The spectrum of symbolic computing for statistical science*

R.W. Oldford
Department of Statistics & Actuarial Science
University of Waterloo

Fall, 1997

Abstract

In this paper a broad definition of symbolic computing is adopted and explored in the context of its application to statistical science. It is argued that the present statistical research in symbolic computing is excessively narrow and could (and should) be broadened considerably. These points are made by reviewing some of the history of symbolic computing, both generally and in the statistical literature, and by considering a variety of examples of symbolic computation in statistics.

1 Introduction

"Already at the time when the first stored program electronic computers were proposed, it was recognized that computation with symbolic expressions was in principle as feasible as computation with numbers. Nevertheless, symbolic computation did not develop as fast as numerical since it was not at all obvious what symbolic computations were of interest, what machine features would help, or how such computation should be programmed. In the last few years, the subject of symbolic computation has been developed at an increasing pace."

*In September 1997, I gave an invited presentation entitled "The spectrum of symbolic computation in statistics" at a workshop on **Symbolic Computation in Statistics** held at the Centre de Recherche de Matheématiques of the Université de Montreal, organized by Jamie Stafford. This is an unfinished initial draft of a paper based on that talk. It should be regarded as fragmentary working notes for such a paper – Sections 2.1 through 3.1 are most polished and provide a summary of the history which may yet be of value to readers though, now, 25 years after it was written. (RWO Fall 2022)

Written nearly forty years ago (CACM 1960, page 163), these words ring familiar today. In statistical science, symbolic computation seems poised at a threshold of exciting possibility yet, perhaps disturbingly, the questions raised above remain difficult to answer.

In this paper I explore some of the possibilities and consider these and like questions as they apply to statistical science. Some history of symbolic computation in general provides the backdrop for this discussion. The early stage of that history is similar to the present stage of symbolic computation in statistics and there is much to learn from it.

For example, as was done then, we must be careful to distinguish non-numerical mathematical computation from the more general notion of symbolic computation – the former being a subset of the latter. To confound the two at this time would be to focus symbolic computation prematurely on the mathematics of statistics.

Throughout the paper the question is raised as to whether we have the beginning of a significant new area of research in statistical computing. If so, then we might ask what are the important questions and directions of such research? Again the historical record to date suggests some important patterns.

The paper has three principal parts: Section 2 on symbolic computation in general, Section 3 on non-numerical mathematical computation in statistical research, and Section 3 on other areas of symbolic computation in statistics.

2 Symbolic computation.

Here, some history of symbolic computation is visited with an eye towards drawing lessons that can be applied to symbolic computation in statistics. Particular, though by no means exclusive, attention is given to non-numerical mathematical computation. This area has received the most attention in the recent statistical literature as researchers (especially mathematical statisticians) have made increasing use of computer algebra systems in their work. Some, notably Andrews and Stafford (199?), have begun to pursue non-numerical mathematical computation in statistics as a research area in its own right and it is to this kind of research that this first part of the paper is primarily directed.

for it is here that much of the recent statistical literature has focussed its attention. It is the use of computer algebra systems by mathematical researchers in statistics that has given rise to much of the present interest

in symbolic computation found in the mainstream statistical literature.

It is largely the recent use of computer algebra systems by mathematical researchers in statistics that has generated much of the present interest in symbolic computation in the statistical literature and so this historical review deals mainly, though not exclusively, with work related to such systems.

– the experience in computer algebra would seem particularly relevant to non-numerical mathematical computation in statistics

By 1986, computer algebra had matured to the point that a review of its principal themes and accomplishments and also its open problems and future directions was in order. These too are summarized in the first part to raise the issue of possible correspondences with research on non-numerical mathematical computation in statistics. The first part closes with some reminders on the progress on symbolic computation in areas other than non-numerical mathematical computation.

2.1 The broad view.

Continuing the opening quote, the breadth of symbolic computation is apparent:

> "The interest has stemmed from three sources: Automatic programming systems which both require symbolic computation in the compilers and which are necessary if symbolic computations are to be programmed without excessive labour, artificial intelligence or heuristic programming which requires symbolic computation, and finally from attempts to make computers carry out non-numerical mathematical computation such as those involved in obtaining analytic solutions to differential equations and in proving theorems by machine."

The first area requires programs which are able to read other programs and to write new ones – one can think of a compiler as such a program. Excessive labour is avoided if the programmer can write at a comfortable level of abstraction, expressing a problem's solution in more natural terms, and have another program construct the corresponding set of instructions for the computation. The second area involves trying to write programs which mimic reasoning processes carried out by humans and as such has a great need for the computational manipulation of symbols. Early but now familiar applications were programs which played games like chess. The third area is the one most closely matched to traditional research topics in statistics and hence the one now seeing the greatest attention in the statistical literature.

The boundaries between these three areas were essentially non-existent. Mathematical reasoning after all is just a kind of human reasoning, and programming language abstractions are important to compilers and human programmers alike, whatever the application. Developments in one area continually fed developments in another.

2.1.1 Early programming languages.

Consider, for example, programming language design. A landmark development was the capability to process lists of indefinite length and possibly unrestricted elements (e.g. including one list as an element of another). List processing operations are mainly concerned with the construction and maintenance of arbitrarly nested list structure including putting information on, deleting it from, or just finding it somewhere on, an arbitrary component of the list structure. Such capabilities were important to other applications like compiler writing, manipulation of formal algebraic expressions, linguistic data processing, and most work in artificial intelligence. According to Sammet (1969 page 383) all these applications could be "lumped together under the general title of *symbolic manipulation*."

The first language designed for list processing was *IPL*, developed by Newell, Simon, and Shaw (see Newell and Simon, 1956) which, as *IPL II*, was first implemented in 1957 and continually improved until 1959 (*IPL-V*). The motivation came from the problem of proving theorems in the propositional calculus but the designers intended it to be used to address any ill-structured and complex problem that required human intelligence. So, in addition to proving theorems, *IPL* was soon used for other applications like chess-playing programs and, most notably, *GPS* the so-called "General Problem Solver" of Newell and Simon (1961) to which the rule-based expert systems so prevalent in the 1980s can trace their roots.

The first comprehensive programming language for symbol manipulation began development in 1959 under the direction of J. McCarthy at the MIT Artificial Intelligence Group. Called *LISP* (an acronym for list processing), the language was

> "... designed to facilitate experiments ... whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit 'common sense' in carrying out its instructions ... [T]he LISP system ... came to be based on a scheme for representing the partial recursive functions of a class of symbolic expressions."
> J. McCarthy (1960), p. 184.

Writing in 1969, J.E. Sammet described LISP as

> "... unusual, in the sense that it clearly deviates from every other type of programming language that has ever been developed. ... However, there has been a considerable practical usage of LISP by a certain group of devotees, who have done interesting work using it for symbolic integration and other forms of algebraic manipulation, theorem proving, solving geometry analogy problems, and contract bridge bidding, etc. ...
>
> ... The great advantage of LISP is its ability to express in a meaningful way solutions to problems which people cannot handle any other way and to express them in a form which is natural to that class of problems."
>
> J.E. Sammet (1969), pp. 406-407, and 410.

A distinguishing feature of *LISP* is that the internal representation of its programs is the same as that of its data (viz. as lists). This made *LISP* unique amongst high-level languages in that it was possible to write a program which could: create another program and execute it, or operate on another program, or even operate on itself.

Originally designed for "common sense" reasoning *LISP* quickly became an important and elegant language for research on non-numerical mathematical computation. In 1961, the first system for symbolic integration called *SAINT* was written in *LISP* by Slagle (1961, 1963). Slagle's approach in *SAINT* was to use general-problem solving methods similar to those of *GPS*. Among the techniques of integration available to it, a promising method was found, tried, and if the technique failed *SAINT* would backtrack and try another one. *SAINT* introduced "semantic" pattern matching whereby appropriate procedures were invoked when a match occurred. *SAINT* was regarded as a triumph for artificial intelligence.

Perhaps because of the strong influence of the lambda calculus of Church (1941) in its design, *LISP* was also highly regarded as a language ideally suited to the development of theoretical computer science vis-a-vis a general theory of computability (Sammet, 1969, p. 416).

This clear focus on symbol processing allowed *LISP* to become an important language for research in computer algebra, artificial intelligence, and programming languages. Indeed, the boundaries between the three areas are easily blurred by users of *LISP* (e.g. Abelson and Sussman, 1985, Norvig, 1992).

Another important programming language concept is that of string processing, where strings are sequences of characters and have indefinite ex-

tent. Many operations will be similar to list processing but now the focus is squarely on searching for patterns within a string and transforming these into different patterns. The first string processing language was *COMIT* (Yngve 1957) whose fundamental executable unit was a rule which would transform one string into another by specifying the target and replacement patterns for the current string. The patterns could be quite complex as strings were organized in *COMIT* to have as constituents either individual characters or whole words. Programs were sequences of such "rewrite" rules, the concept being introduced for the first time by *COMIT*. Although designed for professional linguists in their research, *COMIT* was also used in such applications as programs for symbolic differentiation, theorem proving, and bridge bidding. Though criticized for having poor list handling capabilities, *COMIT*'s string manipulation features became the model for such capabilities in other languages (e.g. *SNOBOL*, Formula *ALGOL*, the *CONVERT* pattern matching extension to early *LISP*).

Ultimately, the application of statistical science requires numerical computation and no programming language has served longer or has figured more prominently in scientific computing than has *FORTRAN*. Interestingly, the first sentence of the 1954 document describing the initial language specification reads

> "The IBM Mathematical Formula Translating System or briefly, FORTRAN, will comprise a large set of programs to enable the IBM 704 to accept a concise formulation of a problem in terms of mathematical notation and to produce automatically a high-speed 704 program for the solution of the problem."
> November 10, 1954, as quoted by J.E. Sammet (1969), page 143.

