D3

SOFTWARE COSTING AND LIFE CYCLE CONTROL

©Lawrence H. Putnam
Quantitative Software Management, Inc.
1057 Waverley Way
McLean, VA 22101

It is remarkable that our $40–50 billion per year computer industry has 1/3 to 1/2 of its effort (and cost) out of control. I am referring to the software generation part of the industry. For 25 years now 200 to 300% cost overruns and up to 100% time slippages have been common, frequent—almost universal—as if there were no pattern, no process, no methodology, no characteristic behavior to the software development process. Indeed, it has become so unfathomable that responsible managers, controllers and corporate officers have tended to avoid the issue, accept the inevitability of overrun, and eat the extra cost—rather than find ways to get the problem solved.

If this were a trivial expense then such managerial responses would make sense. But $16–20 billion a year for the nation is non-trivial. Software development activities for major corporations cost 1–3% of revenues. This is perhaps 10–40% of net profit—thus, an activity worthy of controlling to the same standard as other critical corporate activities.

How can it be? People are aware of these realities. Many seminars, conferences and studies have been (and are still being) conducted to try to provide answers to the management questions:

Can I do it?
How much will it cost?
How long?
How many people?
What's the manloading?
What's the cash flow?
What's the trade-off?
What are the risks?

My studies of the past five years show very conclusively that there is a fundamental characteristic behavior to the software development process. The underlying characteristic is the complex human intercommunication process necessary to permit broad, abstract concepts to be transformed into a set of absolutely specific instructions the machine can respond to. This human intercommunication process is characterized by ambiguity and partial understanding. Progress proceeds in "fits and starts"—"surges"—"two steps forward, one back"—"loop back and start over," etc. These are all expressions to describe complex feedback paths, driven by random interaction among the human participants—all of whom must interact in a highly interdependent way.

People trying to plan and manage software attempt to do it deterministically – linear process flow diagram, decompose into a work breakdown structure and Gantt chart, assign tasks and schedule and then try to execute. Further, in an effort to meet arbitrary schedules, many activities that have sequential or partially sequential dependencies are attempted in parallel in the mistaken belief that what sometimes works in independent manufacturing processes will succeed in software. After 25 years of failure it is time to recognize this approach (by itself) will not work with software. We will have to deal with fundamentals.

The characteristic (average) behavior of software development over time is well described by the Rayleigh equation, a specific form of the Weibull family of reliability functions. The Rayleigh equation appears frequently in random statistical processes – scattering phenomena, narrow band Gaussian processes, diffusion and transport phenomena, quantum mechanics – so it is very reasonable to expect its appearance in software development where we implicitly recognize the unpredictability of the process, yet seem afraid to say it is a statistical process driven by many complex interactions unknown in advance and therefore random. The Rayleigh equation describes the average behavior over time of software development because it is a good model of a large number of Gaussian variables whose phases are random, meaning that many pieces of work will not be "in phase," hence will not "add" constructively, but may indeed "subtract," requiring feedback, rework, and so on.

Fortunately, we don't have to model this behavior in detail. The Rayleigh equation represents the overall time-varying behavior very well. Moreover, the Rayleigh equation parameters yield the management parameters that directly answer the management questions. The Rayleigh/Norden overall manpower equation for large systems is

$$\dot{y} = (K/t_d^2) \cdot t \cdot \exp(-t^2/2t_d^2) \text{ people}$$

where

K is the life cycle effort in manyears, or manmonths,

t_d is the development time in years, or months,

t is elapsed time in years, or months, from the beginning of detailed logic design and coding, and

$\dot{y}$ is manpower in manyears/year, or manmonths/month, or just plain, countable people at any instant in time.

Multiplying this equation by the labor rate turns it into a cost function. Integrating (adding up the curve) over time yields cumulative effort and cost any at time – thus, development effort and cost is an easily extractable subset of the life cycle numbers.

The relationships among the Rayleigh parameters are highly complex. This probably explains why purely empirical approaches have not yielded satisfactory solutions until now. Recently we have found good, practical ways to relate the Rayleigh management parameters to valid system characteristics in ways that answer the management questions directly with numbers that are the best possible answers. These findings are so important they should be commented upon immediately because the economic implications are absolutely awesome. The main points are these:

- A good, accurate method to size a system early in functional development has been developed.
- A software equation relating the system size to the managerial parameters – manmonths of effort (K), development time (t_d), and the state of technology being applied to the development effort – has been developed.
- Empirical verification from hundreds of systems of all types and development environments that the basic Rayleigh/Norden time varying behavior is phenomenologically sound.

- Empirical verification from the 400 odd systems collected by RADC that the parametric software equation and constraint relations are sound and are sophisticated enough to cope with the enormous range these parameters exhibit. (This has been the problem with single and multiple regression approaches – the variance has been enormous – this has been attributed to "poor data" when in reality, it is much more a function of the development environment (development computer, tools, and techniques) and system type (complexity).)

- Two good ways have been found to determine the management parameters from the system size and a set of system and managerial constraints.

  (1) LINEAR PROGRAMMING which produces a pair of constrained optimal solutions for the managerial parameters.

  (2) MONTE CARLO SIMULATION which produces a minimum time solution and uncertainty or risk profiles.

- Better and more straightforward ways to demonstrate the acute time sensitivity of the software development process.

- A dynamic (time varying) approach to measuring progress (not just resource consumption), coping with requirements changes when they occur and continually converging toward the actual system behavior – in effect, a real time process controller.

- The ability to play managerial "what if" games with software development projects AT ANY POINT IN THE LIFE CYCLE (from earliest feasibility analysis through development into the operations and maintenance phase).

I will comment on each of these points.

SIZING

Many software developers will tell you they cannot size a system accurately, that there is too much inherent uncertainty. This is partly true. They usually cannot size a module emanating from a functional description very accurately. But they can estimate ranges quite well. This is good enough because the statistics of aggregation work with us. We use the PERT estimating algorithm (Beta distribution) and ask our design engineers to estimate the size of each functional module in this way:

a – smallest number of source statements
m – most likely number of source statements
b – largest number of source statements

The expected number from a specific functional module is

$$\frac{a + 4m + b}{6}$$

and the associated standard deviation is approximately

$$\frac{|b - a|}{6}$$

When we aggregate all the module estimates into a systems estimate, a remarkable thing happens – the relative uncertainty of the system size ($^{\sigma}TOT/^{E}TOT$) is generally much smaller than the uncertainty (σ_i/E_i) of any of the modules. This is because of the cancelling effect that will occur in execution. Some modules will be smaller than planned, others larger; the net effect is a much smaller aggregate standard deviation than one would intuitively expect. Consider the following set of data obtained from an experienced team of about 15 system designers about twelve weeks into the functional design of a contemporary information retrieval system. Here are their estimates.

