ALAM—Atlas Lisp Algebraic Manipulator*

By R. A. d'Inverno†

ALAM is a system written in ATLAS LISP for carrying out algebraic manipulation. It has been used extensively in General Relativity, where it appears both to be more efficient and to be able to handle more complicated problems than other systems in this field.
(Received June 1968, revised December 1968)

1. Motivation and background

In the past decade a large number of computing systems for carrying out algebraic manipulation have been constructed. The general trend has been towards increasing the capability of the systems, thus allowing a whole spectrum of algebraic problems to be processed. On the other hand, it is clear that with respect to one computer and one language, the more general the capability the less efficient the system is likely to be. The interest of the author is the application of these systems in General Relativity where a large number of algorithmic algebraic problems exist. The systems encountered have a very definite limit on the complexity of the calculation they can handle in terms of what is economically viable. ALAM was constructed to allow more complicated problems to be processed, by restricting the capability of the system to what happens in normal calculations only, thus increasing its efficiency. ALAM possesses some peculiar features, their final justification being that the system appears to work more efficiently, in performing the same task, than other systems encountered, and that ALAM can proceed with problems where other systems cannot.

For the above reasons attention was turned to the largest, fastest available computer, namely Atlas 1. The most convenient (although not necessarily the most efficient) way of representing algebraic structures is in tree-structures or lists. LISP 1.5 is a language which processes lists and has the advantages of being a very powerful language allowing recursive functions to be defined and possessing an automatic garbage collector. Many algebraic systems have been written in LISP; they seem in practice to be more efficient than systems written in other languages. Thus it was decided that a version of LISP under development for the Atlas 1 would be incorporated as a subset of ALAM. This version of LISP did not possess the function COMPILE which translates LISP functions into machine-code, allowing programs to run very much faster,‡ and use less store. For this reason, ALAM was written in LAP, LISP's two-pass assembler (i.e. essentially in machine-code). Although ALAM can and has been written in LISP proper, such machine-code subroutines are probably more efficient than similar LISP functions compiled. In this paper, the structure and distinctive features of ALAM are introduced, and in Section 5 ALAM is compared with representative systems of the two languages most used in this field (LISP 1.5 and FORMAC). Section 6 contains definitions of some of the terms used in this paper.

2. Basic algebraic structure

The structure is written in prefix or Polish notation and stored in the machine as a set of atoms and lists. Its recursive definition is as follows.

An *algebraic expression* is either the atom 0; or an atomic symbol; or a list of two non-zero positive integers; or a list, the first item of which is an operator and the remaining items algebraic expressions.

More explicitly if A1, A2, etc. represent algebraic expressions then the expressions are constructed from:

(i) Atoms, which may represent constants, variables or arbitrary functions;
e.g. C, Y, Z, B1, B2, . . .

(ii) Plus and times, (+, *) which take an indefinite number of arguments;
e.g. (+ A1 A2 . . . A*N*).

(iii) Minus, (—) which takes one argument;
e.g. (— A1).

(iv) Exponentiation, (**) which takes two arguments;
e.g. (** A1 A2) which represents $A1^{A2}$.

(v) Logarithm and exponentiation to base *e*, (LOG, EXP) which take one argument;
e.g. (EXP A1).

(vi) Trigonometric, hyperbolic, inverse trigonometric and inverse hyperbolic functions which take one argument;
e.g. (SIN A1).

(vii) Partial (or ordinary) derivative of an arbitrary funtion (£, $ or π according to character code). This has a list of arguments, the first being the name of the function, and the rest being a list of derivatives.
e.g. (£ A 1 2 3) might be $\partial^3 A(x, y, z)/\partial x \partial y \partial z$; here 1, 2 and 3 refer to x, y and z respectively.

(viii) Numbers. Except for zero (0) all numbers are either rationals, represented by a list of two non-zero positive integers, or are algebraic numbers constructed in combination with previously mentioned structure.
e.g. (1 2) is $\frac{1}{2}$; (** (2 1) (1 2)) is $\sqrt{2}$.

