A General Empirical Solution to the Macro Software Sizing and Estimating Problem

LAWRENCE H. PUTNAM

Abstract—Application software development has been an area of organizational effort that has not been amenable to the normal managerial and cost controls. Instances of actual costs of several times the initial budgeted cost, and a time to initial operational capability sometimes twice as long as planned are more often the case than not.

A macromethodology to support management needs has now been developed that will produce accurate estimates of manpower, costs, and times to reach critical milestones of software projects. There are four parameters in the basic system and these are in terms managers are comfortable working with—effort, development time, elapsed time, and a state-of-technology parameter.

The system provides managers sufficient information to assess the financial risk and investment value of a new software development project before it is undertaken and provides techniques to update estimates from the actual data stream once the project is underway. Using the technique developed in the paper, adequate analysis for decisions can be made in an hour or two using only a few quick reference tables and a scientific pocket calculator.

Index Terms—Application software estimating, quantitative software life-cycle management, sizing and scheduling large scale software projects, software life-cycle costing, software sizing.

I. Introduction

Current State of Knowledge

THE earliest efforts at software cost estimation arose from the standard industrial practice of measuring average productivity rates for workers. Then an estimate of the total job was made—usually in machine language instructions. Machine language instructions were used because this was the way machines were coded in the early years and because it also related to memory capacity which was a severe constraint with early machines. The total product estimate was then usually divided by the budgeted manpower to determine a time to completion. If this was unsatisfactory, the average manpower level (and budget) was increased until the time to do the job met the contract delivery date. The usual assumption was that the work to be done was a simple product—constant productivity rate multiplied by scheduled time—and that these terms could be manipulated at the discretion of managers. Brooks showed in his book, *The Mythical Man-Month* [1] that this is not so—that manpower and time are not interchangeable, that productivity rates are highly variable, and that there is no nice, industry standard that can be modified slightly to give acceptable results for a specific job or software house. This was not serious for small programs built by one person or a small group of persons. But when system programming products of hundreds of programs, hundreds of thousands of lines of code, built by multiple teams with several layers of management arose, it became apparent that severe departures from constant productivity rates were present and that the productivity rate was some function of the system complexity.

Morin [2] has studied many of the methods which have been tried for software cost estimating. She found that most of these methods involved trying to relate system attributes and processing environmental factors to the people effort, project duration, and development cost. The most generally used procedure was multiple regression analysis using a substantial number of independent variables. Morin concluded, "... I have failed to uncover an accurate and reliable method which allow programming managers to solve easily the problems inherent in predicting programming resource requirements. The methods I have reviewed contain several flaws—chief among them is the tendency to magnify errors ... ."

Morin recommended that "... researchers should apply non linear models of data interpretation ... as Pietrasanta [31] states, the use of non linear methods may not produce simple, quick-to-apply formulas for estimating resources, but estimation of computer resources is not a simple problem of linear cause-effect. It is a complex function of multiple-variable interdependence."

"... The most important objective of estimation research should be the production of accurate estimating equations. At this time [1973], the application of non linear methods seems to hold the greatest promise for achieving this objective."

Gehring and Pooch [3] confirm Morin's findings. They state, "There are few professional tools or guidelines which are accepted as accurate predictions or good estimates for various phases of software development." Gehring and Pooch make the important observation that the basic resources in software development are manpower, machine-time, money, and elapsed time and that these resources are interdependent.

I call these resources the management parameters. If they are indeed interrelated in a functional way and if they can in turn be functionally related to software system parameters, then we should be able to produce a mathematical model that will be useful in predicting and controlling the software life-cycle process. Bellman [28] makes the point that by developing mathematical models of a process, we avoid the task of

Manuscript received June 15, 1977; revised December 15, 1977 and March 6, 1978.

The author is with the Space Division, Information Systems Programs, General Electric Company, Arlington, VA 22202.

0098-5589/78/0700-0345$00.75 © 1978 IEEE

trying to store all possible information about the process and instead let the model generate the data we need.

Gehring [4], [5] feels existing knowledge and the state of the art is too meager to do this with software. Morin [2] suggests it should be tried with nonlinear (interpreted to mean curvilinear) models.

I take the position that Morin's approach will work; that reasonable models can be built that reproduce what actually happens within the limits of the data measurement, recording, and interpretation.

A few comments about data are appropriate. The human effort on a project is generally recorded according to accounting rules and standards established by the organization. Each organization establishes its rules according to its own needs. Consistency from organization to organization is uncommon. Recorded data does not come from an instrument (like a voltmeter) whose accuracy is known. Often manpower is aggregated or counted at finite intervals (weekly, say) perhaps from recollection rather than precise notes made at the time. Not all effort recorded against the job is actually done on the job. Typically only 4 to 5 h of an 8 h day are applied to the project. Administration and other needs of the organization absorb the rest. It is rare that this administrative and lost time is charged as such—usually it is billed to the project. This means manpower data are imprecise. Dollar cost data are worse since even more categories of expense get subsumed into the numbers. Generally, I have found that manpower data accumulated to a yearly value is not more accurate than ±10-15 percent of the reported value. If the data are examined at shorter intervals, the percentage variation tends to be even greater. This is why efforts to estimate and control using a bottom-up, many finite elements approach have always failed so far. The process becomes overwhelmed by the imprecision engendered by the "noise." This comes as a blessing in disguise, however, because if we treat the process macroscopically and use the concepts of expected values and statistical uncertainty we can extract most of the useful and predictive value and at the same time know something about the uncertainty or the magnitude of the "noise," which I will assert is ever present. In other words, software development behaves like a time dependent random process. The data reflect this and are not intractable when so considered.

The time-varying character of software development is also highly important. Most practitioners treat the problem statically even though they implicitly recognize time dependence by the phases into which they divide projects. In fact, time is the independent variable in software projects. The dependent variables (manpower, code production) are usually time rates. The software life cycle is dynamic, not static.

Some investigators have noted this in their work, most notably Aron [6], Norden [7], [8], Felix and Walston [14], Daly [15], Stephenson [16], Doty Associates [17], and this author [9]-[13].

What Is Not Known

The phenomenology of the software development (and maintenance) process is not known. The data suggest a fairly clear time-varying pattern. This seems to be of a Rayleigh or similar form. There is considerable "noise" or stochastic components present which complicate the analysis. Further, the observables (manpower, cost, time) are strongly subject to management perturbation. This means that even if a system has a characteristic life-cycle behavior, if that behavior is not known to managers, *a priori*, then they will respond reactively (nonoptimally with time lags) to system demands. Superimposed on this is the ever-present problem of imprecise and continually changing system requirements and specifications.

Customer Perspective at Start of Project

We can get an appreciation of what information we really need to know to plan and control large scale software efforts by looking at the problem the way the customer might at about the time he would ask for proposals. At this time certain things are known:

1) a set of general functional requirements the system is supposed to perform;
2) a *desired* schedule;
3) a *desired* cost.

Certain things are unknown or very "fuzzy," i.e.,

1) size of the system;
2) *feasible* schedule;
3) minimum manpower and cost consistent with a feasible schedule.

Assuming that technical feasibility has been established, the customer really wants to know:

1) product size—±a "reasonable" percentage variation;
2) a "do-able" schedule—±a "reasonable" percentage variation;
3) the manpower and dollar cost for development—±a "reasonable" variation;
4) projection of the software modification and maintenance cost during the operational life of the system.

During development both the customer and the developer want to measure progress using the actual project data. These data should be used to refine or modify earlier estimates in order to control the process.

Traditional Estimating Approach

In the past, estimating the size, development time, and cost of software projects has largely been an intuitive process in which most estimators attempt to guess the number of modules, and the number of statements per module to arrive at a total statement count. Then, using some empirically determined cost per statement relationships, they arrive at a total cost for the software development project. Thus the traditional approach is essentially static. While this approach has been relatively effective for small projects of say less than 6 months duration and less than 10 man-years (MY) of effort, it starts to break down with larger projects and becomes totally ineffective for large projects (2 years development time, 50 man-years of development effort or greater, and a hierarchical management structure of several layers).

Dynamic Life-Cycle Model Approach

The approach we shall use will be more along the line of the experimentalist than the theoretician. We will examine the

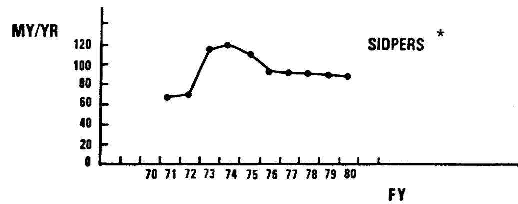


Fig. 1. Manpower loading as a function of time for the Army's Standard Installation-Division Personnel System (SIDPERS).

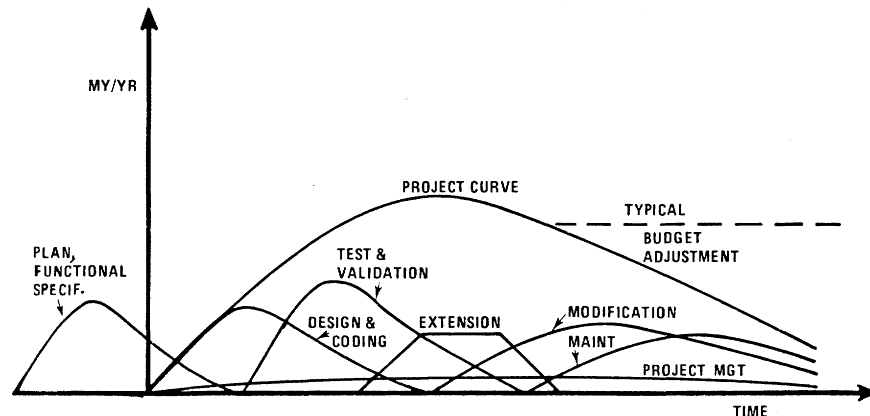


Fig. 2. Typical Computer Systems Command applicator of manpower to a software development project. Ordinates of the individual cycles are added to obtain the project life-cycle effort at various points in time.

data and attempt to determine the functional behavior by empirical analysis. From those results we will attempt to arrive at practical engineering uses.

