that $|\Phi_m(x)|^s \leq \epsilon\,|\psi_n(x)|$

The last lemma corresponded to Lemma 4. The next one corresponds to Lemma 5.

Lemma 14

Let $f \approx \Sigma_{j=0}^{\infty} a_j \varphi_j$ as $x \to \alpha$ in $\mathscr{F}$, where $[\varphi_j]_{j=0}^{\infty}$ is asymptotic as $x \to \alpha$ in $\mathscr{F}$, and let f and all the φ_j be Lebesgue-integrable in some interval $[\lambda, \alpha] \subseteq \mathscr{F}$, with the $\varphi_j \geq 0$ $(j = 0,1,2,\cdots)$ in $[\lambda, \alpha]$. Then

$$\int_x^{\alpha} f(t)dt \approx \Sigma_{j=0}^{\infty} a_j \int_x^{\alpha} \varphi_j(t)dt \; . \qquad (63)^{\dagger}$$

The proof is a straightforward consequence of Lemma 5, Lemma 10 (ii), and (41). It is omitted here.

By Lemma 9 (i), a subsequence of an asymptotic sequence is also asymptotic. Thus we may, without loss of generality, suppose that

$$f \approx \Sigma_{k=0}^{\infty} a_k \varphi_k, \text{ with } a_0 \neq 0, \; a_1 \neq 0 \; . \qquad (64)$$

We call the sequence of φ_j <u>totally multiplicative</u> under the following conditions: the asymptotic sequence $[\varphi_j]_{j=0}^{\infty}$ is such that all the products $\Pi_{i=1}^{k} \varphi_{j_i}$ (with each j_i ranging through the values $0,1,2,\cdots$, and $k = 1,2,3,\cdots$) can be ordered in an asymptotic sequence $[\psi_t]_{t=0}^{\infty}$ (i.e., for each $[j_1, j_2, \cdots, j_k]$, we have a $t = g_k(j_1, j_2, \cdots, j_k)$); and for every j_1 and j_2, with $j_1 < j_2$, there is a power k, such that $\varphi_{j_1}^{k} = \underline{o}(\varphi_{j_2})$, while for every j_1 and k there is a j_2, such that $\varphi_{j_2} = \underline{o}(\varphi_{j_1}^{k})$.

The next lemma presents a result which does not appear to be in the literature.

whenever $x \in \theta_\alpha(\epsilon)$. Also, $\theta_\alpha(\epsilon)$ can be so chosen that $|\Phi_m(x)| < 1/2$, since $\Phi_m(x) \to 0$ as $x \to \alpha$. Thus, for $x \in \theta_\alpha(\epsilon)$,

$$\left|\Sigma_{h=s+1}^{\infty} \frac{r(r-1)\cdots(r-h+1)}{h!}[\Phi_m(x)]^h\right| \le \epsilon\,|\psi_n(x)|\,\Sigma_{h=s+1}^{\infty}\left|\frac{r(r-1)\cdots(r-h+1)}{h!}\right|(1/2)\ h-s;$$

and, since the sum on the right converges, this shows that the sum on the left is $\underline{o}(\psi_n)$. Thus (68) yields that

$$f^r = a_0{}^r\,\varphi_0{}^r\left\{\Sigma_{h=0}^{s}\frac{r(r-1)\cdots(r-h+1)}{h!}\Phi_m{}^h + \underline{o}(\psi_n)\right\}. \qquad (69)$$

We note, too, that any product of φ_j, arising from terms in $\Phi_m{}^h$ with $h > s$, will certainly have a g-function with a value strictly greater than n.

By applying the multinomial theorem to each $\Phi_m{}^h$ in (69), we get

$$f^r = a_0{}^r\varphi_0{}^r\left\{\Sigma_{h=0}^{s}\ \Sigma_{\substack{i_1,i_2,\cdots i_m\\ i_1+i_2+\cdots+i_m \le h}} \frac{r(r-1)\cdots(r-h+1)}{i_1!\,i_2!\cdots i_m!\,(h-i_1-i_2-\cdots-i_m)!}\right.$$

$$\left.\left(\frac{a_1\varphi_1}{a_0\varphi_0}\right)^{i_1}\left(\frac{a_2\varphi_2}{a_0\varphi_0}\right)^{i_2}\cdots\left(\frac{a_m\varphi_m}{a_0\varphi_0}\right)^{i_m}\underline{o}\left[\left(\frac{\varphi_m}{\varphi_0}\right)^{h-i_1-i_2-\cdots-i_m}\right] + \underline{o}(\psi_n)\right\}; \qquad (70)$$

and if we choose m, as we always can under the conditions of this lemma, so that $\varphi_m/\varphi_0 = \underline{O}(\psi_n)$, we can simplify (70) to

$$f^r = a_0{}^r\varphi_0{}^r\left\{\Sigma_{h=0}^{s}\Sigma_{\substack{i_1,i_2,\cdots,i_m\\ i_1+i_2+\cdots+i_m=h}}\frac{r(r-1)\cdots(r-h+1)}{i_1!\,i_2!\cdots i_m!}\left(\frac{a_1\varphi_1}{a_0\varphi_0}\right)^{i_1}\left(\frac{a_2\varphi_2}{a_0\varphi_0}\right)^{i_2}\right.$$

$$\left.\cdots\left(\frac{a_m\varphi_m}{a_0\varphi_0}\right)^{i_m} + \underline{o}(\psi_n)\right\}. \qquad (71)$$

We now see that the finite sum of terms in (71) can be rearranged, in the manner described in the assertion of the lemma, to form terms $p_t \psi_t$. (Any terms with $t > n$ are lumped into the $\underline{o}(\psi_n)$ of (67).)

In the particular case of asymptotic power series, when (48) holds, the sequence $[\varphi_j/\varphi_0]_{j=0}^{\infty} = [\lambda^j]_{j=0}^{\infty}$ is evidently totally multiplicative with $\psi_t = \lambda^t$. Also, if

$$f \approx a_0\varphi_0 + a_1\varphi_0\lambda^u + \Sigma_{k=2}^{\infty} a_k\varphi_0\lambda^{u+k-1} \qquad (72)^{\dagger}$$

with $a_0 \neq 0$ and $a_1 \neq 0$, then Equation (71) simplifies to

$$f^r = {a_0}^r{\varphi_0}^r\Big\{\Sigma_{h=0}^{s}\Sigma_{\underbrace{i_1, i_2, \cdots, i_m}_{i_1+i_2+\cdots+i_m=h}} \frac{r(r-1)\cdots(r-h+1)}{i_1!i_2!\cdots i_m!}$$

$${a_1}^{i_1}{a_2}^{i_2}\cdots{a_m}^{i_m}{a_0}^{-h}\lambda^{hu+i_2+2i_3+\cdots+(m-1)i_m} + \underline{o}(\lambda^n)\Big\} \qquad (73)^{\dagger}$$

and we choose $s = [n/u]$ (where $[x]$ denotes the greatest integer not greater than x) and $m = n - u + 1$.

We see from Lemmas 12 through 15 that, under fairly likely and lax restrictions, we can formally combine the asymptotic series of given functions by addition, subtraction, multiplication, division, and taking arbitrary powers, as well as by forming infinite sums and integrals, and obtain the asymptotic series of the corresponding combinations of functions. By a reversal of Lemma 14, we can also differentiate an asymptotic series when the resulting series satisfies the conditions of Lemma 14, as follows.

Lemma 16

If $f \approx \Sigma_{j=0}^{\infty} a_j\varphi_j$ as $x \to \alpha$ in $\mathcal{F}$, where $\varphi_0 = \underline{O}(1)_{\alpha}$ and $[\varphi_j]_{j=0}^{\infty}$ is asymptotic as $x \to \alpha$ in $\mathcal{F}$, if f and the φ_j are differentiable almost everywhere in some interval $[\lambda,\alpha] \subseteq \mathcal{F}$, with $\varphi_j' \geq 0$ in $[\lambda,\alpha]$, if $[\varphi_j']_{j=0}^{\infty}$ is asymptotic as $x \to \alpha$ in $\mathcal{F}$, and if $f' \approx \Sigma_{j=0}^{\infty} b_j \varphi_j'$, then $a_j = b_j$ $(j = 1,2,3,\cdots)$.

By Lemma 14 we note that, since f' and the φ_j' are Lebesgue-integrable in $[\lambda,\alpha]$, by (63), and since $f \sim a_0\varphi_0 = \underline{O}(1)$ as $x \to \alpha$; if $\lambda < x < y < \alpha$, then

$$f(y) - f(x) = \int_x^y f'(t)dt \approx \Sigma_{j=0}^{\infty} b_j \int_x^y \varphi_j'(t)dt = \Sigma_{j=0}^{\infty} b_j[\varphi_j(y) - \varphi_j(x)] .$$

From this, since $\varphi_j(y) \to 0$ as $y \to \alpha$ if $j = 1,2,3,\cdots$, we get

$$f(x) \approx \lim_{y\to\alpha} [f(y) - b_0\varphi_0(y)] + \Sigma_{j=0}^{\infty} b_j\varphi_j(x) . \tag{74}$$

Formula (44) now shows that $a_j = b_j$ for $j \geq 1$. (For $j = 0$, we have no such result; but this corresponds to the "arbitrary constant" of integration.)

When we deal with asymptotic power series, this result can be somewhat strengthened by Lemma 17.

Lemma 17

If $f \approx \varphi_0 \Sigma_{j=0}^{\infty} a_j \lambda^j$ as $x \to \alpha$ in $\mathcal{F}$, where $\lambda \to 0$ as $x \to \alpha$ in $\mathcal{F}$, and if f, φ_0 and λ are differentiable in $\mathcal{F}$, and if

$$f' \approx \varphi_0' \Sigma_{j=0}^{\infty} a_j \lambda^j + \lambda' \Sigma_{j=1}^{\infty} j b_j \lambda^{j-1} \tag{75}^{\dagger}$$

(i.e., $(f/\varphi_0)'$ has an asymptotic power series in $[\varphi_0\lambda'\lambda^j]_{j=0}^{\infty}$); then $a_j = b_j$ $(j = 1,2,3,\cdots)$.

The proof of this is simple, and it is given in all the references 1, 2, 3, and 4.

Henrici[7] discusses the operations described in Lemmas 12 through 15, as applied to convergent power-series. Although he does not give the detailed results of Lemma 15, he does mention an excellent alternative approach (unfortunately not applicable to general asymptotic series) which he attributes to J.C.P. Miller. Its application to asymptotic power series in which $\varphi_0 = 1$ runs as follows.

$f \approx \Sigma_{j=0}^{\infty} a_j \lambda^j$ and $f' \approx \lambda' \Sigma_{j=1}^{\infty} j a_j \lambda^{j-1}$. If $f^r \approx \Sigma_{k=0}^{\infty} p_k \lambda^k$, then

$$f(f^r)' = r f^r f' \approx r \lambda' \Sigma_{k=0}^{\infty} p_k \lambda^k \Sigma_{j=1}^{\infty} j a_j \lambda^{j-1} \text{ and}$$

$$f(f^r)' = \lambda' \Sigma_{k=1}^{\infty} k p_k \lambda^{k-1} \Sigma_{j=0}^{\infty} a_j \lambda^j .$$

Thus, if we equate coefficients of powers of λ (using lemmas 11 and 13 to justify our action), we get

$$p_{h+1} = \frac{1}{(h+1)a_0} \Sigma_{k=0}^{h} [r(h-k+1)-k] a_{h-k+1} p_k \; (h=0,1,2,\cdots) . \qquad (76)^{\dagger}$$

From this the coefficients $p_1, p_2, p_3, \cdots$ are recursively obtained in terms of p_0 and the a_j $(j=0,1,2,\cdots)$. Since $p_0 = a_0{}^r$, the problem of finding all the p_k is solved, straightforwardly and recursively.

5. IMPLEMENTATION OF AN ASYMPTOTIC SUBPROGRAM

Problems and Assumptions

We now turn to the main purpose of this paper; namely, implementing an asymptotic package, a subprogram to be included in a formula-manipulation system. What is described here is to be seen as just a basic starting package, which hopefully will be expanded in scope and power as larger computers become available.

As was explained in section 3, for any given $\mathfrak{D}$ and any $\alpha \in \tilde{\mathfrak{D}}$, all functions will fall into commensurate equivalence classes. The general theoretical treatment of these classes will lead to difficulties involving the axiom of uncountable choice; but in any actual computational application, we deal with only a finite number of asymptotic series, each truncated to a finite number of terms. Thus, no difficulty should be encountered.

For each $\mathfrak{D}$ and α, we collect and list function subroutines for each successive "easy" function which we encounter and which is not commensurate with any of the functions already listed. Further, whenever two functions in our list can be ordered by an $\underline{o}$ (or equivalently $\underline{\omega}$) relation, this should be indicated. (Certain parts of functions cannot be so ordered, as was exemplified in (61) and (62).) Thus the collection of functions are a partially-ordered set. Any ordered subset of this collection will be an asymptotic sequence for the given $\mathfrak{D}$ and α . Such a sequence will not necessarily be totally multiplicative. An example is the sequence $[\varphi_k]_{k=0}^{\infty}$, defined by

$$1/\varphi_0(x) = x, \quad 1/\varphi_k(x) = \exp\,[1/\varphi_{k-1}(x)] \quad (k=1,2,3,\cdots) \tag{77}$$

which is asymptotic as $x \to \infty$ in $\mathfrak{R}^+ = \{ x : x > 0 \}$, but which is certainly not totally multiplicative, since $x^n = \underline{o}(\exp x)$ for all powers n, as $x \to \infty$.

To carry out the process described above, we must be able to find the limits of ratios of arbitrary functions as their common argument tends to an arbitrary limit. Thus we must presuppose the existence in our main system (or in the asymptotic package) of a limit subprogram such as is described by R. Iturriaga.[8] Any shortcomings of this limit subprogram will emerge in the asymptotic subprogram as consequent shortcomings in the form of an inability to obtain certain limits, and hence an inability to make certain decisions essential to the computation of certain asymptotic expansions. This is, of course, inevitable; and we shall suppose that such cases (which would lead to some kind of interrupt or negative response) are not being considered by the subprogram.

Often only one limit α is considered in any given computation, and this limit is almost always 0 or $+\infty$. It seems, therefore, that it is more efficient to set up the asymptotic subprogram to compute expansions asymptotic as $x \to \infty$ only, or, at most, expansions for $x \to 0$ and for $x \to \infty$. Clearly, any formula manipulation system should be capable of transforming any but the most unusual functions from $f(x)$ to $f(y+\alpha)$ or $f(\alpha + y^{-1})$.

To specify the required asymptotic expansion, one clearly needs the following information:

a. an expression (say f) denoting the function whose asymptotic expansion is required (this could be a formula, the name of a function, or the label of a suitable subroutine);

b. a variable (say x) in terms of which the expansion is to be made (this may appear in the expression f; in any case, it will be an argument of this expression);

c. the limit (say α) to which x tends in the asymptotic process (and this is understood to include mention of the set $\mathcal{S}$ or $\mathcal{F}$, if it is relevant to the the limit process);

d. a label (say L) identifying the particular asymptotic sequence $[\varphi_k]_{k=0}^{\infty}$ relative to which the asymptotic expansion is to be made; and

e. a positive integer (say n) denoting the number of terms in the expansion.

It is suggested that, for practical purposes, it is preferable to let n denote the number of nonzero terms in the asymptotic expansion.

The appropriate instruction could take the form

$$z \leftarrow \text{ASYMP}\ (f;\ x,\ \alpha;\ L;\ n) \tag{78}$$

and if, in fact, $f \approx \Sigma_{j=0}^{\infty} a_j \varphi_j$ as $x \to \alpha$, with

$$\left.\begin{aligned} & a_{j_h} \neq 0 \ \text{ for } h = 1,2,\cdots n\ (j_1 < j_2 < \cdots < j_n)\ , \\ & a_j = 0,\ \text{if } j < j_n \text{ and } j \neq j_h\ (h = 1,2,\cdots,n)\ ; \end{aligned}\right\} \tag{79}$$

then the asymptotic package should return the expression

$$z = \Sigma_{h=1}^{n}\, a_{j_h} \varphi_{j_h}(x) \tag{80}$$

in the form appropriate to the particular formula manipulation system to which it is adjoined.

To be specific, we assume that the limit subprogram is geared to computing limits as $x \to \infty$ by means of an instruction of the form

$$z \leftarrow \text{LIM}\ (f; x)\ . \tag{81}$$

In the expression (78), if α is omitted, it is assumed to be $+\infty$; if L is omitted, the asymptotic sequence is assumed to be the power series (38). Thus, if $f \approx \Sigma_{j=0}^{\infty} a_j x^{-j}$, with (79), the instruction

$$z \leftarrow \text{ASYMP}\ (f; x; n) \tag{82}$$

yields the expression

$$z = \Sigma_{h=1}^{n} a_{j_h} x^{-j_h}\ . \tag{83}$$

Thus we note that, by (44),

$$\text{ASYMP}\ (f; x; 1) = \text{LIM}\ (f; x) \tag{84}$$

whenever the limit on the right exists and is neither zero nor infinite.

Clearly, there will be many pitfalls, of both a theoretical and a practical nature, which can arrest the computation of an asymptotic expansion. Next we consider the procedures to be followed by the asymptotic subprogram to unravel the expansion. The difficulties and pitfalls emerge in the course of the discussion.

Step I

If $\alpha \neq +\infty$, we must use the main formula manipulation system to replace the variable x in the expression f by $(\alpha + y^{-1})$, obtaining a new expression $\hat{f}$. This operation will be denoted by

$$\hat{f} \leftarrow \text{REPL}\ (f; x, \alpha + y^{-1})\ . \tag{85}$$

If L denotes the sequence $[\varphi_j(x)]_{j=0}^{\infty}$ (asymptotic as $x \to \alpha$) and if $\hat{L}$ denotes the corresponding sequence $[\hat{\varphi}_j(y)]_{j=0}^{\infty} = [\varphi_j(\alpha + y^{-1})]_{j=0}^{\infty}$ (asymptotic as $y \to \infty$), then instruction (78) is equivalent to

$$\left.\begin{array}{l} \hat{f} \leftarrow \text{REPL } (f; x, \alpha + y^{-1}), \\ z \leftarrow \text{ASYMP } (\hat{f}; y; \hat{L}; n), \\ z \leftarrow \text{REPL } (z; y, (x - \alpha)^{-1}) \; . \end{array}\right\} \qquad (86)$$

The principal difficulty here is the passage from L to $\hat{L}$. In an initial asymptotic package, this transformation will probably have to be done by the user, and only asymptotic expansions with $\alpha = +\infty$ will be handled automatically. Of course, in the simplest case, when L identifies the power sequence (37), then $\hat{L}$ should refer us to (38). More generally, if L denotes a subroutine which successively presents us with $\varphi_0(x)$, $\varphi_1(x)$, $\varphi_2(x)$, and so on, then we may replace the calling instructions

$$\left.\begin{array}{ll} & \text{INITIALIZE } L\ (\varphi, x), \\ \text{and} & \\ & \text{CALL } L, \end{array}\right\} \qquad (87)$$

by

$$\left.\begin{array}{ll} & \text{INITIALIZE } L\ (\varphi, x), \\ \text{and} & \left\{\begin{array}{l} \text{CALL } L, \\ \\ \text{REPL } (\varphi; x, \alpha + y^{-1}); \end{array}\right. \end{array}\right\} \qquad (88)$$

these last being equivalent to the purely conceptual

$$\left.\begin{array}{ll} & \text{INITIALIZE } \hat{L}\ (\varphi, y), \\ \text{and} & \\ & \text{CALL } \hat{L}, \end{array}\right\} \qquad (89)$$

respectively. (The asymptotic subprogram may initially address itself to the subroutine L, giving it an initialize signal (which sets the index j at zero), the expression name φ, and the variables x; subsequent calling of subroutine

L (without the initialize signal) will put the function $\varphi_j(x)$ at φ and increase j by one; thus successive calls to L will put $\varphi_0(x)$, $\varphi_1(x)$, $\varphi_2(x)$, $\cdots$ at φ .) However, in this case it would be preferable to modify the subroutine L *ad hoc* or to have an appropriate $\hat{L}$ already, in order to increase the efficiency of the process.

Step II

Assuming that the procedure in step I has already been carried out, if necessary, we may sometimes receive an instruction

$$z \leftarrow \text{ASYMP}\ (f; y; L; n) \tag{90}$$

(where the circumflex accents are omitted for simplicity). The expression f is either expressible in the form of an operation T applied to one, two, or more expressions $f_1, f_2, \cdots$, or f is *elementary*. In the latter case, f either does not contain y or it is y itself. If f is elementary, no further unraveling is possible, and the same applies when we encounter an operation T which is not one of the algebraic or calculus-oriented operations discussed in section 4. We must now use the knowledge we have, relative to special functions and their asymptotic expansions and to the powers of the limit subprogram, using (44). Crudely, and in the absence of special knowledge, the procedure at this stage is as follows.

```
        INITIALIZE $L(\varphi,y)$, [Note: this includes $j \leftarrow 0$]
        $h \leftarrow 1$, $z \leftarrow 0$,
(L1)    CALL L, [Note: this includes $\varphi \leftarrow \varphi_j(y)$, $j \leftarrow j+1$]
        $a \leftarrow \text{LIM}((f - z)/\varphi; y)$,
        IF $a \neq 0$ { $z \leftarrow z + a \times \varphi$,
                      IF $h = n$ { end of procedure },
                      $h \leftarrow h + 1$ },
        GO TO L1                                              (91)
```

[In (91), instructions end with a comma; "a ← b" means that the variable a is given the value of the expression b; the notation "GO TO label" denotes an unconditional jump; a label in parentheses in the left-hand margin refers to the instruction immediately to its right; the notation "IF statement{instructions}" means that the instructions in curly brackets are obeyed only if the statement is true, otherwise they are omitted; "end of procedure" is a jump to whatever is to be done after the procedure is completed.]

If all the limits in (91) are obtainable and finite, and if we do not get simply an infinite succession of zero coefficients a beyond some stage (for example, the asymptotic expansion of $1 + e^{-x}$ in terms of (38) is $1 + 0.x^{-1} + 0.x^{-2} + 0.x^{-3} + \cdots$, as $x \to \infty$), then the procedure (91) will give the required expansion in a finite number of operations.

Now we must discuss the algorithms corresponding to the operations of algebra and the differential and integral calculus, as they may appear in the expression f. Let us suppose that the expression z in (80) is stored in our system in the form of a heading [x; L; n], followed by a pair of parallel lists $[j_1, j_2, \cdots, j_n]$ and $[a_{j_1}, a_{j_2}, \cdots, a_{j_n}]$. (This is not a hard-and-fast specification, but just a proposal, to establish a notation.)

We shall adopt the notations

$$a \leftarrow \text{ASCOF}\ (f; x, \alpha; L; h), \tag{92}$$

$$t \leftarrow \text{ASTER}\ (f; x, \alpha; L; h), \tag{93}$$

and

$$j \leftarrow \text{ASIND}\ (f; x, \alpha; L; h); \tag{94}$$

for the instructions which respectively, put, as a, the h^{th} nonzero coefficient of the asymptotic expansion of f (as $x \to \alpha$) in terms of the sequence at L; as t, the corresponding term of the expansion; and as j, the index j_m of this term in the sequence L. (The omission conventions will be the same as that for ASYMP.) It follows that

$$\text{ASTER}\ (f; x, \alpha; L; h) = \text{ASYMP}\ (f; x, \alpha; L; h) - \text{ASYMP}\ (f; x, \alpha; L; h-1)\ . \tag{95}$$

(We adopt the convention that

$$\text{ASYMP}\ (f;\ x,\ \alpha;\ L;\ 0) = 0\ , \tag{96}$$

which makes (95) meaningful when $h = 1$.) In general, ASTER will be a part of ASYMP, e.g., (91) can be written formally as

```
      $h \leftarrow 1,\ \ z \leftarrow 0,$
(L1)  $z \leftarrow z + \text{ASTER}\ (f;\ y;\ L;\ h),$
      IF $h = n$ { end of procedure },                    (97)
      $h \leftarrow h + 1,$
      GO TO L1 .
```

We also know that

$$\text{ASTER}\ (f;\ x,\ \alpha;\ L;\ h) = \text{ASCOF}\ (f;\ x,\ \alpha;\ L;\ h) \times \varphi_{\text{ASIND}}\ (f;\ x,\ \alpha;\ L;\ h)\ . \tag{98}$$

The two lists which follow the heading $[x;\ L;\ n]$ in the assumed representation of the asymptotic expansion (80) will thus be $[\text{ASIND}\ (f;\ x,\ \alpha;\ L;\ h)]_{h=1}^{n}$ and $[\text{ASCOF}\ (f;\ x,\ \alpha;\ L;\ h)]_{h=1}^{n}$ respectively.

Step III

If we wish to perform the operation (90) and if $f(y) = \Sigma_{i=1}^{k} c_i f_i(y)$, where all the $c_i \neq 0$, then Lemma 12(i) yields the algorithm

```
        $z \leftarrow 0$, $i \leftarrow 1$,
(L1)    $h_i \leftarrow 1$, $j_i \leftarrow$ ASIND ($f_i$; y; L; 1),
        IF $i = k$ { GO TO L2 } , $i \leftarrow i + 1$, GO TO L1,
(L2)    $h \leftarrow 1$, $j \leftarrow 0$,
(L3)    $a \leftarrow 0$, $i \leftarrow 1$,
(L4)    IF $j_i = j$ { $a = a + c_i \times$ ASCOF ($f_i$; y; L; $h_i$),                    (99)
                  $h_i \leftarrow h_i + 1$, $j_i \leftarrow$ ASIND ($f_i$; y; L; $h_i$) ,
        IF $i = k$ { GO TO L5 }, $i \leftarrow 1 + 1$, GO TO L4,
(L5)    IF $a \neq 0$ { $z \leftarrow z + a \times \varphi_j(y)$,
                  IF $h = n$ { end of procedure },
                  $h \leftarrow h + 1$ }, $j \leftarrow j + 1$, GO TO L3.
```

If we wish to appeal to parts ii or iii of Lemma 12 to handle infinite series or integrals, we must verify that the conditions are satisfied, and we must have the capacity, in the main formula manipulation system, to sum infinite series or compute integrals. Another case arises, however, which is simpler. If $f = \Sigma_{i=1}^{\infty} c_i f_i$, but, for each index $(h + 1)$, there is an index k_h, such that $f_i \approx \Sigma_{j=h+1}^{\infty} a_{ij} \varphi_j$ for all $i > k_h$ (i.e., the first h+1 functions $\varphi_0, \varphi_1, \varphi_2, \cdots, \varphi_h$ appear only in the asymptotic expansions of the first k_h functions $f_1, f_2, \cdots, f_{k_h}$) [if the k_h are exact bounds, then k_h will be a nondecreasing function of h we can still compute (90) finitely using a modified form of (99), in which the first line and the line labeled L3 are respectively replaced by

$$z \leftarrow 0,\ \ i \leftarrow 1,\ \ k \leftarrow \max(k_0, 1)\ ,$$

and

```
      (L3)  $a \leftarrow 0,\ i \leftarrow 1,$ IF $k \geq k_j;$ $\{$GO TO L4$\}$,
            $m \leftarrow k+1,\ k \leftarrow k_j,$                                   (100)
      (L6)  $h_m \leftarrow 1,\ j_m \leftarrow$ ASIND $(f_m; y; L; 1),$
            IF $m = k$ $\{$GO TO L4$\}$, $m \leftarrow m+1,$ GO TO L6 .
```

Step IV

If we wish to compute (90), if $f(y) = \Pi_{i=1}^{k} f_i(y)$, and if the condition of Lemma 13 is satisfied; then we can construct an appropriate algorithm as follows. First, let us suppose that all the ordered sets of k indices $\underset{\sim}{j} = [j_i]_{i=1}^{k}$ can be ordered in the sequence $[\underset{\sim}{j}_m]_{m=0}^{\infty} = [[j_{im}]_{i=1}^{k}]_{m=0}^{\infty}$, so that every k-tuple $\underset{\sim}{j}$ has a unique place (a value of m) in the sequence and, if $m \leq m'$, $g(\underset{\sim}{j}_m) \leq g(\underset{\sim}{j}_{m'})$. In fact, let the increasing sequence of indices $m_0 = 0, m_1, m_2, \cdots$ be defined by

$$m_t = \min \left\{ m:\ g(\underset{\sim}{j}_m) \geq t \right\}; \tag{101}$$

so that $g(\underset{\sim}{j}_m) = t$ for $m = m_t, m_t+1, m_t+2, \cdots, m_{t+1}-1$, and for no other values of m. In particular, when $[\varphi_j]_{j=0}^{\infty}$ is the sequence (47), we can easily verify that we can make $g(\underset{\sim}{j}) = \Sigma_{i=1}^{k} j_i = t$ in precisely $\binom{t+k-1}{k-1}$ ways. Thus, $m_0 = 0$, $m_1 = 1$ and, for $t \geq 2$, it can be seen that

$$m_t = \Sigma_{s=0}^{t-1} \binom{s+k-1}{k-1} = \binom{t+k-1}{k} . \tag{102}$$

So if $m_t \leq m < m_{t+1}$, we can define j_1 as the unique index such that

$$\binom{t+k-j_1-2}{k-1} \leq m - \binom{t+k-1}{k} < \binom{t+k-j_1-1}{k-1} , \qquad (103)$$

then j_2 as the unique index such that

$$\binom{t+k-j_1-j_2-3}{k-2} \leq m - \binom{t+k-1}{k} - \binom{t+k-j_1-2}{k-1} < \binom{t+k-j_1-j_2-2}{k-2} , \qquad (104)$$

and so on. Thus there is a unique representation of m in the form

$$m = \binom{t+k-1}{k} + \Sigma_{i=1}^{k} \binom{t+k-1-\Sigma_{h=1}^{i} j_h - i}{k-i} . \qquad (105)$$

This representation determines t and then $j_1, j_2, \cdots, j_k$. Generally such values will be denoted by $t = t(m)$, $j_i = q_i(m)$, where $i=1,2,\cdots k$. Clearly, $t(0) = q_i(0) = 0$, since $\psi_0 = \varphi_0^k$ in every case.

Now we proceed to the algorithm below.

```
        $z \leftarrow 0,\ h \leftarrow 1,\ t \leftarrow 0,\ \overline{m} \leftarrow m_1,\ m \leftarrow 0,$

(L1)    $p \leftarrow 0,$
(L2)    $i \leftarrow 1,$
(L3)    $q \leftarrow q_i(m),\ s_i \leftarrow 1,$
(L4)    $j \leftarrow$ ASIND $(f_i;y;\overline{L};s_i),$
        IF $j < q$ {$s_i \leftarrow s_i+1$, GO TO L4},
        IF $j = q$ {IF $i = k$ {GO TO L6}, $i \leftarrow i+1$, GO TO L3},      (106)
```

```
(L5)   $m \leftarrow m + 1$,  IF $m = \overline{m}$ {IF $p \neq 0$ {$z \leftarrow z + p \times \psi_t(y)$,
                                   IF $h = n$ {end of procedure},
                                   $h \leftarrow h + 1$}, $t \leftarrow t + 1$, $\overline{m} \leftarrow m_{t+1}$,
                                   GO TO L1},  GO TO L2,                              (106)
(L6)   $a \leftarrow 1$, $i \leftarrow 1$,                                                        cont'd
(L7)   $a \leftarrow a \times$ ASCOF $(f_i;y;\overline{L};s_i)$,
       IF $i = k$ {GO TO L8}, $i \leftarrow i + 1$,  GO TO L7,
(L8)   $p \leftarrow p + a$,  GO TO L5.
```

Here L refers to $[\psi_t]_{t=0}^{\infty}$ and $\overline{L}$ to $[\varphi_j]_{j=0}^{\infty}$, as specified in Lemma 13.

Step V

If our formula manipulation system has capabilities for integration or differentiation, we can easily use lemmas 14 and 16 to construct simple algorithms to perform (when appropriate) these operations on asymptotic expansions. These will be omitted here, since the algorithms are easy; but the theoretical pitfalls are considerable and are beyond the scope of this paper. More sophisticated asymptotic packages should eventually be able to handle this question.

Step VI

The final question, and the most complicated to be considered here, is that of computing (90) for $f = g^r$ by applying Lemma 15, when the sequence denoted by the label $\overline{L}$ is totally multiplicative. Again, we assume that the set of all possible products of the form (66) (for all values of k) can be ordered by an index m which identifies the integer k and the powers $i_1, i_2, \cdots, i_k$, and hence the index t and the function ψ_t, by

$$t = g_{i_1+i_2+\cdots+i_k}(1, i_1 \text{ times}; 2, i_2 \text{ times}; \cdots; k, i_k \text{ times}) = t(m) \qquad (107)$$

in such a way that, if $m \leq m'$, then $t(m) \leq t(m')$. We then write $k = w(m)$ and $i_h = v_h(m)$ $(h=1,2,\cdots,k)$, with $v_h(m) = 0$ for $h > k_m$. We note, as in (71), that, for any choice of t, the terms of the asymptotic expansion of $f = g^r$ which contain $\psi_0, \psi_1, \psi_2, \cdots, \psi_t$ arise only from the products in which k and $i_1+i_2+\cdots+i_k$ are bounded above (i.e., we have $\bar{k}_t$ and $\bar{s}_t$, such that $k \leq \bar{k}_t$ and $i_1+i_2+\cdots+i_k \leq \bar{s}_t$). Thus such an ordering is possible.

As in step IV, we could now define

$$\left.\begin{aligned} &m_t = \min \{m\colon t(m) \geq t\} \\ &\text{and} \\ &k_t = \max \{w(m)\colon t(m) \leq t\} \end{aligned}\right\} \tag{108}$$

and proceed as in (106). Although a systematic ordering of the products can be achieved, sometimes this can only be done recursively, the indexing of cases cannot be set up in advance. We now assume that this state of affairs prevails (the procedure followed below could also have been adopted in step IV by suitably modifying (106).)