Although numerical evaluation is the goal, there is clear recognition that the problem and its solution be specifiable in terms natural to the problem area (here mathematics) and that the symbolic expression be manipulated automatically (i.e. compiled) to produce the necessary machine instructions for evaluation. The anonymous authors of this report felt compelled to justify this approach arguing, for example, that coding and debugging time will be considerably reduced and that the investigation of mathematical models will be more feasible.

Significantly missing from *FORTRAN*'s design were string and list processing facilities. Early attempts were made to provide these through subroutine calls and include *FLPL* (Gelernter et al, 1960), and *DYSTAL* (Sakoda 1965).

Another significant development from this period is that of languages specially developed for discrete-event simulation. Two of today's best known systems are *GPSS* (Gordon 1961) and *SIMSCRIPT* (Dimsdale and Markowitz 1964). Such systems introduced interesting language features as well.

In *GPSS*, block diagrams representing the system to be simulated form the basis for the subsequent program, each line roughly referring to a block in the flow chart. Basic data elements reflected the nature of the problem: *transactions* (representing units of traffic), *equipment* acted on by transactions (including *facilities*, *storages*, and *logic switches*), and *blocks* (describing the logic of the system). The approach focuses on actions, with little user attention given to particular data structures.

In contrast, the *SIMSCRIPT* model was based on describing the system to be simulated in terms of *entities*, *attributes* of these entities, and *sets* of entities. Programs describe the system in terms of these structures and the actions to be taken on them.

A novel approach was taken by *SIMULA* (Dahl and Nygaard 1966). *SIMULA* extended *ALGOL* adding significant elements of list-processing and, more importantly, the concept of a collection of programs called *processes* which (conceptually at least) could operate in parallel. A process would both carry data *and* execute actions – an encapsulation which, together with its inheritance capability, marks *SIMULA* as the first *object-oriented* programming language.

2.1.2 Earliest computer algebra systems

About the same time that *FORTRAN* was being designed, the first programs on non-numerical mathematical computation were presented. Independently, Nolan (1953) and Kahrimanian (1954) developed programs which could perform analytic differentiation. At the time there was no widespread recognition of non-numerical mathematical computation as being of inherent interest – note, for example, that the work of Kahrimanian (1954) was presented at a symposium on "automatic programming".

By 1959, interest in this broad view of symbolic computing had developed sufficiently that the *Association for Computing Machinery* (the *ACM*) held a symposium on the topic, and devoted a special issue of the *Communications of the ACM* to the symposium papers. A great many authors presented papers describing, for the most part, programs which solved particular problems (typically mathematical ones) symbolically. Some of the excitement of the symposium (as well as some eerily familiar hyperbole) can be seen in a section heading like "A Program That Does 9 Chapters of Prin-

cipia in 9 Minutes" (over 350 theorems on an IBM 704) from Wang (1960, page 224).

Solving particular mathematical problems via symbolic computation soon gave way to the development of more general systems that were designed to be helpful in solving any of a wide variety of mathematical problems. To be truly helpful to the user, these programs needed to be able to perform a variety of operations that are not unlike those which a mathematical worker might perform in solving a problem. Indeed, the closer these operations were to those of the mathematician, the more useful they became. If one could add to this an interface which allowed the user to concentrate more on solving the problem than on proper use of the system, then a truly powerful tool for mathematical use would result.

The first reasonably general system was *ALGY* (Bernick, Callendar & Sanford, 1961). *ALGY* employed a notation like *FORTRAN*'s but had no arithmetic defined, no looping control, and no control transfers. Its operators included: the (re)naming of an algebraic expression, the removal of parentheses from an expression which performed the necessary algebraic multiplication and grouping of identical terms, substitution of one or more expressions in another, factoring of polynomials of one variable, and expansion of products of sine and cosine functions into sums of sine and cosines of multiple angles. *ALGY* was "the first system to provide multiple capabilities on a general class of expressions in one system" (Sammet, 1969, page 474).

The next systems pushed the integration of non-numerical mathematical computation with an existing numerical mathematical language permitting both types of computation in a single system. The obvious base language was *FORTRAN* and the pioneer in this area was *FORMAC*, begun in 1962 at IBM in Boston by Sammet and Tobey (1962, 1964).

The *FOR*mula *MA*nipulation *C*ompiler, or *FORMAC*, was a preprocessing extension of *FORTRAN IV* and designed to look as much like it as possible. *FORMAC* was intended to handle only those mathematical problems which "require large amounts of tedious algebraic manipulation by hand" (Sammet, 1969, p. 475) and did not provide, for example, general list- or string-handling capabilities. *FORMAC* added a number of types and operators that were not available in *FORTRAN* including symbols which could be assigned mathematical expressions as their values, rational and mixed mode arithmetic, and a few symbolic combinatorial functions. Many functions provided direct user control on symbolic expressions. These included simplification methods (i.e. expansion of expressions, forcing common denominators, and separating expressions into terms, factors, exponents, etc.), substitution methods (i.e. replacing one partial expression by another within

a larger expression; if many such substitutions are carried out at once, the user can specify whether these are to be done in parallel, or sequentially), and expression analysis methods (i.e. probing an expression for the coefficient of a given power, the highest and lower powers, or the expression's numerator or denominator and searching for sub-expressions), and partial differentiation (to arbitrary order). Moreover, any symbolic expression could be evaluated at numerical values of its variables. By 1967, a PL/1 version (*PL/1-FORMAC*) existed which additionally was capable of some symbolic integration and some support for simplification and display.

According to one of its authors, Sammet (1969, p. 490), *FORMAC*'s most significant contribution to technology was its successful adding of non-numerical computation facilities "*as a language* to an existing language used for numerical scientific problems." *FORMAC* was a practical system that "could be easily learned and used to solve specific analytic problems arising in the course of engineering and mathematical work." It helped "steer people away from numerical analysis and back to analytic solution of problems."

Another extension of *FORTRAN* which saw some use in statistical research was *ALTRAN*, an early version of which was developed by 1964 at Bell Laboratories at Murray Hill. *ALTRAN* contributed little new to symbolic technology as it was written to take advantage of an existing collection of *FORTRAN* subroutines for handling polynomial functions called *ALPAK* (see CACM 1966 and Sammet, 1969). *ALTRAN* added new data types for polynomials, rationals, and rational functions of polyomials and statements involving these were preprocessed into *FORTRAN* calls to *ALPAK*. *ALTRAN* saw little use outside Bell Laboratories.

By 1966 (see e.g. CACM 1966) there were many systems for non-numerical mathematical computation many of which were based on a number of general purpose list-processing languages developed by this time (e.g. *Formula ALGOL*, *LISP*, *L6*, *SLIP*, and *REFCO*). *LISP* languages stood out in this regard, providing the basis for such systems as *CONVERT*, *FLAP*, and *MATHLAB*.

2.1.3 Early interface considerations

Scientific programming languages were begun so that computation might be expressed relatively easily in a language as much like mathematics as possible. As hardware technology developed some research focussed on developing "input/output capabilities", or "interfaces" in today's language, that would further ease the scientific use of the computer.

Early developments included "on-line" systems for numerical mathemat-

ical computation: from small useful languages like *BASIC* (*B*eginner's *A*ll purpose *S*ymbolic *I*nstruction *C*ode), *CPS* (*C*onversational *P*rogramming *S*ystem) and *JOSS*, to an interactive version of *FORTRAN* called *QUIK-TRAN*. Some systems like *MAP* (*M*athematical *A*nalysis *P*rogram) were designed to be used by persons having little or no knowledge of computers. Consequently, they could be overly anthropomorphic using English expressions, like *MAP*'s "I CAN FIT FUNCTIONS OF THE FORM ...", to interact with the user. (It is interesting to note that in Sammet's (1969) discussion of these languages most example programs deal with statistical calculations such as means, standard deviations, and least-squares estimates.)

MAP is of particular interest for, in addition to the usual mathematical fuctionality, it provided commands to integrate, differentiate, and convolve mathematical expressions, albeit numerically. By 1967, planned improvements for *MAP* included adding symbolic differentiation and substitution functions as well as graphical output.

The first complete "on-line" system for non-numerical mathematical computation was *MATHLAB* (Engelman, 1964). Actions that could be taken on mathematical expressions included substitute, factor, expand, integrate, differentiate, and solve. Unfortunately, the typed i/o and English-like output (e.g. "THANKS FOR THE EXPRESSION") rendered *MATHLAB* ineffective. *MATHLAB* also provided the first program to symbolically integrate rational functions (see Engelman 1971). The program made use of pattern-directed heuristic transformations to reduce the integration problem to one for which an algorithm existed. Engelman, in Hayes-Roth et al (1983, p. 39), describes *MATHLAB* as an expert system.

One way to create a more natural mathematical interface is to increase the usable character set to include mathematical symbols. Sammet (1969) describes some of the early numerical mathematical languages which did just this. An example is *AMTRAN* (*A*utomatic *M*athematical *TRAN*slation) which, in addition to a standard keyboard, had a special keyboard and display scope which allowed Greek letters and mathematical symbols like $\sum$, ∞, $\int$, ∂, $\triangle$, and ∇ to be part of its character set. One of *AMTRAN*'s basic goals was "To use the natural language of mathematics as a programming language without any arbitrary restrictions whatsoever" (Reinfelds, et al 1967 p. 469 as quoted by Sammet 1969, p. 258). Contrast this approach to that of *APL* (Iverson, 1962) which extended the character set to include a novel collection of operators permitting programming in a terse mathematical *style* suited to array manipulation. *APL* enjoyed great success in statistics (e.g. Anscombe 1982) until largely supplanted by interactive statistical systems.