It has been well established from a large body of empirical evidence that software projects follow a life-cycle pattern described by Norden [7], [8]. This life-cycle pattern happens to follow the distribution formulated by Lord Rayleigh to describe other phenomena. Norden used the model to describe the quantitative behavior of the various cycles of R&D projects each of which had a homogenous character. Accordingly, it is appropriate to call the model the Norden/Rayleigh model.

The paper will show that software systems follow a life-cycle pattern. These systems are well described by the Norden/Rayleigh manpower equation, $\dot{y} = 2Kate^{-at^2}$, and its related forms. The system has two fundamental parameters: the life-cycle effort (K), the development time (t_d), and a function of these, the difficulty (K/t_d^2). Systems tend to fall on a line normal to a constant difficulty gradient. The magnitude of the difficulty gradient will depend on the inherent entropy of the system, where entropy is used in the thermodynamic sense to connote a measure of disorderedness. New systems of a given size that interact with other systems will have the greatest entropy and take longer to develop. Rebuilds of old systems, or composites where large portions of the logic and code have already been developed (and hence have reduced the entropy) will take the least time to develop. New stand-alone systems will fall in between. Other combinations are also possible and will have their own characteristic constant gradient line.

These concepts boil down to very practical engineering or businessman's answers to most of the manpower, investment, and time questions that management is concerned with [13]. Since elapsed time is ever present in the model as the independent variable, this approach is dynamic and hence will produce numerical answers at any desired instant in time.

II. Empirical Evidence of a Characteristic Life-Cycle Behavior

It will be instructive to examine the data that led to the formulation of the life-cycle model. The data are aggregate manpower as a function of time devoted to the development and maintenance of large-scale business type applications. Fig. 1 shows a typical budget portrayal from U.S.A. Computer Systems Command.[1] The timeframe of this display is fiscal year (FY) 1975. Note the straight-line *projection* into the years FY'76 and beyond.

Norden [7], [8] found that R&D projects are composed of cycles—planning, design, modeling, release, and product support.

Norden [8] linked cycles to get a project profile. Fig. 2 shows the comparable software cycles laid out in their proper time relationship. When the individual cycles are added to-

[1] U.S.A. Computer Systems Command is the U.S. Army central design agency for Standard Army Management Information Systems that run at each Army installation. It employs about 1700 people. It designs, tests, installs, and maintains application software in the logistic, personnel, financial, force accounting, and facilities engineering areas. Systems range in size from 30 MY of development and maintenance effort to well in excess of 1000 MY.

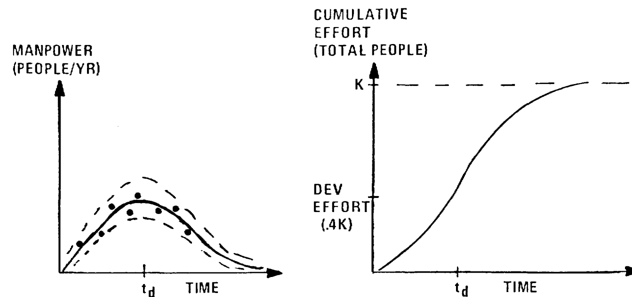


Fig. 3. Expected manpower behavior of a software system as a function of time.

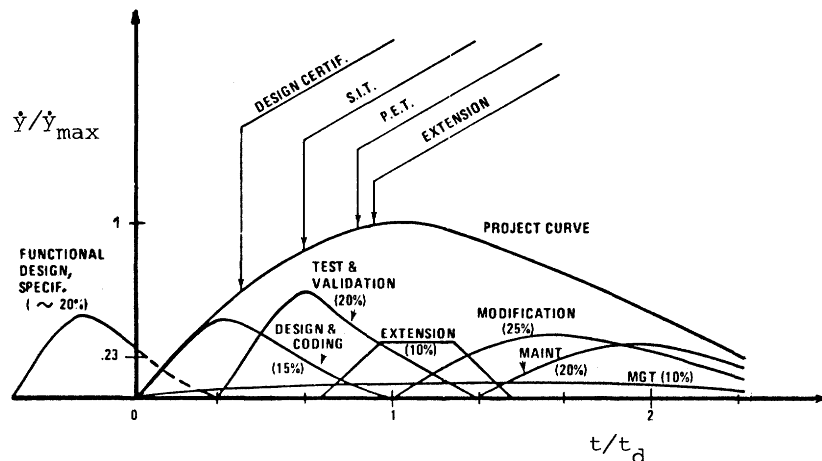


Fig. 4. Large scale software application system life cycle with subcycles and empirically determined milestones added.

gether, they produce the profile of the entire project. This is labeled Project Curve on the figure.

When the model was tested against the man-year budgetary data for about 50 systems of the Computer Systems Command, it was discovered that Computer Systems Command projects followed this life-cycle model remarkably well.

Data from about 150 other systems from other operations have been examined over the past two years [18]-[27]. Many of these also exhibit the same basic manpower pattern—a rise, peaking, and exponential tail off as a function of time. Not all systems follow this pattern. Some manpower patterns are nearly rectangular; that is, a step increase to peak effort and a nearly steady effort thereafter. There is a reason for these differences. It is because manpower is applied and controlled by management. Management may choose to apply it in a manner that is suboptimal or contrary to system requirements. Usually management adapts to the system signals, but generally responds late because the signal is not clear instantaneous with the need.

Noise in the data is present for a variety of reasons. For example, inadequate or imprecise specifications, changes to requirements, imperfect communication within the human chain, and lack of understanding by management of how the system behaves.

Accordingly, we observe the data, determine the expected (average) behavior over time, and note the statistical fluctuations which tell us something about the random or stochastic aspects of the process. Conceptually the process looks like the representation shown in Fig. 3.

The data points are shown on the manpower figure to indicate that there is scatter or "noise" involved in the process. Empirical evidence suggests that the "noise" component may be up to ±25 percent of the expected manpower value during the rising part of the manpower curve which corresponds to the development effort. t_d denotes the time of peak effort and is very close to the development time for the system. The falling part of the manpower curve corresponds to the operations and maintenance phase of the system life-cycle. The principal work during this phase is modification, minor enhancement, and remedial repair (fixing "bugs").

Fig. 4 shows the life-cycle with its principal component cycles and primary milestones. Note that all the subcycles (except extension) have continuously varying rates and have long tails indicating that the final 10 percent of each phase of effort takes a relatively long time to complete.

III. Basic Characteristics of the Norden/Rayleigh Model as Formulated by Norden [7], [8]

It has been empirically determined that the overall life-cycle manpower curve can be well represented by curves of the

Norden/Rayleigh form:

$$\dot{y} = 2Kate^{-at^2} \quad \text{MY/YR} \tag{1}$$

where $a = (1/2t_d^2)$, t_d is the time at which $\dot{y}$ is a maximum, K is the area under the curve from $t = 0$ to infinity and represents the nominal life-cycle effort in man-years.

The definite integral of (1) is

$$y = K(1 - e^{-at^2}) \quad \text{MY} \tag{2}$$

and this is the cumulative number of people used by the system at any time t.

Fig. 5 shows both the integral and derivative form of the normalized ($K = 1$) Norden/Rayleigh equation. When K is given a value in terms of effort these become effort and manloading curves, respectively. Because each of the parameters K and a can take on a range of values, the Norden/Rayleigh equation can represent a wide variety of shapes and magnitudes. Fig. 6 shows the effect of first varying the shape parameter $a = (1/2t_d^2)$, holding the magnitude K constant, and then varying the magnitude K holding the shape parameter a constant.

Now we wish to examine the parameters K and t_d.

K is the area under the $\dot{y}$ curve. It is the total man-years of effort used by the project over its life-cycle.

t_d is the time the curve reaches a maximum. Empirically this is very close to the time a system becomes operational and it will be assumed hereafter than $t_{\dot{y}_{\max}} = t_d$ = development time for a system.

Most software cost is directly proportional to people cost. Thus, the life-cycle ($LC) cost of a system is

$$\$LC = \overline{\$COST/MY} \cdot K. \tag{3}$$

We neglect the cost of computer test time, inflation overtime, etc., all of which can be easily handled by simple extensions of the basic ideas. The development cost is simply the $\overline{\$COST/MY}$ times the area under the $\dot{y}$ curve from 0 to t_d, and the bar over $COST/MY indicates the average cost/MY.

That is,

$$\$DEV = \overline{\$COST/MY} \int_0^{t_d} \dot{y} \cdot dt$$

$$= \overline{\$COST/MY} \cdot (0.3945\,K)$$

$$\$DEV = 40 \text{ percent } \$LC. \tag{4}$$

These are the management parameters of the system—the people, time, and dollars. Since they are related as functions of time, we have cumulative people and cost as well as yearly (or instantaneous) people and cost at any point during the life-cycle.

In the real world where requirements are never firmly fixed and changes to specifications are occurring at least to some extent, the parameters K and t_d are not completely fixed or deterministic. There is "noise" introduced into the system by the continuous human interaction. Thus, the system has random or stochastic components superimposed on the deterministic behavior. So, we are really dealing with expected values for $\dot{y}$, y, and the cost functions with some "noise" superimposed.