	Smallest	Most Likely	Largest	Expected	Std Dev
Maintain	8675.	13375.	18625.	13467.	1658.
Search	5577.	8988.	13125.	9109.	1258.
Route	3160.	3892.	8800.	4588.	940.
Status	850.	1425.	2925.	1579.	346.
Browse	1875.	4052.	8250.	4389.	1063.
Print	1437.	2455.	6125.	2897.	781.
User Aids	6875.	10625.	16250.	10938.	1563.
Incoming Messages	5830.	8962.	17750.	9905.	1987.
System Monitor	9375.	14625.	28000.	15979.	3104.
System Management	6300.	13700.	36250.	16225.	4992.
Comm. Proc..	5875.	8975.	14625.	9400.	1458.
Total				98475.	7081.

Note that the expected number of source statements for the system is just the sum of the expected number for each functional module. The standard deviation for the system is the square root of the sum of squares of the module standard deviations. This is what accounts for the cancelling effect. Note that the ratio $^{\sigma}TOT/^{E}TOT = 7081/98475$ is only about 7 percent, yet the coefficient of variation of one function, SYS MGT, is 4992/16225 – 31 percent, and the absolute magnitude of the standard deviation for SYS MGT, 4992, is 70 percent the size of the standard deviation for the entire system. The upshot of this is that we can predict system size to engineering accuracy even when there is large uncertainty in individual functional modules. This is a proven technique used in 15 years of experience in PERT charting. Counterintuitive –YES; but it works. Another point of major importance is that the engineers asked to provide the estimates are comfortable with the procedure. They are not threatened by range estimates. With this technique they can always be right, rather than always wrong as with any single number estimate they might provide. The more uncertain they are the broader the range they estimate. This is intelligent hedging that is accounted for in a systematic way. The technique has been used five times within GE with excellent results. Engineers and managers all felt comfortable with the procedure and satisfied with the results.

The question frequently arises as to why we estimate source statements instead of executable machine language instructions. The answer is simple and practical. Today, programmers and

analysts can estimate source statements because this is what comes out of their mind and off the tip of their pencil. Few people have any intuitive feel for executable machine langugae statements; the measure does not relate to their thinking or creative process. Both source statements and executable machine language instructions are valid information measures in the Shannon sense – they are ultimately reduced to bits of information in the machine. It is just that today, with most people writing in a language several levels above the machine level, source statements are natural; machine language instructions are not.

THE SOFTWARE EQUATION

The software equation $SS = C_k K^{1/3} t_d^{4/3}$ relates the number of source statements (SS) to the managerial parameters K and t_d. K is the life cycle size in manyears of effort; t_d is the development time in years. These are the Rayleigh/Norden parameters of the overall manpower equation.

$$\dot{y} = K/t_d^2 \cdot t \cdot e^{-t^2/2t_d^2} \text{ people}$$

C_k is a technology constant. It measures any throughput constraints that impede the progress of programmer/analysts – a batch development environment on a production machine (low) versus on-line, interactive program development on a dedicated test-bed machine (high). C_k is quantized. We see this in the data repeatedly. C_k varies in a set sequence of allowable values (Fibonacci sequence). The software equation will not be derived here; an adequate description of that process is contained in IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978.

The important point is that the software equation gives us the linkage between system size, technological tools, effort and schedule. Effort and time are coupled. You cannot change one without changing the other. And the change is dramatic! Rearrange the software equation and you have the trade-off law:

$$\text{Dev Effort} = 0.4 \left(\frac{SS}{C_k}\right)^3 \frac{1}{t_d^4}$$

Note that Dev Effort = 0.4K; i.e., the area under the Rayleigh curve to t_d.

This turns immediately into the software economics law by multiplying by the burdened labor rate, $/MY.

$$\text{Dev Cost} = 0.4 (\$/MY) \frac{SS^3}{C_k^3 t_d^4}$$

All of these parameters can be favorably influenced by management before a project starts. Since they are all power functions and C_k goes up in quantum jumps by a factor of 1.6, then cost improvements by factors of 2 to 10 or more are possible with intelligent planning and good investment sense. The economics of this trade-off law are almost too good to be true. It says take 3 or 4 months longer and cut your cost in half; or better, buy a dedicated test-bed computer, thereby

increasing C_k by 1.6. When you cube this you have cut your cost by a factor of 4. Eliminate 10 percent of the system frills and shrink the number of source statements to 0.90SS. This cuts the cost to 73 percent of the original value. The improvements just cited for a 2-year system that we stretch out to 2.25 years are:

$$\text{Dev Cost}_{\text{before}} = 0.4\,(\$/\text{MY})\,\frac{SS^3}{C_k^3\,2^4} = 0.0625\left(0.4(\$/\text{MY})\,\frac{SS^3}{C_k^3}\right)$$

$$\text{Dev Cost}_{\text{after}} = 0.4\,(\$/\text{MY})\,\frac{(0.9)^3\;SS^3}{(1.6)^3\;C_k^3\;(2.25)^4} = 0.00694\left(0.4\$/\text{MY}\,\frac{SS^3}{C_k^3}\right)$$

The improvement ratio is:

$$\frac{\text{Dev Cost}_{\text{after}}}{\text{Dev Cost}_{\text{before}}} = \frac{0.00694}{0.0625} = \frac{1}{9}$$

The trade-off law is a consequence of system signal-to-noise ratio and bandwidth limitation: when development time ($B - 1/t_d$) is shortened, the bandwidth increases and signal-to-noise ratio decreases (actually, noise increases in the form of more difficult human intercommuncation). With the trade-off indicated, small time decreases soon make a system impossible to do – regardless of how many people or dollars are hurled at it. This is Brooks' Law at play.

We cope with the bandwidth limitation in the form of some empirically observed constraints that relate to the system difficulty K/t_d^2, the initial solpe of our Rayleigh manpower curve. The best measure seems to be the difficulty gradient $|\nabla D| \tilde{} K/t_d^3$. For a certain class of system, (new, stand-alone, etc.), the magnitude of this gradient stays constant.

When we solve the software equation simultaneously with the gradient constraint, we obtain the minimum time that a given size system can be built along with its associated life cycle effort, K(MY); development effort, 0.4K(MY), and development cost ($), $/MY (0.4K). These are expected values, of course, because of the inherent noise in the process.

EMPIRICAL SUPPORT

How can we be sure the software equation and gradient relation work across a broad class of system types and development environments? Classically, we use a set of data to determine the functional behavior, formulate a theory to explain the behavior and then verify the postulated behavior with another independent set of data. In this case, we found the basic behavior from the Army Computer Systems Command data, broadened the range of applicability with the Felix-Walston data (IBM Systems Journal, Vol. 16, No. 1, 1977) and recently have been able to verify the software equation, and gradient relations against the largest collection of software data yet collected. This is the software data base collected by Richard Nelson at Rome Air Development Center. Data for more than 400 systems have been collected and partially analyzed. Of particular interest are the machine generated plots of development effort, development time and average

manpower versus system size in delivered source lines of code (SS). The dependent variables are the management parameters and relate directly to our Rayleigh parameters. Development effort is 0.4K, development time is t_d, and average manpower is $[0.4(K/t_d)]$. When one looks at the Rome data polts (see Figures 1, 2, and 3), one notices the vast range of the independent variable from a hundred or so lines of code (small program) to systems of several million lines. The range of the dependent variables is large also. A clear functional behavior with good correlation is evident, but the value of the functions are severely limited as predictors because of the very large standard deviations. Some attribute this to "poor data." I submit it is inherent in any non-homogeneous data collection spanning many years, different languages, different system types, different design philosophies, etc. The variability combined with the good overall functional character is just what I have observed elsewhere with an independent data set. What it says to me is, "Hey, you've got a parametric variation present or an eigenvalue solution here – no single functional relation can handle it."