‡ claims of up to a factor of 100 have been made.

* This research has been sponsored in part by the AEROSPACE RESEARCH LABORATORIES through the EUROPEAN OFFICE OF AEROSPACE RESEARCH, OAR, UNITED STATES AIR FORCE, under Contract AF 61(052)–877
† *King's College, London*

3. Basic algebraic functions

(i) *Differentiation.* The function DIFF differentiates its first argument, an algebraic expression, with respect to its second argument, an integer which has previously been placed on the property list of the variable it labels, with indicator VAR,

e.g. DIFF ((SIN A) 1) = (* (COS A) (£ A 1))

if A is a function of X say, which has 1 as its VAR. Differentiation works faster by letting the operators themselves differentiate their arguments, for example + is a subroutine which differentiates its arguments. This device avoids a long parent function with many conditional branches.

(ii) *Simplification.* By simplifying expressions in many different ways, it was found, at least in this system, that in large problems a high degree of efficiency can be obtained if the process of simplification is broken down into three stages:

(*a*) The function ZERM truncates expressions with zeros in them. Zeros can arise in expressions either via differentiation or because in many problems subexpressions are zero. ZERM very quickly eradicates such zeros, thus usually causing a reduction in storage occupied by the expression, and preventing unnecessary simplification of some terms.

(*b*) The function EXPD expands the resulting expression using associativity and distributivity. This, again quite quickly, reduces expressions to the general form (+ A1 A2 . . . A*N*) where A*I* is either (* B1 B2 . . . B*M*) or (— (* B1 B2 . . . B*M*)).

(*c*) The function EDIT then simplifies this general form by first of all simplifying the A*I*'s in a normal manner (e.g. collecting up powers, multiplying numbers together, etc.). The resulting expressions are canonicalised in a fairly straightforward manner. Canonicalisation is a time-consuming process, and apart from reducing input expressions to a unique form and canonicalising powers in the A*I*'s, this is the only time it is used in ALAM; and even here it consists only of ordering the remaining B*I*'s in each A*J*. The need for canonicalisation arises in testing the equality of expressions like (* A B) and (* B A). Finally the A*I*'s are collected up according to the normal rules. Before collecting up, the A*I*'s and in the above multiplication simplification the B*I*'s, are written in a form to facilitate their collection. Experience has shown that a fully expanded answer is what is required in most relativity problems (see 'common factors' and Section 5).

(iii) *Output.* The function PRT prints out its first algebraic argument, on the output stream denoted by its second argument, in a 'mathematical format'. Although this is time-consuming the end product is worth the extra effort involved.

4. Distinctive features of ALAM

Some of the main differences between ALAM and other existing systems are:

(i) *Input—Output.* Input is a relatively rare procedure; a well-defined Polish notation has been found to be unambiguous, while difficulties in this respect can arise with FORTRAN-type expressions. Output on the other hand is more frequent, and an attempt has been made to output in a 'mathematical notation', with standard indices, trigonometric and exponential forms,

e.g.

$$S_{12}^{0} = 2U_{12}B_0 + (3/28)\,\mathrm{LOG}\,(U) + E^{-(2G+B)}\,\mathrm{COS}\,(A) - \mathrm{COT}\,(A)R^{-2}U_{22}^{(1/3)}$$

This requires no transformation to a readable format for the mathematician without computer training. On the other hand large FORTRAN-type expressions are not easily readable, and translation by hand into a different notation is not only laborious but can introduce errors.

(ii) *Recursion avoided.* When writing functions in LISP it is natural to write recursive functions. In fact in all functions where it was possible recursion has been dropped in favour of cycling (this of course is not always possible, as for example in the definition of COPY). This has proved to increase the overall efficiency considerably; for example the transfer of information to and from the push-down stack is bypassed; this could otherwise quickly build up in a large problem. This feature is related to the fact that in LISP proper PROG generally allows one to define functions more efficiently than COND does.