The only nonzero terms in (66) arise from factors $(a_h\varphi_h/a_0\varphi_0)^{i_h}$ for which $a_h \neq 0$ (by (64)), $a_0 \neq 0$). Thus we limit ourselves to these factors, i.e., we assume that $i_h \neq 0$ only if $a_h \neq 0$. Instead of the ordering given above, we adopt a "diagonal" ordering of terms. If ASTER $(g;y;\bar{L};m) = a_{h_m}\varphi_{h_m}$ $(m = 1,2,3,\cdots)$, (66) is specified by k and the k-tuple $[i_{h_1}, i_{h_2}, \cdots, i_{h_k}]$, and now these terms are taken in the order (partly anticipated in (71))

[0], [1], [2], [0,1], [3], [1,1], [0,2], [0,0,1],
[4], [2,1], [1,2], [0,3], [1,0,1], [0,1,1], [0,0,2], [0,0,0,1],
[5], [3,1], [2,2], [1,3], [0,4], [2,0,1], [1,1,1], $\cdots$

The principle is to take, first, all $i_h = 0$ (this yields the leading term of the expansion $a_0{}^r \varphi_0{}^r$, corresponding to $p_0 = 1$ and $\psi_0 = 1$); then all terms with $k + s$ constant ($s = i_{h_1} + i_{h_2} + \cdots + i_{h_k}$) together, in the order of increasing $k + s$. For each constant $k + s$ ($k \geq 1$, $s \geq 1$), the order is that of increasing k. For each fixed pair $[k, s]$, the order is the reverse lexicographic one for $[i_{h_1}, i_{h_2}, \cdots, i_{h_k}]$; that is, the ordering is, first, by decreasing i_{h_1}; then, for each fixed i_{h_1}, by decreasing i_{h_2}; then, for each fixed pair $[i_{h_1}, i_{h_2}]$, by decreasing i_{h_3}; and so on; always satisfying the condition that s is a fixed sum, and that, while every $i_{h_j} \geq 0$, we must have $i_{h_k} \geq 1$. It is clear that this ordering of terms is well defined (every term will occur in the sequence, and no term will occur twice). Within each set of terms with fixed k and s, if the k-tuple $[i_{h_1}, i_{h_2}, \cdots, i_{h_k}]$ immediately precedes the

$$\left.\begin{array}{l}
k\text{-tuple } [i'_{h_1}, i'_{h_2}, \cdots, i'_{h_k}], \text{ if } i_{h_1} + i_{h_2} + \cdots + i_{h_{j-1}} + i_{h_j} = \hat{s}, \\
\text{and if } j \leq k-2 \text{ and} \\
\qquad i_{h_1} = i'_{h_1},\ i_{h_2} = i'_{h_2}, \cdots, i_{h_{j-1}} = i'_{h_{j-1}},\ i_{h_j} \neq i'_{h_j}; \\
\text{then} \\
\qquad i'_{h_j} = i_{h_j} - 1,\ i_{h_{j+1}} = i_{h_{j+2}} = \cdots = i_{h_{k-1}} = 0,\ i_{h_k} = s - \hat{s}, \\
\text{and} \\
\qquad i'_{h_{j+1}} = s - \hat{s},\ i_{h_{j+2}} = \cdots = i_{h_{k-1}} = 0,\ i_{h_k} = 1;
\end{array}\right\} \qquad (109)$$

but if $j = k-1$, then $i'_{h_j} = i_{h_j} - 1$ and $i'_{h_k} = i_{h_k} + 1$.

For a given pair (k, s), the least value of t is that corresponding to the k-tuple $[s-1, 0, \cdots, 0, 1]$, which is the first one in order. If this value is

$$t(k.s) = g_s(h_1, (s-1) \text{ times};\ h_k \text{ once}), \qquad (110)$$

then $t(k', s) > t(k, s)$ whenever $k' > k$, and $t(k, s') > t(k, s)$ whenever $s' > s$. Further, for a given $k + s$, the least value of t is dependent on the choice of the sequence at L. If this is

$$\overline{t}_u = \min\{t(k,s) : k + s = u,\ k \geq 1,\ s \geq 1\}\ , \tag{111}$$

then $\overline{t}_u \leq \overline{t}_{u'}$ if $u \leq u'$, and we know that terms with $k + s > u$ cannot contribute to p_t for $t < \overline{t}_u$. (Since the sequence is supposed to be totally multiplicative, it follows that we can make $\overline{t}_u$ as large as we like, by making u sufficiently large.)

On this basis, we may proceed with the algorithm as shown below.

```
        IF $n = 1\{z \leftarrow [\text{ASTER } (g;y;\overline{L};1)]^r$, end of procedure$\}$,
        $z \leftarrow 1$, $h \leftarrow 1$, $s \leftarrow 1$, $k \leftarrow 1$, $\ell \leftarrow 0$, $t_0 \leftarrow 0$, $a_0 \leftarrow$ ASCOF $(g;y;\overline{L};1)$,
(L1)    $f \leftarrow 1$, $w \leftarrow r$,
(L2)    $f \leftarrow f \times w$, IF $w > r - s + 1\{w \leftarrow w - 1$, GO TO L2$\}$, $j \leftarrow 1$,
(L3)    $m_j \leftarrow 0$, IF $j < k - 1\{j \leftarrow j + 1$, GO TO L3$\}$, $m_k \leftarrow 1$, $m_1 \leftarrow m_1 + s - 1$,
(L4)    $q \leftarrow f$, $j \leftarrow 1$,
(L5)    $h_j \leftarrow$ ASIND $(g;y;\overline{L};j+1)$, IF $m_j = 0$ $\{$GO TO L15$\}$, $b \leftarrow$ ASCOF
        $(g;y;\overline{L};j+1)$, $i \leftarrow 0$,
(L6)    $q \leftarrow q \times b/((m_j-1) \times a_0)$, IF $i < m_j-1\{i \leftarrow i + 1$, GO TO L6$\}$,        (112)
(L15)   IF $j < k\{j \leftarrow j + 1$, GO TO L5$\}$,
        $t \leftarrow g_s(h_1, m_1\text{times}; h_2, m_2\text{times}; \cdots; h_k, m_k\text{times})$, $i \leftarrow 0$,
(L7)    IF $t = t_i\{p_i \leftarrow p_i + q$, GO TO L8$\}$,
        IF $t < t_i$ $\{$ $j \leftarrow \ell$, $\ell \leftarrow \ell + 1$,
(L14)             $p_{j+1} \leftarrow p_j$, $t_{j+1} \leftarrow t_j$, IF $j > i$ $\{j \leftarrow j - 1$, GO TO L14$\}$,
                  $p_i \leftarrow q$, $t_i \leftarrow t$, GO TO L8 $\}$,
        IF $i < \ell$ {$i \leftarrow i + 1$, GO TO L7}, $\ell \leftarrow \ell + 1$, $p_\ell \leftarrow q$, $t_\ell \leftarrow t$,
(L8)    IF $k = 1$ {GO TO L10}, IF $m_{k-1} \neq 0$ {$m_{k-1} \leftarrow m_{k-1} -1$, $m_k \leftarrow m_k + 1$,
        GO TO L4}, IF $k = 2$ {GO TO L10}, $j \leftarrow k - 2$,
(L9)    IF $m_j = 0$ {IF $j > 1$ {$j \leftarrow j - 1$, GO TO L9}, GO TO L10},
        $m_j \leftarrow m_j - 1$, $m_{j+1} \leftarrow m_k$, $m_k \leftarrow 1$, GO TO L4,
(L10)   IF $s > 1$ {$s \leftarrow s - 1$, $k \leftarrow k + 1$, GO TO L1}, $i \leftarrow 1$, $j \leftarrow 0$,
(L11)   IF $i > \ell$ {GO TO L13},
        IF $t_i < \overline{t}_{k+2}$ {IF $p_i \neq 0$ {$z \leftarrow z + p_i \times \psi_{t_i}(y)$, $h \leftarrow h + 1$,
                                IF $h = n$ {$z \leftarrow z \chi$ ASTER $(g;y;\overline{L};1)]^{\overline{r}}$,
                                        end of procedure}},
                    $i \leftarrow i + 1$, GO TO L11},
(L12)   IF $i \leq \ell$  $j \leftarrow j + 1$, $p_j \leftarrow p_i$, $t_j \leftarrow t_i$,{$i \leftarrow i + 1$, GO TO L12},
(L13)   $\ell \leftarrow j$, $s \leftarrow k + 1$, $k \leftarrow 1$, GO TO L1.
```

(112 cont'd)

Some explanation is indicated. The variables z, a_0, k, and s have the same meaning as in the discussion. What is calculated in z is initially $\Sigma_{t=0}^{c} p_t \psi_t$, such that exactly n of the coefficients p_t with $0 \leq t \leq c$ are nonzero (including p_c). The last instruction before "end of procedure" (between L11 and L12) then multiplies the sum by $a_0{}^r \varphi_0(y)^r$, as required. Initially,

$$z = 1 = p_0 \psi_0. \tag{113}$$

The number of nonzero terms of the sum already accumulated in z is denoted by h, which is increased whenever a nonzero term is added to z (between L11 and L12). The number of terms of the sum $\Sigma_{t=0}^{c} p_t \psi_t$ which have not yet been added to z, but to which some product (66) has made a contribution is denoted by ℓ. The arrays $[t_1, t_2, \cdots, t_\ell]$ and $[p_1, p_2, \cdots, p_\ell]$ are the corresponding indices and coefficients of ψ_t (as accumulated so far). In L1 and L2, we have the computation of the factor $r(r-1)\cdots(r-s+1)$ occurring in (66), for each new value of s . At L3, we compute the initial k-tuple for each new value of the pair $[k,s]$, namely, $[s]$ if $k = 1$ or $[s-1,0,\cdots,0,1]$ if $k > 1$. This is stored as the array $[m_1, m_2, \cdots, m_k]$ (i.e., $m_j = i_{h_j}$). Between L4 and L7, the complete product (66) is computed; then the index t is calculated. If this index occurs as a t_i $(1 \le i \le \ell)$, the product is added to the corresponding p_i ; while if this t is new, ℓ is increased by 1, the value of t is assigned to the new t_ℓ, and the product is taken as the new p_ℓ (between L7 and L8). If a further k-tuple exists (for the same $[k, s]$), we advance according to the rule (109) (at L8, L9); if no more exist but $s > 1$, we change $[k, s]$ to $[k + 1, s - 1]$ and start calculations on the products belonging to these new parameters; and if $s = 1$, so that all terms with a given $k + s = u$ have been dealt with, then we compute $\bar{t}_{u+1}$ (which is $\bar{t}_{k+2}$ at this point), and search through the list of ℓ terms for those which have been completed $(t_i < \bar{t}_{u+1})$, while adding the nonzero terms to z, discarding all terms with $t_i < \bar{t}_{u+1}$ from the list, and moving all remaining terms up in the list ("garbage-disposal"), so that the list will not become too large (between L11 and L13). The new value of $k + s = u + 1$ is then initiated with $k = 1$ and $s = u$, and the computation continues. When z contains the required number, n, of nonzero terms, the procedure terminates.

In particular, when L refers to a sequence (47), then

$$g_s(h_1, m_1\text{times}; h_2, m_2\text{times}; \cdots; h_k, m_k\text{times}) = \Sigma_{j=1}^{k} h_j m_j \tag{114}$$

and so

$$t(k, s) = (s - 1)h_1 + h_k, \tag{115}$$

whence

$$\overline{t}_u = (u - 1)\, h_1 + \min\{h_k - kh_1 : k = 1, 2, \cdots, u - 1\} \le (u - 1)h_1. \qquad (116)$$

We could still proceed as in (112). But now it is also possible to set up the ordering of terms by value of t, so that a computation similar in style to (106) can be set up. Even better, if $\varphi_0 = 1$, we can use the algorithm of Miller, whose essential principle is contained in the recurrence relation (76). These algorithms are straightforward and will not be explicitly displayed here.

Step VII

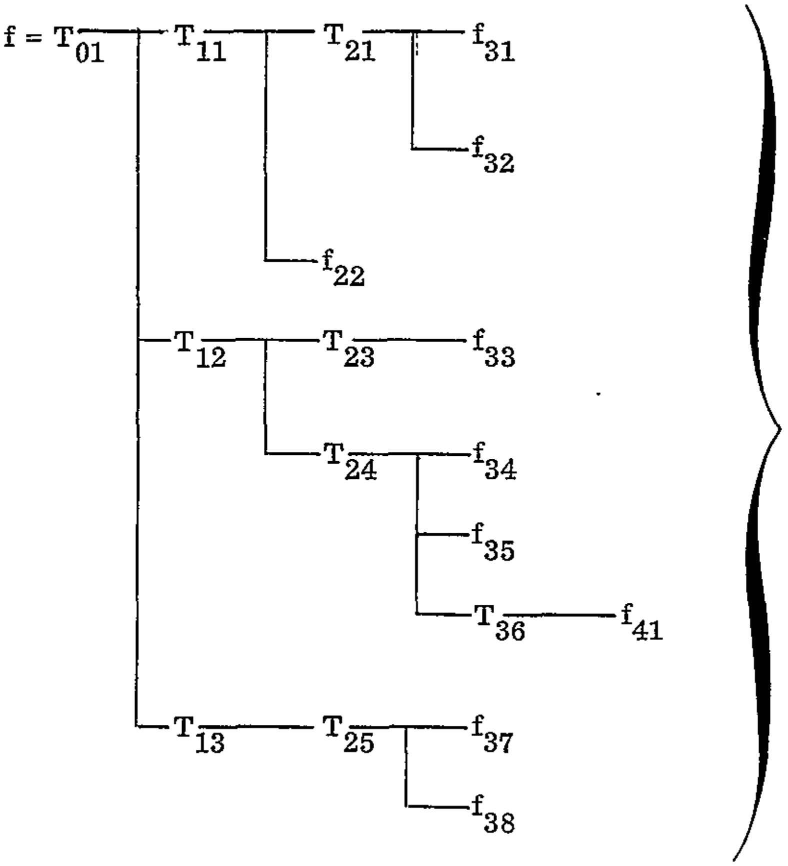
Let us now suppose that the algorithms described in steps II through VI are available to us. That is, suppose that, if

$$f = T(f_1, f_2, \cdots, f_m), \qquad (117)$$

then

$$\begin{aligned} &\text{ASYMP}\ (f; x, \alpha; L; n) = \Psi_T\,(\text{ASYMP}\ (f_1; x, \alpha; \overline{L}; n_1), \\ &\text{ASYMP}\ (f_2; x, \alpha; \overline{L}; n_2), \cdots, \text{ASYMP}\ (f_m; x, \alpha; \overline{L}; n_m)), \end{aligned} \qquad (118)$$

where the sequence $\overline{L}$, the numbers $n_1, n_2, \cdots, n_m$, and the function Ψ_T are directly computable by the algorithms contained in the asymptotic package, given that the ASYMP $(f_h; x, \alpha; \overline{L}; n_h)$ $(h = 1, 2, \cdots, m)$ can themselves be computed. In that case, it is necessary, first, to unravel f into a tree structure, which can be illustrated by the example below.



(119)

This diagram is to be interpreted as meaning that $f = T_{01}(f_{11},f_{12},f_{13})$; $f_{11} = T_{11}(f_{21},f_{22})$, $f_{12} = T_{12}(f_{23},f_{24})$, $f_{13} = T_{13}(f_{25})$; $f_{21} = T_{21}(f_{31},f_{32})$, $f_{23} = T_{23}(f_{33})$, $f_{24} = T_{24}(f_{34},f_{35},f_{36})$, $f_{25} = T_{25}(f_{37},f_{38})$, $f_{36} = T_{36}(f_{41})$; and f_{22}, f_{31}, f_{32}, f_{33}, f_{34}, f_{35}, f_{37}, f_{38}, and f_{41} are elementary (as defined in step II). Since the "tree" is listed in the computer system, the asymptotic expansions of the elementary "twigs" are computed, and then the algorithms described earlier are used to work down the tree to its "root," the function f.

Step VIII

One major problem remains. If multiplications, powers, differentiations, or integrations occur in the tree exemplified by (118), the asymptotic sequence $\bar{L}$ occuring in the component functions (arguments of T) changes (in a predictable way) into a sequence L for the composite function. The reverse process of obtaining $\bar{L}$ from L is not so easily performed. What is more, the question of choosing an asymptotic sequence L is not adequately touched on in the theoretical literature; but is assumed to have been done a priori. From a practical viewpoint, it is clear that a process somewhat as follows is required.

We begin with a collection of "easy" functions: a suitable initial collection might contain x^{r_0}, $[(\exp)^{p_1}x]^{r_1}$, $[(\log)^{p_2}x]^{r_2}$; for all positive integers p_i and all real r_i. We know that, as $x \to \infty$,

$$\left.\begin{array}{l}
\text{if } r_0 < r'_0,\ x^{r_0} = \underline{o}(x^{r'_0}); \\[1em]
\text{if } r_1 < r'_1,\ [(\exp)^{p_1}x]^{r_1} = \underline{o}([\exp)^{p_1}x]^{r'_1}); \\
\text{if } p_1 < p'_1,\ [(\exp)^{p_1}x]^{r_1} = \underline{o}([\exp)^{p'_1}x]^{r'_1}),\ \text{for all } r_1 > 0,\ r'_1 > 0; \\[1em]
\text{if } r_1 < 0,\ r'_1 > 0,\ [(\exp)^{p_1}x]^{r_1} = \underline{o}([\exp)^{p'_1}x]^{r'_1}),\ \text{for all } p_1, p'_1; \\[1em]
\text{if } r_1 > 0,\ x^{r_0} = \underline{o}([\exp)^{p_1}x]^{r_1}),\ \text{for all } r_0, p_1; \\
\text{if } r_1 < 0,\ [(\exp)^{p_1}x]^{r_1} = \underline{o}(x^{r_0}),\ \text{for all } r_0, p_1; \\[1em]
\text{if } r_2 < r'_2,\ [(\log)^{p_2}x]^{r_2} = \underline{o}([\log)^{p_2}x]^{r'_2}); \\[1em]
\text{if } p_2 > p'_2,\ [\log)^{p_2}x]^{r_2} = \underline{o}([\log)^{p'_2}x]^{r'_2}),\ \text{for all } r_2 > 0,\ r'_2 > 0;
\end{array}\right\} \quad (120)$$

$$\left.\begin{array}{l} \text{if } r_2 < 0,\ r'_2 > 0,\ \ [(\log)^{p_2}x]^{r_2} = \underline{o}([\log)^{p'_2}x]^{r'_2}),\ \ \text{for all } p_2, p'_2; \\ \text{if } r_0 > 0,\ [(\log)^{p_2}x]^{r_2} = \underline{o}(x^{r_0}),\ \ \text{for all } p_2, r_2; \\ \text{if } r_0 < 0,\ x^{r_0} = \underline{o}([(\log)^{p_2}x]^{r_2}),\ \ \text{for all } p_2, r_2\ . \end{array}\right\} \begin{array}{c}(120\\ \text{cont'd})\end{array}$$

Thus all these functions have a single ordering. Now, given any function f, we use the limit subprogram to investigate whether any of these functions (call it φ_0) is commensurate with f. If any is commensurate with f, we put $f \approx a_0\varphi_0 + f_1$, determining a_0 as usual (by (44)), and look at f_1 similarly; and so on. A refinement would be to look, not only at the functions listed, but also at products of these functions. If f is not commensurate with any of the functions listed, then we add f to the list (together with its powers f^{r_0}, $[(\exp)^{p_1}f]^{r_1}$, and possibly $[(\log)^{p_2}f]^{r_2}$) . Then $f \approx f$, for our purposes.

This procedure allows the system to "learn" an ever-increasing collection of functions for constructing asymptotic sequences. In addition, it computes natural asymptotic expansions, in the sense that, for instance, $f = 1 + e^{-x}(\frac{x}{x-1})$ would expand to

$$f \approx 1 + e^{-x} + x^{-1}e^{-x} + x^{-2}e^{-x} + \cdots \tag{121}$$

to as many terms as required, rather than the relatively artificial and uninteresting result

$$f \approx 1 + 0.x^{-1} + 0.x^{-2} + 0.x^{-3} + \cdots , \tag{122}$$

which adherence to a given sequence (here, (38)) yields. Although this proposal is tentative and needs further investigation, it seems to be an approximation to the proper approach.

6. CONCLUSION

We have now reviewed the theory of asymptotic series, and considered fairly completely the problems and procedures to be met in the implementation of an asymptotic package for a formula manipulation system. Many questions still remain unanswered or only partly answered: these are indicated, as they arise, in Section 5. However, we hope that we have established enough to enable a programmer to begin writing a basic asymptotic package for an existing system, such as FORMAC, as soon as a limit-subprogram (however rudimentary) has been written. (Initially, it may be necessary to limit consideration to asymptotic power series, based on the sequence (38), and to handle only polynomials, power series, and the results of applying addition, subtraction, multiplication, division, and the formation of arbitrary powers. In that case, the limit-subprogram would be very easy to write and could be part of the asymptotic package.)

It is the author's belief that indeed a working program, however elementary and limited, will prove to be of great help in engendering and encouraging subsequent development of more ambitious systems having greater scope and power. It is in this spirit that the present paper is offered.

Note: Dr. Ralph L. London and the author have examined the algorithms (91), (97), (99), (100), (106), and (112), and have rigorously proved their validity. In the course of discussing the process of proof, quite a number of corrections were found to be necessary (this illustrates the great value of such proving procedures.) The proofs are to be published at a future time.

REFERENCES

1. E. T. Copson, "Asymptotic Expansions," Cambridge Tracts in Mathematics and Mathematical Physics, No. 55, Cambridge University Press, Cambridge, England, 1965.

2. N. G. DeBruijn, "Asymptotic Methods in Analysis," Bibliotheca Mathematica, Vol. IV, North-Holland Publishing Co., Amsterdam and P. Noordhoff Ltd., Groningen, 1961, 2nd. Edition.

3. A. Erdélyi, Asymptotic Expansions, Dover Publications Inc., New York, 1956; also Technical Report No. 3, Office of Naval Research No. NR 043-121, Contract No. Nonr-220(11), Washington, D.C.

4. W. R. Wasow, "Asymptotic Expansions for Ordinary Differential Equations," Pure and Applied Mathematics, Vol. XIV, Interscience Publishers, John Wiley & Sons Inc., 1965.

5. H. Poincaré, Acta Mathematica, Vol. 9, 1886, pp. 295-344.

6. T. J. Stieltjes, Annales Scientifiques de l'Ecole Normale Superieure (3), Vol. 3, 1886, pp. 201-258 and Oeuvres Completes de T. J. Stieltjes, Vol. 2, P. Noordhoff Ltd., Groningen, 1918, pp. 2-58.

7. P. Henrici, "Automatic computation with power series," Journal of the Association for Computing Machinery, Vol. 3, 1956, pp. 10-15.

8. J. R. Iturriaga, "Contributions to Mechanical Mathematics," Ph.D. Thesis, Carnegie Institute of Technology, Pittsburgh, Pennsylvania, April 1967.

COMPUTING TIME ANALYSES FOR SOME ARITHMETIC AND ALGEBRAIC ALGORITHMS

by

George E. Collins
Computer Sciences Department
The University of Wisconsin
Madison, Wisconsin

Abstract

Computing time bounds are derived for the author's polynomial reduced sequence (p.r.s) algorithm[6] for computing the g.c.d. of two polynomials with integer coefficients. A new g.c.d. algorithm which uses congruence arithmetic is presented. The computing time of the new algorithm is analyzed, and it is shown to be more efficient than the old one. Using easily obtained bounds for operations on large integers, computing time bounds are derived for the integer Euclidean algorithm, èxtended Euclidean algorithm, and the Chinese remainder theorem algorithm.

This research was supported by the Wisconsin Alumni Research Foundation, by the University of Wisconsin Graduate School through the University of Wisconsin Computing Center, and by the National Science Foundation, Office of Computing Activities, through Grant GJ239.

COMPUTING TIME ANALYSES FOR SOME ARITHMETIC AND ALGEBRAIC ALGORITHMS

by

George E. Collins

1. INTRODUCTION

In a recent paper (reference 6), the author presented a new algorithm, the reduced polynomial remainder sequences (p.r.s) algorithm, for computing the the greatest common divisor (g.c.d.) of two multivariate polynomials with integer coefficients. It was asserted that the computing time for this algorithm, when applied to two univariate polynomials of degree n whose coefficients are d digits long, is approximately proportional to n^4 d^2. Section 3 contains a thorough and rigorous analysis of the computing time for this algorithm in the univariate case, and proof is given that, if the two polynomials are weakly normal and d bounds their norms, the computing time is bounded by the function $O(n^4(\ln d)^2)$, the norm of a polynomial being the sum of the absolute values of its coefficients. This result is obtained as a corollary of a more general theorem which bounds the computing time as a function of four variables.

Section 4 presents a new and faster algorithm for computing the g.c.d. of two univariate polynomials with integer coefficients, and proves several theorems to show that the algorithm always terminates and produces the greatest common divisor. The new algorithm, which uses congruence arithmetic (arithmetic performed in finite fields GF(p) containing a prime number of elements), is based on the theory of reduced polynomial remainder sequences and subresultants that the author developed in reference 6.

In section 5, the computing time of the new algorithm is analyzed. The bound that is obtained for the computing time of the new algorithm is $O(n^4(\ln d) + n^3 (\ln d)^2)$. We also show that the average computing time for the new algorithm is $O(n^3(\ln d)^2)$, a substantial improvement over previous algorithms. Furthermore, we show that the average computing time for the new algorithm, when applied to polynomials with a g.c.d. of degree zero (which frequently occurs in practice), is $O(n^2 + n(\ln d)^2)$. Finally, we propose a method for extending the algorithm to multivariate polynomials.

To analyze the computing times of polynomial algorithms, we must have bounds for the computing times required to perform operations on large integers. Such bounds, easily obtained for the operations of addition, subtraction, multiplication, and division (with or without a remainder), are stated in section 2. We also obtain bounds, in section 2, for the time required to compute the g.c.d. of two integers using the Euclidean algorithm, and show that these bounds also apply to the extended Euclidean algorithm.

In the congruence arithmetic g.c.d. algorithm, as in other congruence arithmetic algorithms (see, for examples, references 1 and 14), one must apply the Chinese remainder theorem algorithm. A bound for the computing time of this algorithm is also derived in section 2.

2. OPERATIONS ON LARGE INTEGERS

Throughout this paper we assume that integers are represented in radix form using an arbitrary base $\beta \geq 2$. Computing times for arithmetic algorithms are then naturally expressed as functions of the number of β-digits in certain numbers N which occur in the algorithms, i.e., $[\log_\beta N] + 1$. However, since $\log_\beta N = (\ln N)/(\ln \beta)$ where ln is the natural logarithm function and since we will uniformly ignore constant multipliers because they are dependent (1) on the computer that is used, (2) on numerous details of the version of the algorithm used, and (3) on the precise manner in which data is represented in the computer, we shall express computing times in terms of ln N.

The following theorem on addition and subtraction illustrates the general form in which our theorems will be stated.

Theorem 2.1

Let $t(a, b)$ be the time required to compute $a + b$ (or $a - b$). Let $T(d) = \max \{t(a, b): |a|, |b| \leq d\}$. Then $T(d) = O(\ln d)$.

The statement that $T(d) = O(\ln d)$ means that a constant C (independent of d) exists such that $T(d) \leq C \ln d$ for all sufficiently large d. The theorem still appears to be quite ambiguous since it does not specify what algorithm or what computer is to be used. However, if we choose any standard classical algorithm, any familiar data representation which uses radix canonical form, and any well-known computer, then such a C will exist for that combination of choices. The theorem can be readily verified for several choices by consulting references 5, 8, and 13.

The next theorem states a similarly well-known fact about various classical multiplication algorithms.

Theorem 2.2

Let $t(a, b)$ be the time to compute $a \cdot b$. Let $T(d, e) = \max \{t(a, b): |a| \leq d \text{ and } |b| \leq e\}$. Then $T(d, e) = O((\ln d)(\ln e))$.

Here we have applied the O-notation to a function $T(d, e)$ of more than one variable. This means that, for some C, $T(d, e) \leq C(\ln d)(\ln e)$ whenever $d \geq d_0$ and $e \geq e_0$.

Theorem 2.2 applies to the classical multiplication algorithms. As a special case we have $T(d, d) = O((\ln d)^2)$, although the stronger form of the theorem, which contains two variables, will be important in many applications in which one argument is much smaller than the other. In recent years several multiplication algorithms, which are much faster for very large integers, have been devised. One such fast algorithm, based on earlier work by A. L. Toom, is given by Cook in reference 11. It has the property that $T(d, d) = O((\ln d) \cdot 2^{5\sqrt{\ln(\ln d)}})$. It follows that, for every $\epsilon > 0$, $T(d, d) = O((\ln d)^{1+\epsilon})$.

It is easy to construct a simple version of this algorithm for which $T(d, d) = O((\ln d)^{\ln_2 3}) = O((\ln d)^{1.585})$. Throughout the present paper, however, we assume that the classical multiplication algorithm is employed, and Theorem 2.2 is applied.

So far, we have bounded computing times as functions of bounds on the inputs to the algorithms. A tight bound on the time for division must be expressed in terms of bounds on one input (the divisor) and one output (the quotient), as in Theorem 2.3.

Theorem 2.3

Let $t(a, b)$ be the time to compute q and r, given a and b such that $a = b \cdot q + r$, $0 \leq |r| < |b|$, $ar \geq 0$, and $abq \geq 0$. Let $T(d, e) = \max \{t(a, b): |b| \leq d \text{ and } |q| < e\}$. Then $T(d, e) = O((\ln d)(\ln e))$.

The truth of Theorem 2.3 follows from the observation that most of the computation required to produce q and r from a and b is essentially the same as that required to produce a from b, q, and r.

The bound for the computing time for the Euclidean algorithm is derived from Lemma 2.4, which bounds the product of the quotients computed by the algorithm.

Lemma 2.4

Let $q_1, q_2, \ldots, q_n$ be the sequence of quotients obtained when the Euclidean algorithm is applied to a and b where $a \geq b > 0$. Let $c = \gcd(a, b)$. Then $q_n \prod_{i=1}^{n-1} (q_i + 1) < ab/c^2$.

Proof. Let $a_1 = a$, $a_2 = b$, $a_i = a_{i+1} q_i + a_{i+2}$ with $0 \leq a_{i+2} < a_{i+1}$ for $1 \leq i \leq n$, and $a_{n+2} = 0$ so that $c = a_{n+1}$. Then $a_i = a_{i+1} q_i + a_{i+2} >$. $a_{i+2} q_i + a_{i+2} = a_{i+2} (q_i + 1)$ for $1 \leq i \leq n$. Taking the product of these inequalities for $1 \le i \le n-1$, we have $\Pi_{i=1}^{n-1} a_i > \Pi_{i=1}^{n-1} a_{i+2}(q_i + 1)$. When we cancel the a_i's on both sides, we obtain $ab = a_1 a_2 > a_n c^2 \Pi_{i=1}^{n-1} (q_i + 1)$. But since $a_n = q_n a_{n+1} = q_n c$, $ab > q_n c^2 \Pi_{i=1}^{n-1} (q_i + 1)$.

As an interesting sidelight, notice that each $q_i \ge 1$ and $q_n \ge 2$ except when $a = b$. Hence, $2^n \le q_n \Pi_{i=1}^{n-1} (q_i +1) < ab/c^2 \le ab < a^2$ so that $n < 2 \log_2 a$. This bound for the number of divisions compares with a bound of about $1.44 \log_2 a$ obtained from Lamé's theorem.

Theorem 2.5

Let $t(a, b)$ be the computing time for the Euclidean algorithm. Let $T(d, e) = \max\{t(a,b): a > b > 0 \text{ and } b \le d \text{ and } a/\gcd(a, b) \le e\}$. Then $T(d, e) = O((\ln d)(\ln e))$.

Proof. Let $c = \gcd(a, b)$. By Theorem 2.3, there exist constants C_1 and C_2 such that the time for the i^{th} division in Euclid's algorithm is $\le C_1 (\ln q_i)(\ln a_{i+1}) + C_2$. Hence, the time for all divisions is $\le C_1 \Sigma_{i=1}^{n} (\ln q_i)(\ln a_{i+1}) + C_2 n \le C_1 (\ln b) \Sigma_{i=1}^{n} \ln q_i + C_2 n = C_1 (\ln b)(\ln \Pi_{i=1}^{n} q_i) + C_2 n < C_1 (\ln b)(\ln(ab/c^2)) + 2 C_2 \log_2 a < C_1 (\ln b)(\ln (a^2/c^2)) + 3 C_2 \ln a \le 2 C_1 (\ln d)(\ln e) + 3 C_2 \ln ce \le 2 C_1 (\ln d)(\ln e) + 3 C_2 \ln d + 3 C_2 (\ln e) = O((\ln d)(\ln e))$.

In analyzing the computing time for the Chinese remainder theorem algorithm below, we shall also need a bound for the time required to compute the extended Euclidean algorithm which, given a and b where $a \ge b > 0$, computes not only $c = \gcd(a, b)$ but also, simultaneously, integers x and y such that $ax + by = c$. The extended Euclidean algorithm may be defined in the manner that follows (see references 12 and 10.) Let the a_i and q_i be defined as above.

Set $x_1 = 1$, $y_1 = 0$, $x_2 = 0$, and $y_2 = 1$. For $1 \le i \le n - 1$, let $x_{i+2} = x_i - q_i x_{i+1}$ and $y_{i+2} = y_i - q_i y_{i+1}$. Then $x = x_{n+1}$ and $y = y_{n+1}$.

Theorem 2.6

Let $t(a, b)$ be the computing time for the extended Euclidean algorithm. Let $T(d) = \max \{t(a,b)\colon a > b > 0 \text{ and } b \le d \text{ and } a/\gcd(a,b) \le d\}$. Then $T(d) = O((\ln d)^2)$.

Proof. The additional computing time for the extended Euclidean algorithm is that required to compute x_{i+2} and y_{i+2} for $1 \le i \le n - 1$. It was shown in reference 10 that $| x_i | \le b/2c$ and $| y_i | \le a/2c$ for all i. Hence, the time for all the multiplications $q_i x_i$ is, by Theorem 2.2, $\le C_1 \sum_{i=1}^{n-1} (\ln q_i)(\ln x_{i+1}) + C_2 n \le C_1 \ln(b/c) \sum_{i=1}^{n-1} \ln q_i + C_2 n < C_1 \ln(b/c) \ln(ab/c^2) + C_2 n = o((\ln d)^2)$, as in the proof of Theorem 2.5. It is also shown in reference 10 that the x_i (also the y_i) alternate in sign. Hence, $| q_i x_{i+1} | \le | x_{i+2} | \le b/c$ and $| q_i y_{i+1} | \le | y_{i+2} | \le a/c$ for all i. Each subtraction $x_i - q_i x_{i+1}$ can therefore be performed in $C_1(\ln| x_{i+2}|) + C_2 \le C_1 \ln d + C_2$ time units. Since $2^n \le d^2$, the time for all such subtractions is $O((\ln d)^2)$. Likewise, the time to compute all y_i is $O((\ln d)^2)$.

In the application of the extended Euclidean algorithm in the Chinese remainder theorem, as in many other applications, a is a prime number p, so that $c = \gcd(a, b) = 1$. We then have $px + by = 1$, i.e., $by \equiv 1 \pmod p$. Since $| y | \le p/2$, the inverse of b in $GF(p)$ is y or $y + p$ according to whether $y > 0$ or $y < 0$.