When I superimposed the software equation and the gradient constraint relation on the Rome data, I found a remarkably good fit. The slopes of the effort, manpower and duration curves of the functions obtained from the software equation were virtually the same as those determined by the RADC computer. However, no single technology constant (C_k) was capable of spanning the entire Rome data set. Indeed, it took two sequences of six or seven technology constants (ranging from about 600 to 14,000) to do this. No rational range of manpower for one technology constant can span the data range. For example, a range of 1 to 1,000 people working on a project will take in less than 1/2 the data points. The effort data say the same thing. The conclusion is that the real solution has to be parameterized; or be a discrete set of eigenvalues. The software equation, gradient and manpower constraints we have arrived at do span this data set, can rationally explain it, and the functional behavior is virtually the same as the data average; i.e., the Rome data 'proves' the software equation and constraint relations in all practical engineering respects. See Figures 1–5.

SIMULATION AND LINEAR PROGRAMMING

Management answers for effort, schedule and cost can be obtained using two powerful techniques that are well established.

Recall that our PERT estimate of source statements had associated with it a standard deviation reflecting the uncertainty in this estimate (and the nature of the way the programs and modules will be built; i.e., each program could be written many different ways to accomplish the same thing functionally; each of these would have a similar but different information content (bit count). The gradient relations were determined empirically and also had a statistical uncertainty in their determination. Now, if we let these two parameters vary randomly in a simultaneous solution that we run several thousand times, we can generate not only the expected value solutions for K and t_d, but also estimates of the standard deviations (more correctly, standard errors of the estimates). This is extremely valuable, because heretofore we have been totally mired in uncertainty – very precise single-value answers of completely unknown validity. Now, when the track record has been 200–300 percent overruns in cost and 50–100 percent overruns in schedule, decision makers do not believe single-value answers. They want to know the risk – the probability profile – they have been stung too often. In an immature discipline like software development (and the economics of it) managers need the risk information – and are entitled to it.

Linear programming lets us introduce the managerial constraints into the problem. Indeed, we can solve the linear programming problem with only the system size and the managerial constraints of maximum cost, maximum time, maximum peak manpower and minimum peak manpower, since these are all functions of our Rayleigh/Norden parameters; however, we also include system constraints such as difficulty and the difficulty gradient to prevent managers from attempting the impossible. The linear programming solution is possible because we can linearize the relations between our variables by taking logarithms and can express all the relations in terms of the two Rayleigh managerial parameters K and t_d. A two dimensional linear programming problem can be done graphically. Since our relations are linear in logarithms, we do it on log paper. The solution is trivial, but the insight and understanding in being able to visualize the interrelationships is rather profound. The minimum cost solution is immediately evident, the minimum time solution is immediately evident – the duals, maximum time and maximum cost, are also present as they must be in a linear programming solution – and the feasible trade-off range is identified in between the extrema. In being able to invoke this powerful technique, we produce constrained optimal solutions – the best that can be done within the constraints, and all other feasible choices. A graphical linear programming solution along with a brief write-up is attached. It works equally as well in the computerized simplex form. A sample output corresponding to the graphical solution is attached. Ideally, both these approaches should be combined – then managers can interactively iterate optimal solutions graphically on the CRT – using their constraints – until they have the best size, time, cost, manpower combination to meet their needs. With the smart graphics terminals available today, this can be done at negligible cost and time. See Figures 6-8.

SCHEDULE SENSITIVITY

Schedule is the most critical problem in software development. Software development acts like a low pass filter with sharp cut-off characteristics (call it a Rayleigh filter). This means that if the development time is arbitrarily specified by managerial fiat, then there is a high chance the system bandwidth will not match the arbitrary time (bandwidth) specified by management. This means the filter characteristic of the system will shape the input manpower and work profile to match as best it can. Attempts to force the system faster just generate power (manpower) losses. This can all be shown with vector arguments, Fourier analysis, and simulation. All methods give the same results – software development is very time sensitive – development time specification is not the prerogative of management – it belongs to the system. Management must iterate constraints to get into and stay within the feasible (schedule) region. (Fortunately, the linear programming solution bounds this region in time, effort, manpower, and cost.) To get some idea of development time sensitivity, consider the following table generated by simulation showing time sensitivity as a system size for a typical government development environment (CK = 5168).

Size (Source Stmts)		Dev Time (Months)		Dev Effort (MM)		Within Normal Range RADC Data Base?			
Avg	σ	Avg	σ	Avg	σ	MM	Dur	Avg # People	Prod
15,000	1500	12.9	0.6	34.7	6	Y	Y	Y	Y
50,000	5000	21.6	1.1	376.5	60	Y	Y	Y	Y
100,000	10000	29.1	1.4	992.9	152	Y	Y	Y	Y
250,000	25000	43.1	2.1	3188.2	498	N(>)	Y	N(>)	N(<)
500,000	50000	57.9	2.8	7782.3	1204	N(>)	Y	N(>)	N(<)

C_k = 5168, ∇D = 14.7, σD = 2.3

This table tells us that the time window is very narrow. For example, a 15,000 source statement system has a standard error of 0.6 month. If management picks the time at one year (very natural to do) then the probability of successful completion is small. 12.0 - 12.9 = -0.9 month, the no. std errors = -0.9/0.6 = -1.5, and $p\{t_d \leq 12 \text{ months}\}$ = 7% – certainly not odds for the betting man. However, a slip of a few weeks is usually forgivable by managers and customers so we hear little about these cases – nevertheless, the time sensitivity is there, but the absolute magnitude is below most people's response threshold. At the other extreme, 500,000 source statements, it is very hard to guess 57.9 months and 7782 manmonths. More managers and decision-makers would pick 48 months rather than 60 months knowing there is a better chance of getting funding. Yet 48 months is impossible (< 1% probability). Furthermore, 3 standard deviations in time (about 8.5 months) is easy to lose on a 5-year project. There are many external factors that can cause that much delay (late delivery of computer; late start with fixed end date, etc.). The only acceptable solution to this management dilemma is to get realistic time estimates, and then bias them for risk. Managers and decision-makers have to give up guessing schedules if they expect to succeed. The process is too counterintuitive, too time sensitive, making the guessing odds unacceptable.

