option or program in *computer engineering* at the undergraduate level, and 34 more showed some indication that there might be such a program or option within a year. If this happens, then over half of our departments will have such an option or program. In other words, the computer is becoming well established in electrical engineering. It is a pleasure to observe that we can now concern ourselves about second-order effects such as whether the introduction of algorithms hampers or improves engineering insight!

VIII. Conclusions (M. L. Dertouzos)

All panelists agree, as is to be expected, that both insight and algorithms are necessary. Further probing and discussion, however, reveal some distinct views. One such view, in its exaggerated form, is that engineering insight should reign supreme with algorithms and computers as obedient numerical servants. Another view regards the computer as a system worthy of serious study by electrical engineers, for example, through courses that deal with computer architecture and data structures. Yet another view is that of computer acculturation, i.e., the *required* exposure in a serious way to computation through *any one* student-elected avenue, e.g., a course, a project, or industrial co-op work.

Some additional views, expressed during the panel discussion are summarized as follows.

Excessive concentration on algorithms may damage insight, e.g., by allowing the student to think of a capacitor as a resistor in series with a voltage source.

Large and complex systems tend to be counter intuitive. Systematic study, backed up by algorithms is the proper medicine.

The more interesting real-world problems are those dealing with nonlinear circuits and systems. These are best explored interactively through a feedback loop system which contains the student and the programs for analysis, optimization, etc.

A computer must be studied and understood as an integral part of an overall system. Consequently, it must not be used only as a tool, but it must be studied as a system, in terms of machine language, microprogramming, and so forth.

Students are very versatile and can easily develop both their insight and their algorithms. They have proven this by using i or j properly when addressing mathematicians or engineers during oral exams.

Perhaps it is appropriate to close this discussion with a view which was not discussed by the panelists. It is the recently emerging doctrine (among artificial intelligence research groups), that the computer has educational value primarily because it enables the student to model (and therefore analyze and improve) his thinking process. According to this view, algorithms are a *basis* for developing and improving insight.

On Important Current Issues Concerning Computers in Electrical Engineering Education

CHRISTOPHER POTTLE, MEMBER, IEEE, MICHAEL L. DERTOUZOS, MEMBER, IEEE, IMSONG LEE, MEMBER, IEEE, AND KENNETH C. SMITH, MEMBER, IEEE

I. Introduction (C. Pottle)

THE REMARKS which follow were contributed by the members of a panel at the Purdue 1971 Symposium on Applications of Computers to Electrical Engineering Education (Lafayette, Ind.) entitled, "Everything You Always Wanted to Know About Computers—But Were Afraid to Ask." These panelists have a definite bias toward computer science and know a good deal about computer organization and programming. They would all, however, claim to be electrical engineers with a primary interest in the education of future electrical engineers.

These panelists were given considerable latitude to discuss whatever topics they believed to be important issues in a modern computer-oriented curriculum. One would naturally suspect that they would choose for discussion those topics which were being investigated with particular emphasis on their campuses. Their contributions appear to be in two general areas, one quite distinct from the other, and both of great interest to any educator who is trying to keep informed of trends in computers and their application to electrical engineering education. The first of these topics concerns the appropriate hardware and software support for efficient learning on the part of the student while maintaining efficient use of the computer. I. Lee discusses the

Manuscript received April 12, 1971; revised May 4, 1971.

C. Pottle is with the Department of Electrical Engineering, Cornell University, Ithaca, N. Y. He is currently on sabbatical leave with the Thomas J. Watson Research Center, Yorktown Heights, N. Y.

M. L. Dertouzos is with the Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass. 02139.

I. Lee is with the Department of Electrical Engineering, University of Massachusetts, Amherst, Mass.

K. C. Smith is with the Department of Electrical Engineering, University of Toronto, Toronto, Ontario, Canada.

complementary roles of the central time-sharing system and the laboratory minicomputer, giving some cost estimates for both. K. C. Smith contrasts the advantages and disadvantages of the remote batch system with those of the time-sharing facility and proposes a possible alternative—the use of many small computers in a distributed "open shop" environment. The second area of interest has to do with the choice of a programming language for general student use and the sort of programs the student will be called upon to implement. M. L. Dertouzos does not believe the former to be a really serious question, but comments on the latter, favoring the user-written program approach over the canned program, with the canned subroutine interconnected by the student as a viable compromise. C. Pottle finds that the choice of a general programming language has generated a great deal of heat at his university recently and suspects many other colleges are also facing this problem. His remarks are an attempt to discourage the teaching of Fortran as the basic programming language and to encourage the substitution of PL/1 or APL, or both.