Our next theorem bounds the computing time for the Chinese remainder theorem algorithm. The computations performed in this algorithm to make the theorem more precise are summarized below. These steps are also referred to in the proof of the theorem. Input to the algorithm includes a sequence $(p_1, p_2, \ldots, p_n)$ of pairwise relatively prime numbers. We shall assume each $p_i \geq 2$. In our application below, the p_i will be prime numbers, but this need not be assumed in the theorem. Additional input to the algorithm is a corresponding sequence $(a_1, a_2, \ldots, a_n)$, such that $0 \leq a_i < p_i$ for all i. The output of the algorithm is the unique integer A such that $|A| \geq p_1 \cdot p_2 \ldots p_n/2$ and $A \equiv a_i \pmod{p_i}$ for all i.

Chinese Remainder Theorem Algorithm

1. Set $Q_1 = p_1$ and compute $Q_i = Q_{i-1} \cdot p_i$ for $2 \leq i \leq n$. Set $P = Q_n$.
2. Compute $P_i = P/p_i$ for $i = 1, 2, \ldots, n$.
3. Compute q_i and r_i such that $P_i = p_i q_i + r_i$ and $0 \leq r_i < p_i$ for $1 \leq i \leq n$.
4. Compute t_i such that $r_i t_i \equiv 1 \pmod{p_i}$ and $0 < t_i < p_i$ for $1 \leq i \leq n$.
5. Compute $S = \Sigma_{i=1}^{n} P_i t_i a_i$.
6. Compute Q and R such that $S = PQ + R$, $0 \leq R < P$.
7. Compute $H = [P/2]$.
8. Set $A = R - P$ if $R > H$; otherwise, set $A = R$.

Theorem 2.7

Let $t(p_1, p_2, \ldots, p_n, a_1, a_2, \ldots, a_n)$ be the computing time for the Chinese remainder theorem algorithm. Let $T(d) = \max \{ t(p_1, \ldots, p_n, a_1, \ldots, a_n) : \Pi_{i=1}^{n} p_i \leq d \}$. Then $T(d) = O((\ln d)^2)$.

Proof. It suffices to show that the time for each of the eight steps is $O((\ln d)^2)$. The time for step 1 is $\le \Sigma_{i=2}^{n} (C_1(\ln Q_{i-1})(\ln p_i) + C_2) \le C_1 (\ln P) \Sigma_{i=2}^{n} (\ln p_i) + C_2 n \le C_2 (\ln d)(\ln \Pi_{i=2}^{n} p_i) + C_2 n \le C_2 (\ln d)^2 + C_2 n = O((\ln d)^2)$, since $2^n \le P \le d$ so that $n = O(\ln d)$.

The time for step 2 is $\le \Sigma_{i=1}^{n} (C_1 (\ln P_i)(\ln p_i) + C_2) \le C_1 (\ln d) \Sigma_{i=1}^{n} (\ln p_i) + C_2 n = O((\ln d)^2)$. The time for step 3 is $\le \Sigma_{i=1}^{n} C_1((\ln p_i)(\ln q_i) + C_2) \le C_1 (\ln d) \Sigma_{i=1}^{n} (\ln p_i) + C_2 n = O((\ln d)^2)$. By Theorem 2.6, the time for step 4 is $\le \Sigma_{i=1}^{n} (C_1(\ln p_i)^2 + C_2) \le C_1 \Sigma_{i=1}^{n} (\ln p_i)^2 + C_2 n \le C_1 (\Sigma_{i=1}^{n} (\ln p_i))^2 + C_2 n \le C_1 (\ln d)^2 + C_2 n = O((\ln d)^2)$.

The time to compute all products in step 5 is clearly $O(\ln d)^2)$, since $t_i < p_i$, $a_i < p_i$, $P_i < P$, $P_i t_i < P$ and $P < d$. Let $S_j = \Sigma_{i=1}^{j} P_i t_i a_i$. Then $S_j \le P \Sigma_{i=1}^{j} a_i < P \Sigma_{i=1}^{j} p_i \le P \Pi_{i=1}^{j} p_i \le P^2 \le d^2$. Hence, the time for all additions is $\le n (C_1 (\ln d^2) + C_2) = n(2 C (\ln d) + C_2) = O((\ln d)^2)$.

In step 6, $S = S_n < P^2$, so $Q < P$ and, hence, the time is $O((\ln d)^2)$. The time for steps 7 and 8 are clearly $O(\ln d)$.

3. OPERATIONS ON UNIVARIATE POLYNOMIALS

The primary purpose of this section is to give a relatively complete analysis of the time required to compute the g.c.d. of two univariate polynomials using the reduced p.r.s. algorithm of reference 6. And we also bound the computing time for other operations on univariate polynomials with integer coefficients.

It is very useful, for the purpose of such analyses, to define the norm of such a polynomial in the manner that follows.

Definition 3.1. If $P(x) = \Sigma_{i=0}^{n} a_i x^i$ is a polynomial with integer coefficients, the norm of P is defined to be $\Sigma_{i=0}^{n} | a_i |$.

Norm (P) is, in fact, a norm for the ring of polynomials over the integers, as shown by the following theorem.

Theorem 3.2

Norm $(P + Q) \leq$ norm (P) + norm (Q) · norm $(P \cdot Q) \leq$ norm (P) · norm (Q).

Proof. The first part is trivial. Let $P(x) = \Sigma_{i=0}^{m} a_i x^i$, $Q(x) = \Sigma_{i=0}^{n} b_i x^i$, and $R(x) = \Sigma_{i=0}^{m+n} c_i x^i$ where $R = P \cdot Q$. Then norm $(R) = \Sigma_{k=0}^{m+n} | c_k | =$ $\Sigma_{k=0}^{m+n} | \Sigma_{i\ j=k} a_i b_j | \leq \Sigma_{k=0}^{m+n} \Sigma_{i+j=k} |a_i b_j| = \Sigma_{i=0}^{m} \Sigma_{j=0}^{n} | a_i b_j | = \Sigma_{i=0}^{m} |a_i|$ $\Sigma_{j=0}^{n} | b_j | =$ norm (P) · norm (A).

The norm has two other important properties which are frequently used in the material that follows: $|a_i| \leq$ norm (P) for all i and norm $(P) \geq (\Sigma_{i=0}^{m} a_i^2)^{1/2}$. The following notation is frequently useful for simultaneously bounding norm (P) and deg (P).

Definition 3.3. $U(d, m) = \{P$: norm $(P) \leq d$ and deg $(P) \leq m\}$.

The following three theorems bound the computing times for the sum, difference, product, or quotient of two polynomials, using the classical algorithms. It is assumed here that the polynomials are represented by a canonical form as in reference 2, 5, 9, or 13.

Theorem 3.4

The time to compute $P + Q$ or $P - Q$ for $P, Q \in U(d, m)$ is $O((\ln d)m)$.

Here we have, for the first time, stated a theorem using a more elliptic phraseology. If stated in full, the theorem would have the same form as our previous theorems, i.e., let $t(P, Q)$ be the time to compute $P + Q$ (or $P - Q$). Let $T(d, m) = \max \{ t(P, Q) : P, Q \in U(d, m) \}$. Then $T(d, m) = O((\ln d)m)$.

Proof. At most, $m + 1$ coefficient additions or subtractions are required, and each takes $\le C_1 \ln d + C_2$ computing time. But $(m + 1)(C_1 \ln d + C_2) = O((\ln d)m)$.

Theorem 3.5

The time to compute $P \cdot Q$ for $P \in U(d, m)$ and $Q \in (e, n)$ is $O((\ln d)(\ln e)mn)$.

Proof. At most, $(m + 1)(n + 1)$ coefficient multiplications are required, the time for each being $\le C_1 (\ln d)(\ln e) + C_2$. Also, at most, $(m + 1)(n + 1)$ additions are required and, by the proof of Theorem 3.2, the time for each addition is $\le C_3 \ln de + C_4 = C_3 (\ln d + \ln e) + C_4$. But $(m + 1)(n + 1)(C_1 (\ln d)(\ln e) + C_3 (\ln d + \ln e) + C_4) = O((\ln d)(\ln e)mn)$.

Theorem 3.6

The time to compute P/Q for $Q \in U(d, m)$ and $P/Q \in U(e, n)$ is $O((\ln d)(\ln e)mn)$.

Proof. At most, $n + 1$ coefficient divisions are required. By Theorem 2.3, the time for each division is $\le C_1 (\ln d)(\ln e) + C_2$, and $(n + 1)(C_1(\ln d)(\ln e) + C_2) = O((\ln d)(\ln e)mn)$. The other required arithmetic is essentially the same as in multiplying P/Q by Q.

We now begin an analysis of univariate polynomial g.c.d. algorithms by considering the content and primitive part algorithms.

Theorem 3.7

The time to compute cont (P) for $P \in U(d, m)$ and norm $(pp(P)) \le e$ is $O((\ln d)(\ln e)m)$.

Proof. Let $P(x) = \sum_{i=1}^{k} a_i x^{e_i}$ where $e_1 > e_2 \ldots > e_k$ and each $a_i \neq 0$. Let $d_1 = |a_1|$ and $d_{i+1} = \gcd(d_i, |a_{i+1}|)$ for $1 \le i \le k-1$. Then cont $(P) = d_k$. $0 < d_i \le |a_i| \le$ norm $(P) \le d$ for all i. Also, cont $(P) \le d_i$ for all i. Hence, $\max\{d_i, |a_{i+1}|\}/d_{i+1} \le$ norm (P) /cont $(P) =$ norm $(pp(P)) \le e$ for $1 \le i \le k-1$. By Theorem 2.5, the time for the $k-1$ g.c.d.'s is $\le (k-1)(C_1(\ln d)(\ln e) + C_2) =$ $O((\ln d(\ln e)m)$, since $k - 1 \le m$.

Theorem 3.8

The time to compute pp(P) for $P \in U(d, m)$ and norm $(pp(P)) \le e$ is $O((\ln d)(\ln e)m)$.

Proof. To compute pp(P), we first compute cont (P), then divide P by cont (P). By Theorem 3.7, we need show only that the time for the division is $O((\ln d)(\ln e)m)$. The division requires, at most, $m+1$ integer divisions and, in each of these, the divisor is cont(P) $\le d$ while the quotient is a coefficient of pp(P) and, hence, bounded by e. Now apply Theorem 2.2.

Corollary 3.9

The time to compute either cont(P) or pp(P) for $P \in U(d, m)$ is $O((\ln d)^2 m)$.

Proof. Apply theorems 3.7 and 3.8, noting that norm $(pp(P)) \le$ norm (P). Next, we study the time required to compute a reduced polynomial remainder sequence over the integers. For this purpose the following theorem on the standardized Euclidean remainder, $\overline{\mathcal{R}}(P, Q)$, is helpful.

Theorem 3.10

Let $t(P, Q)$ be the time to compute $\overline{\mathcal{R}}(P, Q)$. Let $T(m, n, e) = \max \left\{ t(P, Q): \deg(P) = m \text{ and } \deg(Q) = n \text{ and } m \geq n > 0 \text{ and norm}(P) \leq e \text{ and norm}(Q) \leq e \right\}$. There is a constant C such that $T(m, n, e) \leq C(m + n)(m - n + 2)^2 (\ln e)^2$ for all sufficiently large e.

Proof. Let $P_1, P_2, \ldots, p_{m-n+2}$ be the sequence of polynomials such that $P_1 = P$, $P_{i+1} = \rho(P_i, Q)$ if $\deg(P_i) \geq n$, and $P_{i+1} = L(Q) \cdot P_i$ if $\deg(P) < n$. Then $\overline{\mathcal{R}}(P, Q) = P_{m-n+2}$.

Let M_i be the $i + 1$ by $m + 1$ matrix

$$M_i = \begin{pmatrix} b_n & b_{n-1} & b_{n-2} & \cdots & b_0 & 0 & 0 & \cdots & 0 \\ 0 & b_n & b_{n-1} & \cdots & b_1 & b_0 & 0 & \cdots & 0 \\ 0 & 0 & b_n & \cdots & b_2 & b_1 & b_0 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ a_m & a_{m-1} & a_{m-2} & \cdots & \cdots & \cdots & \cdots & \cdots & a_0 \end{pmatrix}$$

It is easy to see that P_{i+1} is the associated polynomial of M_i. Since the Euclidean norm of each row of M_i is, at most, e, Hadamard's theorem implies that the coefficients of P_{i+1} are bounded by e^{i+1}. At most, $m + n$ multiplications are required to compute P_{i+1} from P_i, and the time for each multiplication is $\leq C_1 (\ln e)(\ln e^i) + C_2 = C_1 (i)(\ln e)^2 + C_2 \leq C_1(m - n + 1)(\ln e)^2 + C_2$. The time for all multiplications is therefore bounded by $(m - n + 1)(m + n)(C_1(m - n + 1)(\ln e)^2 + C_2)$. At most, m additions are required to compute P_{i+1} from P_i, and the time for each is $\leq C_3(\ln e^{i+1}) + C_4 \leq C_3 (m - n + 2)$.

$(\ln e) + C_4$. So the time for all additions is $\leq (m - n + 1)m\ (C_3(m - n + 2)$ $(\ln e) + C_4)$. The validity of the theorem is now evident.

In reference 6 we defined a p.r.s. $P_1, P_2, \ldots, P_k$ to be normal in case $n_i - n_{i+1} = 1$ for $2 \leq i \leq k - 1$ where $n_i = \deg(P_i)$. It was shown that, among other nice properties, a normal reduced p.r.s. agrees, to within signs, with the associated subresultant p.r.s. There it was stated that, empirically, almost all p.r.s.'s are normal. As Knuth observes in reference 13, this is also true in a definite mathematical sense. Notice that if P_1 and P_2 have a g.c.d. of degree greater than one and if $P_1, P_2, \ldots, P_k$ is the complete p.r.s., then P_{k-1} is an associate of the g.c.d. and $P_k = 0$. Hence, $n_{k-1} - n_k > 1$, and $P_1, P_2, \ldots, P_k$ is not normal.

Since, in practice, g.c.d.'s of degree greater than one will occur frequently, our computing time analysis, to be useful, must not be restricted to a normal p.r.s. So we now define a p.r.s. $P_1, P_2, \ldots, P_k$ to be weakly normal in case $\deg(P_i) - \deg(P_{i+1}) = 1$ for $2 \leq i \leq k - 2$. We can now make the stronger assertion that, for any r, almost all complete p.r.s.'s $P_1, P_2, \ldots, P_k$ for which $\deg(\gcd(P_1, P_2)) = r$ are weakly normal.

In the following, we bound the time to compute a complete weakly normal reduced p.r.s. For convenience, we also say that (P,Q) is weakly normal when any complete p.r.s. $P_1, P_2, \ldots, P_k$ for which $P_1 = P$ and $P_2 = Q$ is weakly normal.

By Theorem 1, part b of reference 6, we have the theorem that follows.

Theorem 3.11

Let $P_1, P_2, \ldots, P_r$ be a complete weakly normal reduced p.r.s. Let $P, P_2, S_3, \ldots, S_r$ be the associated complete subresultant p.r.s. Then, for $3 \leq i \leq r$, $P_i = S_i$ if i is even or if $n_1 - n_2$ is odd; otherwise $P_i = -S_i$.

The next theorem bounds the coefficients of a weakly normal complete reduced p.r.s.

Theorem 3.12

Let $P_1, P_2, \ldots, P_r$ be a complete weakly normal reduced p.r.s. such that $\deg(P_1) = m$, $\deg(P_2) = n$, norm $(P_1) \leq e$, and norm $(P_2) \leq e$. Then the coefficients of P_i are bounded by $e^{m-n+2i-4}$ for $3 \leq i \leq r$.

Proof. By weak normality, $\deg(P_i) = n - i + 2$ for $2 \leq i \leq r - 1$ and, by Theorem 3.11, $P_i = \pm S_i$ for $3 \leq i \leq r$. The coefficients of S_i are, by definition, determinants of order $(m + n) - 2(\deg(P_{i-1}) - 1) = m - n + 2i - 4$ of submatrices of the Sylvester matrix of P_1 and P_2. By Hadamard's theorem they are therefore bounded by $e^{m-n+2i-4}$.

Theorem 3.13

The time to compute the complete reduced p.r.s. for P and Q, such that $\deg(P) = m$, $\deg(Q) = n$, $m \geq n > 0$, $\deg(\gcd(P, Q)) = k$, norm $(P) \leq e$, norm $(Q) \leq e$, and (P, Q) is weakly normal, is $O(((m + n)(m - n + 1)^2 + (m + n - 2k + 2)^2 (n^2 - (k - 1)^2))(\ln e)^2)$.

Proof. Let $P_1, P_2, \ldots, P_r$ be the complete reduced p.r.s. By Theorem 3.10, the time to compute $P_3 = \bar{R}(P_1, P_2)$ is $O((m + n)(m - n + 1)^2 (\ln e)^2)$. This completes the proof if $r = 3$.

Suppose $r \geq 4$. For $i \geq 2$, let $P_i' = \rho(P_i, P_{i+1})$. Then $\overline{\mathcal{R}}(P_i, P_{i+1})$ is either $\rho(P_i', P_{i+1})$ or $L(P_{i+1}) \cdot P_i'$ deg $(P_{i+1}) <$ deg $(P_i) = n - i + 2$. Therefore, at most, $2(n-1+2)$ multiplications are required to compute P_i'. By Theorem 3.12, the computing time for each is $\leq C_1 (\ln e^{m-n+2i-4})(\ln e^{m-n+2i-2}) + C_2$ $\leq C_1 (m - n + 2i - 2)^2 (\ln e)^2 + C_2$. The coefficients of P_i' are bounded by $2e^{2m-2n+4i-6} < e^{2(m-n+2i-2)}$. It follows that the time for all multiplications in computing $\overline{\mathcal{R}}(P_i, P_{i+1})$ has a bound of the form $(n - i + 2)(C_1(m - n + 2i - 2)^2 (\ln e)^2 + C_2)$. Such a bound continues to hold when additions are also considered.

Now $P_4 = \overline{\mathcal{R}}(P_2, P_3)/(L(P_2))^{m-n+1}$. The successive powers $(L(P_2))^j$, where $2 \leq j \leq m - n$, can be computed in $(m - n + 1)$ $(C_3(m - n + 1)(\ln e)^2 + C_4)$. The coefficients of P_4 are bounded by e^{m-n+4} and $|(L(P_2))^{m-n+1}| \leq e^{m-n+1}$ Hence, $\overline{\mathcal{R}}(P_2, P_3)$ can be divided by $(L(P_2))^{m-n+1}$ in $(n - 2)(C_5(m - n + 1)$ $(m - n + 4)(\ln e)^2 + C_6)$. The total time to compute P_4 from $\overline{\mathcal{R}}(P_2, P_3)$ is therefore $\leq C_7 (n - 2)(m - n + 1)(m - n + 4)(\ln e)^2 + C_8 m$. Adding to this the time above to compute $\overline{\mathcal{R}}(P_2, P_3)$ gives a bound of the form $(m + n)(C_5(m - n + 1)(m - n + 4)(\ln e)^2 + C_6)$, which is $O((m + n)(m - n + 1)^2(\ln e)^2)$. This proves the theorem for $r = 4$.

Suppose $r > 4$. For $i \geq 3$, $P_{i+2} = \overline{\mathcal{R}}(P_i, P_{i+1})/(L(P_i))^2$. Since deg $(P_{i+2}) = n - i$, P_{i+2} can be computed from $\overline{\mathcal{R}}(P_i, P_{i+1})$ in $(n - i)(C_3 (m - n + 2i)^2 (\ln e)^2 + C_4)$. Altogether, the time to compute P_{i+2} from P_i and P_{i+1} has a bound of the form $(n - i + 2)(C_1(m - n + 2i)^2(\ln e)^2 + C_2)$. Since $r = n - k + 3$ if $k > 0$

and $r = n + 2$ if $k = 0$, then $r \leq n - k + 3$. The total time to compute P_5, P_6, $. . .$, P_r is therefore bounded by $(C_3(m - n + 2(n - k + 1))^2(\ln e)^2 + C_4)$ $\Sigma_{i=3}^{n-k+1} (n - i) \leq (C_3(m + n - 2k + 2)^2(\ln e)^2 + C_4)$ $\Sigma_{i=3}^{n-k+1}$ $(n - i = O((m + n - 2k + 2)^2(n^2 - (k - 1)^2)(\ln e)^2)$, since $\Sigma_{i=3}^{n-k+1}$ $(n - i) \leq \frac{1}{2}((n - 2)(n - 3) - (k - 1)^2) = O(n^2 - (k - 1)^2)$.

Theorem 3.14

The time to compute gcd (P, Q) by the reduced p.r.s. algorithm, such that $\deg(P) = m$, $\deg(Q) = n$, $m \geq n > 0$, $\deg(\gcd(P, Q)) = k$, norm $(P) \leq e$, norm $(Q) \leq e$, and (P, Q) is weakly normal, is $O(((m + n)(m - n + 1)^2 + (m + n - 2k + 2)^2 (n^2 - (k - 1)^2))(\ln e)^2)$.

Proof. The required computations are as follows:

1. $a = \text{cont}(P)$, $b = \text{cont}(Q)$, $P_1 = \text{pp}(P)$, $P_2 = \text{pp}(Q)$, $c = \gcd(a, b)$.

2. Compute the complete reduced p.r.s. $P_1, P_2, . . ., P_r$.

3. If $P_r = 0$, compute $R = c \cdot \text{pp}(P_{r-1})$.

By Theorem 2.6 and Corollary 3.9, the computations in step 1 can be performed in $O((m + n)(\ln e)^2)$. The time for step 2 is $O(((m + n)(m - n + 1)^2 + (m + n - 2k + 2)^2 (n^2 - (k - 1)^2))(\ln e)^2)$ by Theorem 3.13. If $P_r = 0$, $k > 0$ and $r - 1 = n - k + 2$. By Theorem 3.12, the coefficients of P_{r-1} are bounded by e^{m+n-2k}. Next we notice that Corollary 3.9 would still hold under the weaker assumptions that $\deg(P) \leq m$ and that the coefficients of P are bounded by d. Since $\deg(P_{r-1}) = k$, $\text{pp}(P_{r-1})$ can be computed in $O(k(m + n - 2k)^2(\ln e)^2)$.

The multiplication of $pp(P_{r-1})$ by c can be done in $O(k(m+n-2k)(\ln e)^2)$. So the time for step 3 is $O(k(m+n-2k)^2(\ln e)^2)$. But $k \leq n^2 - (k-1)^2$, since $k \leq n$.

By eliminating the variables n and k, singly or together, we obtain the three corollaries of Theorem 3.14 that follow.

Corollary 3.15

The time to compute gcd (P, Q) by the reduced p.r.s algorithm, such that $\deg(P) = m$, $\deg(Q) = n$, $m \geq n > 0$, norm $(P) \leq e$, norm $(Q) \leq e$, and (P,Q) is weakly normal, is $O(((m+n)(m-n+1)^2 + n^2 (m+n)^2)(\ln e)^2)$.

Proof. Use Theorem 3.14, noting that $(m+n-2k+2)^2 \leq (m+n+2)^2 = O((m+n)^2)$ and $n^2 - (k-1)^2 \leq n^2$.

Corollary 3.16

The time to compute gcd (P, Q) by the reduced p.r.s. algorithm, such that $P, Q \in U(e, m)$, deg $(\gcd(P, Q)) = k$, and (P, Q) is weakly normal, is $O((m^3 + (m-k+1)^3(m+k)(\ln e)^2)$.

Proof. Use Theorem 3.14, noting that $(m+n)(m-n+1)^2 \leq 2m(m+1)^2 = O(m^3)$ and $(m+n-2k+2)^2(n^2-(k-1)^2) \leq 4(m-(k-1))^2(m^2-(k-1)^2) = 4(m-k+1)^3(m+k-1) = O((m-k+1)^3(m+k))$.

Corollary 3.17

The time to compute gcd (P, Q) by the reduced p.r.s. algorithm such that $P, Q \in U(e, m)$ and (P, Q) is weakly normal is $O(m^4(\ln e)^2)$.

Proof. Use Corollary 3.16, noting that $m^3 + (m - k + 1)^3(m + k) \leq m^3 + 2m(m + 1)^3 = O(m^4)$.

Having now analyzed the computing time for the reduced p.r.s. algorithm, let us observe that theorems 3.12, 3.13, and 3.14, and corollaries 3.15, 3.16, and 3.17 still hold if we replace "reduced p.r.s." everywhere by "primitive p.r.s." Let $Q_1, Q_2, \ldots, Q_r$ be the complete primitive p.r.s. Then O_i is a divisor and associate of P_i for all i. Hence Theorem 3.12 still holds. In Theorem 3.13 we now have to compute $pp(\overline{\mathfrak{R}}(Q_i, Q_{i+1}))$ in place of dividing $\overline{\mathfrak{R}}(P_i, P_{i+1})$ by $(L(P_i))^{\delta_{i-1}+1}$. The computing time bounds are of the same order for the two operations. In Theorem 3.14, the computation $pp(Q_{r-1})$ is omitted, but this does not affect the bound of the theorem.

It would be interesting to know whether theorems 3.13 and 3.14, stated for a primitive p.r.s. without the assumption of weak normality, would still hold. It seems likely that they would, but we have not attempted a proof. It also seems quite unlikely that they hold for a reduced p.r.s. without the weak normality assumption.

4. A CONGRUENCE ARITHMETIC G.C.D. ALGORITHM FOR UNIVARIATE POLYNOMIALS

In this section we describe a congruence arithmetic algorithm for computing the g.c.d. of two polynomials with integer coefficients. We also prove several theorems to show that the algorithm does what it is supposed to do. In section 5, we will analyze the computing time for the algorithm, showing that, on the average, it is much faster than previous algorithms.

For any prime number p, let GF(p) be the finite field with p elements 0, 1, . . ., p - 1, and let φ_p be the unique homomorphism of the integers I onto GF(p) so that $\varphi_p(i) = i$ for $0 \leq i < p$. Let φ_p^* be the homomorphism from

I[x] onto GF(p) [x] induced by φ_p. That is, $\varphi^*_p (\Sigma^n_{i=0} a_i x^i) = \Sigma^n_{i=0} \varphi_p (a_i) \cdot x^i$. We shall usually just write φ_p in place of φ^*_p.

Given two polynomials with integer coefficients P_1 and P_2, where $\deg(P_1) \geq \deg(P_2) > 0$, the algorithm decides whether $\deg(\gcd(P_1, P_2)) = 0$ and, if it does not, computes the last nonzero term S of the complete subresultant p.r.s. for P_1 and P_2 (this being an associate of $\gcd(P_1, P_2)$). A complete g.c.d. algorithm is constructed from this algorithm in an obvious way.

Let $P_1, P_2, S_3, \ldots, S_r$ be the complete subresultant p.r.s., and let S be the last nonzero subresultant. The general idea is to compute $\varphi_p(S)$ for a sufficiently large number of primes p that S can then be computed by the Chinese remainder theorem algorithm. The required number of primes is determined by Hadamard's theorem as a function of $\deg(P_i)$ and norm (P_i) for $i = 1$ and 2.

In addition to P_1 and P_2, the algorithm requires as input an infinite sequence of distinct prime numbers $p_1, p_2, p_3, \ldots$ and an integer h such that $p_i \geq 2^h$ for all i. In practice the p_i would probably be the first few hundred primes greater than 2^h, where 2^h is about half the largest integer which can be stored in one computer word.

A complete description of the algorithm follows.

Congruence Arithmetic Subresultant Algorithm

1. Compute $d = \text{norm}(P_1)$, $e = \text{norm}(P_2)$, $m = \deg(P_1)$, and $n = \deg(P_2)$. Compute the least integer r such that $2^r \geq d$, and the least integer s such that $2^s \geq e$. Compute $t = ms + nr$ and $u = [t/h] + 1$.

2. Set $I = 0$ and $M = \mathcal{S} = \mathcal{P} = (\ )$.

3. Select the next prime p.

4. Compute $P_i^* = \varphi_p(P_i)$ for $i = 1, 2$. If $\deg(P_1^*) < m$ or if $\deg(P_2^*) < n$, go to step 3.

5. Compute the complete reduced p.r.s. $P_1^*, P_2^*, \ldots, P_k^*$ of P_1^* and P_2^* over $GF(p)$.

6. If $P_k^* \neq 0$, terminate with indication that $\deg(\gcd(P_1, P_2)) = 0$.

7. Set $N = (n_1^*, n_2^*, \ldots, n_k^*)$ where $n_i^* = \deg(P_i^*)$.

8. Compute the subresultant associate S of P_{k-1}^*, using Theorem 1 of reference 6.

9. If $N < M$, go to step 3.

10. If $N > M$, set $\mathcal{S} = (S)$, $\mathcal{P}' = (p)$, $I = 1$, and $M = N$, then go to step 3.

11. Adjoin S to $\mathcal{S}$ and p to $\mathcal{P}$. Set $I = I + 1$. If $I < u$, go to step 3.

12. Let $\mathcal{P} = (p_1, p_2, \ldots, p_u)$ and $\mathcal{S} = (S_1, S_2, \ldots, S_u)$. Each S_i is a polynomial of degree $k > 0$. By $k + 1$ applications of the Chinese remainder theorem, compute the unique polynomial S of degree k such that $\varphi p_i(S) = S_i$ for $1 \leq i \leq u$ and such that the coefficients of S are bounded by $\frac{1}{2} p_1, 1_2 \ldots p_u$.

The remarks that follow explain the above algorithm. In step 1 we are applying Hadamard's theorem to obtain an upper bound u for the number of primes p for which $\varphi_p(S)$ will be needed to determine S. By Hadamard's theorem, the coefficients are bounded by $d^n e^m \leq 2^{ms+nr} = 2^t < 2^{hu} < p_1, p_2 \ldots p_u$ for any u primes greater than 2^h.

$\mathcal{P}$ is a list of primes $(p_1, p_2, \ldots, p_k)$ which were used but not discarded. $\mathcal{S}$ is a corresponding list of polynomials $(S_1, S_2, \ldots, S_k)$ for which hopefully, $S_i = \varphi_{pi}(S)$. If it later turns out that $S_i \neq \varphi_{pi}(S)$, both p_i and S_i are discarded. The value of I is always k; the number of primes in the list $\mathcal{P}$.

For each prime p, the sequence of degrees $(n_1^*, n_2^*, \ldots, n_k^*)$ is computed. All such sequences are ordered lexicographically. The value of the variable M is always the maximum of all degree sequences which were computed. Since ( ), the null sequence, is least among all sequences, M is initialized to ( ). Any prime whose degree sequence proves to be nonmaximal is discarded. It will be proved below that if any u distinct primes all have the same degree sequence, their common degree sequence is that of the complete reduced p.r.s. for P_1 and P_2 over the integers and, hence, their degree sequence is maximal. It will also be shown that if p has a maximal degree sequence, the complete reduced p.r.s. over GF(p) is the homomorphic image under φ_p of the complete reduced p.r.s. over the integers.

Theorem 4.1 and its corollary justify step 6.

Theorem 4.1

Let P and Q be nonzero polynomials over I, and let p be a prime such that $\varphi_p(\mathcal{L}(P)) \neq 0$ and $\varphi_p(\mathcal{L}(Q)) \neq 0$. Then $\deg(\gcd(\varphi_p(P), \varphi_p(Q))) \geq \deg(\gcd(P, Q))$.

Proof. Let $R = \gcd(P, Q)$. Then $P = R \cdot P_1$ and $Q = R \cdot Q_1$. Since φ_p is a homomorphism, $\varphi_p(P) = \varphi_p(R) \cdot \varphi_p(P_1)$ and $\varphi_p(Q) = \varphi_p(R) \cdot \varphi_p(Q_1)$. Hence, $\varphi_p(R)$ is a common divisor in GF(p) [x] of $\varphi_p(P)$ and of $\varphi_p(Q)$. Also, $\mathcal{L}(P) = \mathcal{L}(R) \cdot \mathcal{L}(P_1)$, so $0 \neq \varphi_p(\mathcal{L}(P)) = \varphi_p(\mathcal{L}(R)) \cdot \varphi_p(\mathcal{L}(P_1)$ and $\varphi_p(\mathcal{L}(R)) \neq 0$. Therefore, $\deg(\gcd(\varphi_p(P), \varphi_p(Q))) \geq \deg(\varphi_p(R)) = \deg(R)$.

Corollary 4.2

Let P and Q be polynomials over I, $\deg(P) \geq \deg(Q) > 0$. Let p be a prime such that $\varphi_p(\mathcal{L}(P)) \neq 0$ and $\varphi_p(\mathcal{L}(Q)) \neq 0$. Let $P_1, P_2, \ldots, P_k$ be a complete p.r.s. over GF(p) such that $P_1 = \varphi_p(P)$ and $P_2 = \varphi_p(Q)$. If $P_k \neq 0$, $\deg(\gcd(P, Q)) = 0$.

Proof. If $P_k \neq 0$, $\deg(\gcd(P_1, P_2)) = 0$. Use Theorem 4.1.

Lemma 4.3

Let P and Q be polynomials over I, $\deg(P) \geq \deg(Q) > 0$. Let p be a prime such that $\varphi_p(\mathcal{L}(P)) \neq 0$ and $\varphi_p(\mathcal{L}(Q)) \neq 0$. Then $\varphi_p(\overline{\mathfrak{R}}(P, Q)) = \overline{\mathfrak{R}}(\varphi_p(P), \varphi_p(Q))$.

Proof. Let $m = \deg(P)$, $n = \deg(Q)$, and $R = \overline{\mathfrak{R}}(P, Q)$. R is uniquely determined by the condition that $(\mathcal{L}(Q))^{m-n+1} \cdot P = Q \cdot S + R$ for some S with $\deg(R) < n$. Since φ_p is a homomorphism, $(\varphi_p(\mathcal{L}(Q)))^{m-n+1} \cdot \varphi_p(P) = \varphi_p(Q) \cdot \varphi_p(S) + \varphi_p(R)$. But, $\varphi_p(\mathcal{L}(Q)) = \mathcal{L}(\varphi_p(Q))$, $\deg(\varphi_p(P)) = m$, and $\deg(\varphi_p(Q)) = n$. Also, $\deg(\varphi_p(R)) \leq \deg(R) < n$. So $\varphi_p(R) = \overline{\mathfrak{R}}(\varphi_p(P), \varphi_p(Q))$.

Theorem 4.4

Let $P_1, P_2, \ldots, P_k$ be a reduced p.r.s. over I, and let p be a prime such that $\deg(\varphi_p(P_i)) = \deg(P_i)$ for $1 \leq i \leq k - 1$. Let $P_1^*, P_2^*, \ldots, P_k^*$ be a reduced p.r.s. over GF(p) such that $P_1^* = \varphi_p(P_1)$ and $P_2^* = \varphi_p(P_2)$. Then $P_i^* = \varphi_p(P_i)$ for $1 \leq i \leq k$.