DYNAMICS AND CONTROL OF THE PROCESS

I have described some good ways to estimate software projects BEFORE THEY START. But software development is a dynamic process. Requirements change. Functional descriptions change. Statutes change. All these things impact an ongoing software project. The change process may be so great that it invalidates a superb earlier estimate of size, cost, and schedule. So regardless of how good our prior estimate is, we still need to know what it is now, based on currently available information. We need a real-time process controller. This is nothing strange to us. It has been done in space operations. For example, we wouldn't think of sending astronauts to the moon if we couldn't measure where they were, compare it to where they should be and then make course corrections. The same concept can be applied to software development. We have a time-varying model that describes the expected manpower trajectory. All we need to do is feed it with the real data in real time so it can update and converge to the true (or present best estimate of the) trajectory. If we feed it manpower data, then we measure and predict resource consumption; but more importantly, if we feed it code production rate, we can measure, update, and predict task accomplishment – rate and % of source code complete. This lets us compare consumption versus accomplishment to see if the rates and predicted times are in agreement – a very important control checkpoint heretofor unavailable. This technique lets us control the process based on the existing system dynamics and make revised estimates of where we are heading.

We can also model the requirements change process in real time just before it takes place. This means decision-makers can know how much the change will cost over the life cycle, and what its slippage consequences are. The tecnhique is to use the 2nd order Rayleigh differential equation, solve it numerically in discrete steps and perturb the driving term by an amount proportional to the % change in the system (% of modules impacted, say). This linear approximation is representative and valid for the noise levels we are working within.

We apply the perturbation at the time the change is to occur and then project ahead to study the new predicted behavior compared to the predicted behavior before the change. Very complex situations can be modeled in this way with excellent indications of the expected response. The managerial insight one gains from this procedure is considerable – the "What if" possibilities

abound – "What if I double my effort for a month?" – "What if I am constrained on computer time for 2 months?" – "What if 25% of the system is cancelled half-way through development?" and so on. Graphical presentation of these situations on the CRT lets them be iterated and solved on-line interactively.

With these specific application techniques applied as described, we have been able to quantitatively come to grips with and produce reasonable engineering answers to the software cost estimating and life cycle control problem. We see that it is a more complicated problem than we would like it to be; yet, when we treat it as the time-varying problem it is, we see the solution is not as difficult as some that have been solved before in other fields – (indeed, we are able to pick up and use the best of those solutions in a number of cases) – and the solution can be easily updated wherever one is in the life cycle. The data requirements are small and occur naturally as a consequence of other normal reporting and record keeping; accordingly, the cost of driving the estimating and control system is negligible. The economics of the software development process is startling. The indications are clear that (apparently innocent) management choices can be made that affect cost by multiples of 5 to 10. With that kind of variation on multimillion dollar projects, managers need to know the choices, sensitivities and influences they can bring to bear – and they need it in numbers – over the whole life cycle.

The managerial questions – "Can I do it? How much? How long? How many people? What's the risk? What's the trade-off? – can be answered with numbers.

FIGURE 1.

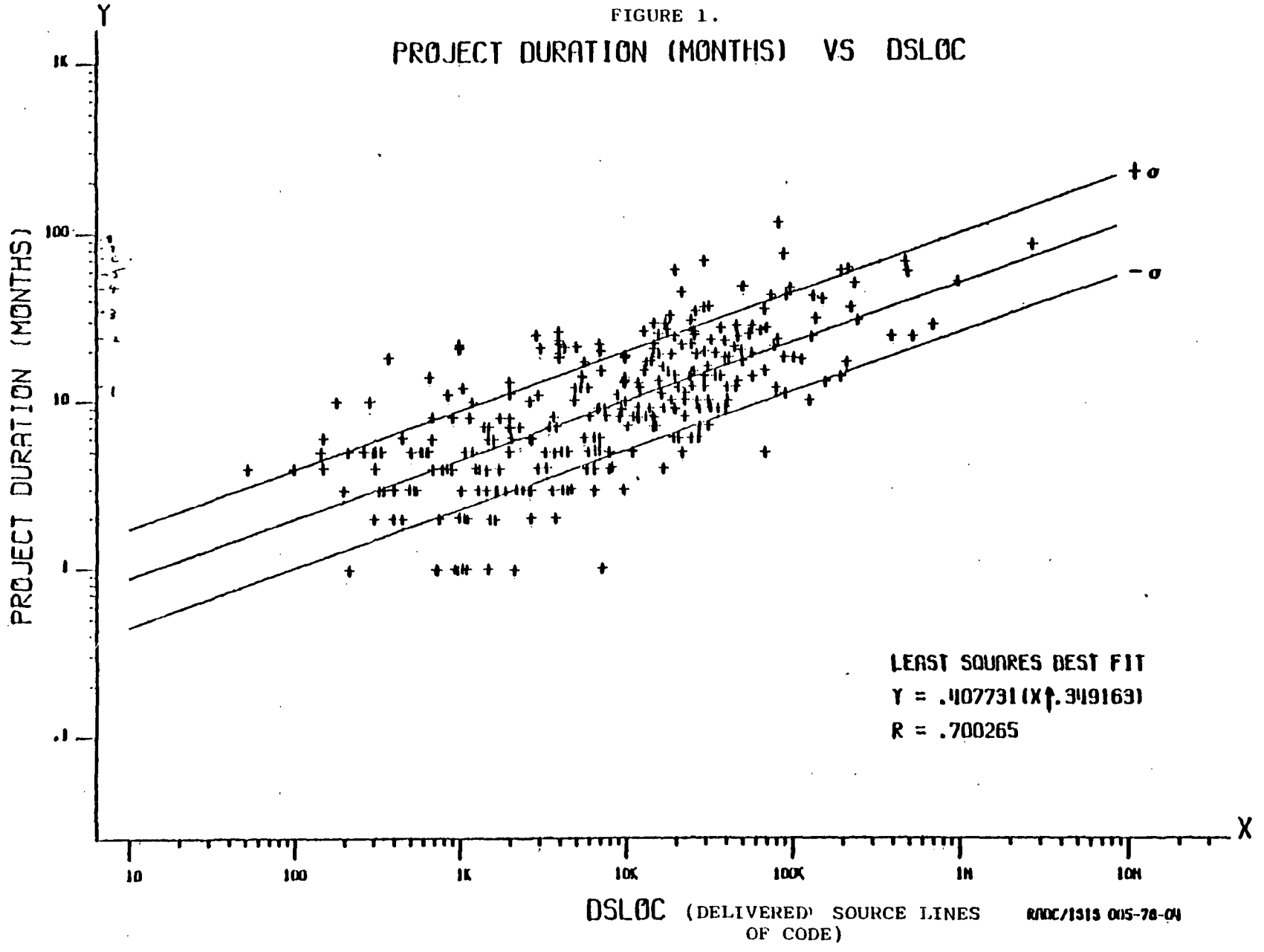

PROJECT DURATION (MONTHS) VS DSLOC
Y
PROJECT DURATION (MONTHS)
1K
100
10
1
.1
+σ
−σ
LEAST SQUARES BEST FIT
Y = .407731(X↑.349163)
R = .700265
X
10
100
1K
10K
100K
1M
10M
DSLOC (DELIVERED SOURCE LINES OF CODE)
RADC/1315 005-78-04

FIGURE 2.