II. The Complementary Roles of Minicomputers and Time-Sharing Systems in Computer-Oriented Electrical Engineering Curricula (I. Lee)

My comments are intended to cover the respective roles and comparative cost of minicomputers and time-sharing systems in a computer-oriented electrical engineering curriculum, based on our experience at the Amherst campus of the University of Massachusetts.

The time-sharing computer system called UMASS has been in operation for the last three years [1]. UMASS is implemented on a CDC 3600 computer system with drums and disks to store user programs and files. It is capable of handling 60 ports (teletype terminals in active use) simultaneously and supports several languages including Fortran, Basic, APL, and Mixal (Mix Assembly Language) [2]. Each user is assigned 8K or 16K words (48-bit) of work space in core memory and the system is available about 13 hours each day. In 1970, there were 3300 student users of UMASS logging 46 689 port hours. This comes to an average of 14 port hours per student per academic year and each port hour costs $5.00.

The average beginning student enrolled in the Computer Systems Engineering Program offered by the Department of Electrical Engineering has been found to use about 140 port hours per student per academic year, approximately 5 hours per student per week.

The time-sharing system has been found extremely useful in teaching assembly and machine language programming using Mixal and nonnumerical processing, i.e., data structure and string manipulations using Mixal and APL. It has also been found very useful in introductory programming courses for freshmen, advanced programming courses for computer science students, and nonproduction-type scientific computations and engineering simulations. The system is not suitable, however, for teaching interrupt handling, overlapping I/O operations, computer control of real-time physical processes, operating systems, and hardware/software interactions in system design and operations.

TABLE I
Estimated Cost of Minicomputers

Machine Configuration	Cost A (per h)	Cost B (per h)
Mini I	$3.30	$5.00
Mini II	$5.00	$7.20

Mini I Configuration: PDP 11/20; 8K words of core (16 bit/word). High-speed paper tape punch/read, real-time clock.
Mini II Configuration: Mini I plus disk (64K words), hardware multiply/divide unit, A/D converter 8 channel, Infoton Video Terminal

Laboratory Minicomputers

An experimental computer laboratory seems to be a necessity in any *balanced* computer-oriented curriculum covering computer technology, methodology, and application. The question is not one of time-sharing versus minicomputer but what sort of computer is best for the laboratory experience of computer-oriented electrical engineers. Recent advances in MSI and LSI technology are already having a profound impact on computer memories, peripherals, central processors, and hardware/software tradeoffs in computer system architecture in general. These important current developments can hardly be taught without laboratory computer facilities. Since funds for laboratory facilities are scarce, it seems useful to estimate the approximate cost of such facilities. Table I summarizes the cost of a computer laboratory for two different minicomputer configurations. The cost figures are approximate and based on the PDP-11 computer manufactured by the Digital Equipment Corporation.

Cost A was based on a purchase option and Cost B on a rental option. It was assumed that the machine would be available 60 hours per week for a 30-week academic year. The annual maintenance cost was assumed to be 10 percent of the purchase price and a five-year depreciation period was used for calculation of Cost A. The figures point out the cost advantage of the purchase option if one can afford it. At the risk of comparing "apples and oranges," one can say that the cost of time-sharing is approximately comparable to that of a minicomputer.

Conclusion

Time-sharing systems and laboratory minicomputers serve different functions and purposes. One cannot replace the other any more than apples can replace oranges. The two cost about the same and both are needed in the education of computer-oriented electrical engineers. If one must choose between them, the choice should be made on the basis of educational goals and needs of a particular department.

References

[1] C. C. Foster, "An unclever time-sharing system," *Computing Surveys*, vol. 3, no. 1, Mar. 1971.
[2] D. E. Knuth, *Fundamental Algorithms, The Art of Computer Programming*, vol. 1. Reading, Mass.: Addison-Wesley, 1969.