Proof. $P_3 = \bar{\mathfrak{R}}(P_1, P_2)$ and $P_3^* = \bar{\mathfrak{R}}(P_1^*, P_2^*) = \bar{\mathfrak{R}}(\varphi_p(P_1), \varphi_p(P_2))$. Therefore $P_3^* = \varphi_p(P_3)$ by Lemma 4.3, since $\deg(\varphi_p(P_i)) = \deg(P_i)$ implies $\varphi_p(\mathfrak{L}(P_i)) \neq 0$. Let $n_i = \deg(P_i)$ and $\delta_i = n_i - n_{i+1}$ for all i. Assume $P_i^* = \varphi_p(P_i)$ and $P_{i+1}^* = \varphi_p(P_{i+1})$ where $2 \leq i \leq k - 2$. Then $P_{i+2}^* = \bar{\mathfrak{R}}(P_i^*, P_{i+1}^*)/(\mathfrak{L}(P_{i-1}^*))^{\delta_{i-1}+1} = \varphi_p(\bar{\mathfrak{R}}(P_i, P_{i+1}))/(\varphi_p(\mathfrak{L}(P_{i-1})))^{\delta_{i-1}+1} = \varphi_p(\bar{\mathfrak{R}}(P_i, P_{i+1})/(\mathfrak{L}(P_{i-1}))^{\delta_{i-1}+1}) = \varphi_p(P_{i+2})$ by Lemma 4.3. By induction, $P_i^* = \varphi_p(P_i)$ for all i.

Theorem 4.5

Let $P_1, P_2, \ldots, P_k$ be a reduced p.r.s. over I. Let p be a prime. Let $P_1^*, P_2^*, \ldots, P_k^*$ be a reduced p.r.s. over GF(p) such that $P_1^* = \varphi_p(P_1)$, $P_2^* = \varphi_p(P_2)$ and $\deg(P_i) = \deg(P_i^*)$ for $1 \leq i \leq k - 1$. Then $P_i^* = \varphi_p(P_i)$ for $1 \leq i \leq k$.

Proof. Proceed by induction on k. For $k = 3$ the theorem is an immediate consequence of Theorem 4.4 applied with $k = 3$. Assume Theorem 4.5 holds for $k = j$, and assume its hypotheses for $k = j + 1$. Then $P_i^* = \varphi_p(P_i)$ for $1 \leq i \leq j$ by induction hypothesis. Hence, $\deg(P_i) = \deg(\varphi_p(P_i))$ for $1 \leq i \leq j$. Hence, $P_i^* = \varphi_p(P_i)$ for $1 \leq i \leq j + 1$, by Theorem 4.4 applied with $k = j + 1$.

The next theorem shows that if u primes produce the same degree sequence, that common degree sequence is the degree sequence over the integers. By the previous theorem, therefore, each prime produces a homomorphic image of the reduced p.r.s. over the integers.

Theorem 4.6

Let $P_1, P_2, \ldots, P_k$ be a complete reduced p.r.s. over the integers. Let $\deg(P_1) = m$, $\deg(P_2) = n$, norm $(P_1) \le d$, and norm $(P_2) \le e$. Let $n_i = \deg(P_i)$. Let $p_1, p_2, \ldots, p_u$ be distinct primes such that $\Pi_{i=1}^{u} p_i > d^n e^m$. Let $\upsilon_1, \upsilon_2, \ldots, \upsilon_r$ be such that: for every i, $1 \le i \le u$, the complete reduced p.r.s. over $GF(p_i)$ for $\varphi_{p_i}(P_1)$ and $\varphi_{p_i}(P_2)$ is a sequence $P_1^{(i)}, P_2^{(i)}, \ldots, P_r^{(i)}$ such that $\upsilon_j = \deg(P_j^{(i)})$ for $1 \le j \le r$, $\upsilon_1 = n_1$, and $\upsilon_2 = n_2$. Then $r = k$ and $\upsilon_i = n_i$ for $1 \le i \le k$.

Proof. Assume $r \ge t$ and $\upsilon_j = n_j$ for $1 \le j \le t$. This holds by assumption for $t = 2$. We show that if it holds for t and if $t < k$, it holds for $t + 1$.

Let $S_1, S_2, \ldots, S_k$ be the complete subresultant p.r.s. over the integers such that $S_1 = P_1$ and $S_2 = P_2$. By Hadamard's theorem we know that the coefficients of all S_i are bounded by $d^n e^m$. Now $\upsilon_t = n_t > n_k = 0$, so $r \ge t + 1$. $\Pi_{i=1}^{u} p_i$ is not a divisor of $\mathcal{L}(S_{t+1})$ so, for some j, p_j is not a divisor of $\mathcal{L}(S_{t+1})$. By Theorem 1 of reference 6, $P_{t+1} = \Pi_{i=2}^{t-1} c_i^{\delta_{i-1}(\delta_i - 1)} S_{t+1}$ where $\delta_i = n_i - n_{i+1}$ and $c_i = \mathcal{L}(P_i)$ for $2 \le i \le t - 1$. By induction hypothesis, $\upsilon_i = n_i$ for $1 \le i \le t$. Hence, $n_i = \deg(P_i^{(j)})$ for $1 \le i \le t$. By Theorem 4.5, $P_i^{(j)} = \varphi_{p_j}(P_i)$ for $1 \le i \le t + 1$. So $\deg(P_i) = \deg(\varphi_{p_j}(P_i))$ for $1 \le i \le t$. Hence, $\varphi_{p_j}(\mathcal{L}(P_i)) = \varphi_{p_j}(c_i) \ne 0$ for $1 \le i \le t$. So p_j is not a divisor of

$\Pi_{i=2}^{t-1} c_i^{\delta_{i-1}(\delta_i - 1)}$, and it is not a divisor of $\mathcal{L}(P_{t+1})$, i.e., $\upsilon_{t+1} = n_{t+1}$.

By induction we now have $r \geq k$ and $\upsilon_i = n_i$ for $1 \leq i \leq k$. This implies $\upsilon_k = n_k = 0$, and therefore, $r = k$.

Let $P_1, P_2, \ldots, P_k$ be the complete reduced p.r.s. over I for the two polynomials P_1 and P_2 which are inputs to the algorithm. Let S_{k-1} be the subresultant associate of P_{k-1}. The next theorem shows that if step 12 of the algorithm is ever reached, $S_i = \varphi_{p_i}(S_{k-1})$ for each p_i in $\mathcal{Q}$

Theorem 4.7

Let $P_1, P_2, \ldots, P_k$ be a complete reduced p.r.s. over I. Let $P_i^* = \varphi_p(P_i)$, and assume $\deg(P_i^*) = \deg(P_i)$ for $1 \leq i \leq k$. Let $S_1, S_2, \ldots, S_k$ be the subresultant p.r.s. over I such that $S = P_1$ and $S_2 = P_2$. Let $S_{k-1}^* = (-1)^{\tau_{k-1}} P_{k-1}^* / \Pi_{i=2}^{k-3} L(P_i^*)^{\delta_{i-1}(\delta_i - 1)}$ where $n_i = \deg(P_i^*)$, $\delta_i = n_i - n_{i+1}$, $\tau_k = \Sigma_{i=1}^{k-3} n_i n_{i+1} + (n_1 + k - 1)(n_{k-2} + 1)$. Then $S_{k-1}^* = \varphi_p(S_{k-1})$.

Proof. Since $\deg(P_i) = \deg(P_i^*)$, $\mathcal{L}(P_i^*) = \mathcal{L}(\varphi_p(P_i)) = \varphi_p(\mathcal{L}(P_i))$. Apply Theorem 1 of reference 6, and use the homomorphism property of φ_p.

We still have to show that if step 12 is reached, S_{k-1} is an associate of gcd (P_1, P_2), i.e., that $S_k = 0$. But this follows easily from the proof of Theorem 4.6.

Finally, we must show that the algorithm will eventually terminate. This is equivalent to showing that only a finite number of primes can ever be discarded by the algorithm in steps 4, 9, and 10. But a prime is discarded only when its degree sequence is nonmaximal, i.e., when it divides $\Pi_{i=1}^{k-1} c_i$ where $P_1, P_2, \ldots, P_k$ is the complete reduced p.r.s. and $c_i = \mathcal{L}(P_i)$. In the next section, we take a closer look at the number of primes which can be discarded.

5. ANALYSIS OF THE CONGRUENCE ARITHMETIC ALGORITHM

In analyzing computing times for the congruence arithmetic g.c.d. algorithm we consider h to be a constant. h will ordinarily be in a range between 30 and 60, depending on the computer word length. Since there will then be a minimum of about $2^{h-1}/h \geq 10^7$ primes in the interval from 2^h to 2^{h+1}, we can safely ignore the size of the primes in our analysis and assume they are all single-precision. There will then be a fixed bound for the time required to perform any arithmetic operation in $GF(p_i)$ for all primes p_i encountered. Likewise, we may safely assume m, n, r, s, t, and u are all single-precision integers.

Theorem 5.1

The time to compute norm (P) such that $P \in U(d, m)$ is $O(m(\ln d))$.

Proof. Obvious.

Theorem 5.2

The time to compute the least r such that $2^r \geq d$ is $O((\ln d)^2)$.

Proof. Let $d_0 = d - 1$ and $d_{i+1} = [d_i/2]$. Pick the smallest k such that $d_k = 0$. Then $r = k$. r divisions are required, and the time for each is $O(\ln d)$. But $r = O(\ln d)$.

Theorem 5.3

The time to compute $\varphi_p(d)$ is $O(\ln d)$.

Theorem 5.4

The time to compute $\varphi_p(P)$ for $P \in U(d, m)$ is $O(m(\ln d))$.

Theorem 5.5

The time to compute the complete reduced p.r.s. $P_1^*, P_2^*, \ldots, P_k^*$ over GF(p) from P_1^* and P_2^*, such that $\deg(P_1^*) = m$ and $\deg(P_2^*) = n$, $m \geq n > 0$, is $O(m^2)$.

Proof. Let $\deg(P_i^*) = n_i$, $\mathfrak{L}(P_i^*) = c_i^*$, $\delta_i = n_i - n_{i+1}$ for $1 \leq i \leq k$, and $\delta_0 = -1$. Then $P_{i+2}^* = \overline{\mathfrak{R}}(P_i^*, P_{i+1}^*)/c_i^{*\,\delta_{i-1}+1}$ for $1 \leq i \leq k-2$. Clearly, $3n_i(\delta_i + 1)$ is the maximum number of operations in GF(p) required to compute $\overline{\mathfrak{R}}(P_i^*, P_{i+1}^*)$. $(c_i^{*\,\delta_{i-1}+1})^{-1}$ can be computed in $\delta_{i-1}+1$ operations, and P_{i+2}^* can then be computed in, at most, $n_{i+2} + 1 \leq n_{i+1}$ operations. But $\Sigma_{i=1}^{k-2}\ 3\,n_i(\delta_i + 1) \leq 3m\ \Sigma_{i=1}^{k-2}(\delta_i + 1) \leq 6m\ \Sigma_{i=1}^{k-2}\ \delta_i \leq 6m^2$, $\Sigma_{i=1}^{k-2}(\delta_{i-1} + 1) \leq 1 + 2\,\Sigma_{i=1}^{k-3}\ \delta_i \leq 1 + 2(m - n_{k-2}) \leq 2m$, and $\Sigma_{i=1}^{k-2}\ n_{i+2} \leq m^2$. So the total number of operations is, at most, $7m^2 + 2m < 8m^2$.

Theorem 5.6

Given a complete reduced p.r.s. $P_1^*, P_2^*, \ldots, P_k^*$ over GF(p) such that $\deg(P_1^*) = m$, the time to compute $P_{k-1}^*/\Pi_{i=2}^{k-3}\ \mathfrak{L}(P_i^*)^{\ \delta_{i-1}(\delta_i-1)}$ is $O(m^2)$.

Proof. $\Sigma_{i=2}^{k-3}\ \delta_{i-1}(\delta_i - 1) \leq \Sigma_{i=2}^{k-3}\ \delta_{i-1}\,\delta_i \leq (\Sigma_{i=2}^{k-3}\ \delta_i)^2 = (m - n_{k-2})^2$. So $d_{k-1} = \Pi_{i=2}^{k-3}\,\mathfrak{L}(P_i^*)^{\ \delta_{i-1}(\delta_i-1)}$ can be computed in a maximum of $(m - n_{k-2})^2$

operations. d_{k-1}^{-1} and $d_{k-1}^{-1} P_{k-1}^{*}$ can then be computed in $n_{k-1} + 2$ additional operations. But $(m - n_{k-2})^2 + n_{k-1} + 2 \leq (m^2 - mn_{k-2}) + (n_{k-2} + 1) \leq m^2 - 2n_{k-2} + 1 < m^2$.

In steps 9 and 10 of the algorithm, we have to compare two degree sequences M and N. Since the maximum length for such a degree sequence is $m + 2$, the time for this operation is $O(m)$. This observation, together with theorems 5.4, 5.5, and 5.6 gives us the theorem that follows.

Theorem 5.7

For each prime p_i selected by the congruence arithmetic subresultant algorithm, the computing time is $O(m^2 + m(\ln d + \ln e))$.

Next, we show that the number of primes selected by the algorithm is $O(n(n \ln d + m \ln e))$.

Theorem 5.8

Let $P_1, P_2, \ldots, P_k$ be a reduced p.r.s. over I. Let $S_1, S_2, \ldots, S_k$ be a subresultant p.r.s. over I such that $S_1 = P_1$ and $S_2 = P_2$. Let p be a prime and $1 \leq i \leq k$. If p divides $\ell(P_i)$, then p divides $\ell(S_j)$ for some $j \leq i$.

Proof. The theorem clearly holds for $1 \leq i \leq 3$. Assume it holds for t where $3 \leq t < k$. Let $c_i = \ell(P_i)$ and $d_i = \ell(S_i)$. By Theorem 1 of reference 6, $c_{t+1} = \pm \Pi_{i=2}^{t-1} c_i^{\delta_{i-1}(\delta_i - 1)} d_{t+1}$. Suppose p divides c_{t+1}, but not d_{t+1}. Then it divides c_j for some $j \leq t - 1$ and hence, by induction hypothesis, d_j. So the theorem holds for $t + 1$.

Theorem 5.9

Let $S_1, S_2, S_3, \ldots, S_k$ be a complete reduced p.r.s. over I with $\deg(S_1) = m$, $\deg(S_2) = n$, norm $(S_1) \le d$, and norm $(S_2) \le e$. Let $d_i = \mathfrak{L}(S_i)$. Then $|\Pi_{i=1}^{k-1} d_i| \le d^{n^2} e^{mn}$.

Proof. By Hadamard's theorem, $|d_i| < d^n e^m$ for all i, so $|\Pi_{i=3}^{k-1} d_i| \le (d^n e^m)^{k-3}$. But $|d_1| \le d$ and $|d_2| \le e$, so $|\Pi_{i=1}^{k-1} d_i| \le (d^n e^m)^{k-2} \le (d^n e^m)^n$.

Theorem 5.10

The number of primes selected by the congruence arithmetic subresultant algorithm is $O(n(n \ln d + m \ln e))$.

Proof. Since every prime discarded by the algorithm is a divisor of some $\mathfrak{L}(P_i)$, by Theorem 5.8 some $\mathfrak{L}(S_i) = d_i$, $1 \le i \le k - 1$. By Theorem 5.9, the product of all discarded primes is, at most, $d^{n^2} e^{mn}$. Since each prime is greater than 2^h, if N is the number of discarded primes, we have $2^{Nh} < d^{n^2} e^{mn}$; hence, $N < (n^2 \log_2 d + mn \log_2 e)/h$. So $N = 0(n(n \ln d + m \ln e))$. The number of primes selected but not discarded is, at most, $u \le t/h + 1 \le (ms + nr)/h + 1 \le (m(s - 1) + n(r - 1) + m + n)/h + 1 < (m(\log_2 e + 1) + n(\log_2 d + 1))/h + 1 = O(n \ln d + m \ln e)$.

After we put all the pieces together, we have the computing time bound for the entire process in Theorem 5.11.

Theorem 5.11

The computing time for the congruence arithmetic subresultant g.c.d. algorithm, such that $\deg(P_1) = m$, $\deg(P_2) = n$, norm $(P_1) \leq d$, and norm $(P_2) \leq e$, is $O(mn(m + \ln d + \ln e)(n \ln d + m \ln e))$.

Proof. By theorems 5.7 and 5.10, the time to process all primes is $O(mn(m + \ln d + \ln e)(n \ln d + m \ln e))$. Therefore we need to show that the same is true of the other parts of the algorithm. By theorems 5.1 and 5.2, the computing time for step 1 is $O(m \ln d + n \ln e + (\ln d)^2 + (\ln e)^2)$. In accordance with the remarks at the beginning of this section, we assume that each prime is $\leq 2^{h+1}$. Hence, $\ln(p_1 p_2 \ldots p_u) \leq (h + 1)u = O(n \ln d + m \ln e)$, as in the proof of Theorem 5.10. By Theorem 2.7, the time for each of the $k + 1$ applications of the Chinese remainder theorem algorithm is $O((n \ln d + m \ln e)^2)$. Since $k + 1 = O(n)$, the computing time for step 12 is $O(n(n \ln d + m \ln e)^2)$. Incorporating this algorithm into a complete g.c.d. algorithm, we must consider the times to compute $pp(P_1)$, $pp(P_2)$, $pp(S)$, and $c \cdot pp(S)$ where $c = \gcd(\text{cont}(P_1), \text{cont}(P_2))$. The times to compute $pp(P_1)$ and $pp(P_2)$ are $O(m(\ln d)^2)$ and $O(n(\ln e)^2)$ by Corollary 3.9. Since the coefficients of S are bounded by $p_1 p_2 \ldots p_u$, Corollary 3.9 shows that the time to compute $pp(S)$ is $O(n(n \ln d + m \ln e)^2)$, the same as step 12, and the time to multiply $pp(S)$ by c is then $O(n(\ln d + \ln e)(n \ln d + m \ln e)$.

The next two corollaries follow easily.

Corollary 5.12

The computing time for the congruence arithmetic subresultant g.c.d. algorithm, such that $\deg(P_1) = m$, $\deg(P_2) = n$, norm $(P_1) \leq e$, and norm $(P_2) \leq e$, is $O(mn(m + n)(m + \ln e)(\ln e))$.

Corollary 5.13

The computing time for the congruence arithmetic subresultant g.c.d. algorithm such that $P_1, P_2 \in U(e, m)$ is $O(m^3(m + \ln e)(\ln e))$.

Comparing corollaries 3.17 and 5.13, we see that the computing time bound for the reduced p.r.s. algorithm is larger than that for the congruence arithmetic algorithm by a ratio of $m(\ln e)/(m + \ln e)$, a ratio which grows indefinitely with m and e. Actually, the superiority of the congruence algorithm is much greater than this ratio indicates. We found that the number of primes discarded is $O(n(n \ln d + m \ln e))$, while the number of primes retained is only $O(n \ln d + m \ln e))$. This suggests that most primes are discarded whereas, it is intuitively clear that, on the average, a prime will be discarded only on very rare occasions. If one chooses an integer N at random, the probability that $\varphi_p(N) = 0$ is only $1/p$. Hence, if the leading coefficients of the subresultant p.r.s. are random in an appropriate sense, the probability that p will be discarded is, at most, $1 - (1 - 1/p)^n \cong n/p$, since n is much smaller than p in all cases of interest. This reasoning suggests that the average number of primes selected is $O(n \ln d + m \ln e)$. Going back over the proof of Theorem 5.11, one can easily show that, under this hypothesis, the average computing time is $O(k(m + n)^2(\ln e)^2 + m^2 (m + n)(\ln e))$, where k is the degree of the g.c.d. Hence, $O(km^2(\ln e)^2 + m^3 (\ln e))$; hence, $O(m^3(\ln e)^2)$. If P_1 and P_2 are relatively prime, there is a very high probability that this will be proved by the value of p, via Corollary 4.2, provided that n is much smaller than p. If we assume that the average number of primes selected in the relatively prime case is less than two (or any other fixed number), then we easily conclude that the average computing time for relatively prime polynomials is $O(m^2 + (m + n)(\ln e)^2)$; hence, $O(m^2 + m (\ln e)^2)$.

The above algorithm can be made faster in two ways. The first way requires only a simple modification of step 12. In step 12, if $M = (n_1, n_2, \ldots, n_a)$, then $n_a = 0$ and n_{a-1} is the degree k of the g.c.d. If $a = 3$, P_2 is the primitive associate of the g.c.d. and the Chinese remainder theorem is not needed. If $a > 3$, then S, the subresultant associate of the g.c.d., has coefficients bounded by $d^{(n-n_{a-2}+1)}$ $e^{(m-n_{a-2}+1)}$, by Theorem 1 of reference 6 and Hadamard's theorem. Hence, only $v = [((m - n_{a-2}+1)\ s + (n - n_{a-2}+1)r)/h] + 1$ primes are needed to determine S by the Chinese remainder theorem. The last $u - v$ elements of $\mathcal{P}$ and $\mathcal{S}$ may be ignored. If, for example, $m = n = 2k$, then $n_{a-2} - 1 \geq k$ and $v = u/2$ approximately. By Theorem 2.7, the computing time for step 12 will therefore be reduced by a factor of about 4.

The algorithm as thus revised computes an a priori bound u for the number of primes needed. After processing u undiscarded primes, it determines k and adjusts downwards its original estimate for the number of primes needed. The second modification avoids this waste by determining the degrees $n_1, n_2, n_3, \ldots, n_a$ sequentially. $n_1 = m$ and $n_2 = n$ are initially given. Let $S_1, S_2, \ldots, S_a$ be the complete subresultant p.r.s. over the integers. The coefficients of S_3 are bounded by $d^{(n-n_2+1)}$ $e^{(m-n_2+1)}$. Hence, at most, $u_3 = [((m - n_2 + 1)s + (n - n_2 + 1)r)/h] + 1$ primes are needed to determine S_3. Having accumulated u_3 undiscarded primes, n_3 is determined as m_3 where $M = (m_1, m_2, m_3, \ldots)$. If $n_3 = 0$, then $S_3 = 0$ and step 12 is undertaken. Otherwise, $u_4 = [((m - n_3 + 1)s + (n - n_3 + 1)r)/h] + 1$ primes are needed to determine S_4. Continuing in this way one eventually goes to step 12 with u_a primes in $\mathcal{P}$. The Chinese remainder theorem is then applied using u_{a-1} of these primes. If, for example, $m = n = 2k$, this modification will reduce by a factor of about 2 the time required to compute reduced p.r.s.'s and subresultant associates over fields $GF(p_i)$.

The univariate congruence arithmetic g.c.d. algorithm can be advantageously used as part of any multivariate g.c.d. algorithm since the calculation of a multivariate g.c.d. requires the calculation of numerous univariate g.c.d.'s. Also, the methods employed in the univariate algorithm can be extended to multivariate algorithms. Viewed abstractly, we wish to compute the g.c.d. of two polynomials P_1 and P_2 over an integral domain ϑ (where ϑ itself may be a polynomial domain). We have at our disposal a sequence $\psi_1, \psi_2, \ldots$ of homomorphisms from ϑ to some integral domains J_i. Instead of computing a complete p.r.s. $P_1, P_2, P_3, \ldots, P_k$ over ϑ, we compute a complete p.r.s. $P_1^{(i)}, P_2^{(i)}, P_3^{(i)}, \ldots$ over J_i such that $P_1^{(i)} = \psi_i^*(P_1)$ and $P_2^{(i)} = \psi_i^*(P_2)$ for $i = 1, 2, 3, \ldots$, ψ_i^* being the homomorphism from $[x]$ to $J_i[x]$ induced by ψ_i. If ψ_i^* preserves the degrees of $P_1, P_2, P_3 \ldots$, we have $P_j^{(i)} = \psi_i^*(P_j)$. One must then have a mechanism for discarding those ψ_i^* which do not preserve degrees. One must also be able to compute a bound u such that P_{k-1} can be determined from $\psi_1^*(P_{k-1})$. $\psi_2^*(P_{k-1}), \ldots, \psi_u^*(P_{k-1})$. If, for example, $\vartheta = I[y]$, one can let ψ_i be the evaluation homomorphism $\psi_i(P) = P(i)$ from $I[y]$ to I. In this case, a bound u can be computed as a function of the degrees of the coefficients of P_1 and P_2, and the interpolation replaces the Chinese remainder theorem for computing P_{k-1} from the $\psi_i^*(P_{k-1})$. Those ψ_i^* which produce nonmaximal degree sequences are discarded as in the univariate algorithm above. One advantage of so choosing the ψ_i is that it leads to an algorithm which is recursive in the number of variables. To compute the complete reduced p.r.s. of $\psi_i^*(P_1)$ and $\psi_i^*(P_2)$ over I, the univariate congruence algorithm above can be used (since only the degree sequence and the subresultant associate of the last nonzero term are needed). From an algorithm for polynomials in n variables, an algorithm for polynomials in n + 1 variables is thus obtained using the evaluation homomorphisms from $I[x_1, \ldots, x_{n+1}]$ to $I[x_1, \ldots, x_n]$.

REFERENCES

1. I. Borosh and A. S. Fraenkel, "Exact Solutions of Linear Equations with Rational Coefficients by Congruence Techniques," Mathematics of Computation, Vol. 20, No. 93, January 1966, pp. 107-112.

2. W. S. Brown, "The ALPAK System for Non-Numerical Algebra on a Digital Computer—I: Polynomials in Several Variables and Truncated Power Series with Polynomial Coefficients," Bell System Technical Journal Vol. 42, No. 3, September 1963, pp. 2081-2119.

3. W. S. Brown, J. P. Hyde, and B. A. Tague, 'The ALPAK System for Non-Numerical Algebra on a Digital Computer—II: Rational Functions of Several Variables and Truncated Power Series with Rational Function Coefficients," Bell System Technical Journal, Vol. 43, No. 1 March 1964, pp. 785-804.

4. J. P. Hyde, "The ALPAK System for Non-Numerical Algebra on a Digital Computer—III: Systems of Linear Equations and a Class of Side Relations," Bell System Technical Journal, Vol. 43, No. 4, July 1964, pp. 1547-1562.

5. G. E. Collins, "PM, A System for Polynomial Manipulation," Communications of the Association for Computing Machinery, Vol. 9, No. 8, August 1966, pp. 575-589.

6. G. E. Collins, "Subresultants and Reduced Polynomial Remainder Sequences," Journal of the Association for Computing Machinery, Vol. 14, No. 1, January 1967, pp. 128-142.

7. G. E. Collins, "The SAC-1 List ProcessingSystem,"University of Wisconsin Computing Center Report (34 pages), Madison, Wisconsin, July 1967.

8. G. E. Collins, "The SAC-1 Integer Arithmetic System," University of Wisconsin Computing Center Report (31 pages), Madison, Wisconsin, September 1967.

9. G. E. Collins, "The SAC-1 Polynomial System," University of Wisconsin Computing Center Technical Reference 2:1968 (68 pages), Madison, Wisconsin, January 1968.

10. G. E. Collins, "Computing Multiplicative Inverses in GF(p)," University of Wisconsin Computer Sciences Technical Report No. 22 (8 pages), Madison, Wisconsin, May 1968.

11. S. A. Cook, "On the Minimum Computation Time of Functions," Harvard University Computation Lab. Report BL-41, Cambridge, Massachusetts, May 1966.

12. D. E. Knuth, The Art of Computer Programming, Vol. I: Fundamental Algorithms, Addison-Wesley, 1968.

13. D. E. Knuth, The Art of Computer Programming, Vol. II: Seminumerical Algorithms, Chapter IV, Arithmetics, Addison-Wesley, 1969.

14. H. Takahasi and Y. Ishibashi, "A New Method for 'Exact Calculation' by a Digital Computer," Information Processing in Japan, Vol. 1, 1961, pp. 28-42.

15. J. V. Uspensky, and M. A. Heaslett, Elementary Number Theory, McGraw-Hill, 1939.

SYMBOLIC METHODS FOR THE COMPUTER SOLUTION OF LINEAR EQUATIONS WITH APPLICATIONS TO FLOWGRAPHS

by

John D. Lipson
Graduate School of Arts and Sciences
Harvard University
Cambridge, Massachusetts

Abstract

The paper considers the computer solution of linear equations with symbolic coefficients. A method of solution based on a multivariate polynomial extension of the two-step integer-preserving elimination algorithm due to E. Bareiss[1] which avoids the necessity for time-consuming rational function arithmetic is proposed. This two-step method is compared with the classical one-step method in the context of multivariate polynomial domains and found to be considerably superior.

The proposed fraction-free algorithm is used as the basis for a new method of computing the generating function of a flowgraph, superior to the traditional node elimination technique of S. J. Mason. The application of flowgraphs to enumeration problems of combinatorial analysis and automata theory is presented.

Proposed algorithms are implemented using the IBM 2250 Scope FORMAC System.[43] The 2250 listings, along with sample output are presented.

The help and encouragement received from Professor Garrett Birkhoff, Dr. Elizabeth Cuthill, Dr. Robert Tobey, and Professor Donald Tufts are greatly appreciated. The support for this research received by IBM and the Office of Naval Research is gratefully acknowledged.

PRECEDING PAGE BLANK NOT FILMED

SYMBOLIC METHODS FOR THE COMPUTER SOLUTION OF LINEAR EQUATIONS WITH APPLICATIONS TO FLOWGRAPHS

by

John D. Lipson

1. INTRODUCTION AND OVERVIEW

Linear Equations With Symbolic Coefficients

The use of computers in conjunction with computer languages, such as FORTRAN, ALGOL, and PL/I, in computing the solution to linear equations with numeric (real) coefficients is well known. (An account of this area along with sample programs is provided by reference 12.) This paper is concerned with the analogous problem in the area of nonnumerical analysis and symbolic mathematics by computer, namely the computer solution of linear equations with symbolic coefficients.

An example of a system of linear equations with symbolic coefficients is provided by

$$\begin{pmatrix} 1 & -x & -x & -1 \\ 0 & 1 & -x & -1 \\ 0 & -x & 1-xy & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \qquad (1.1)$$

Here the coefficients are in $I[x,y]$, the domain of bivariate polynomials over the integers I. The solution, which then lies in the field $I(x,y)$ of rational functions, is

$$u_1 = \frac{-1 - 2x + xy - x^2 + x^2y}{1 - xy - x^2}$$

$$u_2 = \frac{1 + x - xy}{1 - xy - x^2}$$

$$u_3 = \frac{-1 - x^2}{1 - xy - x^2} \tag{1.2}$$

$$u_4 = 1$$

A simple but important observation for the sequel is that any system of linear equations with symbolic coefficients, regardless of the nature or complexity of the coefficients, can be transformed via substitution to a system of linear equations with coefficients in some multivariate polynomial domain. For example, suppose the given system of linear equations is

$$\begin{pmatrix} 1 & -\sin(ar) & -\sin(ar) & -1 \\ 0 & 1 & -\sin(ar) & -1 \\ 0 & -\sin(ar) & 1 - \frac{\sin(ar)}{a^2 + r^2} & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} . \tag{1.3}$$

Substituting x for $\sin(ar)$ and y for $\frac{1}{a^2 + r^2}$ yields (1). The solution to (1.3) is eventually obtained by making the inverse substitutions into (1.2).

Computation With Multivariate Polynomials

The time-honored method of Gaussian elimination is valid over any field. Thus it is applicable to computing the solution to a system of linear equations with polynomial coefficients, provided that a rational function arithmetic capability is available. Indeed, systems have been implemented with this capability, notably ALPAK [5, 6, 20] and MATHLAB, [26] and they employ standard Gaussian elimination to solve linear equations with multivariate polynomial or rational function coefficients.

Gaussin elimination is considerably less efficient over the field of quotients of a polynomial domain* than over the real field due to the nature of computation with quotients of polynomials. First, the arithmetic $\oplus$ and $\odot$ of quotients

$$\frac{a}{b} \oplus \frac{c}{d} = \frac{ad+bc}{bd} \quad \text{and} \quad \frac{a}{b} \odot \frac{c}{d} = \frac{ac}{bd}$$

involves multiple polynomial operations—three multiplications and one addition in the case of quotient addition $\oplus$, and two multiplications in the case of quotient multiplication $\odot$. Second and more important is the problem of reducing quotients to lowest terms. For example, over the field of quotients of $I[x,y]$ one does not want to accommodate rational functions such as $\frac{15x^6y^2 + 5x^5y}{6x^5 + 3x^3y^2 + 2x^4 + x^2y}$; instead one wants the equivalent reduced from $\frac{5x^4y}{2x^3 + xy}$. The reduction of rational functions to lowest terms generally involves computing the greatest common divisor (g.c.d.) of the numerator and denominator polynomials by some variant of the Euclidean algorithm. In spite of the significant results recently achieved by G.E. Collins in this area, [7, 8, 9] the amount of computation required to determine greatest common divisors, especially in the multivariate polynomial case, is formidable.** As remarked in reference 26 on page 88, most computation time in any application involving rational functions is taken up with g.c.d. calculations. Thus we are sorely tempted to employ a method for solving linear equations with polynomial coefficients that does not involve computation with rational functions. Of course one such method is provided by Cramer's rule, but the excessively large number of operations required to directly evaluate higher order determinants more than negates any advantages gained by avoiding g.c.d. calculations.

*Unless specified otherwise, polynomials are understood to have integer coefficients. Although there is no harm algebraically in admitting polynomials with real coefficients, serious computational difficulties arise due to roundoff error.

**In reference 32 it is reported that to compute the g.c.d. of a polynomial of degree 5 in two or three variables can take on the order of a minute on the 7094.

Fraction-Free Gaussian Elimination

Given a system of linear equations $A\underline{x} = \underline{b}$ with integer coefficients and right-hand side, the problem arises of computing the exact solution, i.e., not tolerating any roundoff error. A modification of Gaussian elimination, attributed to Jordan in reference 1, transforms an integer system to triangular form such that:

a. No rational numbers are produced, only integers; i.e., all transformations are integer-preserving or fraction-free, and hence not subject to roundoff error.

b. The magnitude of the coefficients in the successive transformed matrices of the triangularization process are minimized by dividing out a factor which occurs systematically.

c. A by-product of the method is the evaluation of the determinant of the coefficient matrix and, by an appropriate back-substitution scheme, the same solution is obtained as would result from applying Cramer's rule.

The algorithm referred to above is described by Bodewig (reference 3, Chapter 1) and Fox (reference 13, pp. 82-86). The importance of the exact divisibility property of step b can be appreciated by observing the size of the integers in an example of Rosser[38] in which elimination is carried out by a simple cross-multiplication scheme. Luther and Guseman[24] essentially rediscover the fraction-free technique described by Bodewig (reference 3, p. 109) for computing the adjoint of an integer matrix.

Recently Bareiss (reference 1, Equation (2.12)) devised a two-step variant of this fraction-free elimination algorithm. In this scheme, variables are successively eliminated two at-a-time in transforming a given system of linear equations to triangular form. The interesting computational aspect of this two-step algorithm is considerably increased efficiency (reference 1, Section IV) over the classical one-step method.