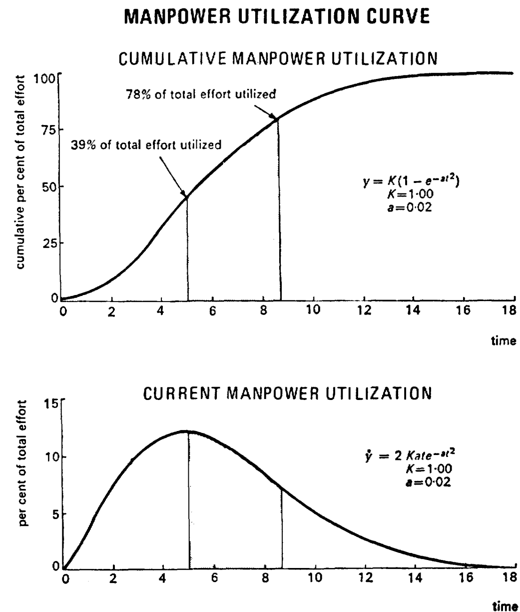


Fig. 5. Norden/Rayleigh model [7].

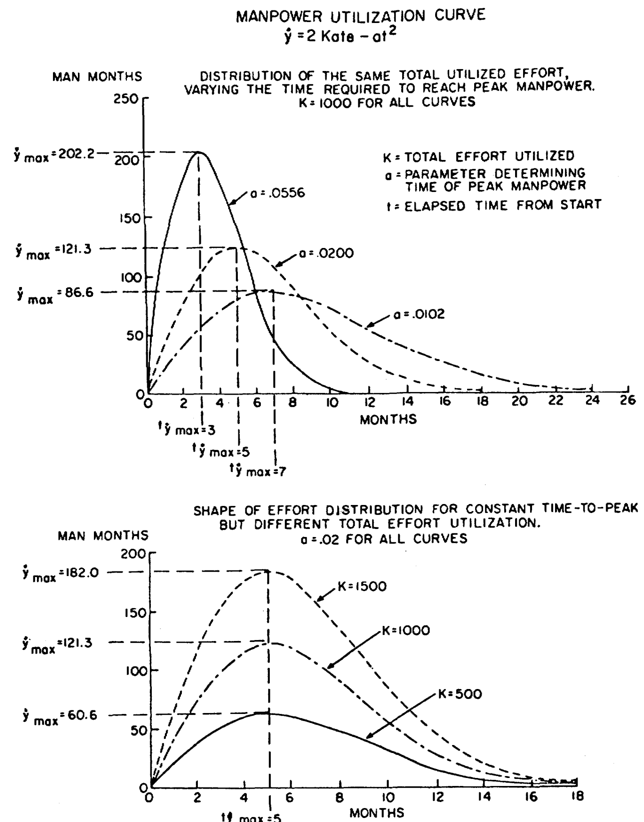


Fig. 6. Life-cycle curve: effect of changing parameter values [8].

Fig. 3 illustrates this—the random character will not be pursued further here except to say it has important and practical uses in the risk analysis associated with initial operational capability and development cost.

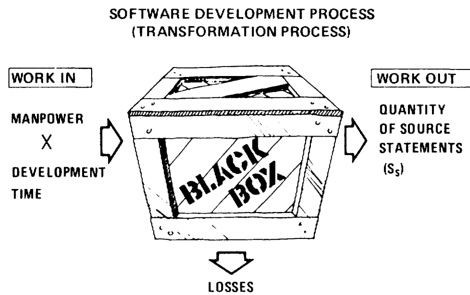


Fig. 7. Software development as a transformation.

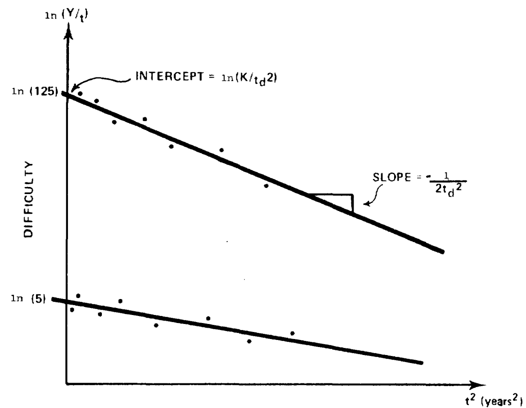


Fig. 8. Difficulty versus time^2 [29].

IV. Conceptual Description of the Software Process

Let us now shift our attention from the empirical observations to some conceptual underpinnings that will shed more light on the software development process. We will look at the Norden/Rayleigh model, its parameters and dimensions and how it appears to relate to other similar processes.

In particular we will consider the management parameters K and t_d so that we may study the impact of them on the system behavior.

Fig. 7 shows the development process as a transformation. The output quantity of source statements is highly variable from project to project. This implies that there are losses in the process.

Returning to the Norden/Rayleigh model, we have noted that in mathematical terms, K is the total life-cycle man-years of the process and a is the shape parameter. But a is more fundamental in a physical sense.

$a = (1/2t_d^2)$, where t_d is the time to reach peak effort. In terms of software projects, t_d has been empirically shown to correspond very closely to the development time (or the time to reach full operational capability) of a large software project. We can write the Rayleigh equation with the parameter t_d shown explicitly by substituting the following for a:

$$\dot{y} = K/t_d^2 \, te^{-t^2/2t_d^2}. \tag{5}$$

Now it is instructive to examine the dimensions of the parameters.

K is total work done on the system.

t_d and t are time units.

$(K/t_d^2) \times t$ has the dimensions of power, i.e., the time rate of doing work on the system, or manpower. But the ratio K/t_d^2 appears to have more fundamental significance. Its dimensions are those of force.

When the Rayleigh equation is linearized by taking logarithms (suggested by Box and Pallesen [29]) we obtain

$$\ln(\dot{y}/t) = \ln(K/t_d^2) + \left(\frac{-1}{2t_d^2}\right)t^2. \tag{6}$$

When the natural logarithm of the manpower divided by the corresponding elapsed time is plotted against elapsed time squared (Fig. 8), a straight line is produced in which $\ln(K/t_d^2)$ is the intercept and $-1/2t_d^2$ is the slope.

When this was done for some 100 odd systems, it was found that the argument of the intercept K/t_d^2 had a most interesting property. If the number K/t_d^2 was small, it corresponded with easy systems; if the number was large, it corresponded with hard systems and appeared to fall in a range between these extremes. This suggested that the ratio K/t_d^2 represented the difficulty of a system in terms of the programming effort and the time to produce it.

When this difficulty ratio (K/t_d^2) is plotted against the productivity $(\overline{PR})$ for a number of systems we get the relationships shown in Fig. 9.

This relationship is the missing link in the software process. This can be illustrated by removing the cover from the black box, as shown in Fig. 10.

Feasible Effort–Time Region

A feasible software development region can be established intuitively. Systems range in effort from 1 to 10 000 MY life-cycle. Development times (t_d) range from 1 or 2 months to 5 to 6 years. For large systems, the range narrows to 2-5 years for most systems of interest. Five years is limiting from an economic viewpoint. Organizations cannot afford to wait more than 5 years for a system. Two years is the lower limit because the manpower buildup is too great. This can be shown a number of ways.

1) Brooks [1] cites Vysottsky's observation that large scale software projects cannot stand more than a 30 percent per year buildup (presumably after the first year). The Rayleigh equation meeting this criterion is with $t_d \geq 2$ years.

2) The manpower rate invokes the intercommunication law cited by Brooks [1]; complexity $= N[(N-1)/2]$ where N is the number of people that have to intercommunicate. Thus, if the buildup is too fast, effective human communication cannot take place because the complexity of these communication paths becomes too great.

3) Management cannot control the people on a large software project at rates governed by the Norden/Rayleigh manpower equation with $t_d < 2$ years without heroic measures.

4) Generally, internal rates of manpower generation (which is a function of completion of current work) will not support an extremely rapid buildup on a new project. New hiring can offset this, but that approach has its own shortcomings.

When the feasible region is portrayed in three dimensions by introducing the difficulty K/t_d^2 as the third dimension, we have

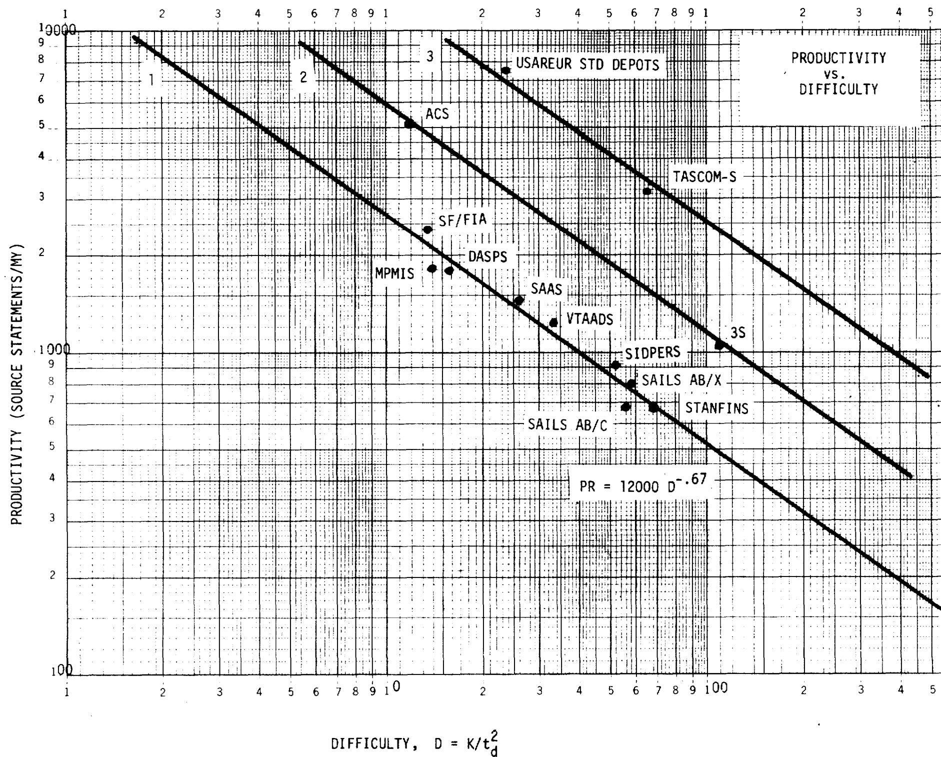


Fig. 9. Productivity versus difficulty for a number of U.S. Army Computer Systems Command systems. Lines 1, 2, and 3 represent different development environments. Line 1 represents a continental United States environment, batch job submission, a consistent set of standards and for standard Army systems that would be used at many installations. Line 2 is typical of a Pacific development environment, (initially) a different set of standards from 1, and the intended application was for one (or only a few) installations. Line 3 is typical of the European environment, different standards, single installation application. The two points shown for lines 2 and 3 are insufficient in themselves to infer the lines. Other more comprehensive data [14] show relations like line 1 with the same basic slope (–.62 to –.72). Based on this corroborating evidence, the lines 2 and 3 are sketched in to show the effect of different environment, standards, tools, and machine constraints.