(iii) *Simplification in three steps.* Again it might be more natural to write a main simplifying function with branches covering all possible simplifications, but functions can be made more efficient if their arguments are of a very particular form, since less tests of the arguments need to be made. The strategy of eradication of zeros, and expansion into a general form allows one to write functions with fewer branches in them.

(iv) *Modification of list structure.* All the main functions act by modifying existing list structure rather than by making modifications while copying existing list structure (consing). This is considerably faster when one realises that the majority of computing time is spent in either modifying or copying list structures. On Atlas, each modification takes only 1 or 2 machine instructions whereas 'consing' takes 15 or 16, and also makes more use of the push-down stack. The disadvantage of this approach, which only shows itself in very large computations, is that each expression must be stored uniquely (without common use of sublists). Both (ii) and (iv) considerably reduce the frequency of garbage collections, a redundant and time-consuming, though necessary, process.

(v) *Restricted capability.* To save run-time some primitive simplifications have been adequate in many problems. For example no simplification of (SIN A1) is attempted because nowhere in ALAM does the system construct such a quantity which could be simplified, as for example (SIN 0) could. Of course the programmer could construct such a quantity as (SIN 0), but if he did he would need to modify ALAM to simplify it fully. Although, for efficiency in most probems, the capability is restricted, it may be extended either by the device mentioned in (vi) or by changing functions or defining new ones. Apart from the substitution device, other

alterations suffer from the disadvantage of requiring the user to have a somwhat detailed knowledge of the system. On the other hand it has been found that extensions are usually straightforward. For example suppose one wished to express a result involving (SIN A) and (COS A) in terms of SIN, COS and powers of SIN only. One could use the substitution device to replace powers of COS by their corresponding SIN expressions. Alternatively one could quite easily write a function which when simplifying ** would check for powers of COS and make the appropriate simplifications.

(vi) *Common factors.* In most algebraic manipulation programs, a large amount of time is spent in trying to obtain common factors. The problem of common factors is not well-defined, and in most applications it has been found that either the answer is not the one desired, or that the added complexity arising from the attempt precludes any answer at all. The simple expedient of substituting an arbitrary function for common factor expressions and resubstituting only when printing out, has so far proved adequate in applications. This same device can be used to abbreviate constantly recurring long expressions in answers, and in substituting one expression for another.

(vii) *Numbers.* Except for zero, all numbers in ALAM are rational, or algebraic numbers defined in terms of rationals. This ensures exact treatment and unambiguous output, unlike some systems which translate all numbers into floating-point form. In problems so far encountered, floating-point numbers have not been required. All arithmetic in ALAM is executed by special functions which take advantage of knowing the specific form of their arguments and which also avoid the need for numeric atoms in intermediary results. This avoids a lot of the redundancy in LISP arithmetic.

(viii) *Use of magnetic tape.* Atlas LISP possesses the function SYSTEM which defines the present system in the computer as a new compiler by outputting it onto magnetic tape. This allows one both to modify ALAM permanently when one wishes and to keep a large number of expressions for future use. ALAM also possesses functions which can output and input individual expressions to and from magnetic tape when required.

(ix) *Overall economy of machine instructions.* Savings which may seem small in a function can achieve significance when the function is entered over and over again during the course of simplification. Attempts have been made to optimise the efficiency of frequently used functions. Three examples are given:

(*a*) If a function of one argument is entered via the push-down stack, then it takes 28 machine instructions to push the stack down and pop it up again. For a larger number of arguments and for use of so-called function variables (variables used in subroutines for intermediary results which must be preserved on the push-down stack) the situation is worse. In some functions, for example numerical functions, it is possible to program so that the push-down stack is not used. In others, the stack need only be used for certain arguments. Thus writing functions in such a way that they use the stack only when it is unavoidable, can save the use of a large number of machine instructions.

(*b*) Savings in machine instructions may be achieved by writing important closed subroutines as open subroutines. This avoids possible unnecessary settings of index registers and the use of the 'call' subroutine. For example the construction of a list of two items could involve the execution of 8 machine instructions when programmed as an open subroutine, as opposed to between 30 and 74 instructions, when programmed as a closed subroutine.