The applications of fraction-free Guassian elimination reported in the above references are primarily numerical in nature. Bodewig and Fox are concerned with computing the exact solution of linear equations with integer coefficients, and Luther and Guseman* report writing a program for the IBM 709 which finds the adjoint of 50 x 50 square matrices using integer arithmetic. Except for brief remarks (reference 1, Sections IV and VI) that multistep fraction-free methods can also be applied to the mechanized algebraic (i.e., nonnumeric) expansion of determinants, the applications that Bareiss discusses are numeric. In addition to the usual application of fraction-free (integer-preserving) methods in computing the exact solution of linear equations with integer coefficients,** Bareiss discusses an interesting new application (reference 1, page 576) of such methods in devising a completely stable (roundoff-free) general elimination routine.

In this report we shall apply a two-step fraction-free elimination method in computing the solution of linear equations with multivariate polynomial coefficients. In section 2 of this paper, a new and elementary proof of the crucial fraction-free property of the elimination methods is presented. The proof holds for an arbitrary integral domain. The algorithms are analyzed from the viewpoint of computational efficiency when applied to multivariate polynomial domains.

Computation of Generating Functions of Flowgraphs

The multivariate polynomial extension of Bareiss' two-step elimination algorithm can be used to compute the generating function or gain of a flowgraph. (It is this problem which served as the starting point and motivation of the research reported here.)

*Note: In the Luther and Guseman paper,[24] Equation (20) on page 448 contains the essence of a two-step elimination algorithm (but it was not recognized or applied as such).

**With respect to the problem of computing the exact solution of linear equations with integer coefficients, the reader is also referred to methods[4, 33, 40] based on congruence (finite field) arithmetic. These methods derive their power by largely avoiding higher precision arithmetic.

Flowgraphs as presented in section 3 of this paper constitute an amalgamation of two powerful mathematical notions: that of a generating function and that of a directed graph. Flowgraphs are frequently applied in such areas as circuit theory,[27, 28, 29] Markov chains,[19, 23, 36, 39] graph theory,[36]., and coding theory[37].

Two methods for computing the generating function of a flowgraph currently exist. Mason's gain formula[28, 29] provides a topological method which requires the detection of certain sets of paths and loops in the flowgraph. This detection generally turns out to be a difficult pattern recognition problem, and Mason's gain formula does not seem suitable as the basis for a computer algorithm. A systematic (i.e., programmable) method, also due to Mason,[27, 29] is based on a node elimination scheme. However, this method involves computation with rational functions, and the proposed fraction-free method for computing the generating functions of flowgraphs promises a great increase in efficiency. In addition, the application of flowgraphs in solving some enumeration problems from combinatorial analysis is presented.

Implementation

A set of routines called FFP (fraction-free package for the solution of linear equations with multivariate polynomial coefficients) was implemented using the IBM 2250 Scope FORMAC System[43]. These routines are used by two applications programs, FGRAPH and STAT. FGRAPH computes the generating function of a flowgraph according to the algorithm presented in section 3; STAT computes the stationary probability vector of a Markov chain specified by a transition matrix with symbolic elements. The above programs, along with sample output, are presented in section 4.

2. FRACTION-FREE GAUSSIAN ELIMINATION OVER INTEGRAL DOMAINS

In this section integer-preserving elimination methods are extended to arbitrary integral domains and the computational aspects of such methods are investigated in the context of multivariate polynomial domains. This extension makes possible their application to the computer solution of linear equations with symbolic coefficients. A new and elementary proof of the important fraction-free property of these methods is also presented.

Let $A = (a_{ij})$ and $B = (b_{ij})$ be an nxn matrix and nxm matrix respectively over an integral domain D, and consider the m systems of linear equations $AX = B$ with common coefficient matrix A . Note that any number m of inhomogeneous terms are allowed; e.g., B is the identity matrix when the inverse of A is desired.

The n x (n+m) augmented matrix of the given systems is defined by

$$A_{ij}^{(0)} = (a_{ij}^{(0)}) = [A \mid B]. \tag{2.1}$$

Division-Free Gaussian Elimination

With $A^{(0)}$ given by (2.1), a sequence of matrices $A^{(k)} = (a_{ij}^{(k)})$ $(k = 1,2,\ldots,n-1)$ is computed according to

$$a_{ij}^{(k)} = a_{kk}^{(k-1)} a_{ij}^{(k-1)} - a_{kj}^{(k-1)} a_{ik}^{(k-1)} \tag{2.2}$$

$$(i = k+1,\ldots,n; \; j = k+1,\ldots,n+m)$$

where it is implicit that

$$a_{ij}^{(k)} = \begin{cases} a_{ij}^{(k-1)} & (i = 1,\ldots,k; \; j = 1,\ldots,n+m) \\ 0 & (i = k+1,\ldots,n; \; j = 1,2,\ldots,k). \end{cases}$$

Provided that A is nonsingular (2.2) generates matrices $A^{(k)}$ which represent systems of linear equations equivalent* to those represented be $A^{(0)}$, and the left-hand n x n submatrix of $A^{(n-1)}$ (the transformed coefficient matrix) is in upper triangular form.

*The usual account must be taken of zero pivots.

The lower right $(n-k) \times (n-k+m)$ submatrix of $A^{(k)}$ (computed according to (2.2)) is denoted by $A_L^{(k)}$, i.e.,

$$A_L^{(k)} = (a_{ij}^{(k)}) \quad (i = k+1,\ldots,n;\ j = k+1,\ldots,n+m). \tag{2.3}$$

Recall that, in an integral domain D, a divides b or b is divisible by a when a = bc for some c in D.

<u>Theorem 2.1.</u> The minors of order two of $A_L^{(k)}$ are divisible by

$$\prod_{\ell=1}^{k} \left[a_{\ell\ell}^{(\ell-1)} \right]^{k-\ell+1}, \quad \text{for } k=1,2,\ldots,n-2$$

The proof is given in Appendix A.

<u>Corollary.</u> The $a_{ij}^{(k)}$ of $A_L^{(k)}$ are all divisible by

$$\prod_{\ell=1}^{k-1} \left[a_{\ell\ell}^{(\ell-1)} \right]^{k-\ell}, \quad \text{for } k=2,3,\ldots,n-1$$

Proof. Each $a_{ij}^{(k)}$ of $A_L^{(k)}$ $(k = 2,3,\ldots,n-1)$ is a minor of order two of $A_L^{(k-1)}$, for (2.2) may be written as

$$a_{ij}^{(k)} = \begin{vmatrix} a_{kk}^{(k-1)} & a_{kj}^{(k-1)} \\ a_{ik}^{(k-1)} & a_{ij}^{(k-1)} \end{vmatrix} \quad \text{———}$$

The desired result now follows from Theorem 2.1.

The division-free algorithm is now analyzed when the integral domain D is specialized to a (multivariate) polynomial domain $I[x_1,\ldots,x_r]$.

The (total) degree of a polynomial

$$a(x_1,\ldots,x_r) = \sum_{\underline{e}} a_{\underline{e}} x_1^{e_1} x_2^{e_2} \ldots x_r^{e_r} \qquad (a_{\underline{e}} \in I)$$

in $I[x_1,\ldots,x_r]$ is defined by the maximum of the degrees $e_1+e_2+\ldots+e_r$ of the monomials $a_{\underline{e}}\ x_1^{e_1}x_2^{e_2}\ldots x_r^{e_r}$ of $a(x_1,\ldots,x_r)$. The degree function deg a obeys the same rules as the familiar univariate degree function

$$\deg ab = \deg a + \deg b$$

$$\deg a+b \leq \max(\deg a, \deg b)$$

and indeed specializes to the univariate degree function for the case of $I[x_1]$.

Assume now that a system of n linear equations $A\underline{x} = \underline{b}$ has polynomial entries a_{ij} and b_i all of degree d. Then the division-free algorithm (2.2) generates polynomials $a_{ij}^{(k)}$ of degree $d \times 2^k$ at stage k $(k = 1,2,\ldots,n-1)$. For example, applying the division-free algorithm to a 10 x 10 system having first degree polynomial entries results in polynomials of degree $2^9 = 512$. The storage and time required to accommodate such large degree polynomials is intolerable. Moreover, Cramer's rule (and the definition of a determinant) indicates that the solution to such a system consists of numerator and denominator polynomials having maximum degree 10 (generally, degree nd). An elimination algorithm is now considered that yields the results promised by Cramer's rule.

Fraction-Free Gaussian Elimination

With $A^{(0)}$ again given by (2.1), the division-free algorithm (2.2) is modified to: for $K = 1,2,\ldots,n-1$

$$a_{ij}^{(k)} = \frac{a_{kk}^{(k-1)}a_{ij}^{(k-1)} - a_{kj}^{(k-1)}a_{ik}^{(k-1)}}{a_{k-1,k-1}^{(k-2)}} \qquad (2.4)$$

$$(i = k+1,\ldots,n;\quad j = k+1,\ldots,n+m)$$

with $a_{00}^{(-1)} = 1$, where again it is implicit that

$$a_{ij}^{(k)} = \begin{cases} a_{ij}^{(k-1)} & (i = 1,\ldots,k;\ j=1,\ldots,n+m) \\ 0 & (i = k+1,\ldots,n;\ j=1,\ldots,k). \end{cases}$$

Clearly the algorithm (2.4) triangularizes $A^{(0)}$. Although the $a_{ij}^{(k)}$ of (2.4) appear to lie in the field of quotients of the integral domain D, the following theorem shows that the transformations of (2.4) are all fraction-free.

Theorem 2.2. The $a_{ij}^{(k)}$ obtained by (2.4) are in D.

Proof. Consider the $a_{ij}^{(k)}$ generated by the division-free algorithm (2.2), and change the notation there to $b_{ij}^{(k)}$ to distinguish these elements from the $a_{ij}^{(k)}$ generated by algorithm (2.4). The corollary to Theorem 2.1 proves that the $b_{ij}^{(k)}$ of (2.2) are all divisible by $\prod_{\ell=1}^{k-1} \left[b_{\ell\ell}^{(\ell-1)} \right]^{k-\ell}$. Consequently, if it is shown that

$$a_{ij}^{(k)} = \frac{b_{ij}^{(k)}}{\prod_{\ell=1}^{k-1} \left[b_{\ell\ell}^{(\ell-1)} \right]^{k-\ell}} \qquad (2.5)$$

then Theorem 2.2 immediately follows. Equation (2.5) is established by a simple proof by induction, which is given in Appendix A.

The explicit relationship given by (2.5) between the elements generated by the division-free algorithm (2.2) and those generated by the fraction-free algorithm (2.4) seems to be new; but it seems interesting only inasmuch as it provides a new and elementary proof of the fraction-free property of algorithm (2.4). Again, knowledge of (2.4) has been attributed to Jordan.

The proof of Equation (2.5) also establishes the following corollary.

Corollary. For the fraction-free algorithm (2.4), any minor of order two of $A_L^{(k-1)}$ is divisible by $a_{k-1,k-1}^{(k-2)}$.

Theorem 2.3. $|A| = a_{nn}^{(n-1)}$

The proof is given in Appendix A.

Next we analyze the fraction-free algorithm (2.4) again for a system of n linear equations $A\underline{x} = \underline{b}$, with the a_{ij} and b_i polynomials of degree d. Then (2.4) generates polynomials $a_{ij}^{(k)}$ of degree $d(k+1)$, a vast improvement over the division-free algorithm (2.2) in which the degree was $d \times 2^k$, resulting in a great reduction in storage and computation time.

The algorithm of key computational importance for this report is a two-step variant due to E. Bareiss [1] of the fraction-free algorithm (2.4). This algorithm, valid over any integral domain, is readily derived from (2.4) as follows. Applying (2.4) to the numerator of the right-hand side of (2.4) gives

$$a_{ij}^{(k)} = \left[\frac{a_{k-1,k-1}^{(k-2)} a_{kk}^{(k-2} - a_{k-1,k}^{(k-2)} a_{k,k-1}^{(k-2)}}{a_{k-2,k-2}^{(k-3)}} \cdot \frac{a_{k-1,k-1}^{(k-2)} a_{ij}^{(k-2)} - a_{k-1,j}^{(k-2)} a_{i,k-1}^{(k-2)}}{a_{k-2,k-2}^{(k-3)}} \right.$$

$$\left. - \frac{a_{k-1,k-1}^{(k-2)} a_{kj}^{(k-2)} - a_{k-1,j}^{(k-2)} a_{k,k-1}^{(k-2)}}{a_{k-2,k-2}^{(k-3)}} \cdot \frac{a_{k-1,k-1}^{(k-2)} a_{ik}^{(k-2)} - a_{k-1,j}^{(k-2)} a_{i,k-1}^{(k-2)}}{a_{k-2,k-2}^{(k-3)}} \right] \Big/ a_{k-1,k-1}^{(k-2)}$$

$$= \frac{a_{k-1,k-1}^{(k-2)}}{a_{k-2,k-2}^{(k-3)}} \left[\underbrace{\left(\frac{a_{k-1,k-1}^{(k-2)} a_{kk}^{(k-2)} - a_{k-1,k}^{(k-2)} a_{k,k-1}^{(k-2)}}{a_{k-2,k-2}^{(k-3)}} \right)}_{①} a_{ij}^{(k-2)} \right.$$

$$- \underbrace{\left(\frac{a_{k-1,1-1}^{(k-2)} a_{ik}^{(k-2)} \quad a_{k-1,k}^{(k-2)} a_{i,k-1}^{(k-2)}}{a_{k-2,k-2}^{(k-3)}} \right)}_{②} a_{kj}^{(k-2)}$$

$$\left. + \underbrace{\left(\frac{a_{k,k-1}^{(k-2)} a_{ik}^{(k-2)} - a_{kk}^{(k-2)} a_{i,k-1}^{(k-2)}}{a_{k-2,k-2}^{(k-3)}} \right)}_{③} a_{k-1,j}^{(k-2)} \right] \Big/ a_{k-1,k-1}^{(k-2)}$$

$$- \frac{1}{\left[a^{(k-3)}_{k-2,k-2}\right]^{2}} \left\{ a^{(k-2)}_{k-1,k} a^{(k-2)}_{k,k-1} a^{(k-2)}_{k-1,j} a^{(k-2)}_{i,k-1} - a^{(k-2)}_{k-1,j} a^{(k-2)}_{k,k-1} a^{(k-2)}_{k-1,k} a^{(k-2)}_{i,k-1} \right\} .$$

Now the term within curly braces $\left[\left\{ \ \right\}\right]$ vanishes and thus need not be computed—this is what reduces computations when the two-step method is used instead of the one-step method. Now the expression in parentheses on the line marked " ① " above is equal to $a^{(k-2)}_{k-1,\ k-1}$ and the expression in parantheses on the line marked " ② " above is equal to $a^{(k-2)}_{ik}$; hence both are in the integral domain D by Theorem 2.2. Also the numerator of the expression in parentheses on the line marked " ③ " above is a minor of order two of $A^{(k-2)}_{L}$ and hence, by the corollary to Theorem 2.2 is divisible by $a^{(k-3)}_{k-2,k-2}$. Thus expression " ③ " in parentheses is in D. Next we can cancel $a^{(k-2)}_{k-1,k-1}$. Finally we note that the entire bracketed term is divisible by $a^{(k-3)}_{k-2,k-2}$ because $a^{(k)}_{ij}$ is in D, again by Theorem 2.2. This establishes the fraction-free property in any integral domain of the following algorithm.

Two-Step Fraction-Free Gaussian Elimination Algorithm (Bareiss)

$$a^{(-1)}_{00} = 1; \quad a^{(0)}_{ij} = a_{ij}; \quad a^{(0)}_{i,n+m} = b_{im}$$

$$\begin{aligned}
c^{(k-2)}_{0} &= (a^{(k-2)}_{k-1,k-1} a^{(k-2)}_{kk} - a^{(k-2)}_{k-1,k} a^{(k-2)}_{k,k-1}) \Big/ a^{(k-3)}_{k-2,k-2} \\
c^{(k-2)}_{i1} &= (a^{(k-2)}_{k-1,k} a^{(k-2)}_{i,k-1} - a^{(k-2)}_{k-1,k-1} a^{(k-2)}_{ik}) \Big/ a^{(k-3)}_{k-2,k-2} \qquad (2.6) \\
c^{(k-2)}_{i2} &= (a^{(k-2)}_{k,k-1} a^{(k-2)}_{ik} - a^{(k-2)}_{kk} a^{(k-2)}_{i,k-1} \Big/ a^{(k-3)}_{k-2,k-2} \\
a^{(k)}_{ij} &= (a^{(k-2)}_{ij} c^{(k-2)}_{0} + a^{(k-2)}_{kj} c^{(k-2)}_{i1} + a^{(k-2)}_{k-1,j} c^{(k-2)}_{i2}) \Big/ a^{(k-3)}_{k-2,k-2}
\end{aligned}$$

for $i = k+1, \ldots n; \quad j = k+1, \ldots n+m$

with the pivot row computed by the one-step formula

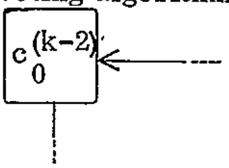
$$a_{kk}^{(k-1)} = c_0^{(k-2)}$$

$$a_{k\ell}^{(k-1)} = (a_{k-1,k}^{(k-2)} a_{k\ell}^{(k-2)} - a_{k-1,\ell}^{(k-2)} a_{k,k-1}^{(k-2)}) \Big/ a_{k-2,k-2}^{(k-3)}$$

for $\ell = k+1, \ldots, n+m$.

The proper sequencing of computation is presented in the flowchart of Figure 1.1 which implements some modifications in the logical flow of control shown on page 569 of reference 1 (see also the following remarks).

A pivoting algorithm for two-step Gaussian elimination is necessarily more complicated than one for a one-step method because of the necessity of checking two pivot elements, $a_{kk}^{(k-1)}$ and $a_{k-1,k-1}^{(k-2)}$, for zero. The reader is referred to the pivoting algorithm of reference 1, Figure 3 which replaces



of Figure 1.1 of this paper.

Comparing Figure 1 of reference 1 with Figure 1.1 of this paper, it is observed that the flow of control is different. Our alteration corrects an error in reference 1, due to the last pivot element $a_{n-1,n-1}^{(n-2)}$ of an even order matrix not being tested for zero; e.g., failure would occur for a coefficient matrix of the form

$$\begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & 0 & * \\ 0 & 0 & * & * \end{pmatrix}$$

Efficiency also increases slightly by assigning the previously computed value $c_0^{(k-2)}$ to $a_{kk}^{(k-1)}$. Also $a_{nn}^{(n-1)}$ should be checked to be sure it is nonzero before exiting as a final test that the coefficient matrix is nonsingular.

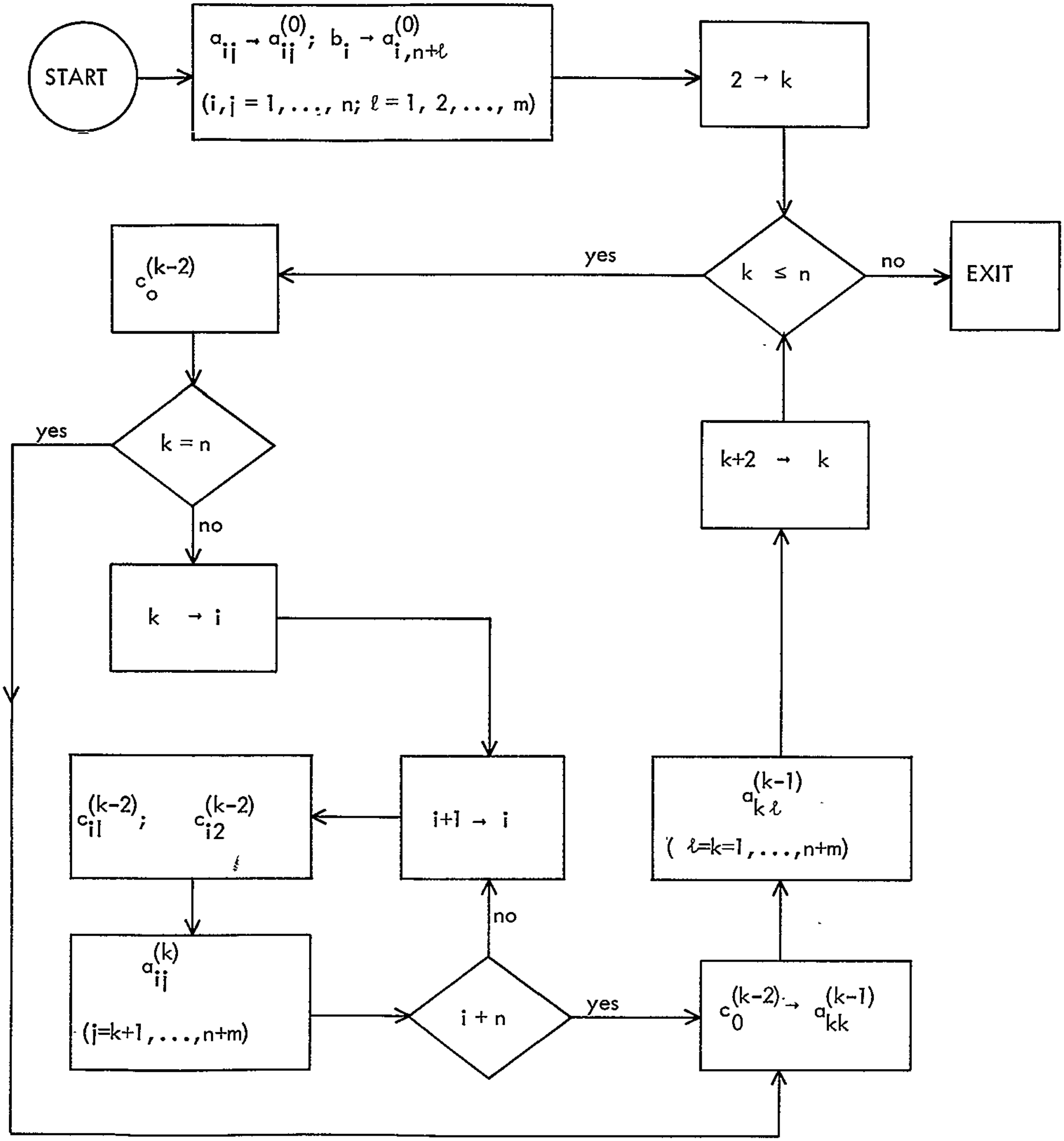


Figure 1.1. Two-Step Fraction-Free Gaussian Elimination

The analysis of the fraction-free elimination algorithms (2.4) and (2.6) is given in terms of the number of integer multiplications and divisions required to carry them out over the multivariate polynomial domain $I[x_1 \ldots, x_r]$.

Let $a(x_1, \ldots, x_r)$ be a polynomial with integer coefficients of degree d in each of r variables of the form

$$a(\underline{x}) = \sum_{e_1=0}^{d} \ldots \sum_{e_r=0}^{d} a_{\underline{e}} x_1^{e_1} \ldots x_r^{e_r} \quad (a_{\underline{e}} \in I) \ .$$

Note that the total degree of such a polynomial is rd. Clearly $a(\underline{x})$ has $(d+1)^r$ terms, assuming there are no missing terms. For simplicity of analysis, this assumption is made throughout.

If $a(\underline{x})$ and $b(\underline{x})$ are two polynomials in r variables of degree d and e in each variable respectively, then the number of integer multiplications required to compute $a(\underline{x})b(\underline{x})$ is

$$[(d+1)(e+1)]^r \tag{2.7}$$

and the number of integer multiplications and divisions required to compute $a(\underline{x})/b(\underline{x})$ assuming, of course, that $b(\underline{x})$ divides $a(\underline{x})$, is

$$[(d-e+1)(e+1)]^r \ . \tag{2.8}$$

Furthermore, the degree of $a(\underline{x})b(\underline{x})$ is d+e and the degree of $a(\underline{x})/b(\underline{x})$ is d-e.

Now consider m systems of n linear equations in n unknowns AX = B with a common coefficient matrix A, and assume that the a_{ij} and b_{ij} of A and B are all polynomials of degree d in each of r variables. Then, referring to (2.4) and (2.6), the $a_{ij}^{(0)}$ are of degree d, the $a_{ij}^{(1)}$ are of degree 2d, and in general the $a_{ij}^{(k)}$ are of degree (k+1)d in each variable.

Let $N_S(m,n,d,r)$ and $N_T(m,n,d,r)$ be the number of integer multiplications and divisions required to carry out the single-step algorithm (2.4) and the two-step algorithm (2.6) respectively. From (2.4) follows

$$N_S(m,n,d,r) = \sum_{k=1}^{n-1} (n-k)(n-k+m)\ [2(kd+1)^{2r} + (kd+d+1)^r(kd-d+1)^r]. \tag{2.9}$$

For the two-step algorithm (2.6) care must be taken to distinguish between n even and n odd. If we define q by n-1 if n is odd and by n-2 if n is even, from (2.6) follows

$$N_T(m,n,d,r) = \overline{\sum_{k=2,4,\ldots,q}} (t_1 + t_2 + t_3 + t_4) \quad + t_5 \tag{2.10}$$

where

$$t_1 = 2(kd-d+1)^{2r} + (kd+1)^r(kd-2d+1)^4$$

$$t_2 = 2(n-k)[2(kd-d+1)^{2r} + (kd+1)^r(kd-2d+1)^r]$$

$$t_3 = (n-k)(n-k+m)[3(kd-d+1)^r(kd+1)^r + (kd+d+1)^r(kd-2d+1)^r]$$

$$t_4 = (n-k+m)\ [2(kd-d+1)^{2r} + (kd+1)^r(kd-2d+1)^r]$$

and

$$t_5 = \begin{cases} 0 \text{ if n is odd} \\ (m+1)[2(nd-d+1)^{2r} + (nd+1)^r(nd-2d+1)^r] \text{ if } n \text{ is even.} \end{cases}$$

The term t_1 corresponds to the computation of $c_0^{(k-2)}$, t_2 to $c_{i1}^{(k-2)}$ and $c_{i2}^{(k-2)}$ $(i = k+1,\ldots,n)$, t_3 to $a_{ij}^{(k)}$ $(i = k+1,\ldots,n;\ j = k+1,\ldots,n+m)$, and t_4 to $a_{k\ell}^{(k-1)}$ $(\ell = k+1,\ldots,n+m)$. For n even, t_5 corresponds to the computation of $a_{n\ell}^{(n-1)}$ $(\ell = n,\ldots,n+m)$.

$$\lim_{mn\to\infty} \frac{N_T}{N_S} = \frac{c_1 d_1^r}{a_1 b_1^r}$$

From (2.9)

$$a_1 = 2(m+1), \; b_1 = (nd-d+1)^2$$

In determining c_1 and d_1 from (2.10) there are two cases to consider:

a. n odd in which case c_1 and d_1 are determined from t_3 of (2.10) as

$$c_1 = 3(m+1), \; d_1 = (nd-2d+1)(nd-d+1)$$

b. n even in which case c_1 and d_1 are determined from t_5 of (2.10) as

$$c_1 = 2(m+1), \; d_1 = (nd-d+1)^2 \; .$$

Noting that $b_1 > d_1$ for n odd and that $a_1 = c_1$ and $b_1 = d_1$ for n even, it follows that

$$\lim_{r\to\infty} \frac{N_T}{N_S} = \begin{cases} 0 \text{ for n odd} \\ 1 \text{ for n even} \end{cases} \tag{2.13}$$

Thus we have the rather surprising result that for higher variate polynomial systems the two-step method offers a potentially unbounded increase in efficiency over the one-step method, provided that the coefficient matrix is of odd order. Conversely, when the coefficient matrix is of even order, then in the limit as r →∞ the two-step method offers no advantage whatsoever over the one-step method. Moreover, the limiting behavior of (2.13) is strongly exhibited for very moderate values of r as indicated by Table 2-1, where the ratio N_T/N_S is computed with m and d arbitrarily set equal to unity.

To ascertain the computational savings afforded by the two-step algorithm (2.6) over the one-step algorithm (2.4) it is useful to examine the ratio $N_T(m,n,d,r)/N_S(m,n,d,r)$ in two limiting cases:

a. large mn (large systems of linear equations)

b. large r (systems with high-variate polynomials).

As for case a, first observe that the number of integer multiplications and divisions in carrying out the transformation $A^{(k-2)} \rightarrow A^{(k)}$ by the one-step algorithm is given from (2.9) by

$$(n-k+1)(n-k+m+1)[2(kd-d+1)^{2r} + (kd+1)^r(kd-2d+1)^r]$$
$$+ (n-k)(n-k+m)[2(kd-1)^r + (kd+d+1)^r(kd-d+1)^r] .$$

Comparing the above with the term t_3 of (2.10), which dominates for large mn, it follows that, for fixed r and d,

$$\lim_{mn\to\infty} \frac{N_T}{N_S} = \frac{2}{3} \quad . \tag{2.11}$$

Thus for large mn, the two-step method requires only two-thirds the number of operations required by the one-step method.

Turning now to case b (large r), it is observed from (2.9) and (2.10) that N_S and N_T have the form

$$N_S(m,n,d,r) = \sum_{i=1}^{u} a_i b_i^r$$
$$N_T(m,n,d,r) = \sum_{i=1}^{v} c_i d_i^r \tag{2.12}$$

where the a_i, b_i, c_i, d_i, u and v are functions of m, n, and d but are independent of r. Assuming in (2.12) that $b_1 > b_2 > b_3 \ldots$ and $d_1 > d_2 > d_3 \ldots$, it follows for fixed m, n, and d that

Table 2-1

SAMPLE VALUES OF N_T/N_S

$(m = d = 1)$

r \ n	2	3	4	5
0	1.	.96	.90	.87
1	1.	.83	.82	.81
2	1.	.64	.76	.75
3	1.	.45	.77	.69
4	1.	.29	.82	.60
5	1.	.19	.88	.50
6	1.	.12	.93	.41

For example, Table 1 indicates that, for a system of three linear equations in three unknowns, $A\underline{x} = \underline{b}$ with the a_{ij} and b_i polynomials of degree 1 in each of three variables. Then the two-step algorithm (2.6) requires less than half, specifically .45, the number of integer multiplications and divisions taken by the one-step algorithm (2.4). However, also note that the advantage of the two-step over the one-step algorithm rapidly disappears with increasing r when $n = 4$, as indicated by (2.13).

The above analysis should be interpreted only in a relative sense, i.e., in indicating the relative superiority of the two-step method over the one-step method. Thus there is nothing, in an absolute sense, to be gained by augmenting a system of linear equations with an even order coefficient matrix in order to obtain an odd order coefficient matrix.

Finally, two distinct limiting behaviors were established for N_T/N_S, one for large mn (2.11), and one for large r (2.13). If both mn and r become large, then (2.13) determines the limiting behavior of N_T/N_S because of the strong exponential dependence of N_T and N_S on r.

Fraction-Free Back-Substitution

Both the one-step (2.4) and two-step (2.6) fraction-free algorithms produce an $n \times (n+m)$ matrix $A^{(n-1)}$ with the left-hand $n \times n$ submatrix (the transformed coefficient matrix of the systems $AX=B$) in upper triangular form. The solution vectors $(\underline{x}_1, \ldots, \underline{x}_\ell, \ldots, \underline{x}_m)$ are then given by the familiar back-substitution formula

$$x_{n,\ell} = \frac{a^{(n-1)}_{n,n+\ell}}{a^{(n-1)}_{nn}} \tag{2.14}$$

$$x_{i,\ell} = \frac{1}{a^{(i-1)}_{ii}} \left[a^{(i-1)}_{i,n+\ell} - \sum_{j=i+1}^{n} a^{(i-1)}_{ij} \, x_{j\ell} \right]$$

$(i = n-1, \ldots, n)$.

The division operation in (2.14) is not generally exact; the $x_{i\ell}$'s are in the field of quotients of the integral domain over which the given systems of linear equations are specified. However by Cramer's rule, the $x_{i\ell}$'s have a common denominator $|A|$. Writing $x_{i\ell} = y_{i\ell} / |A|$ and invoking Theorem 2.3 that $a^{(n-1)}_{nn} = |A|$, the following fraction-free back-substitution formula for the $y_{i\ell}$'s is obtained:

$$y_{n\ell} = a^{(n-1)}_{n,n+\ell} \tag{2.15}$$

$$y_{i\ell} = \frac{1}{a_{ii}} \left[a^{(i-1)}_{i,n+\ell} \, a^{(n-1)}_{nn} - \sum_{j=i+1}^{n} a^{(i-1)}_{ij} y_{j\ell} \right]$$

$(i = n-1, \ldots, 1)$.

By Cramer's rule, the division operation in (2.15) is exact. Reduction to lower terms may be possible for the quotients $x_{i\ell} = \frac{y_{i\ell}}{|A|}$ in the case of Gaussian domains. To determine quotients in lowest terms (i.e., with numerator and denominator relatively prime) the application of a GCD algorithm is necessary.

3. FLOWGRAPHS AND THEIR GENERATING FUNCTIONS

In this section, we define flowgraphs and their generating functions.

As devised by Mason (see references 27 and 28, and reference 27, Chapter 4), a flowgraph is a graphical representation of a system of linear equations. This point of view is pursued by showing both the equivalence of the concepts of the flowgraph generating function as defined in this paper to flowgraph gain or transmission as defined by Mason, and by showing the applicability of Mason's node elimination scheme (see reference 29, Sections 4-1 to 4-6) to computing the generating function of a flowgraph.

A flowgraph technique for solving certain kinds of enumeration problems from combinatorial analysis and automata theory is presented and examples are given to illustrate its value.

This section also gives a new method for computing the generating functions of flowgraphs, based on the two-step fraction-free elimination algorithm (2.6), and compares this method with Mason's node elimination algorithm for computational efficiency.

Basic Definitions

Flowgraph. A flowgraph $F = (\mathcal{G}, s, f, \psi)$ over an integral domain D consists of:

a. A directed graph $\mathcal{G} = [n, A, \phi]$ specified by a set $\underline{n} = \{1,2,\ldots,n\}$ of nodes, a set A of arcs, and a function $\phi : A \rightarrow \underline{n} \times \underline{n}$ which associates with each arc a its endpoints $\phi(a) = (i,j)$;

b. A starting node $s \in \underline{n}$ and a final node $f \in \underline{n}$; and

c. A function $\psi : A \rightarrow D$ which associates a label $\psi(a) \in D$ with each arc $a \in A$.

Transmission Matrix. The transmission matrix $T = (t_{ij})$ of the flowgraph F is the n x n matrix defined by: for i, j $\in \underline{n}$

$$t_{ij} = \overline{\sum_{\{a:\ \phi(a)=(i,j)\}}} \psi(a) \ ; \tag{3.1}$$

where t_{ij} is called the branch transmission from node i to node j.