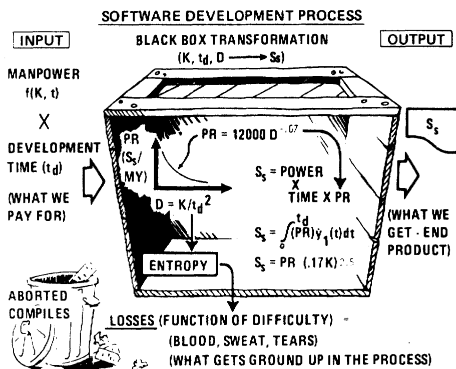


Fig. 10. Black box with cover removed.

a difficulty surface (Fig. 11). Note that as the development time is shortened, the difficulty increases dramatically. Systems tend to fall on a set of lines which are the trace of a constant magnitude of the difficulty gradient. Each of these lines is characteristic of the software house's ability to do a certain class of work. The constants shown [8], [15], [27] are preliminary. Further work is needed to refine these and develop others.

Difficulty Gradient

We can study the rate of change of difficulty by taking the gradient of D. The unit vector $\hat{i}$ points in t_d direction. The unit vector $\hat{j}$ points in K (and \$COST) direction.

grad D points almost completely in the $-\hat{i}(-t_d)$ direction. The importance of this fact is that shortening the development time (by management guidance, say) dramatically increases the difficulty, usually to the impossible level.

Plotting of U.S. Army Computer Systems Command systems on the difficulty surface shows that they fall on three distinct lines:

$$K/t_d^3 = 8, K/t_d^3 = 15, \text{and } K/t_d^3 = 27.$$

The expression for grad D is

$$\text{grad } D = \frac{-2K}{t_d^3}\,\hat{i} + \frac{1}{t_d^2}\,\hat{j}. \tag{7}$$

We note that the magnitude of the $\hat{i}$ component has the form K/t_d^3. The magnitude of grad D is

$$|\text{grad } D| \doteq \partial D/\partial t_d = 2K/t_d^3, \text{since } \partial D/\partial K \ll \partial D/\partial t_d$$

throughout the feasible region. $|\text{grad } D|$ is the magnitude of the maximum directional derivative. This points in the nega-

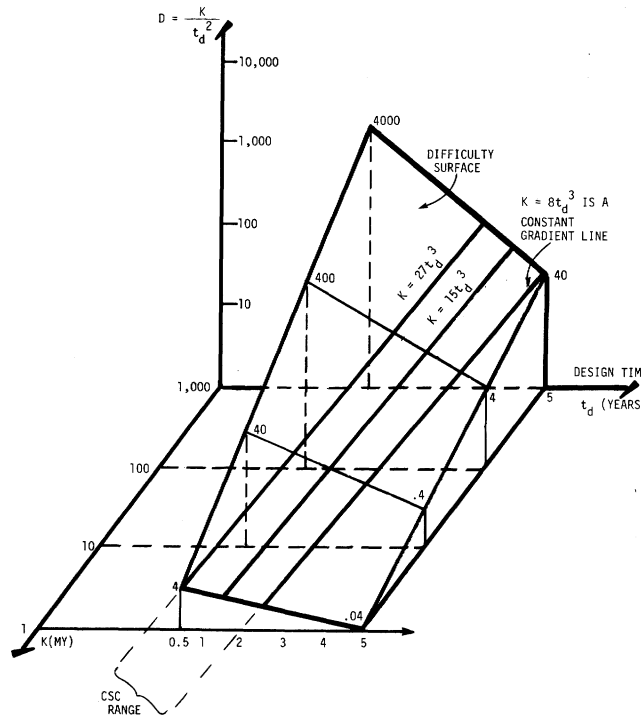


Fig. 11. Difficulty surface.

tive $\hat{i}(t_d)$ direction. The trace of all such points is a line K = constant times t_d^3. Such a line seems to be the maximum difficulty gradient line that the software organization is capable of accomplishing. That is, as system size is increased, the development time will also increase so as to remain on the trace of a line having a constant magnitude of the gradient defined by

$K/t_d^3 = C$, where C can equal either 8, 15, or 27. [2]

Study of all U.S. Army Computer Systems Command systems shows that 1) if the system is entirely new—has many interfaces and interactions with other systems—$C \doteq 8$; 2) if the system is a new stand-alone system, $C \doteq 15$; and 3) if the system is a rebuild or composite built up from existing systems where large parts of the logic and code already exist, then $C \doteq 27$. (These values may vary slightly from software house to software house depending upon the average skill level of the analysts, programmers, and management. They are, in a sense, figures of merit or "learning curves" for a software house doing certain classes of work.)

Significance of Management Directed Changes to the Difficulty

Managers influence the difficulty of a project by the actions they take in allocating effort ($\sim K$) and specifying scheduled completion ($\sim t_d$). We can determine the sensitivity of the difficulty to management perturbation by considering a few examples.

Assume we have a system which past experience has shown can be done with these parameter values: $K = 400$ MY, $t_d = 3$ years. Then $D = K/t_d^2 = 400/9 = 44.4$ MY/YR^2. If management were to reduce the effort (life-cycle budget) by 10 percent, then $K = 360$, MY $t_d = 3$ years, as before, and $D = 360/9 = 40$ (10 percent decrease). However, the more common situation is attempting to compress the schedule. If management were to shorten the time by $\frac{1}{2}$ year, then $K = 400$, $t_d = 2.5$, and $D = 400/6.25 = 64$ (44 percent increase). *The result is that shortening the natural development time will dramatically increase the difficulty.* This can be seen easily by examining Fig. 6 and by noting that $K = \sqrt{e}\ \dot{y}_{max}\ t_d$. D is thus proportional to $\dot{y}_{max}/t_d$ which is the slope from the origin to the peak of the Norden/Rayleigh curves. If the slope is too great, management cannot effectively convert this increased activity rate into effective system work probably because of the sequential nature of many of the subprocesses in the total system building effort.

V. The Software Equation

We have shown that if we know the management parameters K and t_d, then we can generate the manpower, instantaneous cost, and cumulative cost of a software project at any time t by using the Rayleigh equation.

But we need a way to estimate K and t_d, or better yet, obtain a relation that shows the relation between K, t_d, and the product (the total quantity of source code).

It can be shown that if we multiply the first subcycle curve (shown as the design and coding curve in Fig. 4) by the average productivity ($\overline{PR}$) and a constant scale factor (2.49) to account for the overhead inherent in the definition of productivity, then we obtain the instantaneous coding rate curve. The shape of this curve is identical to the design and coding curve

[2] These constants are preliminary. It is very likely that 6 to 8 others will emerge as data become available for different classes of systems.

but its dimensions are source statements per year ($\dot{S}_s$). The area under this coding rate curve is the total quantity of final end product source statements (S_s) that will be produced by time t. Thus, if we include all the area under the coding rate curve we obtain $S_s = 2.49 \cdot \overline{PR} \cdot K/6$ source statements, defined as delivered lines of source code. From our empirical relation relating the productivity $\overline{PR}$ to the difficulty, we can complete the linkage from product (source statements) to the management parameters.

Now the design and coding curve $\dot{y}_1$ has to match the overall manpower curve $\dot{y}$ initially, but it must be nearly complete by t_d; thus, $\dot{y}_1$ proceeds much faster, but has the same form as $\dot{y}$.

This suggests that the Norden/Rayleigh form is appropriate for $\dot{y}_1$ but now the parameter t_0 will be some fraction of t_d. A relation that works well is

$$t_0 \doteq t_d/\sqrt{6}.$$

Let us substitute that into a $\dot{y}_1$ Rayleigh curve,

$$\dot{y}_1 = K_1/t_0^2 \cdot t \cdot e^{-t^2/2t_0^2}.$$

$$\dot{y}_1 = \frac{K_1}{t_d^2/6} \cdot t \cdot e^{-t^2/2t_d^2/6}$$

$$= \frac{6K_1}{t_d^2} \cdot t \cdot e^{-3t^2/t_d^2}$$

but $K_1 = \frac{1}{6} K$, so $\dot{y}_1 = K/t_d^2 \cdot t \cdot e^{-3t^2/t_d^2}$ and $\dot{y}_1 = D \cdot t \cdot e^{-3t^2/t_d^2}$ is the manpower for design and coding, i.e., analysts and programmers doing the useful work on the system.