(*c*) The atoms which represent the basic operators +, —, *, **, etc., are made to occupy consecutive locations in store. This enables the basic tests on algebraic expressions to be made very efficiently by means of a jump table.

5. Applications and comparisons

So far ALAM has been applied extensively to problems in General Relativity, in the construction of tensors, the transformation of expressions, and the expansion of polynomials in power series with algebraic coefficients. Increased efficiency can be achieved in the processing of polynomials by treating them as lists. In all applications the common factor device has proved either adequate or unnecessary. An example of ALAM's application is the function GEOM, which takes a list of ten algebraic expressions—the components of the metric tensor g_{ab} (a symmetric 4×4 matrix of expressions), produces the quantities g (determinant of g_{ab}), g^{ab} (inverse matrix), Γ^a_{bc} (40 combinations of g_{ab}'s and their first derivatives), R_{abcd} (20 combinations of g_{ab}'s and their first and second derivatives), R_{ab}, R and T_{ab} (all combinations of g_{ab} and R_{abcd}). Other related tensors or tetrad components can be obtained in a straightforward manner.

The source program in ALAM for the application of GEOM to the so-called B.V.M. metric consists of declaration of the variable labelling, delarations of the dependent variables of the arbitrary functions, and finally GEOM and its argument, thus:

```
DEFLIST (((T 0) (R 1) (E 2) (P 3)) VAR)
DEFLIST (((B (0 1 2)) (G (0 1 2)) (U (0 1 2))
  (V (0 1 2))) DEP)
GEOM ((
(+ (* V (EXP (* (2 1) B)) (** R (— (1 1))))
  (— (* (EXP (* (2 1) G)) (** U (2 1)) (** R (2 1)))))
(EXP B)
(* U (EXP (* (2 1) G)) (** R (2 1)))
0 0 0 0
(— (* (EXP (* (2 1) G)) (** R (2 1))))
0
(— (* (EXP (— (* (2 1) G))) (** (SIN E) (2 1))
  (** R (2 1))))))
```

Comparisons with a system written in FORMAC and one in LISP 1.5 for this metric are:

(i) Clemens/Matzner using FORMAC, (IBM 7094) about 30 minutes.

Example of the output for the expression Γ^1_{14} is:

```
1 1 4
U*((U*R*FMCEXP(G*2)*2+U*R**2*FMCEXP
(G*2)*FMCDIF(G,(R,1))*2+R**2*FMCEXP
(G*2)*FMCDIF(U,(R,1)))*2**(−1)−FMCEXP
(B*2)*FMCDIF(B,(E,1)))*FMCEXP(B*(−2))+
(U*R**2*FMCEXP(G*2)*FMCDIF(U,(R,1))*(−2)
+U**2*R*FMCEXP(G*2)*(−2)+U**2*R**2*
FMCEXP(G*2)*FMCDIF(G,(R,1))*(−2)−V*R**
(−2)*FMCEXP(B*2)+V*R**(−1)*FMCEXP(B*2)*
FMCDIF(B,(R,1))*2+R**(−1)*FMCEXP(B*2)*
FMCDIF(V,(R,1)))*FMCEXP(B*(−2))*2**(−1)$
```

In the attempt to extract common factors a number of cancellations have been missed.

(ii) Fletcher's GRAD-ASSISTANT, (IBM 7090) unable to complete the calculation because of lack of store. By arranging that expressions be eliminated when they have been printed out, where possible, it was possible to obtain a result in about 17 minutes. The same expression is:

```
(GAM+−−1 1 4) = − 0·5EO * V (T, R, E) * R
**(−2) + (0·5EO * (D V 0 1 0) (T, R, E) + (D B 0 1 0)
(T, R, E) * V(T, R, E)) * R ** (−1) − 0·5EO * EXP
(−2 * B (T, R, E)) * EXP (2 * G(T, R, E)) * U (T, R, E)
* (D U 0 1 0) (T, R, E) * R ** 2 − (D B 0 0 1)
(T, R, E) * U(T, R, E))
```

Here R has been considered to be the common factor.

