At this session it seems to me that you might be interested in several of the more-or-less technical facets of the direction of a large aerospace computer installation. Consequently I will avoid competing with our environment by discussing the ubiquitous problems of recruiting, of personnel motivation, of obtaining cooperation among the members of the various computing groups, or even the basic problems inherent in convincing our computer folks that the whole computer center does not exist for them at all, but rather as a service for the other parts of our company.

Instead, I want to tell you today about a few of the figures we have on the actual costs of "Change"; then go into a few aspects of the "Turn-Around-Problem" from the management point of view; and finish with a few remarks on what a computer center such as ours may reasonably expect in the future.

While there are many fields in which constant change is the order of the day, the operation of virtually any modern computer center is faced with adjustment to changing computers, changing computer languages, and changing software systems with a frequency which is quite notable. In a recent attempt to analyze the economic effects of such changes, several interesting relationships have been noted.

In the past, the speed with which a newly installed computer has been "loaded" has often been of interest, but few figures have appeared which treat as a dynamic quantity the relationship between the program checkout load and the production load. Such relationships must be known, however, before one can evaluate the effects of change, since the purely static "before and after" pictures tend to conceal many of the significant points.

In analyzing the dynamics of the situation, some simple relationships have been postulated, and their predictions compared with those historical data which were obtainable at the LMSC computation center.

First, it was assumed that the loading on a computer could be divided between check-out and production, and that the ratio of these two would vary with time from installation. Since the proportion of computer programs which run for production without previous check-out is vanishingly small, it would seem reasonable that the load on a newly installed computer initially must consist solely of program development work. From such a starting point it follows that the ratio of production to development will show a continuous increase until it either levels off with time, or the pattern becomes confused by changes in language, operating system, or type of work processed by the center.
Both the ratio of production to development at steady state and the rate at which this ratio approaches steady state must be determined in order to understand and to evaluate the economics involved.

Historical data obtained subsequent to the installation of two types of computers, the IBM 709's which replaced Univac 1103's and the IBM 1410's which were installed to handle administrative systems, shows the patterns given in Figs. 1 and 2 respectively.

It can be seen that the data in both cases have a basic similarity, and that the experience in general seems to follow the same type of curve. The smoothed curves themselves were obtained by assuming that the development load would be reduced half-way to the steady state load during each four month period.

Data on the introduction of new computer languages have been somewhat more difficult to obtain, and tend to be less definitive. Hardware changes are necessarily abrupt. Software changes need not be. However, records have been found which show the ratio of production to development following a fairly general shift from Fortran I to Fortran II on the IBM 7090's which started in June of 1963. Since the scatter of these data is considerably greater than it was for the introduction of a computer itself, and since figures are not available for rework as a separate item, smooth curves were not derived from the data. Instead, the curves from Fig. 1, the introduction of the 709's, were plotted.

Figure 1. Load components versus time from installation of 709.
Figure 2. Load components versus time from installation of 1410.

directly in Fig. 3. It would seem that the data at least do not disagree with this pattern.

During the ten-year period from mid-1955 to July 1965 LMSC has employed six distinctly different computers to handle important parts of its work (IBM 650, 709/7090/7094, 1410; Univac 1103, 1107/1108; and RCA 301), and six additional computers either as peripheral units or to serve special purposes (IBM 1401, RPC 4000, CDC 160A, 924, 3200 and GE 415).

During the same ten-year span, in addition to the machine languages of these twelve computers, more than twelve distinct versions of compiler languages or general operating systems have been put into operation.

Each innovation brought improvement. It also brought a new set of problems. With each improvement some portion of the total number of computer users at LMSC has been required to familiarize itself with either a new computer, new computer language, or new system some 24 times in these 10 years.

Since some of these changes affected more than two-thirds of the work load, while others touched only a few percent, it would appear reasonable to assume that on the average each change affected one-sixth of the work load. From this figure it follows, again on an average basis, that there must have been the equivalent of a complete change some four times in ten years. That is not to say that large disruptions can be noted at that frequency, but only that they are actually present in the average figures to that extent. The fact that truly disruptive changes do not show in the records results from several fac-
tors. First, in an operation as large as the one being studied, there are usually several large computers of the same type, which can be phased into or out of the operation over a long enough period of time to give a nearly smooth operation in the aggregate. Second, the introduction of a new language can also be phased in rather smoothly by involving only a few programs at a time. As you all probably realize, phasing a language out is an even more gradual process than introducing it. It may not be completely impossible, but it is almost impossibly difficult.