If we multiply by the average productivity rate $\overline{PR}$ then we should get the rate of code production. But here we have to be careful of the definition of productivity. The most common such definition is

$$\overline{PR} = \frac{\text{total end product code}}{\text{total effort to produce code}}.$$

The total effort to produce the code is a burdened number—it includes the overhead and the test and validation effort, i.e.,

$$y(t_d) = \int_0^{t_d} \dot{y}\, dt = 0.3935\, K.$$

The $\dot{y}_1$ curve is 95 percent complete at $t_d, y_1(t_d) = \int_0^{t_d} \dot{y}_1\, dt = 0.95(K/6)$. Accordingly, the burdened rate $\overline{PR}$ should be multiplied by $0.3935\, K/0.95/6\, K = 2.49$ to adjust for this condition.

Then

$$2.49\, \overline{PR} \cdot \dot{y}_1 = \frac{dS_s}{dt} = \dot{S}_s$$

$$\dot{S}_s = 2.49 \cdot \overline{PR} \cdot K/t_d^2 \cdot t \cdot e^{-3t^2/t_d^2} \quad \text{source statements/year}$$

$$S_s = \int_0^\infty \dot{S}_s\, dt = 2.49 \cdot \overline{PR} \cdot \int_0^\infty K/t_d^2 \cdot t \cdot e^{-3t/t_d^2}\, dt.$$

Therefore, $S_s = 2.49 \cdot \overline{PR} \cdot K/6$ source statements, interpreted to mean delivered lines of source code. From our empirical relation relating the $\overline{PR}$ to the difficulty, we can complete the linkage from product (source statements) to the management parameters.

Recall that we found that $\overline{PR} = C_n D^{-2/3}$ by fitting the data. The constant appears to take on only quantized values. Substituting,

$$S_s = 2.49 \cdot \overline{PR} \cdot K/6$$

$$= 2.49\, C_n \left(\frac{K}{t_d^2}\right)^{-2/3} K/6$$

$$S_s = \frac{2.49}{6}\, C_n K^{1/3}\, t_d^{4/3}.$$

The most dominant value for C_n in Fig. 9 is 12 000. Hence, for systems having that productivity constant C_n

$$S_s = \frac{2.49}{6}\,(12\,000)\, K^{1/3}\, t_d^{4/3}$$

$$S_s = 4980\, K^{1/3}\, t_d^{4/3} \text{ source statements.}$$

So, in general, the size of the product in source statements is

$$S_s = C_k\, K^{1/3}\, t_d^{4/3} \tag{8}$$

where C_k has now subsumed 2.49/6, is quantized, and seems to be a measure of the state of technology of the human-machine system.

The SIDPERS system is shown on Fig. 8. Its parameters are $K = 700$, $t_d = 3.65$. The calculated product size is

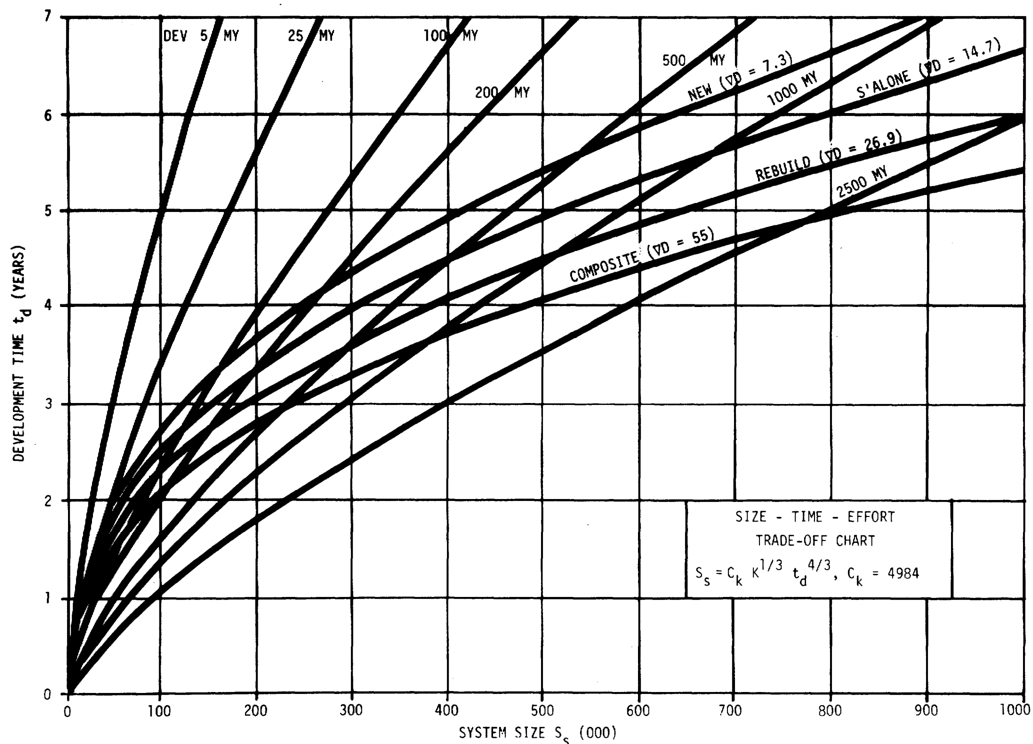
$$S_s = 4980\,(700)^{1/3}\,(3.65)^{4/3} = 247\,000$$

source statements which is reasonably close to the reported 256 000 source statements.

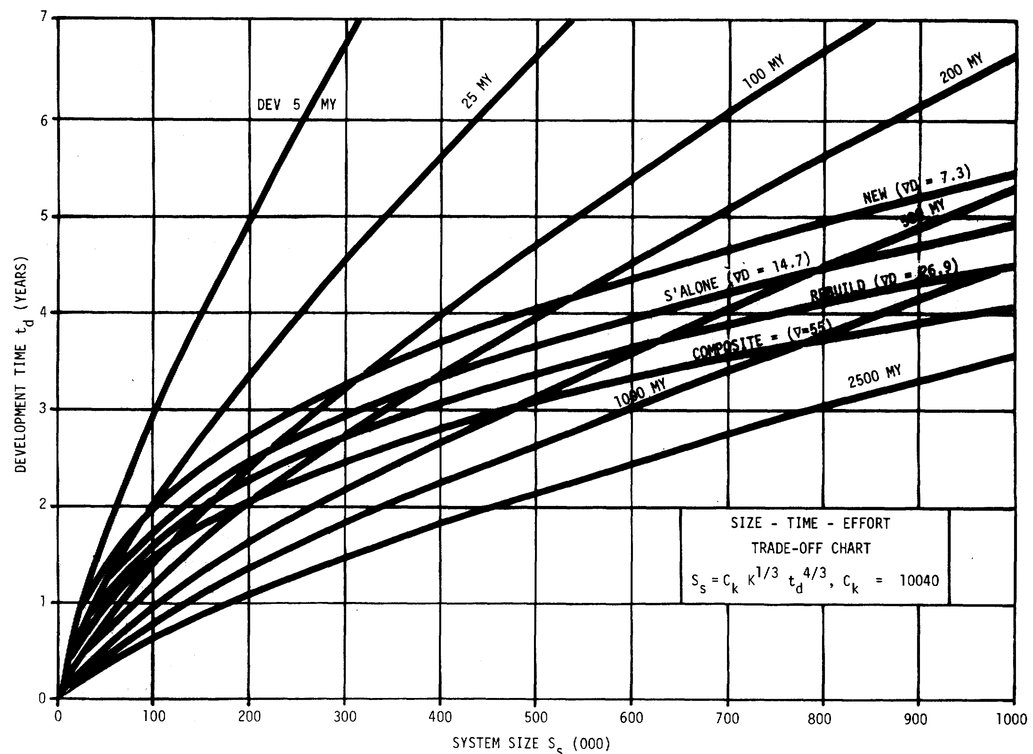
Note that we have now found *a reasonable relation linking the output (S_s) to the input (K, t_d - management parameters) and a constant (C_k) which is somehow a measure of the state of technology being applied to the project.* C_k appears to be quantized. The constant can be increased by applying better technology. The constant seems to relate to machine throughput (or programmer turnaround, available test time, etc.) and other technological improvements like the Chief Programmer Team, Top Down Structured Programming, on-line interactive job submission, etc.

Fig. 12 shows this relation combined with the limiting gradient conditions for the entire ranges of interest for K, t_d, and system size (S_s). This figure is very enlightening because it shows the incompressibility of time (t_d) and the very unfavorable impact of trading effort for time. This is, in fact, a quantitative description of Brook's law [1]. Adding people to accelerate a project can accomplish this until the gradient condition is reached, but only at very high cost.

For example, assume we have a new system of 100 000 S_s to build. If we have 5 years to the job, it can be done with 5 MY of development effort; however, if it has to be done in $3\frac{1}{3}$ years then it will cost about 25 MY of effort—a fivefold increase in effort (and cost). At the shortest possible time, $2\frac{3}{4}$ years, it will take about 55 MY of development effort, a



(a)



(b)

Fig. 12. (a) Size-time-effort tradeoff for function $S_s = 4984\ K^{1/3}\ t_d^{4/3}$. This is representative of about 5 years ago. Batch development environment, unstructured coding, somewhat "fuzzy" requirements and severe test bed machine constraint. (b) Size-time-effort tradeoff for function $S_s = 10040\ K^{1/3}\ t_d^{4/3}$. This is reasonably representative of a contemporary development environment with on-line interactive development, structured coding, less "fuzzy" requirements, and machine access fairly unconstrained. In comparing the two sets of curves note the significant improvement in performance from the introduction of improved technology which is manifested in the constant C_k.

tenfold increase over the 5 year plan. *Clearly, time is not a "free good" and while you can trade people for time within narrow limits, it is a most unfavorable trade and is only justified by comparable benefits or other external factors.*

The Effort-Time Tradeoff Law

The tradeoff law between effort and time can be obtained explicity from the software equation

$$S_s = C_k K^{1/3} t_d^{4/3}.$$

A constant number of source statements implies $K t_d^4$ constant. So $K = \text{constant}/t_d^4$, or proportionally, development effort = constant$/t_d^4$, is the effort-development time tradeoff law.