III. The Cost Efficiency-Learning Efficiency Gap in Undergraduate Use of Computers (K. C. Smith)

Historically there are two incommensurable approaches to provision of computation facilities to undergraduate students. For those whose interest is in computer efficiency and overall economy, the tradition has been batch processing. For those more inclined toward student learning efficiency the choice of individual interactive terminals is made. In recent times, each approach has been trimmed and improved to its own end with scant attention paid to the other.

Batch advocates have provided remote batch facilities which bring the peripheral equipment somewhat nearer to the user while maintaining a large and efficient centralized processing facility. Improved compilers, available under batch operation, make the student user's life somewhat less exasperating. No matter how good the system, however, the feedback loop between student and computation process is not well closed.

The advocates of the terminal-oriented approach, on the other hand, have not been idle. Continuing emphasis exists on improved conversational and interactive programming packages. In the combined software and hardware areas, emphasis has been placed on schemes for servicing more and more terminals from one CPU in order that machine efficiency be improved. Hardware emphasis has been on improved low-cost terminals. However, a major cost of conversational terminal systems remains that of the terminal itself and the space required for the relatively large number of terminals needed per user.

To the user, remote batch offers a collection of advantages and disadvantages. Provided the system is sufficiently well equipped and operated, turnaround can be very quick. At the University of Toronto, for example, the time it takes to hand in a deck, to have it read, returned by an operator, and for printout to be removed from the fast printer is usually less than 1 minute. (A 5- to 10-min queue is, of course, often present.) To maintain this level of turnaround, a large competent team of operators is needed. They constitute, perhaps, one of the largest components of the operating cost of such a system. A difficulty is, of course, that periodically something breaks down. Perhaps the most serious disadvantage of remote batch is its reliance on the virtually perfect operation of a rather large amount of equipment.

Conversational systems, of course, offer the ultimate in closed loop learning processes. The problem is that many users are slow learners, hence the terminal time per user corresponds somewhat more to the desk time of a batch user than to his combined keypunch and computer time. The result is apparently the need for a large number of interactive terminals, occupying considerable, though distributed, space. Though the susceptibility of a conversational terminal system to terminal breakdown is very low by virtue of the number of terminals in any system, this system too suffers fatally from central facility failure. There is, however, with the added expense of dialing terminals, a convenient possibility of using another central service for jobs in which a unique resident program or data base is not a part.

At the present state of technology, a third alternative is available. This alternative appears particularly attractive for large schools in which the terminal cost of conversational systems is very large or in which the operation and administrative costs of remote batch systems often dominate. The alternative is the use of many minicomputers in a distributed user-operated batch system.

Individual machines in the system will be free-standing and independent. Each is to be equipped to run a small number of programs of interest to the majority of the students. Each is user operated with a single roving attendant to ascertain the need for service.

Though it is premature to guarantee that such a system is economically feasible, there are several favorable indications.

The majority of batch runs are aborted compilations. (This is really the major weakness of batch.) Fortunately, a current system extremely well suited to the compilation task is the DEC PDP 11. A PDP 11-20 system with 16K, 16-bit words, a 600K removable disk pack, a 200-card/min reader and an 1100 (short) line/min printer is available for about $50 000. This cost is, incidentally, about one third of the annual operating cost at the University of Toronto for one remote batch terminal having roughly the same I/O capability. The state of the art in compiler writing is such that compilers could be written without too great trauma using compiler-compiler techniques on a large machine at the University's computer center and transferred via disk pack to the small machines.

To avoid swapping inefficiency, each machine of such a multiple minicomputer system would be assigned at any one time to be a "Language X" machine where language assignment would relate to the current user demand for various services. In the event that one machine fails, the task assignments of the remaining machines could be quickly rearranged to suit the demand. Meanwhile, the pressure to repair the offending machine is much reduced and high-priced instant-service contracts are avoided. Furthermore, since there are relatively large numbers of any one machine component, the possibility of having complete spares for the worst mechanical offenders is obviously available.

Not least of the advantages to be obtained in the system as outlined is that which accrues to the state of mind of the student user. Such a system may replace the mixed feelings of awe, despair, and disgust which remote batch often engenders, with a sense of intimacy and satisfaction. The popular misconception

that all computing properly must be assigned to a juggernaut might usefully be dispelled.