Generating Function. The generating function G of the flowgraph F is defined by

$$G = [(I-T)^{-1}]_{sf} \tag{3.2}$$

i.e., G is the $(s,f)^{th}$ element of $(I-T)^{-1}$, where I is the identity matrix. Clearly G exists when $|I-T| \neq 0$.

Consider now a special case of the above which arises when the transmission matrix T has the form T = zA, where A is an n x n matrix (a_{ij}) over an integral domain D. Each branch transmission t_{ij} is a monomial $a_{ij}z$ in the polynomial domain D[z]. From (3.2) the generating function in this case is given by

$$\begin{aligned} G &= [(I-zA)^{-1}]_{sf} \qquad (3.3) \\ &= [\sum_{k=0}^{\infty} z^k A^k]_{sf} \\ &= \sum_{k=0}^{\infty} a_{sf}^{(k)} z^k \end{aligned}$$

where $a_{sf}^{(k)}$ is the (s,f) element of A^k. Thus the flowgraph generating function G is the generating function (in the usual mathematical sense) of the sequence $\left\{a_{sf}^{(k)}\right\}_{k=0}^{\infty}$.

Furthermore, a closed form may be obtained for the power series $\sum_{k=0}^{\infty} a_{sf}^{(k)} z^k$ by observing from (3.3) that

$$G = \frac{C_{fs}}{|I-zA|} \tag{3.4}$$

where C_{fs} is the cofactor of the $(f,s)^{th}$ element of $I-zA$. Thus the generating function of the sequence $\{a_{sf}^{(k)}\}$ is a rational function of the indeterminate z. Also note that the existence and uniqueness of the power series expansion for the rational function (3.4) follows from purely algebraic considerations.

For the denominator polynomial $|I-zA| = 1 + \sum_{i=1}^{n} s_i z^i$ considered as an element of D[[z]], the domain of formal power series over D has an invertible constant term, namely unity; hence it is invertible in D[[z]] for any integral domain D (for example see reference 2, Chapter 13).

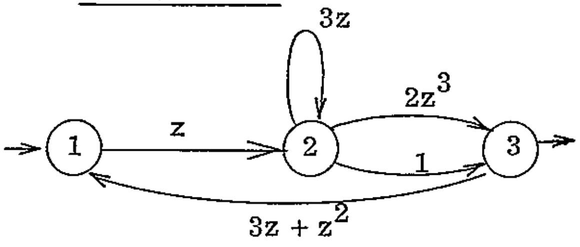
Two choices for the matrix A give rise to classes of flowgraphs which have received much attention in the literature.

a. A is the transition matrix of a Markov chain. Then the higher transition probabilities are given by successive powers of the matrix A (for example see reference 11, Chapter 7) so that $a_{sf}^{(k)}$ in (3.3) is the probability of a transition from the initial state s to the final state f in k steps. The flowgraph analysis of Markov chains originated with the work of Sittler[39] and Huggins.[19] See also Howard[18], Lorens[23], and Ramamoorthy and Tufts.[36]

b. A is the adjacency matrix of a multi-graph, with a_{ij} equal to the number of arcs (paths of length one) from node i to node j. Then the element $a_{ij}^{(k)}$ of A^k is the number of paths from node i to node j of length k (for example see reference 17, Chapter 2). Thus (3.3) and (3.4) in this case yield a path enumeration generating function with respect to a fixed starting node s and final node f of the given multigraph. The generating function approach to path enumeration was considered by Ramamoorthy and Tufts[36] using flowgraph theory, and by Kasteleyn, who does not explicitly use flowgraphs, in reference 17, Chapter 2.

Since our definition of a flowgraph does not restrict the branch transmissions to monomials, a branch transmission may be a polynomial or power series. For example, in a path enumeration problem the branch transmission $t_{ij} = 2z + 4z^3 + 5z^7$ might indicate that there are two paths of length 1, four of length 3, and five of length 7 associated with arc(s) for node i to node j. Similarly, $t_{ij} = (1-az)^{-1}$ might convey the information that there are a^k paths of length k for k = 0,1,2,... associated with the arc(s) from node i to node j. The coefficient g_k of the power series expansion of the generating function $G = \sum_{k=0}^{\infty} g_k z^k$ is then the total number of paths of length k from the starting node s to the final node f. An example involving the above concepts follows. The computations were carried out using the Scope FORMAC program presented in the next section.

<u>Example 3.1.</u>



Flowgraph F
(s=1, f=3)

$$T = \begin{pmatrix} 0 & z & 0 \\ 0 & 3z & 1+2z^3 \\ 3z+z^2 & 0 & 0 \end{pmatrix}$$

Associated Transmission Matrix

$$G = [(I-T)^{-1}]_{13}$$

$$= \frac{z+2z^4}{1-3z-3z^2-z^3-6z^5-2z^6}$$

The first few g_k 's in the power series expansion $G = \sum_{k=0}^{\infty} g_k z^k$ are:

$$\begin{array}{lll}
g_0 = 0 & \qquad & g_4 = 48 \\
g_1 = 1 & & g_5 = 183 \\
g_2 = 3 & & g_6 = 711 \\
g_3 = 12 & & g_7 = 2750 \ .
\end{array}$$

Thus from node 1 to node 3 there are no paths of length 0, one of length 1, three of length 2, twelve of length 3, etc. Also, complex variable techniques may be used to obtain information about the g_k 's. For example, the asymptotic rate of growth of the g_k 's is given by

$$\lim_{k \to \infty} \frac{g_{k+1}}{g_k} = \frac{1}{R}$$

where R is a root of the denominator polynomial of G which is smaller in absolute value than all other roots. For the above generating function, $R \cong .26$.

Flowgraphs and Linear Equations; Flowgraph Reduction

Let F be a flowgraph with n nodes $\underline{n} = \overline{\{1,2,\ldots,n\}}$, starting node s, final node f, and transmission matrix $T = (t_{ij})$. An augmented flowgraph $\tilde{F}$ is constructed from F by introducing two new nodes, a source node 0 and a sink node f, with branch transmissions as shown in Figure 3.1.

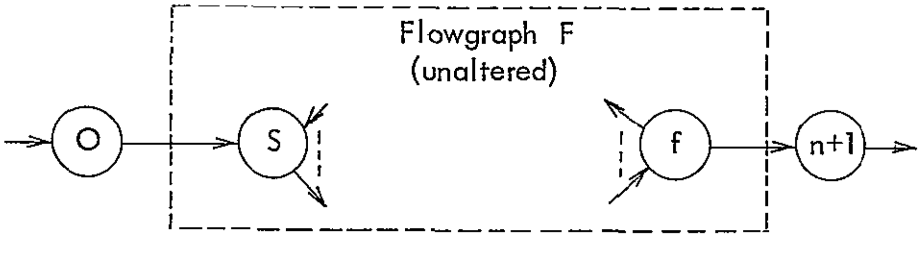


Figure 3.1. Flowgraph $\tilde{F}$

A system of linear equations is associated with $\tilde{F}$ as follows: the source node node 0 represents an independent variable x_0, and the nodes $1, 2, \ldots, n+1$ represent dependent variables $x_1, x_2, \ldots, x_{n+1}$. These variables are related according to the equations

$$
\begin{aligned}
x_j &= \sum_{i=1}^{n} x_i t_{ij} && (j = 1, 2, \ldots, n;\ j \neq s) \qquad (3.5)\\
x_s &= x_0 + \sum_{i=1}^{n} x_i t_{is} \\
x_{n+1} &= x_f \ .
\end{aligned}
$$

By eliminating the dependent variables $x_1, x_2, \ldots, x_n$ from (3.5), a relationship

$$x_{n+1} = G\, x_0 \qquad (3.6)$$

is obtained. Mason refers to G as the gain or transmission of the flowgraph F.

<u>Theorem 3.1.</u> $G = [(I-T)^{-1}]_{sf}$.

The proof is given in Appendix A. Theorem 3.1 states that the gain G of (3.6) is precisely the generating function of the given flowgraph F.

Mason's node elimination algorithm is essentially a graphical variant of Gauss-Jordan elimination. Consider the elimination of a variable x_j in (3.5). Now

$$x_j = \sum_{\substack{i=1\\ i\neq j}}^{n} x_i \frac{t_{ij}}{1-t_{jj}}$$

and substituting for x_j in an equation

$$x_k = \sum_{i=1}^{n} x_i t_{ik}$$

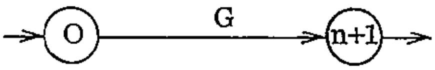
gives

$$x_k = \sum_{\substack{i=1 \\ i \neq j}}^{n} x_i \left(t_{ik} + \frac{t_{ij} t_{kj}}{1-t_{jj}} \right) . \tag{3.7}$$

Equation (3.7) states that when node j (variable x_j) is eliminated, the relationship between node i (variable x_i) and node k (variable x_k) implied by (3.5) is retained if the branch transmission t_{ik} is replaced by

$$t'_{ik} = t_{ik} + \frac{t_{ij} t_{kj}}{1-t_{jj}} . \tag{3.8}$$

Thus a new flowgraph is obtained with one fewer node and with branch transmissions given by (3.8). The dependent nodes 1,2,...,n are eliminated in turn; finally the flowgraph corresponding to (3.6) is obtained



with G the generating function of the original flowgraph.

Note from (3.8) that in eliminating a node, say j, the transmission function from node i to node k is altered only if node i and node k are adjacent to node j. Herein lies the power of the node elimination scheme: in eliminating a single node from a large flowgraph one need only be concerned with a (usually) small part of the total graph, namely those nodes adjacent to the one being eliminated. This principle is illustrated below in Figure 3.2. (Note in particular how the relationship between node i and itself is preserved when node j is eliminated.)

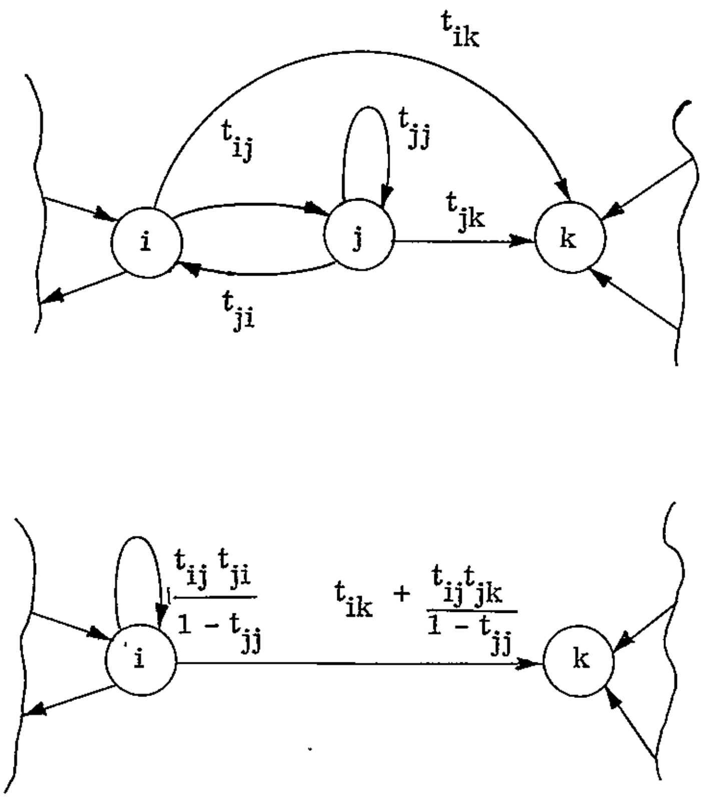

t_{ik}
t_{ij}
t_{jj}
t_{jk}
t_{ji}
i
j
k
$\frac{t_{ij}\,t_{ji}}{1 - t_{jj}}$
$t_{ik} + \frac{t_{ij}t_{jk}}{1 - t_{jj}}$
i
k

Figure 3.2. Node Elimination

The two flowgraphs of Figure 3.2 are equivalent (≡) in the sense that they represent equivalent systems of linear equations.

Application of Flowgraphs to Enumeration Problems in Automata Theory and Combinatorics

The applications of flowgraphs to the analysis of Markov chains [19, 39] and to the enumeration of comma-free code words [37] have suggested to this author the possibility of applying flowgraphs to problems of a more general combinatorial nature.

Many combinatorial problems can be cast in the following form: "over an alphabet A determine the number of sequences $a_1 a_2 \ldots a_n (a_i \in A)$ of length n having some prescribed property P''. The reader is referred to Sections 3-4 and 3-5 of Liu[22] for several examples of problems of the above form.

A method to solve such problems, based on elementary automata theory and flowgraph theory, is now proposed. An outline of the method follows.

Step 1. Construct a machine M (automation, recognition device, Rabin-Scott automation-see, for example, reference 16, Chapter 3 and reference 34) with input alphabet A that accepts or recognizes precisely those sequences specified by the property P.

Step 2. Transform the state diagram of the machine M to an appropriate flowgraph F to enumerate all paths from the starting state to any final state of the machine M.

Step 3. Compute the generating function (3.2) of the flowgraph F.

The flowgraph generating function $G = \sum_{n=0}^{\infty} g_n z^n$ obtained in step 3 then yields the solution to the given combinatorial problem, i.e., g_n is the number of sequences of length n having the prescribed property P. Thus the flowgraph generating function G is precisely what one would get using classical generating function techniques, as described in Riordan (reference 44, Chapter 2) and Liu (reference 22, Chapters 2 and 3). However, the author has found that many problems requiring at least a modicum of ingenuity to solve using traditional combinatorial techniques can be handled $\overline{\text{routinely}}$ using the proposed method. Some examples follow.

Example 3.2. (See reference 22, page 77). Find the number of n-digit binary sequences that have the pattern 010 occurring for the first time in the nth digit.

Step 1.

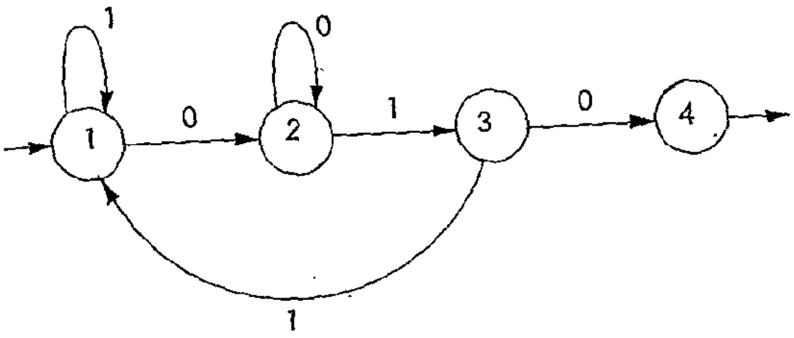


Figure 3.3a Machine M

Step 2.

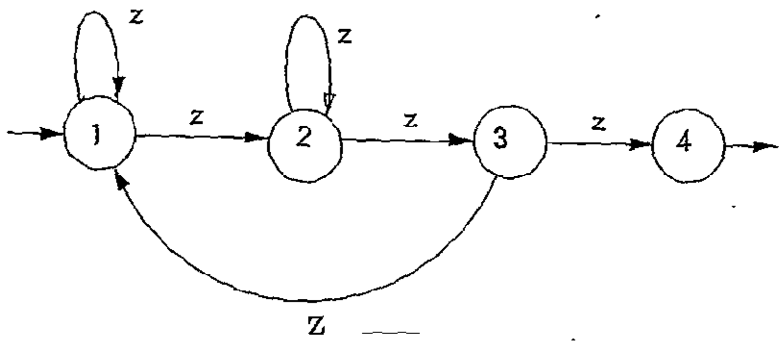


Figure 3.3b. Flowgraph F

Step 3.

$$T = \begin{pmatrix} z & z & 0 & 0 \\ 0 & z & z & 0 \\ z & 0 & 0 & z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$G = [(I-T)^{-1}]_{14}$$

$$= \frac{z^3}{1-2z+z^2-z^3}$$

$$= z^3 + 2z^4 + 3z^5 + 5z^6 + \ldots$$

Thus there is one sequence of length 3, two of length 4, three of length 5, etc. Note that the machine M of Figure 3.3a is incompletely specified in that no transition is indicated when the machine is in state 4 and an input of 1 is received. It is assumed in this case that a transition is made to some absorbing dead state. The generating function is clearly unaltered by ignoring a dead state or any other state from which there is no path to some final state.

Example 3.3. The machine of Figure 3.4a appears in reference 30, p. 165. The generating function G of flowgraph 3.4b enumerates the number of binary sequences of length n recognized by the given machine.

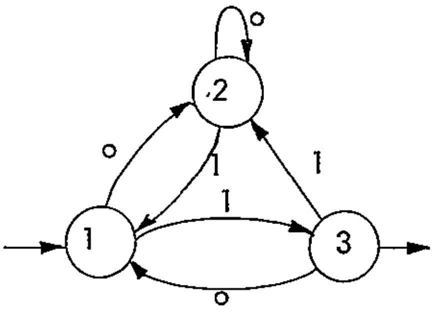


Figure 3.4a

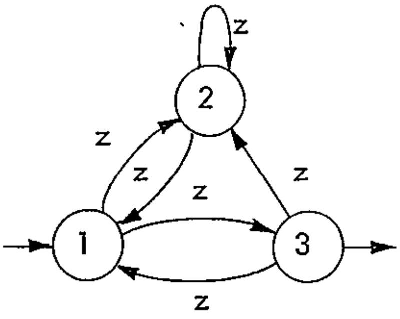


Figure 3.4b

$$T = \begin{pmatrix} 0 & z & z \\ z & z & 0 \\ z & z & 0 \end{pmatrix}$$

$$\begin{aligned} G &= [(I-T)^{-1}]_{13} \\ &= \frac{z-z^2}{1-z-2z^2} \\ &= \frac{\frac{2}{3}\, z}{1+z} + \frac{\frac{1}{3}\, z}{1-2z} \\ &= \sum_{n=1}^{\infty} \frac{2^{n-1} + 2(-1)^{n-1}}{3}\, z^n \end{aligned}$$

Thus the machine recognizes $\frac{2^{n-1} + 2(-1)^{n-1}}{3}$ sequences of length n.

The flowgraph approach to the above enumeration problem also admits the potentially useful generalization of analyzing a machine which processes different symbols in different lengths of time. If the automation requires t_i units time to process an input symbol a_i, then the branches of the flowgraph corresponding to an a_i transition are labeled z^{ti}. The generating function $G = \sum_{t=0}^{\infty} g_t z^t$

then enumerates the number of sequences g_t recognized by the automation at time t.*

The next example considers an infinite machine.and flowgraph.

Example 3.4. In reference 31, Section 4.2.2 Minsky discusses the problem of recognizing "grammatical" or "well-formed" sequences of parentheses; e.g., (()), ( ) ( ) , and ( ) (( )) are well-formed, while (,) ( )(, and (( ))) are not. (See reference 31, Section 4.2.3 for further details.) Consider now determining the enumerator $G = \sum_{n=0}^{\infty} g_n z^n$ for the numbers g_n of well-formed sequences of parentheses of length n. The infinite state automation of Figure 3.5a recognizes precisely those sequences of parentheses that are well-formed, and the corresponding infinite flowgraph of Figure 3.5b serves as a parenthesis counter.

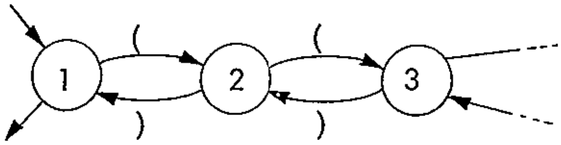


Figure 3.5a. Parenthesis-Checking Machine

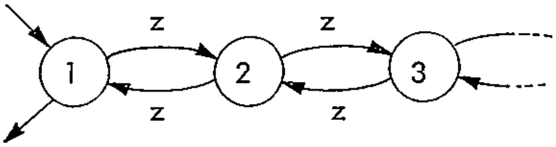


Figure 3.5b. Parenthesis-Counting Flowgraph

Since the transmission matrix T of Figure 3.5b is an infinite matrix, formula (3.2) for the generating function G is not useful for computational

*In reference 21 this problem is treated (using flowgraphs) from a regular expression viewpoint.

purposes. Instead, G will be computed using the node elimination scheme discussed earlier. Eliminating nodes 2, 3, 4,... in Figure 3.5b gives the equivalent flowgraph of Figure 3.5c, while eliminating nodes 3,4,5,... gives the equivalent flowgraph of Figure 3.5d.

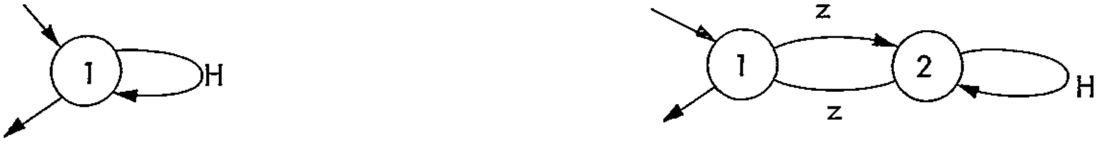


Figure 3.5c

Figure 3.5d

Equivalent Flowgraphs

When we eliminate node 2 in Figure 3.5d and equate the resultant transmission from node 1 to itself to H in Figure 3.5c, we get

$$H = \frac{z^2}{1-H}$$

whence

$$H = \frac{1 - \sqrt{1-4z^2}}{2}$$

the negative sign for the square root being chosen so that the series expansion for H has positive terms. The desired generating function G is then given by

$$G = \frac{1}{1-H}$$

$$= \frac{1 - \sqrt{1-4z^2}}{2z^2}$$

$$= \sum_{n=0}^{\infty} \left[\frac{(2n)!}{n!(n+1)!}\right] z^{2n} .$$

The last equality follows from application of the binomial theorem and subsequent simplification. Thus there are $\frac{(2n)!}{n!\,(n+1)!}$ well-formed sequences of parentheses of length 2n, e.g., there are 14 sequences of length 8 which agree with the enumeration given by Minsky in reference 31, page 75.

It is well known in automaton theory that the properties and capabilities of infinite machines are quite different from those of finite machines. Therefore it is interesting to note that their sequence-enumerating generating functions have different analytic properties: for finite machines the generating function is always a rational function, whereas for infinite machines a generating function with a branch point has been encountered.*

The previous example in conjunction with Theorem 3.1 suggests using flowgraphs to compute specific elements in the inverse of an infinite order matrix with a finite periodic structure. Thus element (1,1) of the inverse of the infinite tridiagonal matrix

$$I-T = \left|\begin{array}{cccccc} 1 & -z & & & & \\ -z & 1 & -z & \bigcirc & & \\ & -z & 1 & -z & & \\ & \bigcirc & & \ddots & & \\ & & & -z & 1 & -z \\ & & & & & \ddots \end{array}\right.$$

is determined as the generating function of flowgraph 3.5b, which has transmission matrix T and node 1 as both the starting and final node. From Example 3.4 the two values for this inverse element are

$$\frac{1 \pm \sqrt{1-4z^2}}{2z}$$

The final example illustrates application of the machine flowgraph technique in deriving a bivariate generating function.

*In reference 39, page 265, the flowgraph analysis of a 2-way infinite random walk results in a generating function which also has a branch point singularity due to a square root.

Example 3.5. (See reference 22, page 83). Find the number of n-digit binary sequences with exactly r pairs of adjacent 1's and no adjacent 0's. (Note: The sequence 111 has two pairs of adjacent 1's.)

A machine M which recognizes sequences having no adjacent 0's is given in Figure 3.6a.

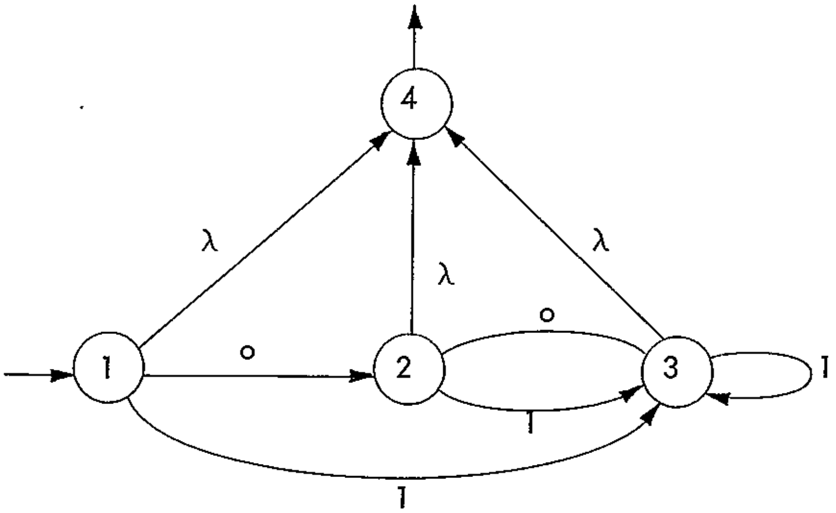


Figure 3.6a. Sequence Recognizing Machine

State 4 which is accessible from states 1, 2, and 3 via the empty tape λ is introduced in order to have a single final state—any sequence accepted by state 1, 2, or 3 is also accepted by state 4, and conversely any sequence accepted by state 4 is accepted by state 1, 2, or 3.

What is desired is the generating function $G(x,y) = \sum_{n,r} a_{n,r} x^n y^r$, where $a_{n,r}$ is the number of sequences of length n having exactly r pairs of adjacent 1's. In the flowgraph corresponding to machine M, the branches are labeled with bivariate monomials $x^{\epsilon_1} y^{\epsilon_2}$, in which $\epsilon_1 = 1$ indicates the occurrence of a 0 or a 1, $\epsilon_1 = 0$ indicates no occurrence of a 0 or 1 (i.e., the empty tape), $\epsilon_2 = 1$ indicates the occurrence of the sequence 11 (i.e., a 1 when the immediate preceding input symbol was also a 1), $\epsilon_2 = 0$ indicates no occurrence of 11. The resulting flowgraph is given in Figure 3.6b.

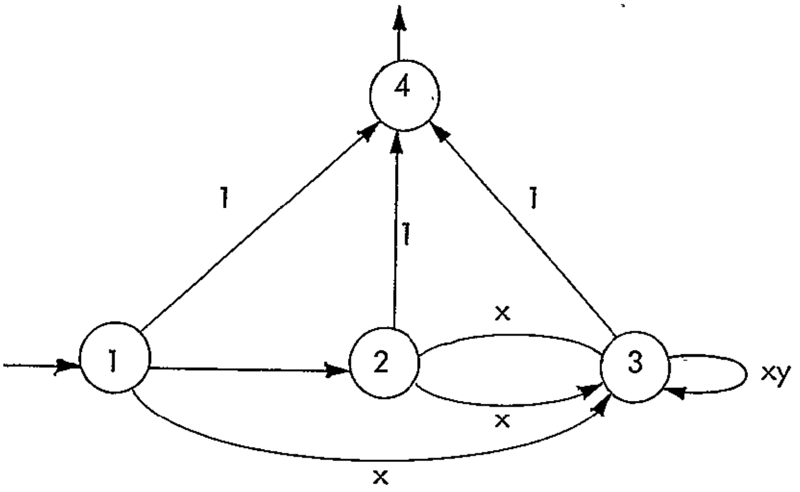


Figure 3.6b. Resulting Flowgraph

the transmission matrix T of the above flowgraph is

$$T = \begin{pmatrix} 0 & x & x & 1 \\ 0 & 0 & x & 1 \\ 0 & x & xy & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$G = [(I-T)^{-1}]_{14}$$

$$= \frac{1-(y-2)x - (y-1)x^2}{1-yx-x^2}$$

This computation was carried out using the FORMAC program which appears in the next section as sample output, along with the first several terms in the expansion of G as a power series in x.

It is instructive to compare the machine-flowgraph method used above with the more traditional method used by Liu (reference 22, page 79) in treating this same problem. The flowgraph approach generalizes effortlessly to bivariate and higher variate problems; traditional generating function methods do not seem to have this property.

Computation of the Generating Function of a Flowgraph

Let F be a flowgraph with starting node s, final node f, and n x n transmission matrix $T = (t_{ij})$. The branch transmissions t_{ij} are in an integral domain (typically a polynomial domain in one or several variables with integer or rational coefficients). Algorithm 3.1 below is proposed for computing the generating function $G = [(I-T)]_{sf}$.

1. Define $A = (I-T^{tr})$, where T^{tr} = transpose of T.
2. Interchange columns f and n of A.
3. Define the n-vector $\underline{b}$ as the s^{th} unit vector.
4. Solve $A\underline{x} = \underline{b}$ for x_n by transforming $A \oplus \underline{b}$ to triangular form according to the 2-step fraction-free algorithm (2.6).

Then $G = x_n = a^{(n-1)}_{n,n+1}/a^{(n-1)}_{nn}$.

The interchange of columns f and n in step 2 means that no back-substitution is required. The above algorithm computes G as $[(I-T^{tr})^{-1}]_{fs}$ instead of as $[(I-T)^{-1}]_{sf}$ in order to take advantage of the fact that step 2 may be omitted, when f=n, as is the case for most flowgraphs.

The proposed method is more efficient than the traditional method of flowgraph reduction by node elimination. The reasons for this are twofold. First, Mason's node elimination scheme is essentially a diagonalization procedure, equivalent to Gauss-Jordan elimination, whereas the proposed algorithm is a triangularization procedure. Moreover a simple interchange of columns (step 2) avoids the necessity for back-substitution. The second and by far the most important advantage of Algorithm 3.1 is that, in the polynomial case, we avoid computations with rational functions and therefore time-consuming GCD calculations.

Mason's celebrated gain formula (see reference 28, page 922) does yeoman service in enabling one skilled in its use to write down by inspection the gain of a flowgraph which has a discernible loop and path structure. However, Mason's gain formula is not suitable for computer implementation because it is generally difficult to detect algorithmically the special loop and path sets required for this formula. This turns out to be a difficult pattern recognition problem.

4. SCOPE FORMAC IMPLEMENTATION OF A SYMBOLIC LINEAR EQUATION SOLVER WITH APPLICATIONS

The IBM 2250 Scope FORMAC System [43] was used to implement a set of routines, called Fraction-Free Package, (FFP) for computing the solution of a given system of linear equations with multivariate polynomial coefficients over the integers or rationals. The routines are:

a. SETDIV and DIVIDE which together constitute an exact division routine for multivariate polynomials;

b. BAREISS, a two-step fraction-free triangularization routine (2.6) for multivariate polynomial domains; and

c. BACKSUB, a fraction-free back-substitution routine for multivariate polynomial domains.

The set of routines FFP is used by two applications programs. The first, program FGRAPH, computes the generating function of a flowgraph according to Algorithm 3.1. (Thus FGRAPH does not invoke BACKSUB). The input.to FGRAPH is an N X N transmission matrix T, a starting node SS, and a final node FF. The elements T(I,J) of T may be arbitrary multivariate polynomials.

The output of FGRAPH is the generating function GG and the first L coefficient G(I) of the power series expansion $GG = \Sigma G(I)\ V^I$, where the variable V and the positive integer L are specified by the user. The expansion of the generating function is carried out using a linear recurring sequence technique (see reference 2, Chapter 13 and reference 36). The coefficients of the expansion of a rational function

$$\frac{a_0 + a_1 z + \ldots + a_m z^m}{b_0 + b_1 z + \ldots + b_n z^n} = \sum_{k=0}^{\infty} g_k z^k$$

over an integral domain (b_0 invertible) satisfy the linear recurrence

$$g_k = -b_0^{-1} \left(\sum_{i=1}^{n} b_i g_{k-i} \right) \quad (k > \max \ \{m, n\} \ )$$

with initial conditions determined by the numerator polynomial.

The second application program which uses the set of routines FFP is STAT which computes the stationary state probability N-vector T of an ergodic (reference 11, Section 7.4) Markov chain specified by an N x N transition matrix P with symbolic elements. Thus T satisfies TP = T with $\Sigma T(I) = 1$.

The IBM 2250 Scope FORMAC listings of FFP, FGRAPH, and STAT are in Appendix B.

Sample Applications of FGRAPH

Consider the labeled directed graph given in Figure 4.1

For FGRAPH, the graph in Figure 4.1 is considered a flowgraph, with starting node s = 1 and final node f = 6 and with transmission matrix $T = (t_{ij})$ given by t_{ij} = label on edge from node i to node j (e.g., $t_{12} = A$ and $t_{13} = B$). The generating function G is consequently obtained as a rational function of two variables A and B.

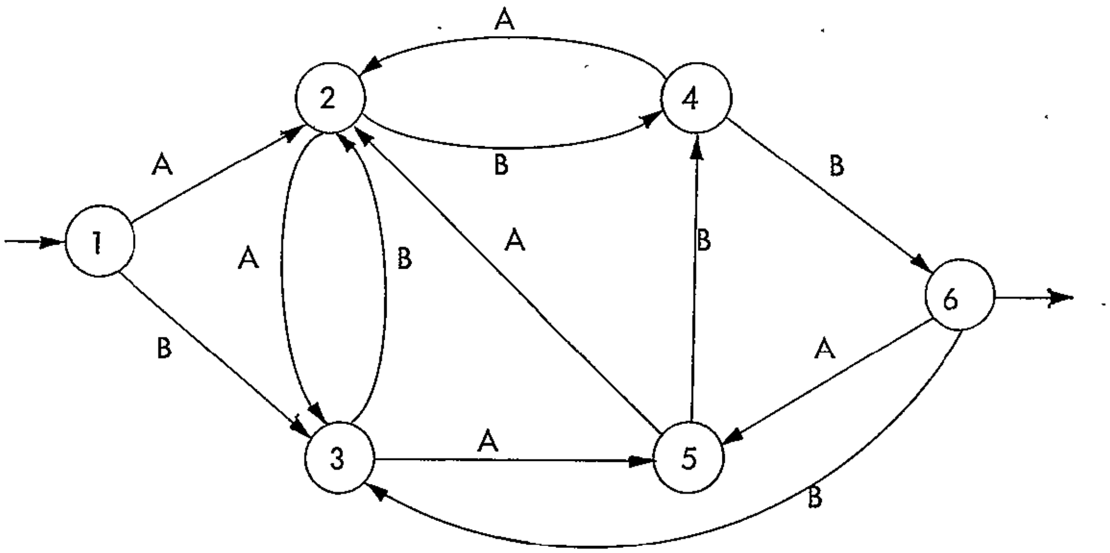


Figure 4.1

For STAT, the above graph is interpreted as a Markov graph, with which the labels A and B = 1-A represent the transition probabilities of p_{ij} of a stochastic matrix P. The stationary probabilities t_i are computed as rational functions of the variable A.

Computational results are now presented, first for FGRAPH then for STAT, in the form in which scope FORMAC displays them.