Note particularly that development time is compressible only down to the governing gradient condition (see Fig. 12); the software house is not capable of doing a system in less time than this because the system becomes too difficult for the time available.

Accordingly, the set of curves relating development effort (0.4 K), development time (t_d), and the product (S_s) permits project managers to play "what if" games and to tradeoff cost versus time without exceeding the capacity of the software organization to do the work.

Effect of Constant Productivity

One other relation is worth obtaining; the one where the average productivity remains constant.

$$\overline{PR} = C_n D^{-2/3} = \text{constant implies that } D = \text{constant}.$$

So the productivity for different projects will be the same only if the difficulty is the same. This does not seem reasonable to expect very frequently since the difficulty is a measure of the software work to be done, i.e., $K/t_d^2 = D$ which is a function of the number of files, the number of reports, and the number of programs the system has. Thus, planning a new project based on using the same productivity a previous project had, is fallacious unless the difficulty is the same.

The significance of this relation for an individual system is that when the $\overline{PR}$ is fixed, the difficulty K/t_d^2 remains constant during development. However, we know that the difficulty is a function of the system characteristics, so if we change the system characteristics during development, say the number of files, the number of reports, and the number of subprograms, then the difficulty will change and so will the average productivity. This in turn will change our instantaneous rate of code production ($\dot{S}_s$) which is made up of both parameter terms and time-varying terms.

In summary then, $S_s = C_k K^{1/3} t_d^{4/3}$ appears to be the management equation for software system building. As technology improves, the exponents should remain fixed since they relate to the process. C_k will increase (in quantum jumps) with new technology because this constant relates to the overall information throughput capacity of the system and (tentatively) seems to be more heavily dependent on machine throughput than other factors.

The Software Differential Equation

Having shown some of the consequences of the Rayleigh equation and its application to the software process let us examine why we have selected this model rather than a number of other density functions that could be fitted to the data. We return to Norden's initial formulation to obtain the differential equation.

Norden's description [8] of the process is this: The rate of accomplishment is proportional to the pace of the work times the amount of work remaining to be done. This leads to the first-order differential equation

$$\dot{y} = 2at\,(K - y)$$

where $\dot{y}$ is the rate of accomplishment, $2at$ is the "pace" and $(K - y)$ is the work remaining to be done. Making the explicit substitution for $a = (1/2t_d^2)$ we have $\dot{y} = t/t_d^2\,(K - y)$. t/t_d^2 is Norden's linear learning law.

We differentiate once more with respect to time, rearrange, and obtain

$$\ddot{y} + t/t_d^2\,\dot{y} + y/t_d^2 = K/t_d^2 = D. \tag{9}$$

This is a second derivative form of the software equation. The Norden/Rayleigh integral is its solution which can be verified by direct substitution. Note that this equation is similar to the nonhomogeneous 2nd order differential equations frequently encountered in mechanical and electrical systems. There are two important differences. The forcing function $D = K/t_d^2$ is a constant rather than the more usual sinusoid, and the $\dot{y}$ term has a variable coefficient t/t_d^2, proportional to the 1st power of time.

Uses Of the Software Equation

The differential equation $\ddot{y} + t/t_d^2\,\dot{y} + y/t_d^2 = K/t_d^2$ is very useful because it can be solved step-by-step using the Runge-Kutta solution. The solution can be perturbed at any point by changing $K/t_d^2 = D$, the difficulty. This is just what happens in the real world when the customer changes the requirements or specifications while development is in process. If we have an estimator for D (which we do) that relates K/t_d^2 to the system characteristics, say the number of files, the number of reports, and the number of application programs, then we can calculate the change in D and add it to our original D, continue our Runge-Kutta solution from that point in time and thus study the time slippage and cost growth consequences of such requirements changes. Several typical examples are given in [13].

When we convert this differential equation to the design and coding curve $\dot{y}_1$ by substituting $t_d/\sqrt{6}$, and $K_1 = K/6$, we obtain

$$\ddot{y}_1 + \frac{6t}{t_d^2}\,\dot{y}_1 + \frac{6}{t_d^2}\,y_1 = \frac{K}{t_d^2} \tag{10}$$

and multiplying this by the $\overline{PR}$ and conversion factor as before we obtain an expression that gives us the coding rate and cumulative code produced at time t.

$$\begin{aligned}\ddot{S}_s + t/(t_d/\sqrt{6})^2\,\dot{S}_s + 1/(t_d/\sqrt{6})^2\,S_s &= 2.49 \cdot \overline{PR} \cdot K/t_d^2 \\ &= C_n(K/t_d^2)^{1/3} \\ &= 2.49\,C_n D^{1/3}. \quad (11)\end{aligned}$$

Since D is explicit, this equation can also be perturbed at any

TABLE I
RUNGE-KUTTA SOLUTION TO CODING RATE DIFFERENTIAL EQUATION FOR SIDPERS

t (years)	Coding Rate (S_s/year) (000)	Cumulative Code (S_s) (000)	
0	0	0	
.5	52.8	13.6	
1.0	89.2	50.0	
1.5	101.0	98.6	
2.0	90.8	147.0	
2.5	68.4	187.0	
3.0	44.2	215.0	
3.5	24.9	323.0	
→ 3.65	20.33	236.0	← Actual size at extension is 256,000 S_s which is pretty close to this
4.0	12.3	241.0	
4.5	5.36	246.0	
5.0	2.09	247.0	

SIDPERS parameters

K = 700 MY
t_d = 3.65 years
D = 52.54 MY/yr
PR = 914 S_s/MY (burdened)

DIFFERENTIAL EQUATION

$$\ddot{S} + \frac{t}{\left(\frac{3.65}{\sqrt{6}}\right)^2} \qquad \dot{S} + \frac{1}{\left(\frac{3.65}{\sqrt{6}}\right)^2} \qquad S = 2.49 . \overline{PR} . D = 2.49 . (12009 D^{-2/3}) . D$$
$$= 2.49 . (12009) . (D)^{1/3}$$
$$= 29902 . (52.54)^{1/3}$$
$$= 111785$$

time t in the Runge-Kutta solution and changes in requirements studied relative to their impact on code production.

An example of code production for SIDPERS using the Runge-Kutta solution is shown in Table I.

This equation in both its manpower forms $\dot{y}$ and $\dot{y}_1$ as well as the code form can be used to track in real-time manpower, cumulative effort, code production rate, and cumulative code. Since the equation relates to valid end product code and does not explicitly account for seemingly good code that later gets changed or discarded, actual instantaneous code will probably be higher than predicted by the equation. Thus, a calibration coefficient should be determined empirically on the first several attempts at use and then applied in later efforts where more accurate tracking and control are sought.

VI. SIZING THE SOFTWARE SYSTEM

Having justified the use of the Rayleigh equation from several viewpoints, we turn to the question of how we can determine the parameters K and t_d early in the process—specifically during the requirements and specification phase before investment decisions have to be made.

One approach that works in the systems definition phase and in the requirements and specification phase is to use the graphical representation of the software equation $S_s = C_k K^{1/3} t_d^{4/3}$ as in Fig. 12. Assume we choose C_k using criteria as outlined in Fig. 12. It is usually feasible to estimate a range of possible sizes for the system, e.g., 1500 000 to 2500 000 S_s. It is also possible to pick the range of desired development times. These two conditions establish a probable region on the tradeoff chart (Fig. 12(b), say). The gradient condition established the limiting constraint on development time. One can then heuristically pick most probable values for development time and effort without violating constraints, or one can even simulate the behavior in this most probable region to generate expected values and variances for development time and effort. The author uses this technique on a programmable pocket calculator to scope projects very early in their formulation.

Another empirical approach is to use a combination of prior history and regression analysis that relates the management parameters to a set of system attributes that can be determined before coding starts. This approach has produced good results.

Unfortunately, K and t_d are not independently linearly related to such system attributes as number of files, number of reports, and number of application programs. However, $K/t_d^2 = D$ is quite linear with number of files, number of reports, and number of application subprograms, both individually and jointly. Statistical tests show that number of files and number of reports are highly redundant so that we can discard one and get a good estimator from the number of application subprograms and either of the other two.