The hands-on experience a minicomputer system provides should in itself be salutary. It is even possible that one of the machines may be assigned for use by those students interested in the details of the machine which lie beyond the facade which its compiler normally presents.

An alternative approach to the problem of providing adequate yet economical undergraduate computation service has been presented. Exact evaluation of the benefits to be gained from such a solution must await actual operational tests.

IV. Computer Languages, Student Programming, and Interactive Computing (M. L. Dertouzos)

In terms of relevant priorities, I do not believe that "the language that all students should learn" is an important issue. Almost any language will do, provided that the student can implement algorithms in that language and can therefore learn the language-independent craft of algorithm synthesis *first*, and then the mechanics of algorithm implementation. If, however, a choice of language is at issue, then the educational objectives will largely influence the choice. For example, if these objectives place a high degree of emphasis on *immediate* and general applicability to as many computers as possible, then Fortran is clearly the answer. If the objectives call for using computers as powerful (and predominantly numeric) slide rules in an interactive environment, then APL is probably the best currently available language. Finally, if the objectives entail the best choice of language for the (uncertain) future, then some derivative of PL/1 is probably in order. Ultimately, all general-purpose languages offer good educational potential—we have seen the success of mass-interactive computing at Dartmouth with Basic; numerous active scientists and engineers learned computing through Fortran, Mad, Algol, and are probably on their second or third language by now. Indeed, the learning of a second language, besides being relatively easy, is also educationally and practically important. It corresponds, in a very crude sense, to the use of linear or circular slide rules, for the solution of engineering problems.

Looking next at the range of possible programs—namely canned, semicanned (i.e., canned routines which are interconnected by the student), and totally uncanned—I would like to strongly discourage the use of the former in any educational environment. There is something very vague and confusing about a (huge) canned program which miraculously solves a 100-node network! The student can seldom comprehend in any meaningful way how such a program works and is therefore left with the totally unmotivated alternative of using the program to check results. The totally uncanned approach is probably most desirable in a free environment, where there is plenty of time and opportunity to construct nontrivial programs. In the typical overcrowded electrical engineering curriculum of today, the middle way is the best practical alternative—students can still enjoy the motivational and educational benefits of synthesizing something nontrivial with a relatively small effort. These observations are substantiated by several educational experiments conducted at the Department of Electrical Engineering at M.I.T. since the early 1960's.

There is no question in my mind that interactive computing is extremely valuable and essential to education. The ability to work in an on-line environment has advantages which go well beyond the obvious savings in turnaround time—e.g., increased motivation, and the ability to construct programs or solve problems in a closed-loop environment, where the effects of one's actions are immediately visible and the desired goals can be progressively approached.

At M.I.T., we have arrived at the conclusion that one half to 1 hour per week of sit-down time is essential for the student to achieve some sort of educationally meaningful "critical mass." Unfortunately, the cost of on-line services is still considerably higher than that of batch, so that economic factors tend to dominate any such decision.

V. Fortran and the Undergraduate Curriculum (C. Pottle)

During the 1960's, any engineering college that did not already teach its beginning undergraduates elementary programming began to do so. Almost without exception, the language used was Fortran. There were virtually no alternatives. Fortran was universally used outside the university and was the language of all scientific programming. A few disciples of Algol were relegated to the lunatic fringe. The establishment of these courses was often opposed by those who claimed that teaching programming was no different from teaching a course in how to use a slide rule, and had no academic content. In a sense they were correct, for often these courses taught nothing but elementary Fortran programming. Today, however, the situation has changed considerably. An effective first course, as Prof. Dertouzos points out in Section IV has to do with the "craft of algorithm synthesis." Most schools have instituted a first course with this objective, or have one in the final planning stages. This introductory course also introduces the general elements of computer programming and, in particular, a high-level programming language. The practical importance of this course cannot be overemphasized, for it is here that the student will learn (hopefully) good programming habits, the principles of debugging programs, and a language which his school can count on his knowing in later course work. Those schools which claim to have answered the problem by teaching undergraduates a short-term noncredit Fortran programming course in the evening have only succeeded in removing nonacademic material from the curriculum. In the remarks which follow, I assume that we are planning or operating a modern course of the type described above with a purpose which

goes beyond the mere teaching of a programming language for use as a tool in later course work.