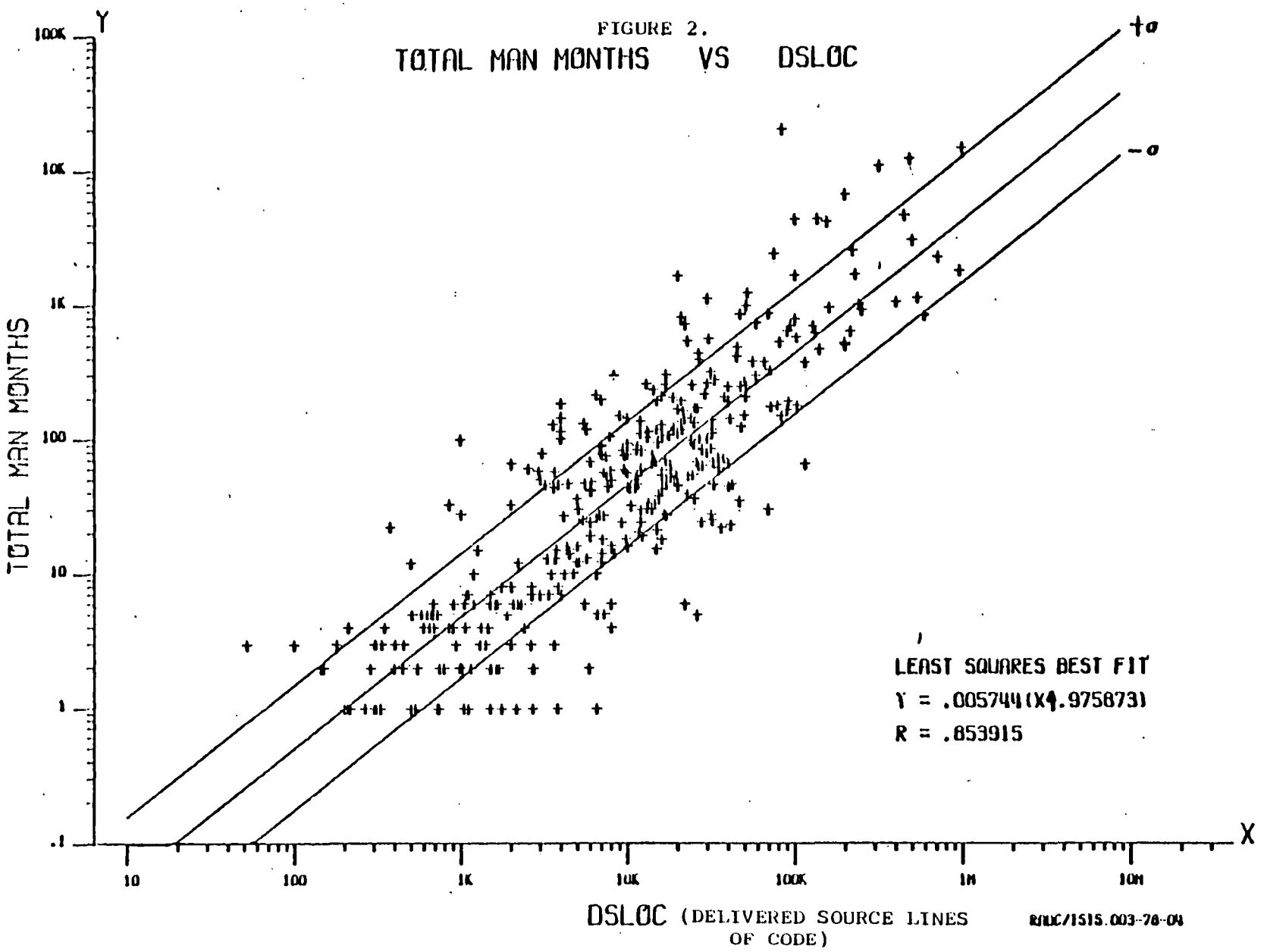

TOTAL MAN MONTHS VS DSLOC
Y
X
TOTAL MAN MONTHS
100K
10K
1K
100
10
1
.1
10
100
1K
10K
100K
1M
10M
+σ
−σ
LEAST SQUARES BEST FIT
Y = .005744(X↑.975873)
R = .853915
DSLOC (DELIVERED SOURCE LINES OF CODE)
RADC/1515.003-78-04

FIGURE 3.