There are relationships other than the difficulty that have significantly high correlation with measures of the product. These relationships are

$$K/t_d^3 = f_2(x_1, x_2, x_3)$$
$$K/t_d^2 = f_3(x_1, x_2, x_3)$$
$$K/t = f_4(x_1, x_2, x_3)$$
$$K = f_5(x_1, x_2, x_3)$$
$$Kt_d = f_6(x_1, x_2, x_3)$$
$$Kt_d^4 = f_7(x_1, x_2, x_3)$$

where K and t_d are the parameters of the Norden/Rayleigh equation. x_1 is the number of files the system is expected to have, x_2 is the number of reports or output formats the system will have, and x_3 is the number of application subprograms. These independent variables were chosen because they directly relate to the program and system building process, that is, they are representative of the programmer and analyst work to create modules and subprograms. More importantly, they are also a measure of the integration effort that goes on to get programs to work together in a systems programming product. The other important reason for using these variables is practical: they can be estimated with reasonable accuracy (within 10-15 percent) early enough in the requirements specification process to provide timely manpower and dollar estimates. Note also that x_1, x_2, x_3 are imprecise measures—the *number* of subprograms, say—this implies an average size of an individual program with some uncertainty or "random noise" present. It seems prudent to let the probability laws help us rather than try to be too precise when we know that there are many other random-like perturbations that will occur later on that are unknown to us. For business applications, the number of files, reports, and application subprograms work very well as estimators. For

TABLE II
USACSC SYSTEM CHARACTERISTICS

System	Life Cycle Size K (MY)	Development Time t_d (YRS)	Number of Files x_1	Number of Rpts. x_2	Number of Appl. Progs. x_3
MPMIS	73.6	2.28	94	45	52
MRM	84	1.48	36	44	31
ACS	33	1.67	11	74	39
SPBS	70	2.00	8	34	23
COMIS	27.5	1.44	14	41	35
AUDIT	10	2.00	11	5	5
CABS	7.74	1.95	22	14	12
MARDIS	91	2.50	6	10	27
MPAS	101	2.10	25	95	109
CARMOCS	153	2.64	13	109	229
SIDPERS	700	3.65	172	179	256
VTAADS	404	3.50	155	101	144
BASOPS-SUP	591	2.73	81	192	223
SAILS AB/C	1028	4.27	540	215	365
SAILS AB/X	1193	3.48	670	200	398
STARCIPS	344	3.48	151	59	75
STANFINS	741	3.30	270	228	241
SAAS	118	2.12	131	152	120
COSCOM	214	4.25	33	101	130

other applications other estimators should be examined. It is usually apparent from the functional purpose of the system which estimators are likely to be best.

The dependent variables $K/t_d^3 \cdots K, Kt_d, Kt_d^4$ also have significance. K/t_d^3 is proportional to the difficulty gradient, K/t_d^2 is the difficulty, a force-like term which is built into our time-varying solution model, the Rayleigh equation. K/t_d is proportional to the average manpower of the life-cycle and K is the life-cycle size (area under Rayleigh manpower curve).

Kt_d is the "action" association with the action integral, and Kt_d^4 is a term which arises when the change in source statements is zero.

A numerical procedure to obtain estimators for the various $Kt_d^4, Kt_d, K \cdots K/t_d^3$ terms is to use multiple regression analysis and data determined from the past history of the software house. A set of data from U.S. Army Computer Systems Command is shown in Table II.

We will determine regression coefficients from x_2, x_3 to estimate K/t_d as an example of the procedure. We write an equation of the form

$$\alpha_2 x_2 + \alpha_3 x_3 = K/t_d$$

for each data point $(K/t_d, x_2, x_3)$ in the set. This gives a set of simultaneous equations in the unknown coefficient α_2 and α_3. We solve the equations using matrix algebra as follows:

$$[x]^T [x]\ [\alpha] = [x]^T [K/t_d] \quad (12)$$

$$[\alpha] = ([x]^T [x])^{-1} [x]^T [K/t_d] \quad (13)$$

$$\begin{bmatrix} \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} c_{22} & c_{23} \\ c_{32} & c_{33} \end{bmatrix}^{-1} \begin{bmatrix} \Sigma K/t \cdot x_2 \\ \Sigma K/t_d \cdot x_3 \end{bmatrix} \quad (14)$$

where the α matrix is a single column vector of 2 elements, the X matrix is a matrix of N rows and 2 columns, the X^{-1} inverse matrix is a square matrix of 2 rows and 2 columns, and the $\Sigma K/t_{d_i} \cdot x_1$ matrix is a column vector of two elements. Solving for the α_i provides coefficients for the regression estimator. For the data set of Table II, we obtain

$$K/t_d = 0.2200\, x_2 + 0.5901\, x_3.$$

Estimators for each of the other $K/t_d^3 \cdots K, Kt_d, Kt_d^4$ are determined in the same way. These estimators for the data set of Table II are listed below:

$$K/t_d^2 = 0.1991\, x_2 + 0.0859\, x_3$$

$$K/t_d = 0.2200\, x_2 + 0.5901\, x_3$$

$$K = -0.3447\, x_2 + 2.7931\, x_3$$

$$Kt_d = -4.1385\, x_2 + 11.909\, x_3$$

$$Kt_d^4 = -483.3196\, x_2 + 791.0870\, x_3.$$

Assuming we have estimates of the number of reports and the number of application subprograms, we can get estimates of K and t_d by solving any pair of these equations simultaneously, or we can solve them all simultaneously by matrix methods.

Since we have observed that different classes of systems tend

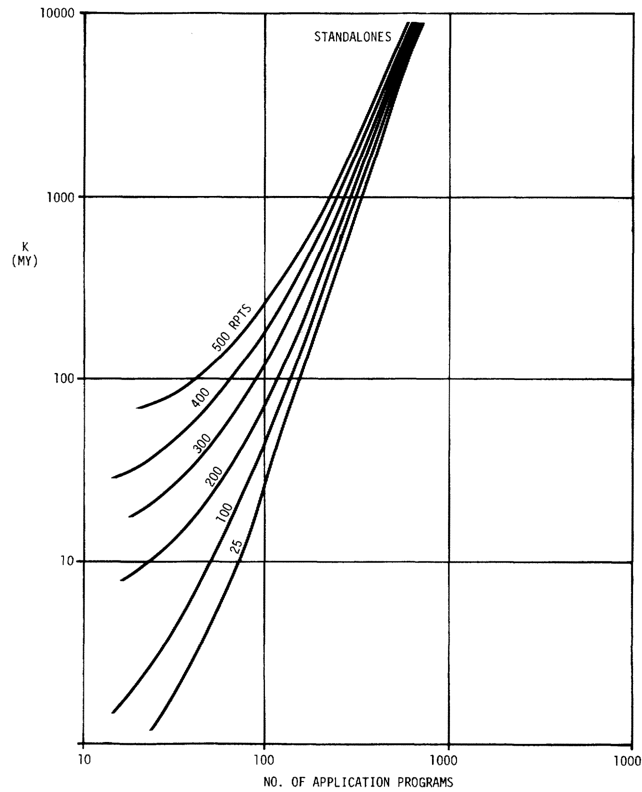


Fig. 13. Life-cycle size as a function of number of application programs and number of reports.

to fall along distinct constant difficulty gradient lines, we can exploit this feature by forcing the estimators to intersect with such a predetermined constant gradient line. We can illustrate using the matrix approach. For example, let us assume we have a new stand-alone system with an estimate from the specification that the system will require 179 reports and 256 application programs. The gradient relation for a stand-alone is

$$K/t_d^3 = 14.$$

The regression estimators are

$$K/t_d^2 = 57.63$$

$$K/t_d = 190.45$$

$$K = 652.54$$

$$Kt_d = 2307.91$$

$$Kt_d^4 = 116\,004.76.$$

Taking natural logarithms of these six equations we obtain the matrix equation

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -3 & -2 & -1 & 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} 1 & -3 \\ 1 & -2 \\ 1 & -1 \\ 1 & 0 \\ 1 & 1 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} \ln K \\ \ln t_d \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -3 & -2 & -1 & 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} 2.6391 \\ 4.0540 \\ 5.2494 \\ 6.4809 \\ 7.7441 \\ 11.6614 \end{bmatrix}$$

which yields

$$\ln K = 6.5189; \hat{K} = 677.86 \text{ MY}$$

$$\ln t_d = 1.2772; \hat{t}_d = 3.59 \text{ YRS}.$$

The data used for these examples are that of SIDPERS (K = 700, t_d = 3.65, as of 1976). The estimators perform quite well in this general development time and size regime. They are less reliable and less valid for small systems ($K < 100$ MY; $t_d < 1.5$ years) because the data on the larger systems are more consistent. Extrapolation outside the range of data used to generate the estimators is very tenuous and apt to give bad results.

Once the estimating relationships have been developed for the software house as just outlined above, then they can be run through the range of likely input values and system types to construct some quick estimator graphs. An example of an early set of quick estimator charts for U.S. Army Computer Systems Command is shown in Figs. 13-15. These figures give considerable insight into the way large scale software projects

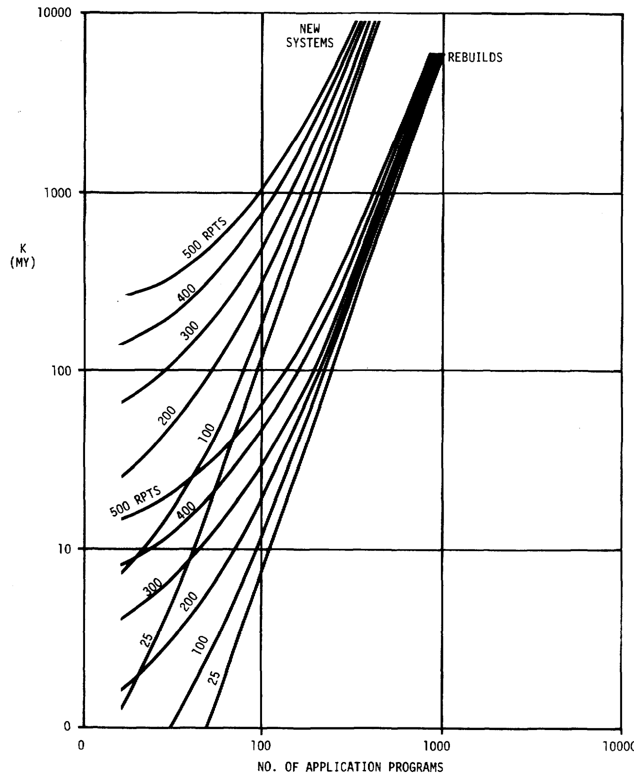


Fig. 14. Life-cycle size as function of number of application programs and number of reports.