Begun in 1961, the *Culler* system (also called *Culler-Fried*) provided scientists and engineers with a system that allowed display and easy manipulation of real and complex valued functions (see Culler and Fried, 1965). Immediate graphical feedback and "push-button" programming were the system's key features. Convenient manipulation was achieved by a software organization based on the central role of functions in classical analysis and by special hardware including an electronic display scope and, for input, two specially designed keyboards. Operands were entered using an alphanumeric keyboard and operators via a separate keyboard of push-buttons, one per operator. In an early hardware version of operator overloading (or generic functions), each keyboard had several levels of operation: I Real functions or vectors, II Matrices, III Display operations, IV Complex Functions or vectors, and V & VI Systems Management and Data Transfer. Pushing the plus button could add vectors if on level I and matrices if on level II; the meaning of each button depended on the level. Note that arguments could be "functions" and although represented numerically (as x and y vectors), the combination of graphical display and of buttons for differentiation and integration (albeit numerically) would contribute to the illusion of working directly with the analytic objects. Whole programs could be written within the system, assigned by the user to a button, and subsequently invoked by pushing that button. The Culler system was extended as *LOLITA* by Blackwell (1967) to include some list processing and symbol manipulation facilities. *LOLITA* could be used to produce such non-numerical mathematics programs as differentiation rules.

Another numerical mathematical system that could immediately display its results was *DIALOG* (Cameron et al, 1967; see also Sammet 1969) which relied on an electronic stylus to build programs from characters displayed on the screen. Reading one character at a time, *DIALOG* would only make available to the user for selection those characters which could legally appear next in the program statement. Functions could be plotted on the display and the location of the stylus on the screen sensed to provide numerical input from the display.

The first computer algebra system to make use of such specialized equipment was the *MAGIC PAPER* system (Clapp and Kain, 1963; Clapp et al, 1966). It used a keyboard which could switch between Latin and Greek letters, a push-button panel of system commands, a display screen for output of equations, a light pen to select command arguments from the screen, and two foot switches that allowed the user to view tables of transformation that could be applied to the selected expression! *MAGIC PAPER* introduced scrolling to interface design; the user would work within a single "scroll" –

entering equations, editing them, performing actions on them, etc. – and could move forward or backward one equation at a time by "advancing" or "rewinding" the scroll. Scrolls could also be saved for further work later. No automatic simplification was employed. Instead the user could apply mathematical operators to expressions, make substitutions, combine terms, combine fractions, simplify products and fractions, move expressions to the other side of an equal sign, and apply a selected simplifying transform from a table (e.g. changing $(a+b)^2$ to $a^2+2ab+b^2$). The user could add new transformations to this table and, as in the Culler system, could define new procedures to be assigned to the push-buttons. Expressions could also be evaluated numerically.

A near contemporary, the *Symbolic Mathematical Laboratory* of Martin (1967) used standard hardware components (i.e. keyboard, light-pen, display scope, calcomp plotter) to build an interactive system that displayed and printed expressions in essentially standard mathematical form (e.g. using $\sum_{i=1}^{\infty}$, $\frac{d^2}{dt^2}X(t)$, etc.). Expressions were manipulated either by using the light pen to select commands from the display scope, or by typing in command expressions directly from the keyboard in a syntax similar to *FORTRAN* (although the system was implemented entirely in *LISP*). In either case, expressions could be selected from the display to be arguments to any command. The system was capable of symbolic differentiation, definite integration, finding limits, and solving equations (as far as possible). The user had much control over rearranging expressions, selecting subexpressions, and simplification. There also appears to have been a command for automatic simplification.

By 1967, then, the seeds of future interfaces had been sown.

2.1.4 Patterns

The common objective here is the development of computational tools which enhance a scientist's ability to solve relevant problems of a mathematical nature. The ideal pursued is well described by the "magic paper" metaphor. The scientist expresses him or herself in a natural way as if scratching out ideas on magic paper. This paper is such that the scientist may probe, simplify, or transform, any scratching on it and the result would be magically expressed in a way natural to the scientist for this problem. All the while, the technology takes care of tedious details and calculations which would otherwise distract the user from gaining insight into the scientific problem.

Human-computer interfaces are to be designed to encourage discourse at a level natural to the scientist. This means using notation, displays,

and interactive i/o so as to effect the illusion of dealing directly with the mathematical concepts. Whether the underlying implementation is strictly numerical (e.g. *Culler*) or non-numerical (e.g. *Symbolic Mathematical Laboratory*) is unimportant to the user, provided the interface is convenient and the results mathematically sound.

While standard mathematics provides the concepts and operations to form the basis of these systems, implementation forces their explicit representation and this leads to new problems and sometimes to new solutions of old problems. Expressions and equations become concrete data structures which can be probed, re-expressed, simplified, expanded, and transformed. These operations and others (e.g. automatic symbolic integration) also require separate explicit representation. Implementation stimulates new computational and mathematical research.

The activity these systems are meant to support is an open-ended one, often tentative and exploratory. The environment must therefore be much the same. Organizing the software explicitly around the relevant mathematical concepts seems to pay off. User manipulation of mathematical concepts are then more easily matched by programmatic manipulation of software constructs. The user should be able to make such modifications, preferably on the fly, and to extend the system as appropriate, minimally by defining new procedures.

These requirements place certain demands on the programming language. Manipulation of symbols, strings, and list structures figure importantly, and, especially in scientific applications, numerical computation and graphics. Most important is having the means to build new data and procedural abstractions as required. This helps in the design of the system, its maintenance, and its extension. A language capable of writing programs which can themselves write programs (new procedures and data structures) is valuable as it encourages the abstraction and automation of process.

2.2 Computer Algebra

By the late 1960s, non-numerical mathematical computation was becoming a separate field of research. By 1986, it had sufficiently matured that a review of its major accomplishments and research milestones seemed in order and was published in the discipline's new *Journal of Symbolic Computation* (Caviness, 1986). As the name of the journal suggests, this research area has been taken to be synonymous with symbolic computation by those in it as well as by most statisticians (and mathematicians). It seems to be worthwhile, then, to briefly consider these achievements.

In reviewing the period 1966-1986, Caviness (1986) identifies the following to be among the most important achievements in computer algebra:

1. Practically fast algorithms for computation of polynomial GCDs and factorization. These are algorithms for multivariable polynomials having integer coefficients. Caviness (1986) is careful to separate the practical from the theoretical. Algorithms described here have some poor theoretical properties – such as worse case computing time that is exponential in the degrees of the polynomial! Nevertheless, they seem to perform well enough for those cases encountered in practice. So well that "without these advances, computer algebra as we know it today simply would not exist" (Caviness, 1986, p. 219). Achieving the "best asymptotic computing-time bound" may not be necessary.

2. Theoretical results in GCD computation and polynomial factorization. Here Caviness describes results on the number of divisions required by Euclid's algorithm when applied to rationals and Gaussian integers. Factoring multivariable polynomials (integer coefficients) or finding the roots in polynomial time were important theoretical problems. The solutions of these problems, and some of their practical drawbacks or restrictions, are described.

3. Algorithmic polynomial ideal theory. Polynomial ideals provide a useful framework for studying many important computational problems including solution of systems of polynomial equations and simplification. An important set of basis polynomials for any ideal, the Gröbner basis, is constructed en route to solving some problems. Such constructive algorithms and theoretical work describing their complexity constitute important progress.

4. Decision procedures for logical theories. The important developments here are theory and algorithms which take an expression involving quantifiers $\exists$ or $\forall$ and find a logically equivalent one having no quantifiers. The resulting expression is now effectively decidable.

5. Integration and summation in finite terms. Watershed work here is that of Risch (1969, 1970). Prior to this, indefinite integration was attacked as a search problem much as it was in *SAINT*. The first partial implementation of the Risch integration algorithm was due to Moses (1967) in a follow-up to *SAINT* called *SIN* which eliminated search but, being unable to back up and try another method, would sometimes fail to solve a problem it could have. With the work of Risch

and others *algorithms* were developed which worked on most functions of interest (e.g. algebraic, logarithmic, or exponential extensions of rational functions) and would either return the indefinite integral or determine that there was no closed-form integral available which could be expressed in terms of elementary functions. According to Caviness (1986, p. 224), the Risch integration algorithm, its' improvements, and implementations "played a major role in the early acceptance of computer algebra systems as useful and interesting tools". Still, implementations with good "human engineering factors" required an interplay between heuristic (i.e. elementary calculus methods) and algorithmic methods.

6. Algorithms for solving differential equations in closed form. These are relatively recent accomplishments with the first breakthrough appearing in Kovacic's (1979) algorithm for solving second-order linear homogeneous differential equations. The n'th order problem was solved two years later (Singer, 1981). As with integration, the key to solution was the development of an appropriate algebraic theory and setting for the problem.

7. Production of computer algebra systems. "[C]omputer algebra would not exist as we know it today without ... the development of computer algebra systems" (Caviness 1986, p. 226). Those having had the most impact at this time were *MACSYMA* (e.g. Martin and Fateman 1971) for the broadest mathematical coverage implemented, *REDUCE* (e.g. Hearn 1971) for its coverage and portability, *ALDES/SAC-2* (Collins and Roos 1986) for its careful and complete documentation of its algorithms, and *muMATH* (e.g. Stoutmeyer 1985) because it would run on a PC. Lastly, Caviness hoped that the recent commercialization of computer algebra systems would result in well documented and maintained systems.

With the exception of the last item the focus is decidedly on algorithms (e.g. see also Davenport et al, 1993, or Buchberger et al 1985). The appropriate mathematical theory provides a setting for finding and assessing algorithms; implementation sorts out which of these are, or can be made, practically useful. This pattern is applied to a few broadly useful, and consequently important, mathematical problem areas.

To most outsiders, however, that which has had greatest impact is the delivery of these algorithms in software systems that can be easily understood and used. Today's familiar computer algebra systems all date from

this time period.