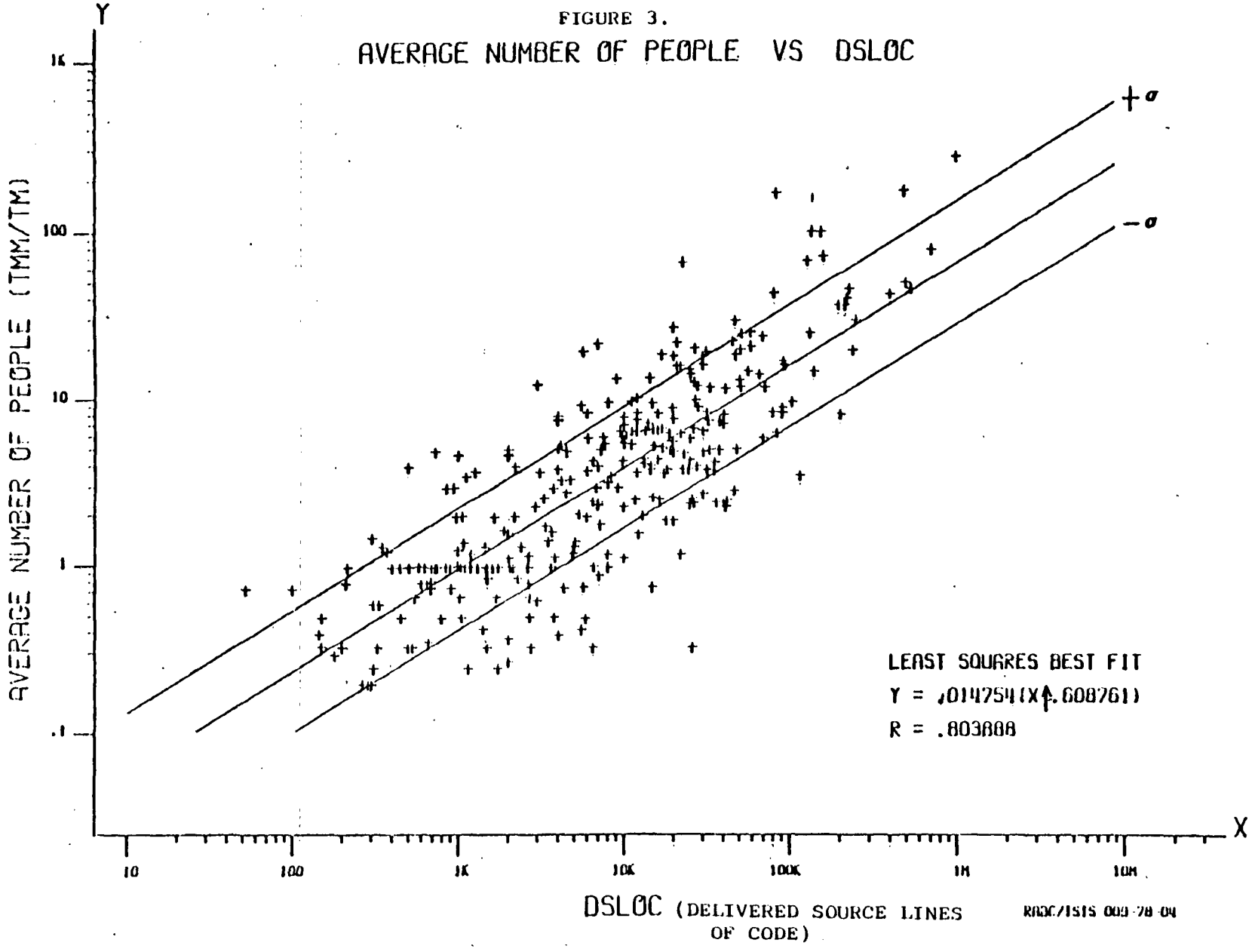

AVERAGE NUMBER OF PEOPLE VS DSLOC
Y
AVERAGE NUMBER OF PEOPLE (TMM/TM)
1K
100
10
1
.1
+σ
−σ
LEAST SQUARES BEST FIT
Y = .014754(X↑.608761)
R = .803888
X
10
100
1K
10K
100K
1M
10M
DSLOC (DELIVERED SOURCE LINES OF CODE)
RADC/ISIS 009-78-04

FIGURE 4.

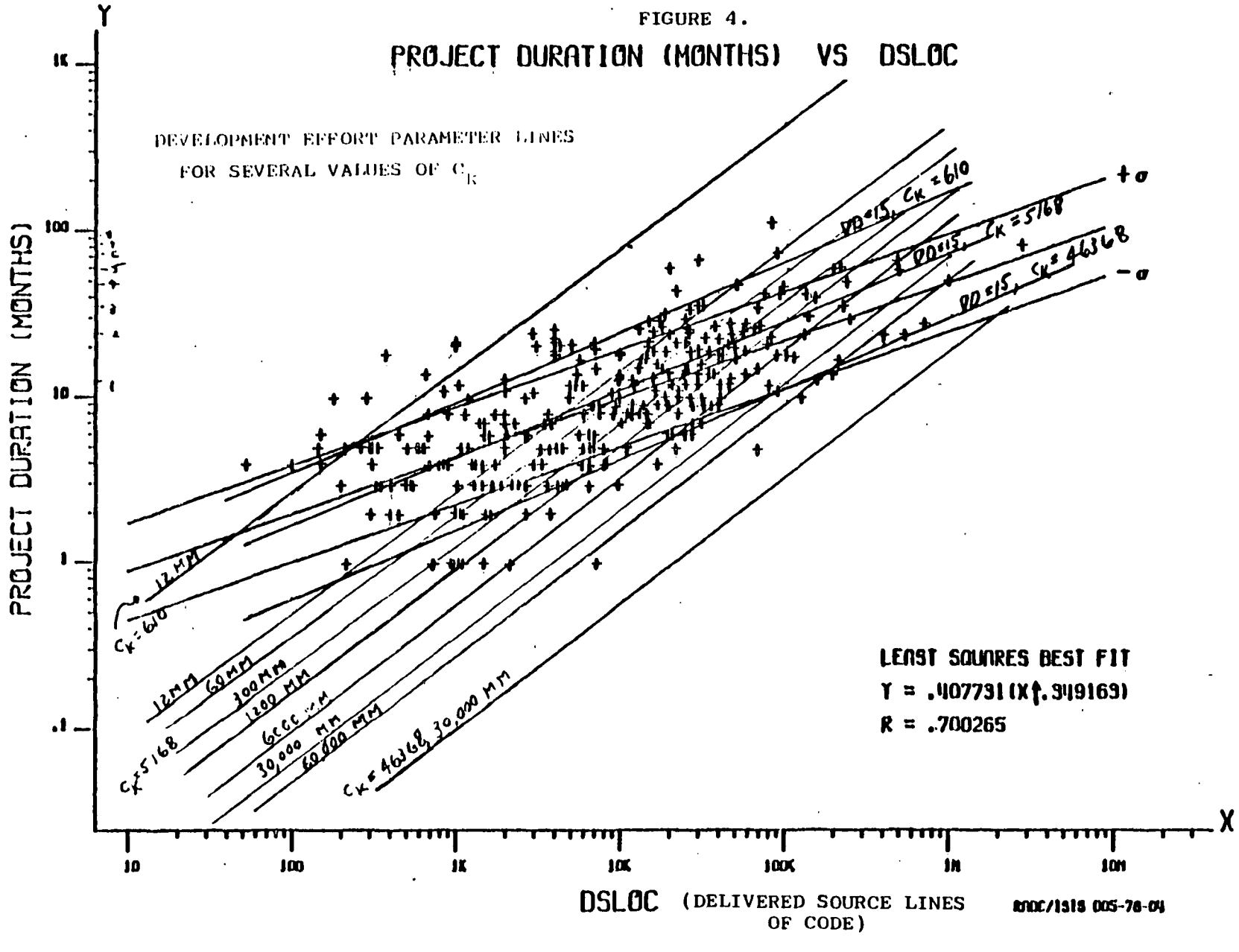

PROJECT DURATION (MONTHS) VS DSLOC
DEVELOPMENT EFFORT PARAMETER LINES
FOR SEVERAL VALUES OF C_K
PROJECT DURATION (MONTHS)
Y
X
1K
100
10
1
.1
10
100
1K
10K
100K
1M
10M
DSLOC (DELIVERED SOURCE LINES OF CODE)
+σ
-σ
PD=15, C_K=610
PD=15, C_K=5168
PD=15, C_K=46368
C_K=610
12 MM
60 MM
300 MM
1200 MM
6000 MM
30,000 MM
60,000 MM
C_K=5168
C_K=46368, 30,000 MM
LEAST SQUARES BEST FIT
Y = .407791 (X↑.349169)
R = .700265
BOOZ/1319 005-78-04

FIGURE 5.