The generating function GG of the flowgraph of Figure 4.1 is

```
          2      3   2       3     2   3     4                     3
GG = ( A B   +  A   B   + A B   + A   B   + B   ) /  (  - 2 A B - A   B
------------------------------------------------------------------------
       2     2   2       3     4     3
  - A B   - A   B   - A B   - B   - A   + 1 )
---------------------------------------------
```

For path enumeration from node 1 to node 6 the substitutions A = Z and B = Z are made in GG (using the FORMAC EVAL routine 42) to obtain the generating function

```
            3           4           5                          2           3           4
GG = ( Z      + 2 Z       + 2 Z       ) / ( - 2 Z       - 2 Z       - 4 Z       + 1 )
----------------------------------------------------------------------------------------
```

The coefficients G(K) in the power series expansion of GG are

```
                                          G(13) = 944
G(0) = 0                                  -----------
--------                                  G(14) = 1840
G(1) = 0                                  ------------
--------                                  G(15) = 3760
G(2) = 0                                  ------------
--------                                  G(16) = 7392
G(3) = 1                                  ------------
--------                                  G(17) = 14976
G(4) = 2                                  -------------
--------                                  G(18) = 29664
G(5) = 4                                  -------------
--------                                  G(19) = 59776
G(6) = 6                                  -------------
--------                                  G(20) = 118848
G(7) = 16                                 --------------
---------                                 G(21) = 238784
G(8) = 28                                 --------------
---------                                 G(22) = 4759C4
G(9) = 60                                 --------------
---------                                 G(23) = 954368
G(1C) = 112                               --------------
-----------                               G(24) = 1904768
G(11) = 24C                               ---------------
-----------                               G(25) = 3815680
G(12) = 456                               ---------------
-----------
```

Thus, for example, there are 112 paths of length 10 from node 1 to node 6 in Figure 4.1. Note the rapid convergence of G(K+1)/G(K) to its asymptotic value of 2 (R = 1/2 is the smallest root in absolute value of the denominator polynomial of GG).

When Figure 3.1 is interpreted as a Markov graph, in GG A is replaced by 3/4 Z and B is replaced by 1/4 Z, indicating 1-step transition probabilities of 3/4 and 1/4 respectively. This results in the generating function which follows.

```
A=3/4 Z, B=1/4 Z
----------------

-
             3            4             5                  2            3
GG = ( 3/64 Z   + 1/64 Z   + 9/256 Z    ) / ( - 3/8 Z   - 15/32 Z   -
-----------------------------------------------------------------------
       4
5/32 Z    + 1 )
---------------
```

```
  Expanding GG yields

     G(0) = 0                                   G(13) = .03498888
     --------                                   -----------------
     G(1) = 0                                   G(14) = .03527426
     --------                                   -----------------
     G(2) = 0                                   G(15) = .03506509
     --------                                   -----------------
     G(3) = .046875                             G(16) = .03507772
     --------------                             -----------------
     G(4) = .015625                             G(17) = .03515123
     --------------                             -----------------
     G(5) = .05273437                           G(18) = .03510251
     ----------------                           -----------------
     G(6) = .02783203                           G(19) = .03510331
     ----------------                           -----------------
     G(7) = .03442382                           G(20) = .03512147
     ----------------                           -----------------
     G(8) = .03759765                           G(21) = .03511042
     ----------------                           -----------------
     G(9) = .03419494                           G(22) = .03511
     ----------------                           --------------
     G(10) = .03458404                          G(23) = .03511449
     -----------------                          -----------------
     G(11) = .03582572                          G(24) = .03511199
     -----------------                          -----------------
     G(12) = .03487253                          G(25) = .03511175
     -----------------                          -----------------
```

Thus,. for example, the probability of being in state (node) 6 after a 10-step transition is .03458404.

As a final application of FGRAPH, the generating function of flowgraph 3.6b of Example 3.5 is computed and expanded below as a power series in x (with coefficients that are polynomials in y).

$$GG = ((-Y+2)X + (-Y+1)X^2 + 1)/(-XY - X^2 + 1)$$

-

-

FIRST L TERMS OF THE PCWER SERIES EXPANSION OF GG

$$G(0) = 1$$

$$G(1) = 2$$

$$G(2) = Y + 2$$

$$G(3) = 2Y + Y^2 + 2$$

$$G(4) = 3Y + 2Y^2 + Y^3 + 2$$

$$G(5) = 4Y + 4Y^2 + 2Y^3 + Y^4 + 2$$

$$G(6) = 5Y + 6Y^2 + 5Y^3 + 2Y^4 + Y^5 + 2$$

$$G(7) = 6Y + 9Y^2 + 8Y^3 + 6Y^4 + 2Y^5 + Y^6 + 2$$

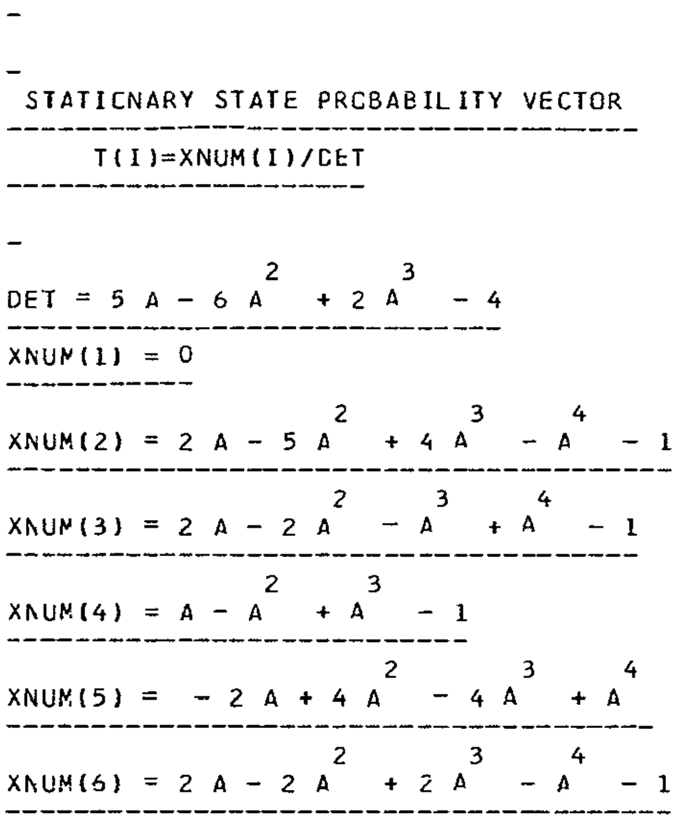
$$G(19) = 18Y + 81Y^2 + 240Y^3 + 540Y^4 + 924Y^5 + 1386Y^6 + 1584Y^7 + 1782Y^8 + 1430Y^9 + 1287Y^{10} + 728Y^{11} + 546Y^{12} + 210Y^{13} + 135Y^{14} + 32Y^{15} + 18Y^{16} + 2Y^{17} + Y^{18} + 2$$

$$G(20) = 19Y + 90Y^2 + 285Y^3 + 660Y^4 + 1254Y^5 + 1848Y^6 + 2508Y^7 + 2574Y^8 + 2717Y^9 + 2002Y^{10} + 1729Y^{11} + 910Y^{12} + 665Y^{13} + 240Y^{14} + 152Y^{15} + 34Y^{16} + 19Y^{17} + 2Y^{18} + Y^{19} + 2$$

Thus, for example, there are 2717 sequences of length 20 having exactly 9 pairs of adjacent 1's.

Sample Applications of STAT

Program STAT produces the output

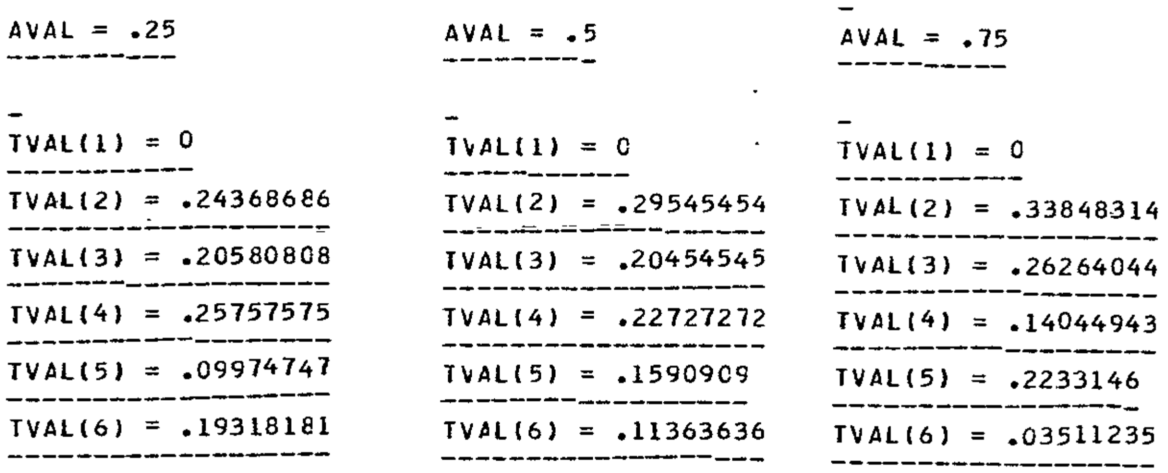


```
-

-
 STATICNARY STATE PRCBABILITY VECTOR
-------------------------------------
      T(I)=XNUM(I)/CET
---------------------

-
                    2        3
DET = 5 A - 6 A     + 2 A     - 4
--------------------------------
XNUM(1) = 0
-----------
                        2       3      4
XNUM(2) = 2 A - 5 A     + 4 A    - A    - 1
-----------------------------------------
                        2     3      4
XNUM(3) = 2 A - 2 A     - A    + A    - 1
-------------------------------------
                  2      3
XNUM(4) = A - A     + A    - 1
----------------------------
                             2       3      4
XNUM(5) =  - 2 A + 4 A     - 4 A    + A
---------------------------------------
                        2       3      4
XNUM(6) = 2 A - 2 A     + 2 A    - A    - 1
-----------------------------------------
```

Evaluating T = (T(I)) for A = .25, .5, and .75 gives the stationary state probability vectors below.



```
                                                  -
AVAL = .25                 AVAL = .5              AVAL = .75
----------                 ---------              ----------

-                          -                      -
TVAL(1) = 0                TVAL(1) = 0            TVAL(1) = 0
-----------                -----------            -----------
TVAL(2) = .24368686        TVAL(2) = .29545454    TVAL(2) = .33848314
-------------------        -------------------    -------------------
TVAL(3) = .20580808        TVAL(3) = .20454545    TVAL(3) = .26264044
-------------------        -------------------    -------------------
TVAL(4) = .25757575        TVAL(4) = .22727272    TVAL(4) = .14044943
-------------------        -------------------    -------------------
TVAL(5) = .09974747        TVAL(5) = .1590909     TVAL(5) = .2233146
-------------------        ------------------     ------------------
TVAL(6) = .19318181        TVAL(6) = .11363636    TVAL(6) = .03511235
-------------------        -------------------    -------------------
```

Thus when $A = .75$ the probability of being in state 6 after a large number of steps is .03511235. The transient probabilities of being in state 6 when $A = .75$ were previously determined using FGRAPH, in which the k-step probabilities are seen to converge quite rapidly toward the above asymptotic value, e.g. $G(25) = .03511175$. Thus FGRAPH and STAT constitute two routines for analyzing Markov chains; FGRAPH yields a transient analysis, whereas STAT yields a stationary analysis.

5. SUMMARY AND CONCLUSIONS

A method is proposed for the computer solution of linear equations with symbolic coefficients, based on a two-step elimination method for computing the exact solution of linear equations with integer coefficients. 1 Section 2 presents a new and elementary derivation of the fraction-free property of the elimination algorithm (2.4) and establishes the algorithm's applicability to solving linear equations over arbitrary integral domains. The primary integral domains that arise in solving linear equations with symbolic coefficients are multivariate polynomial domains with integer (or rational or real) coefficients. For such polynomial domains the fraction-free elimination scheme is valuable because it avoids multivariate rational function manipulations with attendant time-consuming GCD calculations, resulting in considerably increased efficiency over ordinary (fraction-producing) Gaussian elimination. The fraction-free algorithms (2.2), (2.4), and (2.6) are analyzed in the context of polynomial domains, and the superiority of the two-step method (2.6) due to E. Bareiss[1] is established.

A new method for computing the generating functions of flowgraphs was presented (Algorithm 3.1). This algorithm is superior to Mason's node elimination algorithm because it avoids rational function arithmetic and because it triangularizes rather than diagonalizes a coefficient matrix. Furthermore, it does not require use of a back-substitution step.

The usefulness of the definition (3.2) of the generating function of a flowgraph lies in the (formal) identity

$$[(I-T)^{-1}]_{sf} = \sum_{k=0}^{\infty} [T^k]_{sf} .$$

The higher powers of an appropriate matrix T yield important information in various problems, such as the higher transition probabilities in a Markov chain and the number of distinct k-step paths in a multi-graph.

The use of flowgraphs in conjunction with both finite-and infinite-state machines in solving problems arising in combinatorial mathematics is presented; the examples of section 3 indicate the power of the flowgraph concept.

The IBM 2250 Scope FORMAC system was used to implement a package of routines, called FFP, for solving linear equations with coefficients that are polynomials in one or several variables with integer or rational coefficients. Two applications programs, FGRAPH for computing the generating function of a flowgraph and STAT for computing the stationary probabilities of a Markov chain, invoked this linear equation solver. Experience with the Scope FORMAC System has indicated the usefulness and convenience of an interactive system, both at the program development state and program running stage.

Linear equations with nonnumeric coefficients also arise naturally in numerical contexts, and the linear equation solver FFP should be useful in such situations. Suppose, for example, that the coefficient matrix and right-hand side of the system of linear equations $A\underline{x} = \underline{b}$ involve two parameters r and s, i.e., $A = A(r,s)$ and $\underline{b} = \underline{b}(r,s)$. Either of two courses could be followed in computing the solution $\underline{x} = \underline{x}(r,s)$ for r and s with each taking on say ten numerical values. First, the system could be solved numerically, using standard numerical methods in conjunction with a language such as FORTRAN, ALGOL, or PL/I one hundred times, one time for each pair of (r,s) values. Alternatively, one could solve the system $A\underline{x} = b$ symbolically, obtaining the solution vector $\underline{x} = \underline{x}(r,s)$ with the parameters r and s explicitly displayed. Clearly there are situations in which the latter course of action may be preferable from the viewpoint of efficiency alone.

Moreover, other considerations that make symbolic solution highly preferable may exist. An example is the problem of computing

$$\max_{(r,s)} \quad f(x)$$

subject to

$$A\underline{x} = \underline{b}$$

where f is some specified scalar-valued function. Solving $A\underline{x} = \underline{b}$ symbolically, one obtains the solution vector $\underline{x} = \underline{x}(r,s)$ with each component of $\underline{x}$ in the form of an analytic expression involving r and s. The maximization problem then reduces to the form

$$\max_{(r,s)} \quad f(r,s)$$

which can be solved analytically using the calculus $\left(\frac{\partial f}{\partial r} = \frac{\partial f}{\partial s} = 0 \right)$.

A numerical approach to this same problem would be most cumbersome. Even if one had an iterative scheme for moving from one (r, s) value to an improved value, the multiple numerical solutions of a system of linear equations could be prohibitively time-consuming.

Appendix A

PROOFS

Proofs are given for Theorem 2.1, Equation (2.5) of Theorem 2.2, Theorem 2.3 and Theorem 3.1. These proofs are established mainly with the use of the following basic property of determinants: replacing a row R_i of a determinant $|(a_{ij})|$ by a linear combination $\alpha R_i + \beta R_j$ of rows i and j $(i \mp j)$ results in a determinant with value $\alpha \times |(a_{ij})|$.

THEOREM 2.1

The minors of order two of $A_L^{(k)}$ $(k = 1, 2, \ldots, n-2)$ are divisible by

$$\prod_{\ell=1}^{k} \left[a_{\ell\ell}^{(\ell-1)} \right]^{k-\ell+1} .$$

Proof. The notation of Gantmacher (reference 14, page 2) for specifying minors of order p from a given $n \times m$ matrix $A = (a_{ij})$ which follows is used.

$$A \begin{pmatrix} i_1 \, i_2 \ldots i_p \\ j_1 \, j_2 \ldots j_p \end{pmatrix} \qquad \text{denotes the determinant} \quad |(a_{i_k j_\ell})|$$

$$(1 \le i_1 < i_2 < \ldots < i_p \le n; \qquad 1 \le j_1 < j_2 < \ldots \, j_p \le m).$$

Consider any minor of order two of $A_L^{(k)}$

$$A_L^{(k)} \begin{pmatrix} i_1 \, i_2 \\ j_1 \, j_2 \end{pmatrix} \qquad (k+1 \le i_1 < i_2 \le n; \quad k+1 \le j_1 < j_2 \le n+m)$$

where the $a_{ij}^{(k)}$ of $A_L^{(k)}$ are computed according to (2.2). From (2.2) it is seen that the transformation $A^{(\ell-1)} \to A^{(\ell)}$ $(\ell = 1, 2, \ldots, k)$ results in each of the last $k - \ell + 2$ rows of $A^{(\ell-1)} \begin{pmatrix} 1 \; 2 \ldots k \; i_1 \, i_2 \\ 1 \; 2 \ldots k \; j_1 j_2 \end{pmatrix}$ being multiplied by the factor $a_{\ell\ell}^{(\ell-1)}$. By the fundamental property of determinants it follows that

$$A^{(\ell)} \begin{pmatrix} 1\ 2\ \ldots\ k\ i_1\ i_2 \\ 1\ 2\ \ldots\ k\ j_1\ j_2 \end{pmatrix} = A^{(\ell-1)} \begin{pmatrix} 1\ 2\ \ldots\ k\ i_1\ i_2 \\ 1\ 2\ \ldots\ k\ j_1\ j_2 \end{pmatrix} \times \left[a_{\ell\ell}^{(\ell-1)} \right]^{k-\ell+2}$$

and consequently that

$$A^{(k)} \begin{pmatrix} 1\ 2\ \ldots\ k\ i_1\ i_2 \\ 1\ 2\ \ldots\ k\ j_1\ j_2 \end{pmatrix} = A^{(0)} \begin{pmatrix} 1\ 2\ \ldots\ k\ i_1\ i_2 \\ 1\ 2\ \ldots\ k\ j_1\ j_2 \end{pmatrix} \times \prod_{\ell=1}^{k} \left[a_{\ell\ell}^{(\ell-1)} \right]^{k-\ell+2} .$$

But columns 1, 2, ..., k of $A^{(k)}$ have zeros below the main diagonal, so that

$$A^{(k)} \begin{pmatrix} 1\ 2\ \ldots\ k\ i_1\ i_2 \\ 1\ 2\ \ldots\ k\ j_1\ j_2 \end{pmatrix} = A_L^{(k)} \begin{pmatrix} i_1\ i_2 \\ j_1\ j_2 \end{pmatrix} \times \prod_{\ell=1}^{k} a_{\ell\ell}^{(\ell-1)} .$$

Combining the last two equations gives

$$A_L^{(k)} \begin{pmatrix} i_1\ i_2 \\ j_1\ j_2 \end{pmatrix} = A^{(0)} \begin{pmatrix} 1\ 2\ \ldots\ k\ i_1\ i_2 \\ 1\ 2\ \ldots\ k\ j_1\ j_2 \end{pmatrix} \times \prod_{\ell=1}^{k} \left[a_{\ell\ell}^{(\ell-1)} \right]^{k-\ell+1}$$

which completes the proof of Theorem 2.1.

PROOF OF EQUATION (2.5)

$$a_{ij}^{(k)} = \frac{b_{ij}^{(k)}}{\prod_{\ell=1}^{k-1} \left[b_{\ell\ell}^{(\ell-1)} \right]^{k-\ell}}$$

The proof is by induction on k starting with the basis $k = 2$. First note that $a_{ij}^{(0)} \equiv b_{ij}^{(0)}$ and $a_{ij}^{(1)} = b_{ij}^{(1)} \ / \ 1 = b_{ij}^{(1)}$. From (2.4) follows

$$a_{ij}^{(2)} = \frac{a_{kk}^{(1)}a_{ij}^{(1)} - a_{kj}^{(1)}a_{ik}^{(1)}}{a_{11}^{(0)}}$$

$$= \frac{b_{kk}^{(1)}b_{ij}^{(1)} - b_{kj}^{(1)}b_{ik}^{(1)}}{b_{11}^{(0)}}$$

$$= \frac{b_{ij}^{(2)}}{b_{11}^{(0)}}$$

which establishes the basis. Assume now that the $a_{ij}^{(\ell)}$ $(i = \ell+1,\ldots,n;$ $j = \ell+1,\ldots,n+1)$ are given by (2.5) for all $\ell = 2,3,\ldots,k-1$. Then

$$a_{ij}^{(k)} = \frac{a_{kk}^{(k-1)}a_{ij}^{(k-1)} - a_{kj}^{(k-1)}a_{ik}^{(k-1)}}{a_{k-1,k-1}^{(k-2)}} \qquad \text{by (2.4)}$$

$$= \frac{\dfrac{b_{ij}^{(k)}}{\left\{\prod_{\ell=1}^{k-2} b_{\ell\ell}^{(\ell-1)\,k-\ell-1}\right\}^{2}}}{\dfrac{b_{k-1,k-1}^{(k-2)}}{\prod_{\ell=1}^{k-3}\left[b_{\ell\ell}^{(\ell-1)}\right]^{k-\ell-2}}} \qquad \text{by the induction hypothesis}$$

which yields the right-hand side of (2.5) by routine cancellation, completing the proof.

The corollary to Theorem 2.2 is readily obtained by replacing $a_{ij}^{(k)}$ and $b_{ij}^{(k)}$ in (2.5) by arbitrary minors of order two of $A_L^{(k-1)}$ and $B_L^{(k-1)}$ (the proof goes through exactly as before). Applying Theorem 2.1 then gives the desired result.

THEOREM 2.3

$$|A| = a_{nn}^{(n-1)}$$

Proof. Observe from (2.4) that for $k = 1,2,\ldots,n-1$, the $n-k-1$ rows $k+1$ through n of $A^{(k-1)}$ are multiplied by the factor $a_{kk}^{(k-1)} / a_{k-1,k-1}^{(k-2)}$.

Ignoring columns $n+1,\ldots,n+m$ of the $A^{(k)}$ due to the inhomogeneous terms, it follows by the fundamental property of determinants stated at the beginning of the appendix that

$$|A| = \prod_{k=1}^{n-1} \left[\frac{a_{k-1,k-1}^{(k-2)}}{a_{kk}^{(k-1)}}\right]^{n-k-1} \times |A^{(n-1)}|$$

$$= \frac{\prod_{k=1}^{n-2} \left[a_{kk}^{(k-1)}\right]^{n-k-2}}{\prod_{k=1}^{n-1} \left[a_{kk}^{(k-1)}\right]^{n-k-1}} \times \prod_{k=1}^{n} a_{kk}^{(k-1)} \qquad (\text{Recall: } a_{00}^{(-1)} \equiv 1)$$

$$= \frac{1}{\prod_{k=1}^{n-1} a_{kk}^{(n-1)}} \times \prod_{k=1}^{n} a_{kk}^{(k-1)}$$

$$= a_{nn}^{(n-1)} \quad . \qquad \text{Q.E.D.}$$

THEOREM 3.1

$$G = [(I-T)^{-1}]_{sf} \quad .$$

Proof. Writing out (3.5) in matrix form gives

$$\begin{pmatrix} x_1 \\ x_2 \\ \\ \cdot \\ \\ \cdot\cdot \\ \\ \cdot \\ \\ x_n \\ x_{n+1} \end{pmatrix}^{tr} \begin{pmatrix} 1-t_{11} & \cdots & t_{1s} & \cdots & t_{1n} & 0 \\ t_{21} & \cdots & t_{2s} & \cdots & t_{2n} & 0 \\ \cdot & & \cdot & & \cdot & \cdot \\ \cdot\cdot & & \cdot & & \cdot & \cdot \\ \cdot & & \cdot & & \cdot & \cdot \\ t_{f1} & \cdots & t_{fs} & \cdots & t_{fn} & -1 \\ \cdot & & \cdot & & \cdot & \cdot \\ \cdot & & \cdot & & \cdot & \cdot \\ \cdot & & \cdot & & \cdot & \cdot \\ t_{n1} & \cdots & t_{ns} & \cdots & 1-t_{nn} & 0 \\ 0 & \cdots & 0 & \cdots & 0 & 1 \end{pmatrix} = x_0\underline{e}_s^{tr}$$

where $\underline{e}_s$ is the s^{th} unit (n+1) vector. Denoting the above coefficient matrix by A and applying Cramer's rule gives

$$x_{n+1} = \frac{|\widetilde{A}|}{|A|}$$

where $\widetilde{A}$ is obtained by replacing row n+1 of $|A|$ by $x_0\underline{e}_s^{tr}$. Expanding $|A|$ along row n+1 gives

$$\begin{aligned} |A| &= (-1)^{2n}|I-T| \\ &= |I-T| \quad . \end{aligned}$$

Expanding $|\widetilde{A}|$ along row n+1 gives

(*) $$|\widetilde{A}| = (-1)^{n+s+1}x_0 \ M$$

where M is the minor obtained by deleting row n+1 and column s from $|A|$. Expanding M along its last column gives

$$(**) \qquad M = (-1)^{n+f+1} M_{fs}$$

where M_{fs} is the minor of $|I-T|$ obtained by deleting row f and column s. From (*) and (**) follows

$$|\tilde{A}| = (-1)^{n+f} x_0 \ M_{fs}$$

and consequently

$$x_{n+1} = \frac{(-1)^{n+f} M_{fs}}{|I - T|} \quad x_0$$

$$= [(I-T)^{-1}]_{sf} \, x_0$$

completing the proof of Theorem 3.1.

Appendix B

LISTINGS OF FFP, FGRAPH, AND STAT

LISTING OF FFP

```
"  THE FOLLOWING THREE ROUTINES CONSTITUTE A FRACTION-FREE PACKAGE    "
"FOR THE SOLUTION OF LINEAR EQUATIONS WITH SYMBOLIC COEFFICIENTS.     "
"THEY ARE:                                                            "
"   (1)  SETDIV AND DIVIDE: TOGETHER THEY CONSTITUTE AN EXACT         "
"        DIVISION ROUTINE FOR MULTIVARIATE POLYNOMIALS.               "
"   (2)  BAREISS: A TWO-STEP FRACTION-FREE TRIANGULARIZATION ROUTINE  "
"        OF E. BAREISS (SEE MATH. COMP., P.565, JULY 1968), EXTENDED  "
"        TO MULTIVARIATE POLYNOMIAL DOMAINS.                          "
"   (3)  BACKSUB: A FRACTION-FREE BACK SUBSTITUTION ROUTINE.          "


SETDIV:  IF AA=0
            DO PUT "** ZERO DIVISOR **" SET STOP END
         NVAR=0  EE=AA
 Q1:     IF LOP(EE)=24
            DO TA=EE
               FOR ID=1(1)4
               DO TA=ARG(1,TA)
                  IF LOP(TA)=46 TO Q2
               END
               PUT "** BAD DATA IN SETDIV **"  SET STOP
 Q2:           NVAR=NVAR+1
               V(NVAR)=TA
               HPV(NVAR)=HIGHPOW(EE,TA)
               EE=COEFF(EE,TA**HPV(NVAR))
               TO Q1
            END
         CF=EE
         END "SETDIV"


DIVIDE:  QQ=0  RR=BB  SC=1
 Q3:     PROD=1  CC=RR
         IF NVAR=0 TO BBG
         FOR II=1(1)NVAR
         DO HP=HIGHPOW(CC,V(II))
            PROD=PROD*V(II)**(HP-HPV(II))
            CC=COEFF(CC,V(II)**HP)
         END
 BBG:             DD=CF/CF
                  IF DD=1 DO FF=CC/CF*2/2  TO BG  END
                  DD=-CF  FF=-CC/CF*2/2
 BG:     TT=FF*PROD
         IF DENOM(TT)=1 TO DV
            DO PUT "AA DOES NOT DIVIDE BB"  SET STOP END
 DV:     QQ=QQ+TT
         RR=RR-TT*AA
         IF RR=0 TO ED
            SC=SC+1  IF SC<15 TO Q3
            PUT "SAFETY CHECK EXCEEDED IN DIVIDE"  SET STOP
 ED:     END "DIVIDE"
```

```
BAREISS: A(0,0)=1
         K=2
 AG:     IF K>N TO FB
         PUT "******",K,"******"
         AA=A(K-2,K-2)
         DO SETDIV
 "PIVOT ALGORITHM"
         IF A(K-1,K-1)=0
           DO FOR I=K(1)N
              DO IF A(I,K-1)=0 TO L1
                 FOR J=K-1(1)(N+1)
                 DO DD=A(I,J) A(I,J)=A(K-1,J) A(K-1,J)=DD END
                 TO L2
 L1:          END
              TO SING
 L2:       END
         FOR I=K(1)N
         DO BB=A(K-1,K-1)*A(I,K)-A(K-1,K)*A(I,K-1)
            IF BB=0 TO LL  TO L3
 LL:     END
         TO SING
 L3:     DO DIVIDE  CO=QQ
         IF K=N TO EV
         IF I¬=K FOR J=K-1(1)(N+1)
                 DO DD=A(I,J) A(I,J)=A(K,J) A(K,J)=DD END
 "PIVOT ALGORITHM COMPLETED; CO COMPUTED"
         FOR I=K+1(1)N
         DO BB=A(K-1,K)*A(I,K-1)-A(K-1,K-1)*A(I,K)
            DO DIVIDE  CI1=QQ
            BB=A(K,K-1)*A(I,K)-A(K,K)*A(I,K-1)
            DO DIVIDE  CI2=QQ
            FOR J=K+1(1)(N+1)
            DO BB=A(I,J)*CO + A(K,J)*CI1 +  A(K-1,J)*CI2
               DO DIVIDE  A(I,J)=QQ
            END
         END
 EV:     A(K,K)=CO
         FOR J=K+1(1)(N+1)
           DO BB=A(K-1,K-1)*A(K,J)-A(K,K-1)*A(K-1,J)
              DO DIVIDE  A(K,J)=QQ
           END
         K=K+2  TO AG
 FB:     IF A(N,N)=0 TO SING  TO EB
 SING:   PUT "** COEFFICIENT MATRIX IS SINGULAR **"  SET STOP
 EB:     PUT "TRIANGULARIZATION COMPLETED"," "," "
         END "BAREISS"
```

```
BACKSUB: DET=A(N,N)
         XNUM(N)=A(N,N+1)
         FOR IJ=1(1)(N-1)
         DO I=N-IJ
            BB=DET*A(I,N+1)
            FOR J=I+1(1)N  BB=BB-A(I,J)*XNUM(J)
            AA=A(I,I)
            DO SETDIV  CO DIVIDE
            XNUM(I)=QQ
         END "IJ LOOP"
         END "BACKSUB"

 "END OF FRACTION-FREE PACKAGE"
```

LISTING OF FGRAPH

```
FGRAPH:
     SORT=CHAIN(I,H,G,F,E,D,C,B,A,S,R,Q,P,O,N,M,L,K,J,Z,Y,X,W,V,U,T)
PUT " FGRAPH (FLOWGRAPH) IS A FORMAC SCOPE PROGRAM FOR COMPUTING        "
PUT "THE GENERATING FUNCTION GG=GG(SS,FF) OF A FLOWGRAPH SPECIFIED BY "
PUT "AN NXN TRANSMISSION MATRIX  T=(T(I,J)) , A STARTING NODE  SS ,     "
PUT "AND A FINAL NODE  FF . EACH  T(I,J)  IS AN ARBITRARY               "
PUT "(MULTIVARIATE) POLYNCMIAL.                                         "
PUT " "

INIT:     SET NE
          V=Z  L=C
          FOR I=1(1)7,J=1(1)7 T(I,J)=0
          GET DATA
          PUT "STARTING NODE SS",SS,"FINAL NODE FF",FF," "
          FOR I=1(1)N
          DO FOR J=1(1)N PUT T(I,J)
             PUT " "
          END

PUT " "," "
PUT " SETUP FOR TRIANGULARIZATION ROUTINE: DEFINE  A=(I-T)-TRANSPOSE, "
PUT "COLUMN N+1 OF A =DELTA(I,SS) . THUS THE NX(N+1) MATRIX  A  IS THE"
PUT "AUGMENTED MATRIX OF THE SYSTEM OF LINEAR EQUATIONS  AX=B , WHERE "
PUT "B(I)=DELTA(I,SS). COLUMNS  FF  AND  NN  ARE INTERCHANGED TO AVOID"
PUT "THE NECESSITY FOR BACK SUBSTITUTION.                              "
PUT " "

 SU:      SET E
          FOR I=1(1)N,J=1(1)N A(I,J)=-T(J,I)
          FOR I=1(1)N DO A(I,I)=1+A(I,I)  A(I,N+1)=0 END
          A(SS,N+1)=1
          IF FF¬=N
             FOR I=1(1)N DO DD=A(I,N)  A(I,N)=A(I,FF)  A(I,FF)=DD END

PUT " INVOKE TWC-STEP FRACTION-FREE TRIANGULARIZATION ROUTINE BAREISS "
          DO BAREISS
          CC=A(N,N+1)  DC=A(N,N)
          SET NE  GG=CC/DD  HH=GG

PUT " GENERATING FUNCTION  GG",GG," "
PUT "******************************************************************"
PUT "    MAKE ANY DESIRED REPLACEMENTS FOR VARIABLES IN  GG. THEN      "
PUT "   THE FOLLOWING RCUTINES MAY BE INVOKED:                         "
PUT "   1. GF: OUTPUTS  GG  AS A RATIONAL FUNCTION OF A SPECIFIED      "
PUT "          VARIABLE  V .                                           "
PUT "   2. PS: COMPUTES THE FIRST  L  TERMS OF THE POWER SERIES        "
PUT "          EXPANSION OF  GG  W.R.T.  V FOR SPECIFIED  L .          "
PUT "******************************************************************"
          PUT " "  SET STOP
 GF:      CC=NUM(GG)  DD=DENOM(GG)  SET E
          CC=CC  DD=DD
          MM=HIGHPOW(CC,V)  NN=HIGHPOW(DD,V)
          CC(0)=COEFF(V*CC,V)  DD(0)=COEFF(V*DD,V)
          CCC=CC(0)  DDD=DC(0)
          SET NE
          FOR I=1(1)MM DO CC(I)=COEFF(CC,V**I)  CCC=CCC+CC(I)*V**I END
          FOR I=1(1)NN DO CD(I)=COEFF(DD,V**I)  DDD=DDD+DD(I)*V**I END
          GG=CCC/DDD  PUT GG," "," "
          SET E

          PUT "FIRST  L  TERMS OF THE POWER SERIES EXPANSION OF  GG "
          PUT "*** FOR P.S. EXPANSION SPECIFY L AND DO PS ***"  SET STOP
 PS:      IF DD(0)=0 DO PUT "NO POWER SERIES EXPANSION FOR GG"," "," "
                       SET STOP END
          RDDO=1/DC(0)*2/2
          G(0)=RDDO*CC(0)  PUT G(0)
          FOR I=1(1)L
          DO SM=0
             MINN=NN  IF I<NN MINN=I
             FOR K=1(1)MINN  SM=SM-DD(K)*G(I-K)
             IF I<=MM  SM=SM+CC(I)
             G(I)=RDDO*SM
             PUT G(I)
          END

 THRU:    PUT " ","*** EXECUTION OF FGRAPH COMPLETED ***"
          PUT " "," "," "," "," "," "
          END "FGRAPH"


     "** DATA **"


     D1: N=4  SS=1  FF=4   V=X
T(1,2)=X  T(1,3)=X    T(1,4)=1
T(2,3)=X  T(2,4)=1
T(3,2)=X  T(3,3)=X*Y  T(3,4)=1
     END

     D2: N=4  SS=1  FF=4
T(1,1)=Z  T(1,2)=Z
T(2,2)=Z  T(2,3)=Z
T(3,1)=Z  T(3,2)=Z  T(3,4)=Z
T(4,1)=Z  T(4,3)=Z  T(4,4)=Z
     END
```

LISTING OF STAT

```
STAT:
     SORT=CHAIN(I,H,G,F,E,D,C,B,A,S,R,Q,P,O,N,M,L,K,J,Z,Y,X,W,V,U,T)
PUT " STAT IS  A FORMAC SCOPE PROGRAM FOR COMPUTING THE STATIONARY       "
PUT "PROBABILITY S-VECTOR  T  ASSOCIATED WITH AN SXS MATRIX  P  OF       "
PUT "TRANSITION PROBABILITIES ; I.E.,  T  SATISFIES  TP=T.               "
PUT "THE MATRIX  P  MAY HAVE SYMBOLIC AND/OR NUMERIC ENTRIES.            "
PUT " ",", "

INIT:     SET NE
          FOR I=1(1)7,J=1(1)7 P(I,J)=0
          GET DATA (S AND P)
          FOR I=1(1)S
          DO FOR J=1(1)S PUT P(I,J)
          PUT " "
          END
          PUT " "

SETUP:    SET E
          N=S-1
          FOR I=1(1)N,J=1(1)N  A(I,J)=P(J,I)-P(S,I)
          FOR I=1(1)N DO A(I,I)=A(I,I)-1  A(I,S)=-P(S,I) END

PUT " INVOKE ROUTINES OF FRACTION-FREE GAUSSIAN ELIMINATION PACKAGE      "
PUT "TO COMPUTE THE SOLUTION TO THE RESULTING SYSTEM OF LINEAR           "
PUT "EQUATIONS.                                                          "
PUT " "
          DO BAREISS
          DO BACKSUB

PUT " STATIONARY STATE PRCBABILITY VECTOR "
PUT "     T(I)=XNUM(I)/DET"," "
          SET NE
          SS=DET  PUT DET
          FOR I=1(1) N
          DO SS=SS-XNUM(I)
             PUT XNUM(I)
             T(I)=XNUM(I)/DET
          END
          XNUM(S)=SS  PUT XNUM(S)  T(S)=SS/DET
 THRU:    PUT " ","*** EXECUTION OF STAT COMPLETED ***"
          PUT " "," "," "," "," "," "
      END "STAT"


      "** DATA **"
```

REFERENCES

1. E. H. Bareiss, "Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination," Mathematics of Computation, Vol. 22, No. 103, July 1968, pp. 565-578.