The first implementation of *REDUCE* appeared in 1967 and a version was available to the public by 1970. *REDUCE* is a relatively small and portable system that has enjoyed wide popularity. It is easily extended and its wide user community has contributed many freely available packages. Principally designed by A.C. Hearn, *REDUCE* has undergone many changes and extensions since (e.g. see Hearn, 1987, or Fitch 1985). *IRENA* (Interface between REduce and NAg) is a relatively recent extension which allows the *REDUCE* user to call numerical methods from the *NAG* library (see e.g. Dewar 1989, or Davenport et al 1992). *IRENA* is presently distributed by *NAG*.

In contrast, *MACSYMA* (e.g. see Pavelle and Wang, 1985) is a relatively large system developed from 1969 to 1982 by the Project MAC group (*MACSYMA* stands for Project MAC SYMbolic MAthematics program). A team effort from the beginning, *MACSYMA* developed largely by incorporating new or improved algorithms into it as they developed. The result is a powerful broadly based non-numerical mathematical system. Like *REDUCE*, *MACSYMA* is written entirely in *LISP* although like essentially all other computer algebra systems the user language is more "ALGOL-like". In the late 1970s *MACSYMA* became a commercial product. Though based on relatively early *MACSYMA*, a public domain version called *MAXIMA*, is also available that runs on the public-domain *AKCL* (Austin-Kyoto Common Lisp).

The first computer algebra system written specifically for microcomputers was *muMATH*, written by D. Stoutmeyer and A. Rich in the late 1970s. The same team then produced *DERIVE*, a menu-based system, as the successor to *muMATH*. Both systems are relatively limited in scope but have been praised as easy to learn and as good introductory systems for teaching (e.g. Glynn 1989 targets young teenagers!). *DERIVE* became commercial in 1988.

By 1980, it was felt by some that the time was ripe to redesign a computer algebra system from scratch to take "advantage of the software engineering technology that [had] become available" in the approximately 10 years since the design of *REDUCE* and *MACSYMA* (Geddes et al 1982). Principal design goals of the *Maple* system included compact size, portability, and efficiency (e.g. Geddes, 1984). The low-level language of choice was *C* and *Maple* was soon available on Unix systems, Macintoshes, and PCs. The high-level user language had "Algol68-like" syntax that was felt to be "more suitable for describing algebraic algorithms" (Geddes et al 1982). *Maple* was generally available in 1983 and soon became a major competitior to

REDUCE and *MACSYMA*. *Maple* became a commercial product in 1989 (Char et al, 1986, 1992).

In the late 1970s to early 1980s, *SMP* was developed by S. Wolfram and others (Wolfram et al 1983) but became largely superseded in 1988 by the launch of *Mathematica*. Again Wolfram was the key developer (Wolfram 1988). *Mathematica* began life as a commercial product and was quickly marketed by Wolfram with much fanfare, being bundled as part of the base software suite for Steve Jobs's *NeXT* machine. Marketing aside, *Mathematica* ran on a variety of platforms and, most importantly, gave careful consideration to its user interface, providing exceptional graphical capabilities well beyond that available at the time from other computer algebra systems. Other systems, like *Maple* (e.g. see Char et al 1992), have since devoted considerable effort to providing their own high-quality graphical interface. Like *Maple*, *Mathematica* is implemented in *C*.

Perhaps no other computer algebra system has been praised more highly (including Caviness 1986) or over a longer period of time than has *AXIOM* (Jenks and Sutor 1992), known previously as *SCRATCHPAD* (e.g. see Jenks 1984, Sutor 1985). *AXIOM* is the culmination of research at IBM's T.J. Watson Research Center in Yorktown that began in the mid-1970s. Like other systems *AXIOM* has many mathematical capabilities, including interactive dynamic two and three-dimensional graphics of functions – much attention has been paid to the interface design. Like *REDUCE* and *MACSYMA*, *AXIOM* is coded in a dialect of *LISP* (*COMMON LISP*, see Steele 1990) but has its own user language that includes a compiler.

Where *AXIOM* differs so dramatically and importantly from the others, however, is in its fundamental structural design. This has important consequences for its use. *AXIOM* is designed from the ground up to model the fundamental mathematical objects directly in the software. Data structures in *AXIOM* include **Groups**, **Rings**, **Fields**, **Algebras**, and **Vector Spaces**. These, and many other such classes of mathematical objects, are related in the software in a class inheritance hierarchy that directly matches the basic algebraic structure (e.g. an **Algebra** is necessarily also a **Ring**). The user can write algorithms that operate at this level. For example, one might write an algorithm in *AXIOM* to solve equations of polynomials where the polynomials are defined over an arbitrary **Field**; code correct at this level would then work for any field whether it be "floating-point numbers", say, or "power series". Computational realizations of sophisticated mathematical objects can be constructed in *AXIOM* just as one would construct them conceptually in mathematics. It is precisely this ability which has generated the consistent praise and excitement surrounding *AXIOM*/*SCRATCHPAD*.

A somewhat dated comparison of most of these systems is given in Harper et al (1991). Notably missing from this comparison is *SCRATCHPAD*. Though not formally compared, the authors felt compelled to mention *SCRATCHPAD* in passing because "it is a powerful system which is likely to become popular if it becomes widely available."

2.3 Other developments

Symbolic computation, in the broad sense, has also made remarkable development outside of computer algebra. Research areas like artificial intelligence, programming languages, and interface design have all made important contributions. As in the early history, assigning credit exclusively to one area or another would be difficult and possibly misleading.

The key to symbolic computation is abstraction – the ability to build software abstractions, to give them visual representation, to manipulate them programmatically and interactively. The abstractions are to model concepts, programming or scientific, that are natural to the problem at hand. The goal is to provide the illusion of dealing directly with those concepts.

The first programs were sequences of machine instruction executed serially in an *imperative* fashion. With the first "high-level" languages these instructions became more abstract, more distant from the physical machine characteristics. With the ability to write subroutines or functions within the language, a new *algorithmic* or *procedural* kind of imperative programming became possible. Examples include *FORTRAN*, *C*, and *LISP*. Some researchers interested in developing and maintaining large programs developed a mathematically cleaner (i.e. easier to prove program properties) style of programming called *functional* programming. In this style of programming every procedure, or function, is able to access only those parameters passed to it and return the value of the function applied to these parameters (e.g. no changes by side-effect). An example is early *LISP*.

In some problems, it is easier to describe what needs to be done rather than how to do it. The *declarative* style of programming has the user specify what is to be done and the language is responsible for ordering tasks so as to accomplish it. An example is *rule-based* programming. Here a collection of rules (e.g. if-then rules) are specified along with a goal. Execution is carried out by an "inference engine" that chains through these rules typically backwards from the goal (or forward from the conditions), finding rules which apply (i.e. pattern matching), backing up from those which fail, until ultimately a series of rules can be produced which lead from fulfilled

conditions to the achievement of the goal. These rule-based systems, dating back to the 1960s, were used as the basis for the "symbolic reasoning" of many expert systems (e.g. Shortliffe, 1976, see also Hayes-Roth et al 1983 who include *MACSYMA* as an expert system).

Another type of declarative programming is *logic programming*. Here the idea is that a collection of axioms form a data-base of "facts" expressed in some uniform way typically, as in *PROLOG* (e.g. see Sterling and Shapiro, 1986), as a collection of relations. One programs by adding relations to the database and by forming queries about new relations. These relations are patterns of known functions, symbols, and "logic variables". The latter act as named wildcards to be used in matching one pattern with another. The pattern matching is called unification. Queries asked can be quite abstract with the result being a collection of conditions for which the query would be true (which can also be quite abstract). The result is a logical proof of the query, constructed automatically by the system.

Also declarative, and somewhat familiar to spreadsheet users, is *constraint-oriented programming*. Here the idea is to specify constraints amongst a number of program objects. The set of constraints form a network and changing the state of any of object causes information to be propagated through the netwoek so as to update the states of the remaining constrained objects. For some problems, this is the natural means of expressing the known structure. The idea appears first in Sutherland's (1963) *SKETCH-PAD* system for computer-aided design. Important early constraint systems include *ThingLab* (Borning 1977, 1981) and *Constraints* (Steele 1980, Sussman and Steele, 1980). See also Leler (1988) for a general overview of the area and definition of the constraint programming language *Bertrand*.

2.3.1 Object Oriented Programming

An important style of programming having both imperative and declarative features is *object-oriented programming*. Defining characteristics can be taken to be: the existence of template data structures called *classes* which containing named fields called slots, instances of classes called *objects*, *inheritance* relations between classes, and procedures called generic functions which dispatch to different *methods* depending on the class of their arguments. Together these provide powerful means of abstraction.

Properly done, classes isolate common structure in a single piece of code that can be accessed by newly defined code through inheritance. Complex concepts can be identified with classes and related one to another through the inheritance mechanism (e.g. recall *AXIOM*). Generic functions define

a general behaviour which can be specialized as appropriate. For example, '+' might be used to capture general notion of "addition" and so execute different code (methods) depending on whether the arguments are numbers, matrices, or perhaps even vector spaces. At execution, the system selects and executes the most specific method for the argument(s) given.

Many different designs exist for object-oriented programming languages. At present, important differences are whether methods can dispatch on the class of a single argument (typically implemented as 'message-passing') or on multiple arguments (the generic function approach), and whether class inheritance is single (one parent, multiple children) or multiple (multiple parents, multiple children). These differences follow the history of the object-oriented language development.

Things begin with *SIMULA*, where data and methods are encapsulated together in a single object. Single inheritance is possible and methods inherited from a superclass can be overridden by a subclass. These ideas were picked up by Kay (1969) and developed further during the 1970s with others at Xerox PARC to produce the explicitly object-oriented language *SMALLTALK* (*-72, -76, -80*) (Goldberg and Robson, 1983). Here methods are invoked by sending a message to the object; following the single inheritance path, the most specific method of that name is found and executed. Steele (1976ab) showed how this style of programming could be implemented in *LISP*.