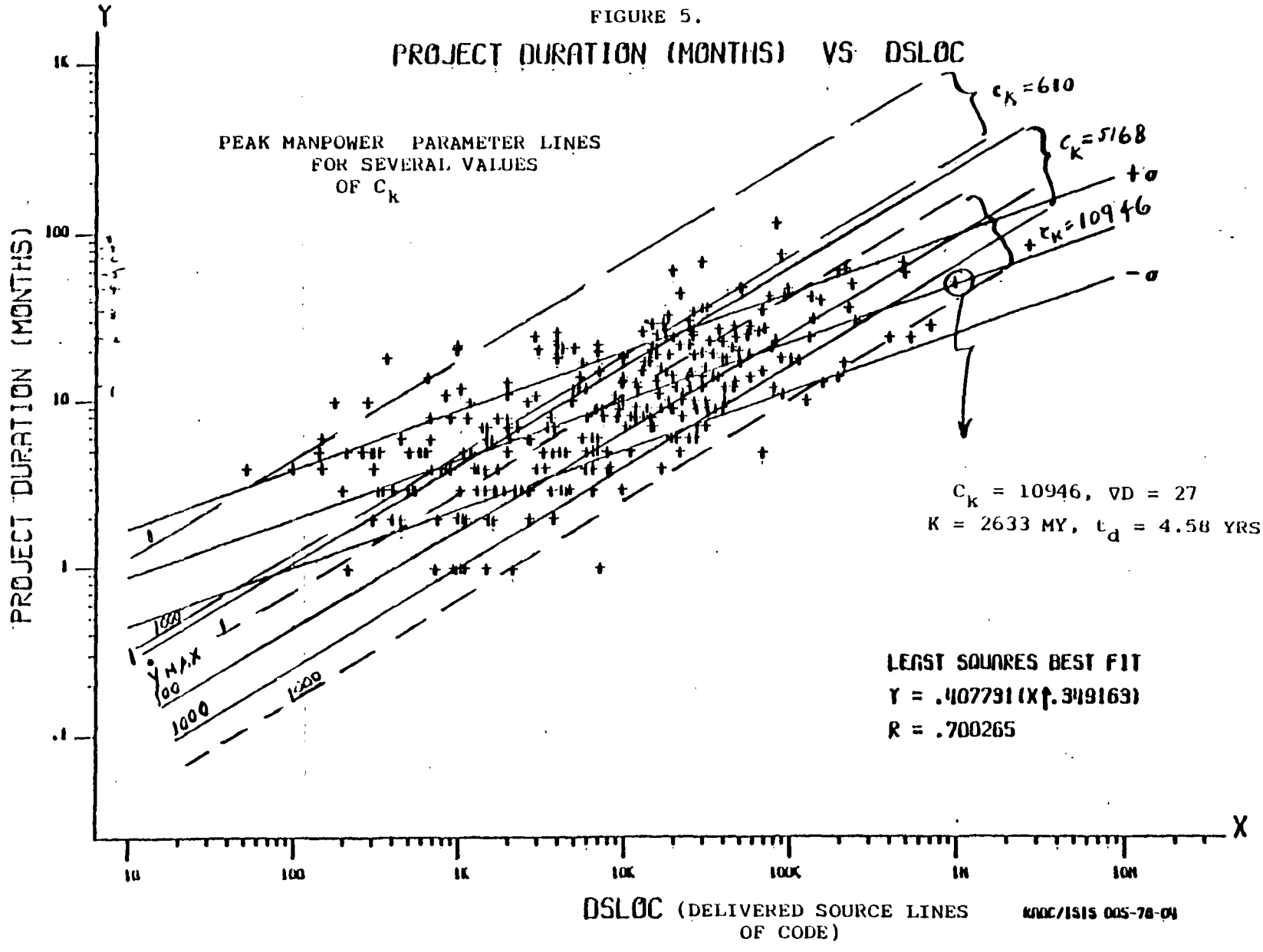

PROJECT DURATION (MONTHS) VS DSLOC
PEAK MANPOWER PARAMETER LINES FOR SEVERAL VALUES OF C_k
C_k = 610
C_k = 5168
+σ
C_k = 10946
−σ
C_k = 10946, VD = 27
K = 2633 MY, t_d = 4.58 YRS
LEAST SQUARES BEST FIT
Y = .407791 (X↑.349163)
R = .700265
PROJECT DURATION (MONTHS)
Y
X
1K
100
10
1
.1
10
100
1K
10K
100K
1M
10M
DSLOC (DELIVERED SOURCE LINES OF CODE)
RADC/ISIS 005-78-04

FIGURE 6.

Linear Programming Alternative

An alternative method for the Rayleigh parameter determination is linear programming. Since we are dealing with only two unknowns, K and t_d, and have a number of constraint conditions involving these parameters, we can easily turn it into a two dimensional linear programming problem which can be solved graphically. The nice feature of this approach is that a number of the constraints can be expressed directly in management terms. Design to cost and design to contract time is possible within the constrained optimization procedure. This procedure is outlined below. The following constraint conditions apply:

$S_s = C_K K^{1/3} t_d^{4/3}$	Software equation
$K/t_d \leq \sqrt{e}\ \dot{y}_{max}$	Maximum peak manpower
$K/t_d \geq \sqrt{e}\ \dot{y}_{max}$	Minimum peak manpower
$K/t_d^2 \leq \lvert D \rvert$	Maximum difficulty
$K/t_d^3 \leq \lvert \nabla D \rvert$	Maximum difficulty gradient
$t_d \leq$	contract delivery time
$\overline{S/MY}\ (.4K) \leq$	Total budgeted amount for development

These constraint conditions can be linearized by taking logarithms and using the simplex method of solving the linear programming problem. The simplest objective functions are cost and time. One generally wants to minimize one or the other of these. Typically we do both and then trade-off in the region in between.

Assume these constraints applied to SAVE:

Number of S_s = 98475

Maximum development cost $\leq$ $2 million

Maximum time (contract delivery) $\leq$ 2 years

Maximum manpower available at peak manning (hiring constraint, say) $\leq$ 28 people

Minimum manpower you desire to employ at peak manning $\geq$ 15 people

Maximum difficulty gradient $\leq$ 15

Maximum difficulty $\leq$ 50

Minimum productivity $\geq$ 2000 S_s/ MY

These translate into:

$$1/3 \log K + 4/3 \log t_d = \log 98475 - \log 10040$$

$$\log K = \log (2 \times 10^5/5 \times 10^4(.4))$$

$$\log t_d = \log 2$$

$$\log K - \log t_d = \log (\sqrt{e}\ 28)$$

$$\log K - \log t_d = \log (\sqrt{e}\ 15)$$

$$\log K - 3 \log t_d = \log 15$$

$$\log K - 2 \log t_d = \log 50$$

$$\log K = \log (98475/\ .4(2000))$$

The intersection of these lines bound the feasible region. An optimal solution will be at some intersection point. Further, because of the equality constraint it must be along the S_s = 98475 line. The limiting conditions in this case are: , $t_d \leq$ is 2 years, maximum peak manpower $\leq$ 28 people and S_s = 98475 source statements. Figure 4 shows the solution.