What does Fortran have going for it that, after developing a sound introductory computer course, it is chosen as the companion high-level programming language? The reasons seem to be the following.

1) Fortran is still the language used by engineers to solve their problems, even the increasingly nonnumerical problems currently confronting them.

2) Implementations of the Fortran language (e.g., Watfor) are available to handle student jobs efficiently and economically.

3) Fortran is the only language understood by most faculty members.

4) Fortran is still the language of choice for obtaining the most efficient solution to large numerical problems.

5) Many higher level undergraduate and graduate courses use the computer in a mode in which the student interconnects already programmed modules to obtain his results. These modules (usually subroutines) are written in Fortran and are often distributed nationally.

It is my contention that these arguments are no longer valid ones for choosing Fortran as the first language learned by undergraduates. I offer several reasons for my position.

1) Fortran was developed 15 years ago with the characteristics of computer hardware in mind rather than how a scientifically designed language should be defined. Many of the devices in Fortran, such as only two types of variables, fixed and permanent dimensioning of arrays, and lack of character handling ability are restrictive in the sense that general programming concepts are difficult to communicate to students when the companion language is full of exceptions and exclusions. An arithmetic example concerns the fact that Fortran truncates toward 0 when converting from floating to fixed-point representation. The reason is simply that in the 1950's, IBM computers used a sign-plus absolute-value number representation where truncation toward 0 is easy to do. What is usually desired in this conversion, however, is the greatest integer contained in the number. To obtain this result, instead of $K = Y$ one must first write the arithmetic statement function

$$\text{IFLOR}(X) = \text{IFIX}(X - 0.5 + \text{SIGN}(0.5, X))$$

at the head of the program and then use

$$K = \text{IFLOR}(Y).$$

Tricks such as this one have no place in an introductory course.

2) What the state of the art is and has been in industry should not be a factor in making decisions about teaching programming any more than the current methods used to design circuits are used to make decisions about teaching circuit theory.

3) Implementations of other programming languages such as PL/1 are now available which handle many small student jobs as efficiently as their Fortran counterparts. For example, the Cornell College of Engineering maintains and distributes a core-resident compiler which implements a proper subset of PL/1 called PL/C. It is at least as efficient as Watfor.

4) Perhaps the primary difficulty in introducing computers into the undergraduate curriculum in the 1960's was due to opposition of faculty who did not know a programming language. It appears that the same situation has now emerged with respect to faculty learning a second language.

5) It seems clear that for the next few years, most practicing engineers who write production programs will be writing Fortran programs, no matter which language they first learned. The desirability of using canned Fortran subroutines in course work in the upperclass years at a university indicates that many students will write Fortran programs even before they graduate. What I wish to emphasize is the ease with which Fortran can be learned as a *second* language. We need have no qualms about requiring students in our departments to have acquired a working knowledge of Fortran from a short noncredit course before the start of his third or fourth year, if we have a good reason for this requirement. No student with a good background in the algorithmic approach and a high-level programming language is really handicapped by going into the world without Fortran.

None of the preceding remarks have given any hint of what language I would suggest as a replacement for Fortran. The omission has been deliberate, for I am far more interested in seeing Fortran replaced in the first course than I am in dictating what its replacement will be. A particular college's own specific situation will probably be the major factor. Two very different programming languages, either of which would be excellent replacements, are APL and PL/1.

1) APL is an interactive language. Any reader who is not familiar at all with APL is urged to become at least minimally informed. It cannot be learned efficiently without terminal sessions where the student is allowed considerable freedom. APL is beautifully designed to implement programming concepts the way they are taught. Its lack of input–output features and of interfacing possibilities with other systems is of no consequence in a first course. For generating enthusiasm in a student for computing, APL is without compare. Perhaps the factor which will eliminate APL from consideration is the calculation of the costs involved in supplying 500 freshmen each semester with 4 hours of terminal time each per week. The cost of about $4.00 per terminal hour for APL (based on estimates gathered at the Purdue Symposium) is lower than might be expected, however, and will continue to drop. The APL missionaries whose papers appear elsewhere in this special issue should be listened to with attention. It is already true that a hoard of APL functions for use in electrical engineering courses have been developed and are available for use in the "canned subroutine" approach.