The first system to support multiple inheritance was the *LISP* based system called *Flavors* from MIT (e.g. Cannon 1980, Weinreb and Moon 1980, Moon et al 1983, Moon 1986). Multiple parents appeared as "mixins", classes representing roughly orthogonal behaviours which might be 'mixed into' (as multiple parents) to provide the selected behaviours for any new class. This added enormous power but meant that method lookup was no longer straightforward; the class inheritance hierarchy was now a directed (acyclic) network as opposed to a tree as in single inheritance. *Flavors* also allowed modification to existing methods via, for example, so-called 'before' and 'after' methods which were executed before and after the 'primary' method. These are very useful but together with multiple inheritance meant that detemination of the approriate method would need to be determined dynamically at run-time.

Another *LISP* based system called *LOOPS* out of Xerox PARC (Bobrow, 1982) was much like *Flavors* in using multiple-inheritance and message-passing. A significant difference is that *LOOPS* also offered rule-based programming, and active-values (a value of an object slot which executed a program whenever the slot was accessed).

Multi-methods, methods which dispatch on the classes of multiple arguments, were introduced by the *COMMON LISP* based *CommonLoops* (Bobrow et al, 1986, Stefik and Bobrow 1986). Method determination is more complicated and the message passing metaphor is replaced by that of a generic function. Relieved of message passing's strong association with a single object, generic functions are developed independently as abstract actions. In many applications, such as mathematics, this seems the more natural metaphor.

This research culminated in the *COMMON LISP* object system *CLOS* (Bobrow et al. 1988) appearing in the ANSI standard language (Steele 1990) and supported by all commercial vendors of *COMMON LISP*. *CLOS* supports multi-method multiple inheritance object-oriented programming. For an overview, example applications, and comparison with other object-oriented programming languages, see Paepcke (1993). For a quick comparison see Appendix A of Kiczales et al (1991).

It can be argued that *CLOS* has moved significantly beyond object-oriented programming. Besides basic 'primary' methods, *CLOS* supports 'before', 'after', and 'around' methods. It is the first language to introduce a *metaobject protocol* (Kiczales et al, 1991) which opens the system itself up allowing the user to adjust the language design to better suit their needs. For example, implementation of generic functions, classes, object instantiation, methods, and method-combination can all be specialized by the user if so desired. This provides the researcher with extraordinary power to explore the use of software in symbolic computation.

2.3.2 Interfaces

The feature of *SMALLTALK* which most captured people's attention was its integrated direct manipulation interface. A large graphical display and a mouse allowed the user to interact directly with the objects; pop-up menus meant fewer typed in commands and more immediate interaction. The immediacy of the interface together with the ability to inspect, copy, or edit program objects gave the impression of an *environment* consisting of objects which could be manipulated. This was a stunning accomplishment at the time.

Xerox PARC was the home to *SMALLTALK* development and it is not surprising that most of the ideas seen today (and some not yet seen!) in windowed environments everywhere were first developed at PARC. High resolution bitmapped displays were combined with the dynamic computation available in *LISP* to produce the first 'exploratory programming environ-

ments' (see Teitelman and Masinter 1981, Sheil 1983). The goal was to provide programmers with 'power tools' designed to maximize their efficiency (the real bottleneck) rather than the machine's.

Tools well in advance of their time included data structure inspectors, source code editors tailored to the language, stack backtrace inspectors, program steppers, network file-browsers, and bitmap editors. At the type-in level, the *dwim* ("do what I mean") editor completed symbol names and offered corrected spelling. If a program failed, computation would be suspended, source code produced in an editor (possibly with an unknown symbol highlighted and a suggested correction), the code changed on the fly by the programmer (all local values would be available for execution of any part of the displayed source), and the computation restarted from where it left off. Other programming tools included interactive graphical displays of program structure which could dynamically highlight procedures as they were executed and calculate the amount of time spent within each procedure. [1]

Programming 'power tools' require the ability to write programs which symbolically manipulate other programs according to a clear model of the target programming language. As the model improves, say from as simple text tokens to one which incorporates the language syntax to one which includes language semantics, the tools become more powerful, say from source code bookkeeping to program structure editors to symbolic interpreters and program verification. The next step in this sequence was to go beyond a model of the language and to model recognized programming patterns as well. This approach is perhaps best typified by the *Programmer's Apprentice* project at MIT (e.g. Rich 1981, 1990). In this project, a library of standard programming forms culled from experienced programmers is set up. Called plans, these programming forms include familiar mathematical objects (e.g. functions, sets, etc.), abstract tools for describing algorithms (e.g. digraphs, threads), and named programming concepts and techniques (e.g. "search loop", "accumulation loop", "trailing pointer", etc.) The library of plans is used by the experienced programmer to build new programs; the system, as a good apprentice, provides the details to the programmer for approval.

[1] Other advances at Xerox by the early 1980s include the first "office automation system" with file-folders, etc., a wysiwyg word processing system (which included mathematical symbols) and the first hypertext system *Notecards*.

3 Symbolic computation in Statistics

In the spring semesters of 1965 and 1966, Martin Schatzoff delivered a graduate statistics seminar at Harvard entitled "Machine Aided Statistical Modelling". The objective was to better exploit the computer in the teaching and the practice of statistical data analysis or, more concretely, to "provide a high-degree of *statistically meaningful* man-machine interaction" (my emphasis, see Schatzoff 1968 page 194).

At the time, there were libraries of statistical programs widely available but these had unforeseen negative effects. First, they encouraged users to fit all problems to suit the available software which was in Schatzoff's words "of limited value in attacking a wide class of data analysis problems". Second, Schatzoff noticed that they actually effected a de-emphasis in the teaching of statistical computation and data analysis!

Because statistical analysis requires "human decision making in the computational sequence", on-line interactive systems seemed better suited. Conversely, it was hoped that by attacking problem areas like statistical analysis, which required rapid interaction, "valuable problem solving approaches not previously available" would result.

Of the three systems used in his course, two were created by Schatzoff and others at IBM's Cambridge Scientific Center. Both were interactive command language systems capable of data manipulation, random sample generation, data transformation, and some plotting facilities. Both systems were "couched in terms familiar to the statistician" and included the capability to write named procedures. Both were to be used in teaching and research. The important difference between the two lay in their focus.

The first system, *COMB* (Console Oriented Model Building, Schatzoff 1965), was tailored to residual analysis of two-way fixed effects anova models. Consequently it had additional commands for various statistical tests and residual plots peculiar to fitting and assessing this model (including adding or dropping an interaction term). *COMB* also had a more english-like interaction with the user. Besides teaching purposes, *COMB* was intended for research on residual analysis.

The second system, *COSMOS* (Console Oriented Statistical Matrix Operator System) is possibly the first interactive general purpose statistical computing system. *COSMOS* focussed on matrix operations, both basic and those of significant interest in statistics such as cross-product matrix and sweep calculations promoted by Beaton (1964). *COSMOS* also provided selection of subsets of the data according to satisfying a specified Boolean condition. It was intended that the user would use these "familiar" building

blocks for statistical model building and data analysis.

One aim was to "encourage statisticians to analyze data themselves at computer consoles, and to experiment with new techniques of data analysis" – hence the strong preference for an open-ended system over use of closed canned programs. Interestingly, *COSMOS* was also intended to help "provide a data bank consisting of real data from a variety of application fields."

The third system used in the course was the *Culler* system seen earlier in Section 2.1.3. Through a series of one-button pushes Schatzoff's class could construct and display density and distribution functions, draw samples from those distributions, and display cumulative sample functions on appropriate scales (e.g. normal probability). They would introduce outliers into samples and observe the effect on any display before and after Winsorizing, trimming, or rejecting outliers. Schatzoff (1968, p. 206) writes: "Utilization of the Culler system in this manner provided considerable insight into the operation of the indicated techniques as well as lively classroom discussion, which would frequently lead to suggestions which could be implemented spontaneously by pushing a few buttons. Such sessions were generally informative and enjoyable."

From our point of view, the important insight is Schatzoff's clear emphasis on bringing the machine closer to the statistician through statistically meaningful and immediate interaction. Though numerical calculations were the ultimate outcome, they were provided by directly interacting with computational representations of statistical concepts. In this way, *COMB*, *COSMOS* and Schatzoff's use of the *Culler* system are very similar to early symbolic computation efforts for applied mathematical purposes.

The first non-numerical mathematical computation paper appearing in the statistical literature seems to be Chambers[2] (1970). In now familiar words, Chambers (1970) is interested in automating "voluminous and tedious, but mechanical" mathematical manipulations – in this case of multivariate power series expansions as used in multivariate statistical theory. *ALPAK* is tried but soon abandoned; among other things it is unable to handle arrays whose dimensions are symbolic. Instead Chambers writes his own system called *SYMPOL*. *SYMPOL* is capable of handling polynomials of arbitrary order, arithmetic operations, differentiation, and generalized tensor products. Arrays of polynomials are handled and can be manipulated in the same way as arrays of variables; rules on scalar polynomials are extended

[2]A statistics graduate student at Harvard during the time of Schatzoff's course but not, apparently, an attendee. (Personal communication, Chambers, 1998.)

to symmetric arrays of polynomials. And finally, the formulae produced are said to be more compact, readable, and closer to conventional notation than those produced by *ALPAK*, *ALTRAN*, or *FORMAC* (see Section 2.1.2).

By 1970, then, we already see novel symbolic computation research and systems directed specifically toward statistics. Together these two early papers cover the spectrum of symbolic computational needs in statistical research, application, and teaching. Moreover, to quote Chambers (1970) and to echo Schatzoff (1968):

> "[O]ur computing needs will lead to systems which have a different emphasis than those developed for other applications. At the same time the systems will often have wider applicability than just to statistical research."