(iii) ALAM, (Atlas 1). Since Atlas is a time-shared machine, execution time depends on the amount of store requested (maximum store is approximately 200 blocks of 512 48-bit words). For 100 blocks of store, execution time was about 4 minutes, a large amount of which was spent in printing in 'mathematical format' and in executing garbage collections, which take a very long time in Atlas LISP. Because of this the efficiency of ALAM becomes more marked in more complicated problems. The same expression is:

$$GAM_{14}^{1} = VB_1R^{-1} + (1/2)V_1R^{-1} - UB_2 - (1/2)UU_1E^{2G-2B}R^2$$

6. Appendix of definitions

Algorithm. An unambiguous procedure for a mechanisable solution of a problem.

Atom. There are two types:

(i) *Atomic symbol.* A string of no more than thirty numerals, letters and other legal characters the first of which must not be a number. In some LISP systems + and − are considered to be numbers and may neither represent atoms in themselves nor commence an atomic symbol.

(ii) *Numeric symbol or number.* These may be floating-point, fixed-point or octal numbers, represented in a fairly standard way.

Canonical form. A rule for writing expressions of a given class in a unique way.

Consing. Consing two expressions together involves placing the pointers to the expressions in the half-words of a word, thus using new memory. Modification involves overwriting existing information with new information.

Garbage collector. Reclaims that part of storage which in some definite sense is no longer in use, thus allowing some programs which would otherwise run out of storage to continue.

List. An ordered set, each of whose elements may in turn be a list, or (equivalently) a vector each of whose elements may itself be a vector. In LISP a list of the elements A1 A2 A3 could be written (A1 A2 A3).

Pointer. A way of referring to the address or location of information in memory.

Polish notation. Polish notation or prefix notation consists of writing an expression with the operator first followed by its arguments, as opposed to infix notation where the operator is written between the arguments. For example:

NORMAL MATHEMATICAL	INFIX	POLISH
$ABC + D$	$A*B*C + D$	$(+ (*A\ B\ C)\ D)$

Property list. The list associated with an atomic symbol comprising properties of the atom, such as its print name. The elements of a property list are indicators, usually followed by their respective property definitions.

Push-down list. The last-in-first-out memory area for saving partial results of recursive functions.

References

SAMMET, J. E. (1966). Survey of the use of computers for doing non-numerical mathematics, IBM Sys. Develop. Div., Tech. Rep. No. TROO. 1428.

Comm. ACM (1966), Vol. 9, No. 8.

CLEMENS, R. W., and MATZNER, R. A. (1967). A system for symbolic computation of the Riemann tensor, University of Maryland, Tech. Rep. No. 635.

THORNE, K. S., and ZIMMERMAN, B. A. (1967). ALBERT—a package of four computer programs for calculating general relativistic curvature tensors and equations of motion, Joint Tech. Rep., Cal. Tech.

FLETCHER, J. G. (1965). GRAD-ASSISTANT—a program for symbolic algebraic manipulation and differentiation, University of California, Lawrence Radiation Laboratory, Livermore, UCRL–14624–T.

MARTIN, W. A. (1967). Symbolic Mathematical Laboratory, Project MAC, M.I.T., MAC–TR–36 (Thesis).

RUSSELL, D. B. (1966). ATLAS LISP, Atlas Computer Laboratory, Chilton, Didcot, Berkshire.

BONDI, H., VAN DER BURG, M. G. J., and METZNER, A. W. K. (1962). Gravitational waves in general relativity, VII. Waves from axi-symmetric isolated systems, *Proc. Roy. Soc. A*, Vol. 269, p. 21.

MCCARTHY, J., ABRAHAMS, P. W., EDWARDS, D. J., HART, T. P., and LEVIN, M. I. (1962). LISP 1.5 Programmer's Manual. Cambridge, Mass.: The M.I.T. Press.