2) PL/1 is primarily a batch processing language of

extraordinary richness. Only a small subset of the language would be taught in an introductory course. Efficient compilers for student use are becoming available. The experience reported by several institutions that have used PL/1 as their programming language is entirely favorable. The basics of nonnumerical programming which are becoming more important to engineering are handled with ease by PL/1. Many students who enter college with Fortran experience are delighted to find a language which has removed many of the incomprehensible constraints appearing in the Fortran language. It seems clear that no more time is required to teach PL/1 than Fortran. This time is better spent, moreover, since it can be used to show how general concepts of programming languages are implemented in a particular language.

APL: A Natural Language for Engineering Education?

PART I (GARTH H. FOSTER, MEMBER, IEEE)

I have found you an argument; but I am not obliged to find you an understanding.

—Samuel Johnson

I. INTRODUCTION

THIS paper represents an effort to describe the effect that some electrical engineering educators feel APL will have upon undergraduate and graduate education in that field. Accordingly, the sections of this paper deal with the following areas. Section II attempts to succinctly convey the flavor of APL and to summarize its features. Section III discusses the strengths and weaknesses of current implementations and the general application of APL to engineering education. Section IV gives a brief introduction of each of the panelists and contains the position papers of each participant.

II. APL: AN OVERVIEW[1]

"What is one and one and one and one and one and one and one and one and one and one?"

"I don't know," said Alice, "I lost count."

"She can't do addition," said the Red Queen.

—Lewis Carroll

```
+\10⍴1
```

—APL

APL, standing for *A Programming Language*, derives from the title of the book by the same name, written by K. E. Iverson and published by Wiley in 1962. That volume, modest in size but not in scope, was the formalization of an algorithmic notation which K. Iverson first began to set down in the late 1950's while at Harvard. Further refinements of the language after Iverson joined IBM surfaced when APL was used to describe the IBM System 360 in 1964. Initially, the lack of implementation tended to discount the utility of APL; first steps to remedy that state of affairs were taken when L. M. Breed and P. S. Abrams at Stanford University implemented an interpreter for APL. Subsequently, Breed joined Iverson and A. D. Falkoff at IBM to collaborate on the design and to implement APL on the IBM System 360. APL\360 was available internally to IBM and a few researchers on the outside in late 1966; the implementation was obtainable in Canada in 1967 and the U. S. by the fall of 1968. From 1967 to the present a number of independent implementations of APL have been undertaken so that APL may now be found (or will shortly be obtainable) on the IBM 1130 and 1500, XDS Sigma 7 (at least two separate implementations), CDC 3600, Burroughs B5500, CDC 6600 and 7600, Honeywell (GE)635, Univac 1108, Burroughs 700 line, and possibly DEC's PDP10 and RCA Spectra 70.

Many of the implementations are being done by equipment users rather than by vendors of hardware and/or software, but the number and the range of machines are indicative of the growing interest in APL.

Most of the implementations have followed the directions laid out by APL\360 in that the implementation is a time-shared interpreter often using the IBM 2741 "Selectric" typewriter terminal or its equivalent. Major variations may exist in the system supervisor and its command structure or in the type and range of I/O devices which a particular implementation may support. The following discussion refers to APL\360 but the statements are, in general, applicable to other implementations.

An APL terminal system provides for the managing, saving, and merging collections of user-defined functions and data into workspaces, and collections of workspaces into libraries, and, in general, makes the language available to a user at a terminal.

To the user the input interface to the APL terminal system appears to be the keyboard shown in Fig. 1 and the output is the terminal printer which is for most installations an IBM 2741 with the Selectric I/O Writer; the golfball normally has the same character

Manuscript received February 22, 1971; revised April 28, 1971.
The author is with the Department of Electrical and Computing Engineering, Syracuse University, Syracuse, N.Y. 13210.

[1] This overview addresses the form and content of the language and not its programming or application. The latter topics are treated more fully in the section of the paper, "APL—A Natural Language for Engineering Education," by W. R. LePage.