3.1 Non-numerical mathematical computation.

In statistical papers, non-numerical mathematical computation often arises as a means to solve some highly specified statistical problem. Maynard and Chow (1972) provides an early example. These authors needed the mean squared error of a complicated estimator. They had available to them a technique to produce a series expansion whose first few terms would suffice as an approximation. However, algebraic derivation of the terms by hand would be long and tedious. Instead, the desired result was had by writing an *ALTRAN* program.

This use is typical of that found in the statistical literature. A specific problem arises in some research and is addressed by writing a tailor-made program that implements a known algorithm which would otherwise have been followed by hand. To date, most statistical papers involving non-numerical mathematical computation are of this kind (e.g. check statistical references found in Kendall's 1993 review paper).

In contrast, in Quednau (1976) it is as if the problem was selected *because* computer algebra systems existed. Quednau's interest is in making more statistical models available to the practitioner. While the theory, here likelihood methods, had been developed in general the details would typically need to be worked out for any new model developed and a program written to yield the numerical results. Typical practice would be to restrict the choice of models to those for which programs existed and were accessible (typically normal models). Quednau (1976) expanded this choice by providing a system of programs which would "generate automatically the subroutines necessary for performing a special likelihood ratio test."

More specifically, the user supplied the density or spectral function in symbolic form of each population from which observations had been drawn and then identified those parameters hypothesized to be equal. Subroutines were then generated symbolically which would calculate negative log-likelihoods and their derivatives. These were then to be used in conjunction with some numerical minimization routine to perform the necessary analysis. The system was written in *PL/I-FORMAC* and produced *PL/I* code.

Quednau (1976) is notable for two things. First, the problem attacked is of wide interest; most statisticians would like to have access to that type of software. Second, the problem area does not end with symbolic computation but with numerical evaluation based on realized data. Consequently, the target population of users is quite large.

These first three papers – Chambers (1970), Maynard and Chow (1972), and Quednau (1976) – demonstrate three distinct possibilities for non-numerical mathematical computation in statistics. Maynard and Chow (1972) use a system to solve their problem. Chambers (1970) develops a symbolic computation system to provide tools to researchers in a particular area. Elements of *SYMPOL* could be useful outside this research area. And Quednau (1976) integrates the symbolic and the numeric to provide tools that could see wide use in practice.

With their increased availablity in the 1980s, we see increased use of computer algebra systems in statistical science (e.g. see Kendall, 1993). Systems used include *REDUCE*, *MACSYMA*, *Maple*, and *Mathematica*. Notably absent, to date, is *AXIOM/SCRATCHPAD* (although John Nelder's endorsement appears on the cover of the *AXIOM* manual). The motivation is often to avoid tedious but well understood calculation.

More ambitious use is what Steele (1987) calls the 'honest path of [computer algebra] application' (in this case *MACSYMA*). Steele (1987) suggests that a computer algebra system be used for encouragement rather than proof. Rather than a "frontal assault on the general problem", the system is engaged with the problem through exploration of a significant example. Moreover, the system should not be reserved only for the hard problems, but also for performing mathematical calulations which could be performed with "much humbler tools". An example of this kind of use is described in Steele (1987).

Following this honest path, one is soon led to constructing computational abstractions useful to the problem at hand. Pursuing problems related to the statistical analysis of shape has led Kendall in a series of papers (see Kendall 1993 for references) to develop some rather general computational abstractions in *REDUCE* and *Mathematica* for the stochastic calculus based

on the Îto formula. Kendall(1993) also describes the work of others in this area who, interestingly, have built quite different computational structures. This experience leads Kendall (1993) to draw a clear distinction between "the use of computer algebra packages to *support* investigations . . . and their use to *implement structure*" (his emphasis).

Andrews and Stafford (1993) use *Mathematica* to capture and exploit the mathematical structure common to derivation of asymptotic expansions of functions of sums of independent and identically distibuted random variables. The motivation is traditional: relegation of tedious calculational detail to the software, here freeing researchers "to concentrate on the structure of the calculation rather than on the detail of term-by-term evaluation" (Andrews and Stafford, 1993, p. 627). The implementation strategy is similarly straightforward: "emulate techniques that one would normally carry out by hand" (Stafford et al, 1994, p. 244). The structure implemented is intended to be general enough so as to raise the level of discourse for researchers in this area – "Here the presentation of new statistical results appears as examples instead of entire publications" (Stafford and Andrews, 1993, p. 716).

In the implementation, one sees computational abstractions which match concepts peculiar to this specialized research area (e.g. `BarnNeilCorr`, `Conditionalscore`, `Delta`, etc.). More general sounding abstractions such as `GenExpand`, or `ParameterDerivative` exist but their meaning is restricted to this specialized research domain. Very general statistical abstractions are also captured, among them `RV` to define a random variable, `Law` to define a probability law, and `Likelihood` as a curious choice to define the log-likelihood of a law. Some attention is also given to presentation of the results in a format which is relatively standard for the research area.

Stafford and Bellhouse (1997) have implemented similar *Mathematica* procedures to compute complicated algebraic expressions which can arise in sample survey theory. This work has led them to consider algorithms for automatically generating all partitions of a finite set.

3.2 Other developments.

In the 1960s, batch mode statistical systems gathered together libraries of statistical routines and made them more easily accessible to a wide variety of users. These provided the user simpler means to specify the problem and output tailored to the problem, a level of abstraction above the subroutines themselves but only just. Systems familiar today dating from this time include *BMDP*, *SPSS*, and *SAS*.

From the mid-1960s through the late 1970s many 'interactive statistical systems' were developed.[3] To encourage the illusion of immediacy, clear and crisp communication between the user and the system was required. The verbosity prevalent in batch-oriented systems could not be tolerated (e.g. see Ryan et al, 1975 on *Minitab*). Pseudo-natural language was rejected in favour of statistically meaningful commands like regress or cancorr as seen for example in *ISP* (Interactive Statistical Processor, Bloomfield 1977). Programming capabilities in these systems essentially stopped at sequential execution of the system commands. Any new functionality would require programming in some underlying language like *FORTRAN*. As pointed out by Guthrie (1975), this meant that statistical analysts would often choose to use only those statistical procedures offered by the system, the result being

> "Even the statistician's operating language, context and syntax, became formed from the names of available programs and functions. In order to regain his individuality, it became necessary for the thinking statistician to teach computers to do his wishes ... That is, he had to learn to program or hire a programmer." ... Guthrie (1975, pp 8-9)

An early exception was the econometric modelling system called *TROLL* (Time Reactive On Line Laboratory). Begun 1n 1966, by 1969 *TROLL* allowed the user to write named 'macros', which could take arguments and execute sequences of commands. Eisener and Hill (1975) describe how, together with *TROLL*'s *BASIC*-like algebraic manipulation facility (including symbolic differentiation), they were able to program in *TROLL* to produce new commands (e.g. robust regression) relatively quickly. Even so, the designers had not foreseen the macro facility being used as a programming language and it was a rather crude one as a result.

Eisener and Hill (1975) planned to rectify that in a redisign by introducing a three layer programming model. The base component would be a collection of subroutine libraries callable from the next component, an algorithmic language with *APL*-like semantics and *ALGOL*-like syntax. The final component was a control language in which users and programmers alike could define new commands. ... like Chambers later?

compiler compilers ... chambers

Together with A distinguishing feature of these systems was their immedi Most early systems provided little more than a statistical command

[3] Too many to survey here as even a cursory look over the proceedings of the annual symposia on *Computer Science and Statistics: the Interface* from this period would show.

language. Because the interaction was immediate, care needed to be taken to simplify input and output so as to keep the communication clear and crisp between the user and the system (e.g. Ryan et al, 1975 on *Minitab*). Sometimes the statistical commands could be very high-level reflecting recent research interests (e.g. *ISP* – Interactive Statistical Processor, Bloomfield 1977).

programming ... statistical abstract. troll ... data abstraction

p (see *Minitab*'s earl by, like the early interactive *Minitab* system (Ryan et al 1975) provided a simplified statistical command language Some of these systems like *Minitab* (Ryan et al 1975) and *ISP* (Interactive Statistical Processor, Bloomfield 1977) provided little more than a statistical command line interface. Others like *TROLL* (Time Reactive On Line Laboratory) recognized early on the need to provide the user some programming capabilities and found themselves Each computational innovation spawns a suite of new systems – from time-shared network computers, to the low cost but memory constrained 'mini-computers', to the more ubiquitous but severely constrained early personal computers, to the increased availability of high-resolution screens, to today's revived[4] interest in internet based systems. Older systems adapt as best they can with each new innovation.

The amount of abstraction in these systems vari Notable in this work is the *TROLL* (Time Reactive On Line Laboratory) for its use of distinct data structures for time series and for systems of equations complete with parameters, random variables, and distinction between exogenous and endogenous variates. Early on *TROLL* (e.g. see Eisener and Hill, 1975 and references therein). Guthrie (1975) describes the s The nature of much of this work is the direct exploitation of the new technology. Of interest here, are developments which

Here I will only highlight some of those developments that fit In the 1975 proceedings of the *Computer Science and Statistics Interface* symposium, I count fully n systems.

Statisticians are typically evaluators and wishers of software. Guthrie's warning ... Looking over the interface proceedings of the 1970s one sees that statisticians become the system developers. Interest in language semantics increases, examples. Worth noting here are TROLL for its time series and modelling data structures. S for Chamber's hierarchical data structures.

Graphics follows suit coming into its own in dynamic graphics in the 1980s. Though 3d-point cloud rotation is older. PRIM-9 PRIMH PRIM79

[4]E.g. see Marks (1975) and comments in Guthrie (1975) who organized the Workshop entitled 'Interactive statistical computing and computer networks'.