2. G. Birkhoff and T. Bartee, Modern Applied Algebra, McGraw-Hill (to appear).

3. E. Bodewig, Matrix Calculus, North-Holland, 1959.

4. I. Borosh, and A. S. Fraenkel, "Exact Solution of Linear Equations with Rational Coefficients by Congruence Techniques," Mathematics of Computation, Vol. 20, No. 93, January 1966, pp. 107-112.

5. W. S. Brown, "The ALPAK System Nonnumerical Algebra on a Digital Computer -- I: Polynomials in Several Variables and Truncated Power Series with Polynomial Coefficients," Bell System Technical Journal, Vol. 42, No. 3, September 1963, pp. 2081-2120.

6. W. S. Brown, J. P. Hyde, and B. A. Tague, "The ALPAK System for Nonnumerical Algebra on a Digital Computer -- II: Rational Functions of Several Variables and Truncated Power Series with Rational Function Coefficients," Bell System Technical Journal, Vol. 43, No. 1, March 1964, pp. 785-804.

7. G. E. Collins, "PM, A System for Polynomial Manipulation," Communications of the Association for Computing Machinery, Vol. 9, No. 8, August 1966, pp. 578-589.

8. G. E. Collins, "Subresultants and Reduced Polynomial Remainder Sequences," Journal of the Association for Computing Machinery, Vol. 14, No. 1, January 1967, pp. 128-142.

*9. G. E. Collins "Computing Time Analyses for Some Arithmetic and Algebraic Algorithms," these Proceedings.

10. W. Feller, An Introduction to Probability Theory and Its Applications, Vol. I, Third Edition, Wiley, 1968.

11. M. Fisz, Probability Theory and Mathematical Statistics, Third Edition, Wiley, 1963.

12. G. Forsythe and C. B. Moler, Computer Solution of Linear Algebraic Equations, Prentice-Hall, 1967.

13. L. Fox, An Introduction to Numerical Linear Algebra, Clarendon Press, 1964.

14. F. R. Gantmacher, The Theory of Matrices, Vol. I, Chelsea, 1959.

15. I. Gerst, "The Bivariate Generating Function and Two Problems in Discrete Stochastic Processes," SIAM Review, Vol. 4, No. 2, April 1962, pp. 105-114.

*This article appears in these Proceedings.

16. A. Ginzburg, Algebraic Theory of Automata, Academic Press, 1968.

17. F. Harary, ed., Graph Theory and Theoretical Physics, Academic Press, 1967.

18. R. A. Howard, Dynamic Programming and Markov Processes, MIT Press, Cambridge, Massachusetts, 1960.

19. W. H. Huggins, "Signal Flow Graphs and Random Signals," Proceedings IRE, Vol. 45, January 1957, pp. 74-86.

20. J. P. Hyde, "The ALPAK System for Nonnumerical Algebra on a Digital Computer -- III: Systems of Linear Equations and a Class of Side Relations," Bell System Technical Journal, Vol. 43, No. 2, July 1964, pp, 1547-1562.

21. J. D. Lipson, "The Analysis of Finite Automata," Term Paper for Applied Mathematics 297, Harvard University, Cambridge, Massachusetts, January 1967.

22. C. L. Liu, Introduction to Combinatorial Mathematics, McGraw-Hill, 1968.

23. C. S. Lorens, Flowgraphs for the Modeling and Analysis of Linear Systems, McGraw-Hill, 1964.

24. H. A. Luther and L. F. Guseman, Jr., "A Finite Sequentially Compact Process for the Adjoints of Matrices over Arbitrary Integral Domains," Communications of the Association for Computing Machinery, Vol. 5, No. 8, August 1962, pp. 447-448.

25. S. Mac Lane and G. Birkhoff, Algebra, Macmillan, 1967.

26. M. Manove, S. Bloom, and C. Engelman, "Rational Functions in MATHLIB," IFIP Working Conference on Symbol Manipulation Languages, Pisa, September 1966.

27. S. J. Mason, "Feedback Theory; Some Properties of Signal-Flow-Graphs," Proceedings IRE, Vol. 41, September 1954, pp. 1144-1156.

28. S. J. Mason, "Feedback Theory; Further Properties of Signal-Flowgraphs," Proceedings IRE, Vol. 44, July 1956, pp. 920-926.

29. S. J. Mason, and H. J. Zimmerman, Electronic Circuits, Signals, and Systems, Wiley, 1960.

30. R. McNaughton and H. Yamada, "Regular Expressions and State Graphs for Yamada," E. F. Moore, ed., Sequential Machines: Selected Papers, Addison-Wesley, 1964, pp. 157-174.

31. M. L. Minsky, Computation: Finite and Infinite Machines, Prentice-Hall, 1967.

32. J. Moses, "A Quick Fail-Safe Procedure for Determining Whether the GCD of Two Polynomials is 1," MIT Artificial Intelligence Memo 126, Cambridge, Massachusetts, March 1967.

33. M. Newman, "Solving Equations Exactly," Journal of Research of the National Bureau of Standards, Vol. 71B, No. 4, October-December 1967, pp. 171-179.

34. M. O. Rabin and D. Scott, "Finite Automata and their Decision Problems," E. F. Moore, ed., Sequential Machines: Selected Papers, Addison-Wesley, 1964, pp. 63-91.

35. C. V. Ramamoorthy, "Generating Functions of Abstract Graphs with Systems Applications," Ph.D. Thesis in Applied Mathematics, Harvard University, Cambridge, Massachusetts, 1964.

36. C. V. Ramamoorthy and D. W. Tufts, "Generating Functions of Abstract Graphs with Applications," Cruft Laboratory Technical Report No. 439, Harvard University, Cambridge, Massachusetts, March 1964.

37. C. V. Ramamoorthy and D. W. Tufts, "Reinforced Prefixed Comma-Free Codes," IEEE Trans. Information Theory, Vol. IT-13, No. 3, July 1967, pp. 366-371.

38. J. B. Rosser, "A Method for Computing Exact Inverses of Matrices with Integer Coefficients," Journal of Research of the National Bureau of Standards, Vol. 49B, 1952, pp. 349-358.

39. R. W. Sittler, "Systems Analysis of Discrete Markov Processes," IRE Trans. in Circuit Theory, Vol. CT-3, December 1956, pp. 257-266.

40. H. Takahasi and Y, Ishibashi, "A New Method for Exact Calucation by a Digital Computer," Information Processing in Japan, Vol. 1, 1961, pp. 28-42.

41. R. G. Tobey, "Algorithms for Antidifferentiation of Rational Functions," Ph.D. Thesis, Harvard University, Cambridge, Massachusetts, May 1967.

42. R. G. Tobey et al., "PL/I-FORMAC Interpreter, User's Reference Manual," IBM Contributed Program Library, 360D 03.3.004, Hawthorne, New York, October 1967.

*43. R. G. Tobey and J. D. Lipson, "The Scope FORMAC Language," these Proceedings.

44. J. Riordan, An Introduction to Combinatorial Analysis, Wiley, 1958.

*This article appears in these Proceedings.

SUMMARY

SIGNIFICANT PROBLEMS IN SYMBOLIC MATHEMATICS

by

Robert G. Tobey
IBM Boston Programming Center
Cambridge, Massachusetts

Abstract

Problems of major practical significance in extending the scope and power of present-day systems for performing literal mathematics are adumbrated.

The author is currently with the Applied Mathematics Division of Argonne National Laboratory, Argonne, Illinois.

PRECEDING PAGE BLANK NOT FILMED

SIGNIFICANT PROBLEMS IN SYMBOLIC MATHEMATICS

by

Robert G. Tobey

1. INTRODUCTION

It has been 17 years since the first work in symbolic mathematics by computer was begun independently by Kahrimanian[14] (at Temple University) and Nolan[18] (at MIT). Eleven years ago interest in symbolic mathematical computation was sufficient among astronomers to stimulate discussion at a celestial mechanics conference held in March 1958.[4] Three problems were recognized as significant:

a. the generally slow speed of computers,

b. their small storage capacity,

c. the nonexistence of algebraic compilers for literal calculations.

The first two problems are familiar. They raise the fundamental question of the availability of sufficient resources. Naturally, this problem persists today, but on a different scale. The third statement is no longer true, although its natural descendents are live problems today. To confirm this one need only read the Proceedings of the ACM Symposium on Symbolic and Algebraic Manipulation.[25]

The significant problems of ten years ago were nebulous and ill-defined. This paper attempts to be more precise and specific; this task is simplified by the context provided by these Proceedings. (Page numbers cited refer to these Proceedings.) The discussion is limited to problems which are significant in the development of practical tools for solving practical problems in a batch processing environment.

The restriction to a batch processing environment is artifical, since methods for coping with several of the problems may involve human decision-making in an interactive environment. However, the size and complexity of expressions and the clerical detail involved is often sufficient to preclude effective human decision-making in all but the simplest cases. Clearly, the development of algorithms for batch operation will enhance interactive possibilities.

The restriction to practical problems discloses the author's prejudice. The problems cited as significant reflect the author's experience carrying out literal computations and working with FORMAC users over the past five years.

The problems cited are quite varied. Their solutions will require expertise from many disciplines, including applied mathematics, combinatorial analysis, numerical analysis, computational linguistics, abstract algebra, complex function theory, and graph theory, in addition to basic system design and programming skills.

The problems discussed here are divided into six categories: Simplification, Partitioning of the Elementary Transcendental Functions, Design of Polynomial Systems, Encoding and Representation, Development of Mathematical Algorithms, and Analysis and Evaluation. Simplification is, by far, the most ambiguous and most complex of these categories. This is testified to by the fact that five of the ten problems cited here fall within that category. Analysis and Evaluation is the least developed area. This is not surprising for it mirrors computing practice and the weakness of analytic tool development in computer science. The remainder of this paper is divided into seven sections; one for each of the above categories and a summary. For each problem, the context of the problem is defined, the problem stated succinctly, and an attempt made to identify the prerequisite knowledge and skills.

2. SIMPLIFICATION

The term "simplification" encompasses all the data reduction and report generating functions which arise in performing symbolic mathematics by computer. In some contexts, simplification can be thought of as a function which is isolated from other mathematical operations and which is performed after a computation to clean up the results. This is the mode of operation of the automatic simplification capability in FORMAC. On the other hand, certain simplifications can only be effectively performed as an integral part of basic algorithms. Hearn[13] indicates the nature of the problem in his discussion of substitution. A further example is the work of Bomberault and Eisenpress[6,21]. In designing systems for performing large-scale symbolic mathematical computations, it is important that simplification be an integral part of other algorithms, i.e., it is essential to minimize intermediate expression swell continuously. Put another way, frequent (as opposed to continuous) data reduction is not adequate to optimize the utilization of space. Similarly, report generation can either be performed by an output editor after the computation is complete or become an integral part of the computation.

The ambiguity of the term "simplification" introduces further complications. Simplification is not only ambiguous in the large—it can mean simplification (1) to prepare for optimal numerical evaluation, (2) to automatically minimize intermediate expression swell, or (3) to make expressions "intelligible"—but it is also complex in the small. Investigators frequently agree on the generic use of the word "simplified," but cannot agree on the status of a specified expression. Consider the example

$$C \sqrt{\frac{2a}{d\,n(2dF + C)}} = \theta \sqrt{\frac{a}{n(F + \theta/2)}}$$

where $\theta = C/d$. One side of this equation is the simpler form for antenna design engineers while the other side is the simpler form for engineers designing the production process.

In addition undecidability is a problem. Regardless of the sense in which the word "simplification" is used, the tacit assumption is usually made that one can recognize like terms or like factors, i.e., that one can decide whether two subexpressions are equivalent. This is trivially true, for manipulating polynomials, rational functions, or trigometric series, and it is true for a large class of functions represented by the symbolic forms which can be constructed using the exponential and trigonometric functions. However, it is not true for suitably complex function classes. The example due to Richardson[19] which Risch discusses (page 136) indicates that if the absolute value function (or the logarithm of the absolute value function) is admitted as an elementary operation, then we will encounter expressions for which the question of equivalence is undecidable, i.e., we cannot determine whether it is possible to further reduce symbolic data.

The discussion of simplification is divided into five paragraphs: Intuitive Simplification, Application of Identities, Minimization of Intermediate Expression Swell, Preparation for Numerical Evaluation, and Production of Intelligible Expressions. Although these problem areas are not mutually disjoint and results in one may contribute greatly to another, each area deals with significant independent issues.

Intuitive Simplification

Despite the ambiguity inherent in the term "simplification," a large majority of users can agree on a basic subset of simplification operations. That subset is quite close to the operations implemented in the 7090 FORMAC System and described in reference 24. This approach does not attempt to reduce expressions to a canonical form, but is satisfied with a "pseudo-canonical" form. This is analogous, at least philosophically, to obtaining a deep structure from the surface structure of a sentence using transformational linguistic techniques. In this sense the simplication problem is as complex as developing adequate transformations to reduce natural language sentences.

This fact is highlighted by Fenichel's experience[8] when he tried to replicate the FORMAC simplification capability in FAMOUS. He was unsuccessful, mainly because FAMOUS did not provide for interaction between individual transformations. Such interaction is a topic of continuing study in computational linguistics. But the interaction between basic simplification transformations and the implications for the design of simplification algorithms has never been thoroughly or systematically studied.

PROBLEM 1: Systematically study the appropriateness and interactions of transformations in the design of simplification algorithms.

PREREQUISITES: Familiarity with the theory of functions of a real and a complex variable, knowledge and experience in the design of transformational grammars (recognition grammars), and experience in complex algorithm design and implementation.

While studying this problem it is important to consider distinct function subsets (see section 3). For example, one should always assume that addition, multiplication, subtraction, and division are permitted. The complexities introduced by the inclusion of additional functions should be precisely understood. This study is important to understand what costs are incurred in the design of a simplification algorithm if additional functions are included and hence how one can extend the capability of a system at minimal cost.

Application of Identities

It is natural to assume that identities such as

$$\sin^2 x + \cos^2 x = 1$$

will be applied automatically by an automatic simplification routine. But general application of this identity requires recognition of the pattern

$$A_1 \sin^2 A_2 + A_1 \cos^2 A_2 + A_3.$$

Where A_1, A_2 and A_3 are arbitrary expressions. Once the pattern is found, it must be replaced by the form

$$A_1 + A_3 .$$

The final replacement is easy; finding such patterns is difficult. In the worst case it can require a combinatorial search. In simpler cases exploitation of the sort as suggested by Marks[16] (see Page 32) will help. It even helps when the sine and cosines are imbedded in products as in the above example; the two terms affected by the identity can still be made to sort adjacently. However, is it possible to make the sort accommodate a large group of system identities? It is clear that this approach is not sufficient to handle arbitrary identities (side conditions) introduced by system users. This is further illustrated by Hearn's discussion of Figure 2 on Page 15.

> PROBLEM 2: Develop techniques for sorting, pattern recognition, and replacement in symbolic expressions which minimize the growth of combinatorial search factors.

> PREREQUISITES: Familiarity with artifical intelligence techniques, the theory of functions of real and complex variables, and perseverance in immersing oneself in the application of identities to large expressions.

Minimization of Intermediate Expression Swell

In his discussion of substitution Hearn[13] emphasizes the importance of performing substitutions at the right point in the calculation in order to minimize intermediate swell in expressions (recall discussion of (2.11) and (2.12), Page 9). The application (or misapplication) of the distributive law can have similar consequences. In many calculations expansion or factoring can make a crucial difference in the size and complexity of the expressions manipulated. Consider the following examples:

$$A = (x-y)^{1000}$$
$$B = x^{1000} - y^{1000}$$
$$C = 7xy^2 + 3x(z-2y^2) - xz(\frac{y^2}{z} + 3) \quad .$$

It is undesirable to either expand A or factor B. On the other hand, expansion of C is necessary for simplification to occur. Expansion produces enormous intermediate expression swell in A and greatly reduces C. (It is due to the frequency of expressions like C in physics that REDUCE always expands.) The issue is more complicated: when is it desirable to reduce the ratio of two expressions by their greatest common factor (G.C.F.)? It is frequently taken for granted (see Page 237) that the ratio of two expressions should be automatically reduced by their G.C.F. But consider

$$D = \frac{(x-y)^{1000}}{x^{1000}-y^{1000}} \quad .$$

Is it not clear that D should be left alone?

In his fraction-free algorithm for the efficient solution of linear equations with symbolic coefficients, Lipson[15] shows that it is frequently possible to systematically remove common factors without a G.C.F. algorithm thus saving execution time and greatly reducing intermediate expression swell. Prior to Lipson's algorithm, it was tacitly assumed that any algorithm for solving linear equations with symbolic coefficients would require frequent application of G.C.F. extraction.

Hartt[12] proposes two additional strategies for reducing intermediate expression swell: the automatic splitting out of subfunctional parts and FAEF, the function variable associated with evaluated functions. Precise definition of these suggestions and detailed design of data structures and algorithms are yet to be accomplished.

PROBLEM 3: Develop algorithms to minimize intermediate expression swell automatically.

PREREQUISITES: Well-rounded mathematical background with a knowledge of factoring in polynomial rings, experience in complex algorithm design, and familiarity with artificial intelligence techniques.

Preparation for Numerical Evaluation

A major use of symbolic mathematical systems is to obtain symbolic expressions which are to be evaluated numerically. Hartt[12] presents the case for a powerful SINCON (Symbolic Numeric Conversion) capability from a physicist's viewpoint. All major production applications of the 7090 FORMAC System[21] were instances of generating symbolic expressions in preparation for eventual numerical calculation. This has also been a major use of PL/I-FORMAC. The symbolic mathematical computation which occurs most frequently is that of taking derivatives; not only obtaining the Jacobian but also mixed second and third order partials. Brute force derivative calculation followed by either interpretation or code generation is generally inadequate (see discussion of nonlinear maximum likelihood estimation in reference 21). An improvement of at least two orders of magnitude over either of these approaches was made by Bomberault and Eisenpress[6, 21]; they simplified the structure of derivatives while differentiation was performed to obtain a fairly optimal sequence of arithmetic statements for evaluation of the derivatives. However, no one has looked at the problem of optimally structuring these arithmetic statements so as to minimize the propagation of roundoff error in the numerical evaluation.

Three aspects of this problem must be distinguished: (a) an algorithm which automatically splits the function into subfunctional parts while other manipulations are being performed; (b) development of an algorithm to handle the clerical detail required to relate various subexpressions to the complete expressions of which they are a part; and (c) development of algorithms for structuring subexpressions so as to minimize roundoff error propagation. These are significant aspects of the capabilities which Hartt proposes as extensions to Eisenpress' and Bomberault's work.

PROBLEM 4: Develop techniques and algorithms for the optimal implementation of symbolic to numeric data conversion.

PREREQUISITES: General knowledge of classical applied mathematics and numerical analysis, with an emphasis on techniques for analyzing roundoff error propagation; and familiarity with the encoding of complex data structures.

Production of Intelligible Expressions

Frequently, the desired result of a symbolic computation is an intelligible expression. Intelligibility requires that: (a) the expression is not so large as to be incomprehensible. (Frequently a FORMAC program will output expressions requiring three to six pages each of densely printed listing.) (b) the variables and subexpressions within the expression are so ordered and arranged that basic physical relationships (relationships inherent in the problem definition or to be discovered from the problem solution) are readily perceived. (c) Basic symmetries are retained and emphasized.

The expression editing displayed by Hearn[13] illustrates intelligibility. Extension of such techniques as skeletal structure extraction which was implemented by Baker[22] will be of value in addressing this problem.

PROBLEM 5: Define intelligibility in the context of specific problems, and develop techniques and algorithms for extracting intelligible expressions.

PREREQUISITES: Familiarity with the techniques of artifical intelligence and with data structures and encoding, in addition to expertise in the discipline of the specific problem under study.

Since intelligibility is such an amorphous concept, it is imperative to consider substantive problems from specific disciplines. A significant part of the problem is to define precisely what constitutes an intelligible expression within the context of the specific problem at hand.

3. PARTITIONING OF THE ELEMENTARY TRANSCENDENTAL FUNCTIONS

Much of the current work in symbolic mathematics is based on mathematical intuition rather than a rigorous knowledge of the properties of the class of functions being manipulated. This is due mainly to our lack of knowledge concerning basic subclasses of functions and their special properties. For example, which classes of the elementary transcendental functions have canonical forms? Which classes of functions are closed with respect to integration? Which classes of functions are closed with respect to the iterative substitution and integration encountered by Gershwin[10] in applying Picard interation?

Answers to such questions will greatly increase our knowledge of the basic functions which arise in most current symbolic mathematical computations, and give us increased insight into which techniques are applicable to which function classes. For example, when does it make sense to avoid decidability problems in simplification by limiting one's calculation to a subset of the transcendental functions for which there is a canonical form? This approach was suggested by Brown informally at the ACM Symposium on Symbolic and Algebraic Manipulation in March 1966. His mathematical results in this direction were recently published.[1] Caviness[2] pursued this concept further in his Ph.D. thesis. An important question still to be answered, however, is what are the properties of particular function classes which make them useful in solving practical problems? Furthermore, which other functions can be neatly represented (or approximated, if approximation is adequate to the problem under consideration) by functions from this class?

PROBLEM 6: Define and study the properties of "useful" subclasses of the elementary transcendental functions.

PREREQUISITES: Familiarity with the properties of the elementary functions, with techniques and results of decideability theory, and with the types and properties of functions which frequently arise in the solution of practical problems.

It is clear that results and techniques from 19th century mathematics will be quite relevant in this problem area. This is already indicated by the work of Brown and Caviness.

4. DEVELOPMENT OF MATHEMATICAL ALGORITHMS

The development of numerical analysis has been accelerated by the development of computers. This is natural since the computer has made possible the utilization of numerical techniques which were impractical in the past. As computer systems for performing symbolic mathematics develop, practical algorithm development for many frequently applied mathematical operations must accelerate in a similar manner. Among the more important of these operations are: symbolic integration, matrix manipulation (determinant calculation, matrix inversion, calculation of the characteristic polynomial and the eigenvalues), asymptotics, greatest common factor extraction, and factoring. The papers by Collins,[3] Feldman,[7] Halton,[11] Lipson,[15] Moses,[17] and Risch[20] indicate the varied and complex mathematical issues which must be dealt with. Studying integration alone requires extensive mathematics. Feldman traces the difficulties he encountered with finite field arithmetic. All these difficulties apply to integration, since algebraic extension field arithmetic is central to obtaining the transcendental part of the integral of a rational function (see section III.2 of reference 23). Moreover, the integration of rational functions is central to the design of more general integration algorithms. The specific problems in this sequence of dependent algorithms are traced in Feldman,[7] Risch,[20] and Tobey.[23]

> PROBLEM 7: Develop precise, efficient algorithms for applying standard mathematical operations to symbolic quantities.
>
> PREREQUISITES: An in-depth wide-ranging knowledge of theoretical and applied mathematics coupled with the perseverance to explicitly define practical processes for implementing mathematical operations which are often conceptually simple.

In developing mathematical algorithms, it is important to know the theoretical limitations. Knowledge of undecidability results is valuable. It is more important, however, not to let such results cloud your approach to practical problems. Risch is right when he says of Richardson's results, "These artificial examples do not give us any real insight."[20] Moreover, they have yet to arise in the solution of specific practical problems. It is important to identify undecidability results which are significant for practical problems.

5. ENCODING AND REPRESENTATIONS

Hearn[13] and Marks[16] touched on several specific encoding and representation issues including expanded versus nonexpanded forms, handling of kernels (Hearn) or common subexpressions (Marks), and encoding polynomial forms (see also reference 23, chapter V).

Many specific as well as nebulous problems can be posed with respect to the issues indicated above. We choose, however, to limit this discussion to one quite specific and promising issue.

Marks[16] (section 2) poses the question of utilizing successive degrees of freedom in data organization. He observes that a character string representation is compact and more efficient than a tree structure representation for operations which use the expression only as a temple, i.e., for algorithms which treat the expression as "read only" data. Such operations as substitution, differentiation, and numerical evaluation scan the input expression interpretively while constructing a new expression. Because they require insertion and deletion of subexpressions as well as rearrangement, automatic simplification and expansion require a list structure for efficient dynamic storage allocation. The overhead due to copying and compactification required to dynamically allocate variable-size contiguous blocks of storage can be prohibitively large (recall the discussion of ALPAK and PM in chapter V of reference 23).

All systems which have been implemented for symbolic mathematics have chosen one expression encoding and maintained that encoding throughout all calculations. This is true not only with respect to contiguous string versus list structure encoding, but also with respect to expanded or recursive representations for polynomials in polynomial manipulation systems.

PROBLEM 8: Determine the conditions under which it is desirable to have more than one representation and/or encoding for expressions.

PREREQUISITES: Familiarity with list processing, tree processing, and dynamic storage techniques, with considerable experience in algorithm implementation and design at the assembly coding level.

It is clear that such issues as whether to have several representations for one expression co-resident in the system or convert between representations will have to be considered. It will be instructive and intriguing to learn for which algorithms it is desirable to convert from one encoding or representation to another. In this study one cannot avoid using the analytic techniques proposed in problem 10. They will be essential in determining tradeoffs between various encodings and representations under the action of various algorithms.

6. DESIGN OF POLYNOMIAL SYSTEMS

In August 1967 at the meeting of the International Astronomical Union[5] in Prague, Herget was asked, "May we expect to find a general language to treat analytic developments on electronic computers or must we always be prepared to use a special apparatus?". He replied, "I expect that we will need special programs to save time on the computer." General purpose expression manipulation systems waste both time and space when the desired manipulation is strictly polynomial. Moreover, the polynomial systems implemented to date waste time and space when one has a specific series structure to manipulate, such as those commonly encountered in astronomy. It is easy to defend Herget's reply.

Too little is known concerning the tradeoffs among encoding, representations, and algorithm designs for polynomial manipulation systems. The problem is further aggravated by the fact that it is frequently desirable to have a polynomial capability which is consistent with and embedded in a general expression manipulation system. (Feldman's recommendations[7] regarding FORMAC move in this direction.) Recent analysis of ALPAK and PM performed by the author[23] and Marks' suggestions concerning the sort (Page 32) encourage one to believe that polynomial and general expression manipulation systems can be merged successfully.

The basic tradeoffs in polynomial systems are between contiguous arrays and list structures for encoding and between completely expanded and recursive representations for multivariant polynomials. Any thorough study of polynomial systems will pose the questions raised in Problem 8.

PROBLEM 9: Develop the necessary techniques and perform a definitive study of polynomial encodings and representations with an eye to both a stand-alone capability and a capability embedded in a general expression manipulation system.

PREREQUISITES: Familiarity with the algebraic theory of polynomial rings and the theory and use of analytic series expansions, and an in-depth knowledge of implementation techniques previously utilized for polynomial systems.

7. ANALYSIS AND EVALUATION

Many of the problems posed require precise combinatorial and analytic tools for assessing design tradeoffs. Collins[3] and Lipson[15] present time analyses for their algorithms. However, more precise and general techniques are required. Systems for symbolic mathematical computations will themselves be of great value here since they can be used to develop and manipulate quite complicated generating functions. (Recall Lipson's discussion in section 3 of reference 15.)

PROBLEM 10: Define the relevant quantities to be measured and develop practical analytic techniques for counting and measuring data structure and algorithm resource requirements and tradeoffs.

.PREREQUISITES: Knowledge of combinatorial analysis, signal flowgraph theory, graph theory, and asymptotic analysis, in addition to familiarity with assembly-level encoding and algorithm design.

It is clear that this is one of the basic problems in computing science. Any results obtained will be of general value.

8. SUMMARY

Ten basic problems in design and implementation of practical batch processing systems for performing symbolic mathematical computation have been discussed. An attempt was made to specify the prerequisite skills and knowledge required to attack each problem.

REFERENCES

1. W.S. Brown, "Rational Exponential Expressions and a Conjecture Concerning π and e," American Mathematical Monthly, Vol. 76, No. 1, January 1969.

2. B.F. Caviness, "On Canonical Forms and Simplification," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1968.

* 3. G.E. Collins, "Computing Time Analyses for Some Arithmetic and Algebraic Algorithms," these Proceedings.

4. M.S. Davis, "Programming Systems for Analytical Developments on Computers," The Astronomical Journal, Vol. 73, No. 3, April 1968.

5. W.J. Eckert, "The Use of Electronic Computers for Analytic Developments in Celestial Mechanics: A colloguium held by Commission 7 of IAU in Prague, 28-29 August 1967," The Astronomical Journal, Vol. 73, No. 3, April 1968.

6. H. Eisenpress and A. Bomberault, "Efficient Symbolic Differentiation Using PL/I-FORMAC," IBM New York Scientific Center Technical Report No. 320-2956, New York, September 1968.

* 7. H.E. Feldman, "Some Symbolic Computations in Finite Fields", these Proceedings.

8. R.R. Fenichel, "An On-Line System for Algebraic Manipulation," Ph.D. Thesis, MIT, Cambridge, Massachusetts, December 1966.

* 9. S.B. Gershwin, "The Use of Computer-Aided Symbolic Mathematics to Explore the Higher Derivatives of Bellman's Equation," these Proceedings.

*10. S.B. Gershwin, "An Attempt to Solve Differential Equations Symbolically," these Proceedings.

*11. J.H. Halton, "Asymptotics for Formula-Manipulation," these Proceedings.

*12. K. Hartt, "Symbolic-Numeric Eigenvalue Problems in Quantum Mechanics," these Proceedings.

*13. A.C. Hearn, "The Problem of Substitution," these Proceedings.

14. H.G. Kahrimanian, "Analytic Differentiation by a Digital Computer," Master's Thesis, Temple University, Philadelphia, Pennsylvania, May 1953.

*15. J.D. Lipson, "Symbolic Methods for the Computer Solution of Linear Equations with Applications to Flowgraphs," these Proceedings.

*These articles appear in these Proceedings.

*16. P. Marks, "Design and Data Structure: FORMAC Organization in Retrospect," these Proceedings.

17. J. Moses, "Symbolic Integration," Ph.D. Thesis, MIT, Cambridge, Massachusetts, December 1967.

18. J.Nolan, "Analytical Differentiation on a Digital Computer," Master's Thesis, MIT, Cambridge, Massachusetts, May 1953.

19. D. Richardson, "Some Unsolveable Problems Involving Functions of a Real Variable," Ph.D. Thesis, University of Bristol, Bristol, England, 1966.

*20. R.H. Risch, "Symbolic Integration of Elementary Functions," these Proceedings.

21. R.G. Tobey, "Eliminating Monotonous Mathematics with FORMAC," *Communications of the Association for Computing Machinery*, Vol. 9, No. 10, October 1966.

22. R.G. Tobey, "Experience with FORMAC Algorithm Design," *Communications of the Association for Computing Machinery*, Vol. 9, No. 8, August 1966.

23. R.G. Tobey, "Algorithms for Antidifferentiation of Rational Functions," Ph.D. Thesis, Harvard University, Cambridge, Massachusetts, May 1967.

24. R.G. Tobey, R.J. Bobrow, and S.N. Zilles, "Automatic Simplification in FORMAC," AFIPS Conference Proceedings, Vol. 27, part 1, Spartan Books, Washington, D.C., December 1965.

25. "Proceedings of the ACM Symposium on Symbolic and Algebraic Manipulation," *Communications of the Association for Computing Machinery*, Vol. 9, No. 8, August 1966.

*These articles appear in these Proceedings.

Federal
Systems
Division

18100 Frederick Pike, Gaithersburg, Maryland 20760