Nevertheless, the net effect of changes must have the same cumulative effect as would a complete change every two and one-half years. The following analysis is based upon that frequency.

Figures 1, 2, and 3, together with the mean change rate of 0.4 changes per year, provide the data required to calculate the average annual cost of change.

Subtracting the asymptotic values of the curves on these figures gives a value of \( (98 - 30) \) = 68 percent for the steady state, which must be contrasted with the integrated value for production, which has an average value of 59.7 percent. Since the production time on a computer is the only time which is of actual value to the user, we may compare these figures directly with the result showing that the steady state condition would provide \( (68 - 56.7)/56.7 \) or 20 percent more useful time than that obtained with the change frequency prevailing.

In addition to the information on machine time which these curves give, they also show us something about the way in which our programming manhours are spent. Specifically, they show that the
amount of development which must be directly attributed to this change frequency is the difference between the integrated mean value, 38.8 percent, and the asymptotic value, 30 percent. These data provide the estimate that the gross programming cost of the change frequency is \((38.8 - 30)/30\) or 29.3 percent of the total programming costs.

At this point, then, we can state that a proposed change must promise a gross cost reduction equaling the total of 20 percent of current machine cost plus 30 percent of current programming costs in order to break even.

Now let us take a moment to look at the “Turn-Around-Time.”

Some of you will remember the queues that formed and the average times that prevailed when a simple diagram was used to show the path of an individual job. It was something like the following:

<table>
<thead>
<tr>
<th>JOB</th>
<th>DISPATCH</th>
<th>SET-UP</th>
<th>Q</th>
<th>CTT</th>
<th>Q</th>
<th>CPU</th>
<th>Q</th>
<th>TTC</th>
<th>Q</th>
<th>DISPATCH</th>
<th>TTP</th>
<th>HOURS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESK</td>
<td>0.05</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
<td>0.05</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
</tr>
<tr>
<td>TABLE</td>
<td>1.00</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
<td>0.10</td>
<td>1.00</td>
</tr>
</tbody>
</table>

This diagram purports to show that while CPU time for the average job may be 6 minutes, and the peripheral equipment time another 12 minutes, the five queues which form may add 5 hours. In the past, we have occasionally been guilty of jumping to the conclusion that by combining operations and eliminating 4 of the 5 queues, we could cut the queue length from 5 hours to one. As many of you realize, in practice it only works that way as long as there is an excess of machine capacity over work load. As in most shops, the various items of equipment are already fairly well-balanced to each other before the figures above could be obtained, rather like a series of five synchronized traffic signals.

Actually, of course, the basic problem lies not in balancing the equipment (even though that must be done) nor in eliminating all but one queuing point (the one remaining could be nearly as long as the sum of those replaced), but in balancing the equipment to the job load.

Fortunately, or perhaps unfortunately, I’m not sure, we have one rather automatic balancer inherent in our system, to the effect that whenever the turn-around-time goes up, the rate at which we receive jobs goes down. The mechanism behind this effect is obvious when one considers that a programmer can’t remove a bug and resubmit a developing program if he never gets it back.

The balancing of equipment work load is thus the real problem, which management must solve on the basis of the most economical operation in the broadest sense.

While no one has yet come up with an accurate estimate of the cost of delay in turn-around, we can start with a highly over-simplified picture, making use of Erlang’s basic queuing equations:

\[
D = \frac{LP}{(c-a)} \\
\text{and } P = \frac{ae}{(c-1)! (c-a)}
\]

\[
\left[ 1 + \frac{a}{1!} + \frac{a^2}{2!} + \cdots + \frac{a^c}{(c-1)! (c-a)} \right]^{-1}
\]

to obtain the average delay \(D\) as a function of mean job length \(L\), the mean number of jobs received \(a\) in the time interval \(L\), and the number of computers \(c\).