Dataviewer XGOBI

spawns a whole suite of such systems Developments in statistical computing paralleled those in computing more generally, albeit sometimes lagging by a few years. Focusing on those developments related to the broad view of symbolic computing the mid 1970s there is The best sources on developments in statistical computing, especially prior to the 1990s are the proceedings of the annual symposia on the *Interface of Computer Science and Statistics* There has been much research on symbolic computation in statistical science that is not non-numerical mathematical computation, a few examples of which are given to indicate the breadth of the subject. The third part draws largely from the work of myself and others to illustrate the broad spectrum of symbolic computation in statistics. This breadth is essentially a result of statistics not being a branch of mathematics but a scientific discipline in itself, one which makes heavy use of mathematics at times but whose problems, objectives, and standards can be quite different. So too exploring non-numerical mathematical computation for statistical uses is important, but restricting symbolic computation in statistics to this would make little sense.

4 Conclusions

Steele (1987) is suggesting a computer algebra system become a convenient interactive computational environment which *supports* the mathematician's problem solving activity (as described for example by Polya 194?), an exploratory mathematical environment. it seems is much like that of the exploratory programming environments of Section As was the case with the exploratory programming environments of Section 2.?.?, except that the activity takes place with the support of an interactive computational environment.

Draw on Steele 1987, refer to Quednau average performance.

5 References

Abelson, H. and G.J. Sussman, with J. Sussman (1985). *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge MA USA

Andrews, D.F. and J.E. Stafford (1993). "Tools for the Symbolic Computation of Asymptotic Expansions", *JRSS-B*, Vol. 55, # 3, pp. 613-627.

Anscombe, F.J. (1982). *Computing in Statistical Science Through APL.* Springer-Verlag, NY USA

Beaton, A.E. (1964). "The Use of Special Matrix Operators in Statistical Calculus", *Research Bulletin RB-64-51*, Educational Testing Service, Princeton NJ USA.

Blackwell, F.W. (1967). "An On-Line Symbol Manipulation System", *Proc. ACM 22nd Nat'l Conf.*, pp. 203-209.

Bloomfield, P. (1977). "An Interactive Statistical Processor for the Unix Time-Sharing System" *Proc. of Comp. Sci. and Stats.: 10th Ann. Symp. on the Interface* (eds. D. Hogben and D.W. Fife), pp. 2-8, Nat. Bur. of Standards, Washington USA.

Bobrow, D.G. (1982). *LOOPS: An Object-Oriented Systemm for Interlisp*, Xerox PARC, CA USA.

Bobrow, D.G., L.G. Demichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon (1988) *Common Lisp Object System Specification*, ANSI committee X3J13 Document 88-002R, June; also appears in *Lisp and Symbolic Computation*, Vol. 1, Jan. 1989, pp. 245-394, and as Chapter 28 of Steele (1990), pp. 770-864.

Borning, A.H. (1977). "ThingLab – An object-oriented system for building simulations using constraints." *Proc. if the 5th Int'l Joint Conf. on AI.*

Borning, A.H. (1981). "The programming language aspects of ThingLab, a constraint-oriented simulation laboratory.", *ACM Trans. on Prog. Lang. and Sys.*, Vol. 3, pp. 353-387.

Buchberger, B. et al (1985). "Symbolic Computation (An Editorial)", *J. of Symb. Comp.*, Vol. 1, pp. 1-6.

CACM (1960). Editorial introduction to special issue on symbolic computation, *Comm. of the ACM* Vol 3, No. 4 (April).

CACM (1966). Special issue on the March 1966 ACM Symposium on Symbolic and Algebraic Manipulation, *Comm. of the ACM* Vol. 9, No. 8 (August).

Cannon, H.I. (1980). "Flavors", *AI Lab Technical Report*, MIT USA.

Church, A. (1941) *The Calculii of Lambda Conversion*, Princeton U. Press, Princeton, NJ USA.

Cameron, S.H., Ewing, D., and M. Liveright (1967). "DIALOG: A Conversational Programming System with a Graphical Orientation", *Comm. ACM*, Vol. 10 No. 6 (June), pp. 349-357.

Cannon, H.I. (1980). "Flavors", AI Lab Tech. Report, MIT, Cambridge MA USA.

Caviness, B.F. (1986). "Computer Algebra: Past and Future", *J. Symb. Comp.* Vol. 2, pp. 217-236.

Chambers, J.M. (1970). "Computers in Statistical Research: Simulation and Computer-Aided Mathematics" *Technometrics*, Vol. 12, #1, pp. 1-15.

Char, B.W., G.J. Fee, K.O. Geddes, G.H. Gonnet, and M.B. Monagan (1986). "A Tutorial Introduction to Maple", *J. of Symb. Comp.*, Vol. 2, pp. 179-200.

Char, B.W., K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt (1992). *First Leaves: A tutorial introduction to Maple V*, Springer-Verlag, New York USA.

Clapp, L.C. and R.Y. Kain (1963). "A Computer Aid for Symbolic Mathematics", *Proc. FJCC*, Vol 24 (Nov), pp. 509-517.

Clapp, L.C. et al (1966) *Magic Paper: An On-Line System for the Manipulation of Symbolic Mathematics*, Computer Research Corp., Report No. R 105-1 (April), Newton MA USA.

Collins, G.E. and R. Loos (1986). *ALDES/SAC-2 Reference Manual*, Springer-Verlag, Vienna Austria.

Culler, G.J. and B.D. Fried (1965). "The TRW Two-Station On-Line Scientific Computer: General Description", *Computer Augmentation of Human Reasoning* (M. Sass and W. Wilkinson, eds.), Spartan Books, Washington,

DC USA.

Dahl, O. and K. Nygaard (1966). "SIMULA – An ALGOL-Based Simulation Language", *Comm. of the ACM*, Vol. 9, No. 9 (Sept.), pp. 671-678.

Davenport, J.H., M.C. Dewar andf M.G. Richardson (1992). "Symbolic and Numeric Computation: the Example of IRENA", in Donald et al (editors) 1992, pp. 347-362.

Davenport, J.H., Siret, Y., and E. Tournier (1993). *Computer Algebra: Systems and Algorithms for Algebraic Computation*, (Second Edition), Academic Press, London UK.

DesVignes, G. and R.W. Oldford (1988). "Graphical Programming", Video TR STAT-88-19, University of Waterloo, also appearing in the *ASA Stat. Graphics Sect.* video library.

Dewar, M.C. (1989) "IRENA – An Integrated Symbolic and Numeric Computation Environment", *Proc. ISSAC '89* (ed. G.H. Gonnet), ACM NY USA, pp 171-179.

Dimsdale, B. and H.M. Markowitz (1964). "A Description of the SIMSCRIPT Language", *IBM Systems Journal*, Vol. 3, No. 4, pp. 57-67.

Donald, B.R., D. Kapur and J.L. Mundy (1992). (editors) *Symbolic and Numerical Computation for Artificial Intelligence*, Academic Press, New York USA.

Engelman, C. (1965). "MATHLAB – A Program for On-Line Machine Assistance in Symbolic Computations", *Proc. FJCC* Vol 27, pt. 2, November, pp. 117-126.

Engelman, C. (1971). "The Legacy of MATHLAB 68" *Proc. 2nd Symp. on Symbolic and Algebraic Manipulation* Los Angeles.

Fitch, J. (1985). "Solving Algebraic Problems with REDUCE", *J. of Symb. Comp.*, Vol. 1, pp. 211-227.

Geddes, K.O. (1984). "On the design and efficiency of the Maple system", *Proc. of the 1984 MACSYMA Users' Conference*, 199, (ed E.V. Golden).

Geddes, K.O., G.H. Gonnet, and B.W. Char (1982). *MAPLE User's Manual (Second Edition)*, Dept. of Comp. Sci,, University of Waterloo, Canda.

Gelernter, H., Hansen, J.R. and C.L. Gerberich (1960) "A FORTRAN-Compiled List-Processing Language", *J. ACM*, Vol. 7, No. 2 (April), pp. 87-101.

Glynn, J. (1989). *Exploring Math from Algebra to Calculus with Derive, A Mathematical Assistant*, MathWare.

Goldberg,A. and D. Robson (1983). *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading MA USA.

Gordon, G. (1961). "A General Purpose Systems Simulation Program", *Proc. EJCC*, Vol. 20, pp. 87-104.

Guthrie, D. (1975). "Dangers in Interactive Statistical Systems" *Proc. of Comp. Sci. and Stats.: 8th Ann. Symp. on the Interface* (ed. J.W. Frane), pp. 8-10, UCLA, USA.

Harper, D., C. Wooff, and D. Hodgkinson (1991). *A Guide to Computer Algebra Systems*, John Wiley & Sons, NY USA.

Hayes-Roth, F., D.A. Waterman and D.B. Lenat (1991). (eds.) *Building Expert Systems*, Addison-Wesley, Reading MA, USA.

Hearn, A.C. (1971). "Reduce-2: A System and language for Algebraic Manipulation" *Proc. of the Second Symp. on Symbolic and Algebraic Manipulation*, (ed. S.R. Petrick), SIGSAM ACM, pp. 115-127.

Hearn, A.C. (1987). *REDUCE-3 User's Manual, version 3.3* Rand Corporation Publication CP78 (7/87).

Hurley, C.B. and R.W. Oldford (1988). "Hierarchical views of statistical objects", Video TR STAT-88-20, University of Waterloo, also appearing in the *ASA Stat. Graphics Sect.* video library.

Hurley, C.B. and R.W. Oldford (1991). "A software model for statistical graphics", *Computing and Graphics in Statistics*, Institute for Mathematics

and its Applications, Vol. 36, Springer-Verlag.

Iverson, K.E. (1962). *A Programming Language*, John Wiley & Sons, NY USA.