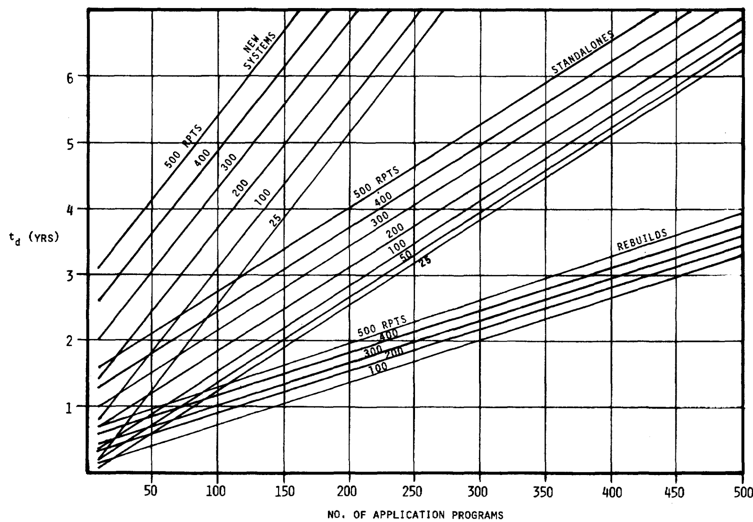


Fig. 15. Development time as a function of number of application programs and number of reports.

have to be done in terms of number of application programs, reports, and the effort to integrate these pieces of work.

Can the estimators developed here be used by other software houses? Probably not—at least not without great care and considerable danger. This is because each software house has its own standards and procedures. These produce different average lengths of application programs. Accordingly, another software house might have an average length of program of 750 lines of Cobol compared with about 1000 lines of Cobol for Computer Systems Command. Thus, the number of application programs would reflect a different amount of work/per application program than CSC experience. Scaling would be necessary. Scaling may or may not be linear. There is no evidence to suggest scaling should be linear since most everything else in the software development process is some complex power function. Furthermore, we have shown earlier the influence of the state-of-technology constant. The multiple regression technique does not explicitly account for this; it is

implicit within the data. Therefore, the regression analysis has to be done for each software house to account for those factors which would be part of a state-of-technology constant.

Engineering and Managerial Application

Space does not permit the development of management engineering applications of these concepts to planning and controlling the software life-cycle. The applications have been rather comprehensively worked out and are contained in [13]. Error analysis, or determining the effect of stochastic variation, again has not been treated due to space limitations.

VII. Summary

Software development has its own characteristic behavior. Software development is dynamic (time varying)—not static.

Code production rates are continuously varying—not constant.

The software state variables are

1) the state of technology C_n or C_k;
2) the applied effort K;
3) the development time t_d; and
4) the independent variable time t.

The software equation relates the product to the state variables:

$$S_s = \int_0^\infty \overline{PR} \cdot \dot{y}_1 \, dt = C_k K^{1/3} \, t_d^{4/3}.$$

The tradeoff law $K = C/t_d^4$ demonstrates the cost of trading development time for people. Time is not free. It is very expensive.

References

[1] F. P. Brooks, Jr., *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
[2] L. H. Morin, "Estimation of resources for computer programming projects," M.S. thesis, Univ. North Carolina, Chapel Hill, NC, 1973.
[3] P. F. Gehring and U. W. Pooch, "Software development management," *Data Management*, pp. 14-18, Feb. 1977.
[4] P. F. Gehring, "Improving software development estimates of time and cost," presented at the 2nd Int. Conf. Software Engneering, San Francisco, CA, Oct. 13, 1976.
[5] —, "A quantitative analysis of estimating accuracy in software development," Ph.D. dissertation, Texas Univ., College Station, TX, Aug. 1976.
[6] J. D. Aron, "A subjective evaluation of selected program development tools," presented at the Software Life Cycle Management Workshop, Airlie, VA, Aug. 1977, sponsored by U.S. Army Computer Systems Command.
[7] P. V. Norden, "Useful tools for project management," in *Management of Production*, M. K. Starr, Ed. Baltimore, MD: Penguin, 1970, pp. 71-101.
[8] —, "Project life cycle modelling: Background and application of the life cycle curves," presented at the Software Life Cycle Management Workshop, Airlie, VA, Aug. 1977, sponsored by U.S. Army Computer Systems Command.
[9] L. H. Putnam, "A macro-estimating methodology for software development," in *Dig. of Papers, Fall COMPCON '76, 13th IEEE Computer Soc. Int. Conf.*, pp. 138-143, Sept. 1976.
[10] —, "ADP resource estimating: A macro-level forcasting methodology for software development," in *Proc. 15th Annu. U.S. Army Operations Res. Symp.*, Fort Lee, VA, pp. 323-327, Oct. 26-29, 1976.
[11] —, "A general solution to the software sizing and estimating problem," presented at the Life Cycle Management Conf. Amer. Inst. of Industrial Engineers, Washington, DC, Feb. 8, 1977.
[12] —, "The influence of the time-difficulty factor in large scale software development," in *Dig. of Papers, IEEE Fall COMPCON '77 15th IEEE Computer Soc. Int. Conf.*, Washington, DC, pp. 348-353, Sept. 1977.
[13] L. H. Putnam and R. W. Wolverton, "Quantitative management: Software cost estimating," a tutorial presented at COMPSAC '77, IEEE Computer Soc. 1st Int. Computer Software and Applications Conf., Chicago, IL, Nov. 8-10, 1977.
[14] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, pp. 54-73, 1977.
[15] E. B. Daly, "Management of software development," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 229-242, May 1977.
[16] W. E. Stephenson, "An analysis of the resources used in the safeguard system software development," Bell Lab., draft paper, Aug. 1976.
[17] "Software cost estimation study: Guidelines for improved cost estimating," Doty Associates, Inc., Rome Air Development Center, Griffiss AFB, NY, Tech. Rep. RADC-TR-77-220, Aug. 1977.
[18] G. D. Detletsen, R. H. Keer, and A. S. Norton, "Two generations of transaction processing systems for factory control," General Electric Company, internal paper, undated, circa 1976.
[19] "Software systems development: A CSDL project history," the Charles Stark Draper Laboratory, Inc., Rome Air Development Center, Griffiss AFB, NY, Tech. Rep. RADC-TR-77-213, June 1977.
[20] T. C. Jones, "Program quality and programmer productivity," *IBM Syst. J.*, vol. 17, 1977.
[21] "Software systems development: A CSDL project history," the Charles Stark Draper Laboratory, Inc., Rome Air Development Center, Griffiss AFB, NY, Tech. Rep. RADC-TR-77-213, June 1977.
[22] A. D. Suding, "Hobbits, dwarfs and software," *Datamation*, pp. 92-97, June 1977.
[23] T. J. Devenny, "An exploratory study of software cost estimating at the electronic systems division," Air Force Inst. of Technology, WPAFB, OH, NTIS AD-A1030-162, July 1976.
[24] D. H. Johnson, "Application of a macro-estimating methodology for software development to selected C group projects," NSA, internal working paper, May 6, 1977.
[25] J. R. Johnson, "A working measure of productivity," *Datamation*, pp. 106-108, Feb. 1977.
[26] R. E. Merwin, memorandum for record, subject: "Data processing experience at NASA-Houston Manned Spacecraft Center," Rep. on Software Development and Maintenance for Appollo-GEMINI, CSSSO-ST, Jan. 11, 1970.
[27] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," Graduate School of Management, UCLA, DDC ADA 034085, 1976.
[28] R. E. Bellman, "Mathematical model making as an adaptive process," ch. 17, in *Mathematical Optimization Techniques*, The Rand Corporation, R-396-PR, Apr. 1963.
[29] G. E. P. Box and L. Pallesen, "Software budgeting model," Mathematics Research Center, University of Wisconsin, Madison, (Feb. 1977 prepublication draft).
[30] "Management Information Systems," *A Software Resource Macro-Estimating Procedure*, Hq. Dep, of the Army, DA Pamphlet 18-8, Feb. 1977.
[31] A. M. Pietrasanta, "Resource analysis of computer program system development," in *On the Management of Computer Programming*, G. F. Weinwurn, Ed. Princeton, NJ: Auerbach, 1970, p. 72.

Lawrence H. Putnam was born in Massachusetts. He attended the University of Massachusetts for one year and then attended the U.S. Military Academy, West Point, NY, graduating in 1952. He was commissioned in Armor and served for the next 5 years in Armor and Infantry troop units. After attending the Armor Officer Advanced Course, he attended the U.S. Naval Post Graduate School, Monterey, CA, where he studied nuclear effects engineering for 2 years, graduating in 1961 with the M.S. degree in physics. He then went to the

Combat Developments Command Institute of Nuclear Studies, where he did operations research work in nuclear effects related to target analysis, casualty, and troop safety criteria. Following a tour in Korea, he attended the Command and General Staff College.

Shortly thereafter he returned to nuclear work as a Senior Instructor and Course Director in the Weapons Orientation Advanced Course at Field Command, Defense Nuclear Agency, Sandia Base, NM. Following that, he served in Vietnam with the G4, I Field Force, commanded a battalion at Fort Knox, KY, for 2 years, and then spend 4 years in the Office of the Director of Management Information Systems and Assistant Secretary of the Army at Headquarters, DA. His final tour of active service was with the Army Computer Systems Command as Special Assistant to the Commanding General. He retired in the grade of Colonel. He is now with the Space Division, Information Systems Programs, General Electric Company, Arlington, VA. His principal interest for the last 4 years has been in developing methods and techniques for estimating the life-cycle resource requirements for major software applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

المنارة للاستشارات

www.manaraa.com