From this equation, and assuming that computer time costs $35,000 per week and that the cost of an hour’s delay is 4 percent of the cost of an hour’s chine time, Fig. 4 can be obtained.

There are several features worth noting in this chart. If one looks at the cost of doing a job when only one computer is involved, it is apparent that the cost of turn-around delay is not very significant with respect to the economic advantage to be gained by loading the single machine as heavily as is feasible. Even when we allow a large economic factor for the hourly cost of delay, as we have here, examination of the first cycle of Fig. 4 shows why the programmer is seldom satisfied with the turn-around-time in an efficiently operated one machine shop.

On the other hand, when the job load has gone
up to the point where four or five machines are required, the relative importance has considerably shifted.

Examining the fourth or fifth cycles, we note that the generalizations based on the first cycle no longer apply. In fact, the greatest danger of excessive cost at this point lies in hitting an over-load condition, rather than an under-load. Further, the difference between the cost per job with optimum loading of four machines, and with the lowest loading of five machines, is less than 7 percent.

While Fig. 4 illustrates the point I wish to make, the simplifications involved are much too gross for efficient operation, so that we have found that a rather elaborate computer program is really required. Such a program, called LOMUSS (Lockheed Multiprocessor Simulation System) has been developed, and will be reported upon separately.

Now I would like to take a few minutes more to make some observations on the evolutionary course that we might expect the next few years to follow.

A few minutes ago we noted that over the last decade there had been a large number of changes in our computers, our computer languages, and our operating systems. One's first reaction might be that this is inherently bad (and of course it is) and that to avoid a continuous repetition we should standardize upon a single hardware configuration supported by a single language and operating system. However, if we plot the pertinent data from Dr. Knight's intensive study of the first 225 digital computers, we obtain Fig. 5, from which it is obvious that the computer field has not yet reached the point at which we can afford to standardize on hardware.

On the software side the situation does not appear to be as obvious as Fig. 5 showed it to be for hardware. I assume that you will all agree that, despite the advances made, from machine language through assemblers, Macro-assemblers, procedure-oriented compilers and problem-oriented languages, the economic advances which have been made are still unimpressive when compared with hardware improvements shown in Fig. 5.
Consequently, I have the feeling, just as many of you have, that despite the advances yet to be realized in hardware, it is in the software area that the greater improvements should be sought.

Rather than expecting revolutionary advances, such as software which merely accepts a few sets of test data and automatically generates the algorithm required, it would seem more reasonable to expect that progress in software will be evolutionary.

If we look at the way that man communicates we see that, while he uses a native language as his basic tool, most of his working time is spent using some special jargon imbedded in the basic language. This is true whether he is a physician, a chemist or a truck driver. In order to explain something to another physician, chemist or truck driver, he uses the jargon of his trade in order to save the
large amounts of time and effort which would be required to communicate in the basic language.

This point is easily illustrated by the story of the gangsters who had taken over a legitimate theater. They didn't realize that when a stagehand yelled "Take the silks off the broads," he meant to remove the light diffusers from a certain type of stage light.

While this idea of a jargon imbedded in a basic language is neither revolutionary nor glamorous, it does seem to be the way that software might progress. Individual jargons could gradually be developed, in the form of special-purpose libraries of functions, subroutines, procedures or what you will, to serve each of the definable areas of need. At LMSC we might expect a dozen or so, in areas such as thermodynamics, trajectory work, real-time control, accounting, medical support, etc.

Before we can develop these various jargons,

Figure 6. Past and potential percent production versus time.
however, we must have a basic language in which to imbed them; for remember, these jargons are not to be distinct languages, each requiring a new compiler and operating system.

Further, such a basic language must itself have certain characteristics, such as ease of use, efficiency, open-endedness, complete machine independence, perhaps even manufacturer independence, and ease of implementation for new computers.

Given such a basic language, and the development of the needed jargons, then we might expect that the composite of Fig. 1, 2, and 3 would more nearly approach Fig. 6, where the economy in hardware costs shown by the increased ratio of production to development load is also a reflection of increased programming efficiency. Further, since there would be no reprogramming, all of the programming time would be devoted to new programs.

REFERENCES