Jenks, R.D. (1984). "The New SCRATCHPAD Language and System for Computer Algebra", *Proc. of the 1984 MACSYMA Users' Conference*, 409, (ed. E.V. Golden).

Jenks, R.D. and R.S. Sutor (1992). *AXIOM: The Scientific Computation System*, Springer-Verlag NY USA.

Kahrimanian, H.G. (1954). "Analytical Differentiation by a Digital Computer", *Symposium on Automatic Programming for Digital Computers*, Office of Naval Research, Dept. of the Navy, Washington D.C. pp. 6-14.

Kay, A. (1969). *The Reactive Engine*, Ph.D. thesis, University of Utah, USA.

Kendall, W.S. (1993). "Computer algebra in probability and statistics", *Statistica Neerlandica* Vol. 47, #1, pp. 9-25.

Kiczales,G., J. des Rivières, and D.G. Bobrow (1991). *The Art of the Metaobject Protocol*, MIT Press, Cambridge MA USA.

Kovacic, J.J. (1986). "An algorithm for solving second-order linear homogeneous differential equations", (ms. appearing first in 1979), *J. Symb. Comp.*, Vol. 2, pp. 3-43.

Leler, Wm. (1988). *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, Reading MA USA.

MacCallum, M.A.H. and F.J. Wright (1991). *Algebraic Computing with Reduce*, Oxford University Press, Oxford U.K.

Marks, G.A. (1975) "Design of a Statistical System for a Network Environment" *Proc. of Computer Science and Statistics: 8th Annual Symposium on the Interface*, Edited by J.W. Frane, UCLA USA.

Martin, W.A. (1967) *Symbolic Mathematical Laboratory*, M.I.T., MAC-TR-36 (Ph.D. thesis), Project MAC, Cambridge MA USA.

Martin, W.A. and R.J. Fateman (1971). "The MACSYMA System", *Proc. of the Second Symp. on Symbolic and Algebraic Manipulation*, (ed. S.R. Petrick), SIGSAM ACM, pp. 59-75.

Maynard, J.M. and B. Chow (1972). "An approximate Pitman-type 'close' estimator for the negative binomial parameter p", *Technometrics*, Vol. 14, pp. 77-88.

McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Pt. 1", *Comm. ACM*, **3**, No. 4 (April), pp. 184-95.

Moon, D., R. Stallman and D. Weinreb (1983). *The Lisp Machine Manual*, AI Lab, MIT, Cambridge MA USA.

Moon, D. (1986). "Object-Oriented Programming with Flavors", *Proc. ACM Conf. on Object-Oriented Systems, Langugaes, and Applications.*

Moses, J. (1967). *Symbolic Integration.* Ph.D. thesis, also as Project MAC Tech. Report TR-47, MIT Cambridge MA, USA.

Newell, A. and H.A. Simon (1956). "The Logic Theory Machine – A Complex Information Processing System", *IRE Trans. Information Theory*, Vol. IT-2, No. 3 (September), pp. 61-79.

Newell, A. and H.A. Simon (1961). "GPS, A Program That Simulates Human Thought", *Computers and Thought*, (E. Feigenbaum and J. feldman, eds.). McGraw-Hill, New York, pp. 279-293.

Nolan, J. (1953). *Analytical Differentiation on a Digital Computer* (M.A. thesis), M.I.T., Cambridge MA USA.

Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufman, San Mateo CA USA.

Oldford, R.W. (1987). "Abstract Statistical Computing", *Bull. of the Int'l Stat. Inst.: Proc. of the 46th Session*, Vol. LII, Book 4, pp. 387-398.

Oldford, R.W. (1990). "Software abstraction of elements of statistical strategy", *Annals of Mathematics and Artificial intelligence*, Vol. 2, pp. 291-308.

Paepcke, A. (1993). Editor, *Object-Oriented Programming: The CLOS perspective*, MIT Press, Cambridge MA USA.

Pavelle, R. and P.S. Wang (1985). "MACSYMA from F to G", *J. of Symb. Comp.*, Vol. 1, pp. 69-100.

Quednau, H.D. (1976). "The Comparison of Parameters Estimated from Several Different Samples by Maximum Likelihood", *Biometrics* Vol. 32, pp. 683-688.

Reinfelds, J. et al (1966). "AMTRAN, A Remote-Terminal, Conversational-Mode Computer System", *Proc. ACM 21st Nat'l Conf.*, pp. 469-477.

Rich, C. (1981). "Inspection Methods in Programming" Ph.D. thesis revised as Tech. Report 604, AI Lab. MIT, Cambridge MA USA.

Rich, C. (1990). *The programmer's apprentice.* ACM Press, Addison-Wesley, Reading MA USA.

Risch, R.H. (1969). "The problem of integration in finite terms", *Trans. Amer. Math. Soc.*, Vol. 139, pp. 167-189.

Risch, R.H. (1970). "The solution of the problem of integration in finite terms", *Bull. Amer. Math. Soc.*, Vol. 76, pp. 605-608.

Ryan Jr., T., R.F. Kohm and B. Joiner (1973). "Interactive Statistics" *Proc. Comp. Sci. and Stats: 8th Ann. Symp. on the Interface*, ed. J.W. Frane, pp. 66 - 72, UCLA USA.

Sakoda, J.M. (1965) *DYSTAL Manual – Dynamic Storage Allocation Language in FORTRAN.* Brown U., Dept. of Sociology and Anthropology, Providence, RI, USA.

Sammet, J.E. (1969). *Programming Languages: History and Fundamentals*, Prentice-Hall, Inc. Englewood Cliffs, NJ USA.

Schatzoff, M. (1965). "Console Oriented Model Building", *Proc. of the 20th*

Nat'l. Conf. of the ACM, pp. 354-374.

Schatzoff, M. (1968). "Applications of Time-Shared Computers in a Statistics Curriculum" *JASA*, Vol. 63, # 321, pp.192-208.

Sheil, B.A. (1983). "Power Tools for Programmers", *Datamation*, Feb., pp. 131-144.

Singer, M.F. (1981). "Liouvillian solutions of nth order homogeneous linear differential equations", *Amer. J. Math.*, Vol. 103, pp. 661-682.

Slagle, J.R. (1961). "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus. Symbolic Automatic Integrator," Ph.D. thesis, Rept. 5G-0001, Lincoln Lab., MIT Cambridge, MA USA.

Slagle, J.R. (1963). "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus," *J. ACM*, *Volume 10*, *No. 4* (Oct. 1963), pp. 507-520.

Shortliffe, E.H. (1976). *Computer-Based Medical Consultation: MYCIN*. American Elsevier.

Stafford, J.E. and D.F. Andrews (1993). "A symbolic algorithm for studying adjustments to the profile likelihood", *Biometrika*, Vol. 80, #4, pp. 715-730.

Stafford, J.E., D.F. Andrews, and Y. Wang (1994). "Symbolic Computation: a unified approach to studying likelihood" *Statistics and Computing*, Vol. 4, pp. 235-245.

Stafford, J.E. and D.R. Bellhouse (1997). "A Computer Algebra for Sample Survey Theory", unpublished manuscript, 28 pages.

Steele, G.L. Jr. (1976a). "LAMBDA: The Ultimate Imperative", AI Lab Memo 353, MIT, Cambridge MA USA.

Steele, G.L. Jr. (1976b). "LAMBDA: The Ultimate Declarative", AI Lab Memo 379, MIT, Cambridge MA USA.

Steele, G.L. Jr. (1980). "The Definitions and Implementation of a Computer Programming based on CONSTRAINTS", Ph.D. thesis (also Tech.

Report 595) AI Lab., MIT Cambridge MA, USA.

Steele, G.L. Jr. (1990). *Common Lisp: The Language (Second Edition)*, Digital Press.

Steele, M.J. (1987). "An Application of Symbolic Computation to a Gibbs Measure Model", *Computer Science and Statistics: Proc. of the 19th Symposium on the Interface*, pp. 237-240.

Sterling, L. and E. Shapiro (1986). *The Art of Prolog*, MIT Press, Cambridge MA USA.

Stoutmeyer, D.R. (1985). "A preview of the next IBM-PC version of mu MATH", *Proc. of Eurocal '85*, Vol I, (ed. B. Buchberger), *Springer-Verlag Lecture Notes in Computer Science*, Vol. 203, pp. 33-44.

Sussman, G.J. and G.L. Steele Jr. (1980). "Constraints – A langugae for expressing almost-hierarchical descriptions." *AI Journal* Vol. 14, pp. 1-39.

Sutherland, I.E. (1963). "SKETCHPAD: A man-machine graphical communication system." Tech. Report 296, MIT Lincoln Lab., Lexington MA USA.

Sutor, R.S. (1985). "The Scratchpad II computer algebra language and system", *Proc. of Eurocal '85*, Vol II, (ed. B.F. Caviness), *Springer-Verlag Lecture Notes in Computer Science*, Vol. 204, pp. 32-33.

Teitelman, W. and L. Masinter (1981). "The INTERLISP programming environment", IEEE *Computer*, Vol. 14, # 2, pp. 25-33.

Wang, H. (1960) "Proving Theorems by Pattern Recognition I", *Comm. ACM* **3**, No. 4 (April).

Weinreb, D. and D.A. Moon (1980). "Flavors: Message Passing in the Lisp Machine", AI Memo no. 602, Project MAC, MIT, Cambridge MA, USA.

Wolfram, S. et al (1983). *SMP Reference Manual*, Inference Corp. Los Angeles USA.

Wolfram, S. (1988). *Mathematica: A system for doing mathematics by computer*, Addison-Wesley, Reading MA USA.

Yngve, V.H. (1957) "A Framework for Syntactic Translation", *Mechanical Translation* Vol. 4, No. 3 (December), pp. 59-65.