Reading off the solutions we see that:

	t_d (yrs)	K (MY)	E (MY)	$\overline{PR}$ (S_s/MY)
Minimum Time	1.83	84	33.6	98475/33.6 = 2931
Minimum Cost	2.0	61	24.4 $1.22M	98475/24.4 = 4036

Trade-off is possible along the S_s line between t_d = 1.83 years, K = 84 MY and t_d = 2 years, K = 61 MY without violating constraints.

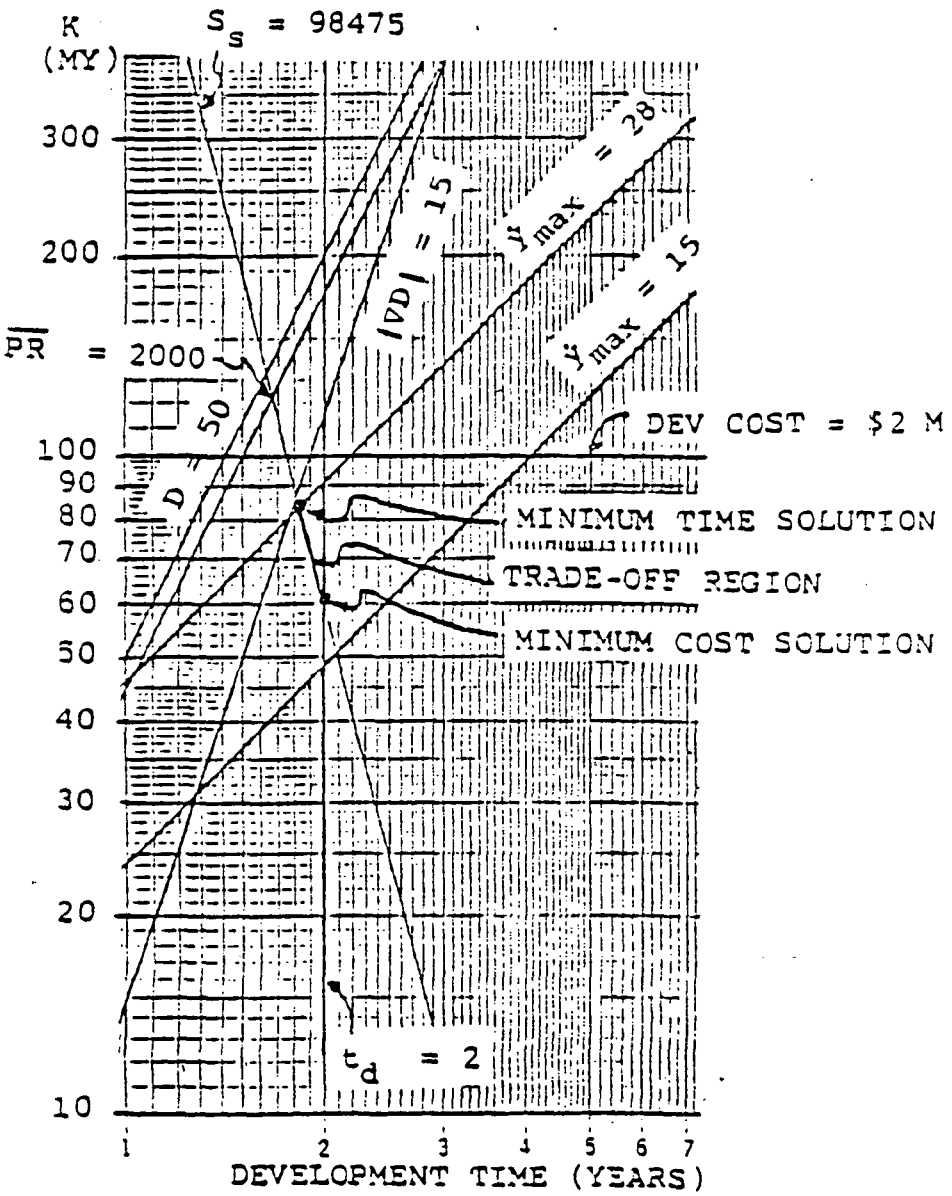
Here it is easy to see the counter intuitive nature of productivity. Note that productivity increases with development time because the required effort (E) goes down as time is increased.

One other point is important. If the technology constant is smaller, the S_s = 98475 line would shift parallel to the right (direction of increasing time). If the constraints remained numerically the same, the feasible region would change because of the relocation of the S_s line. The time constraint could probably not be met and a relaxation of that constraint would have to be sought.

This is a deterministic solution. However, by extending the idea of simulation, the linear programming concept can be embedded within a simulation and the uncertain constraints can be allowed to vary randomly about their mean values and the statistical uncertainty for the minimum time and minimum cost solutions can be obtained by running the problem a few thousand times.

FIGURE 7.

LINEAR PROGRAMMING SOLUTION
FOR SAVE



```
**********************************************************************
                          LINEAR PROGRAM                          20
**********************************************************************
TITLE:  SAVE                                        DATE: 17-Jan-79

   THIS FUNCTION USES THE TECHNIQUE OF LINEAR PROGRAMMING (SIMPLEX ALGORITHM)
   TO DETERMINE THE MINIMUM EFFORT (AND COST) OR THE MINIMUM TIME IN WHICH
   A SYSTEM CAN BE BUILT.  THE RESULTS ARE BASED ON THE ACTUAL MANPOWER, COST,
   AND SCHEDULE CONSTRAINTS OF THE USER, COMBINED WITH THE SYSTEM CONSTRAINTS
   YOU HAVE PROVIDED EARLIER TO YIELD A CONSTRAINED OPTIMAL SOLUTION.

 ENTER THE MAXIMUM DEVELOPMENT COST> 2000000

 ENTER MAXIMUM DEVELOPMENT TIME IN MONTHS> 24

 ENTER THE MINIMUM AND MAXIMUM NUMBER OF PEOPLE YOU
 CAN HAVE ON BOARD AT PEAK MANLOADING TIME> 15,28
```

	TIME	EFFORT	COST (X $1000)
MINIMUM COST	24.0 MONTHS	278. MM	1159.
MINIMUM TIME	21.9 MONTHS	399. MM	1662.

```
YOUR REALISTIC TRADE-OFF REGION LIES BETWEEN THE LIMITS OF THE TABLE ABOVE.

(INTERPOLATION IN THE TRADE-OFF TABLE BETWEEN THESE LIMITS WILL PRODUCE ALL
ACCEPTABLE ALTERNATIVES.  WOULD YOU LIKE TO SEE A TRADE-OFF ANALYSIS WITHIN
THESE LIMITS (Y OR N) ? Y
```

TIME	MANMONTHS	COST (X $1000)
21.93	399.	1662.
22.43	364.	1519.
22.93	334.	1390.
23.43	306.	1276.
24.00	278.	1159.

FIGURE 8.