f^2 COCOMO: Estimating Software Project Effort and Cost

Allan Caine and Anne Banks Pidduck
School of Computer Science, University of Waterloo, Waterloo, Ontario Canada
{adcaine, apidduck}@cs.uwaterloo.ca

Abstract

The COnstructive COst MOdel, COCOMO, was developed to estimate the effort measured in staff-months to complete a software project. Given an estimate of the cost per staff-month, COCOMO can be used to estimate the cost of developing software. All business enterprises involved in developing software must know their costs to maintain their long-term viability. However, COCOMO has one significant drawback. It requires that the size of the project be measured in lines of code, but function points are a better metric in measuring project size compared to lines of code. In the event that the size of the project is measured in function points, COCOMO uses a function points to lines of code converter. We present a new model which uses function points as a direct input into the model. By using actual software project data, we show that the software project data can be analyzed on a programming language by programming language basis. We claim that our proposed model would take the programming language into account through pre-computed constants. We conclude that our model is superior to the existing model because it eliminates the errors introduced by arbitrary function point indices and replaces them with constants that are scientifically and statistically verifiable.

1. Introduction

This paper begins by briefly explaining the COCOMO 81 and COCOMO II models. In section 2, we show that both models have a major deficiency. They cannot take function points as a direct input. Yet function points are a superior metric to measuring project size compared to lines of code [3]. Instead, a function point index is used to convert the function points to an equivalent number of lines of code. Unfortunately, the values of these indices are not universally agreed upon and any errors in the values of these indices materially affect the estimate of the software project effort and consequently the estimated cost of producing the software.

In section 3, we propose a new model, which we call f^2 COCOMO. Our proposed model takes function points as a direct input. We claim that our proposed model is superior because it eliminates the arbitrary function point indices. With the elimination of the function point indices in our model, however, we introduce a problem. The model must have some way of accounting for the programming languages.

In section 4, we present our experimental results, which show that programming language can be accounted for through pre-computed constants. In the first phase of the experiment, we start with a small data set from the US Army. Our intent is to show that the Gauss-Newton method is appropriate for re-computing the constants used in the COCOMO 81 formula. Our experimental results confirm that this method is appropriate. We then go on to analyze Boehm's data [1] which consists of 63 software projects. We separated the data by programming language, and re-computed the constants in the COCOMO 81 model. Our experimental results confirm that we can indeed re-calibrate COCOMO 81 for each programming language and reduce the sum of the absolute value of the errors.

We end our paper with our conclusions. First, we conclude that our proposed model f^2 COCOMO is quite workable. Second, we conclude that analyzing data on a programming language by programming basis is a sensible approach to developing a model which uses function points as its primary input in estimating project effort. Together with an estimate of the cost per staff-month, an estimate of the total cost can be found as the product of the number of staff months multiplied by the cost per staff month. We indicate that further research needs to be conducted to improve COCOMO II. COCOMO II needs to be improved so that it can take function points as a direct input.

2. COCOMO 81 and COCOMO II

The COnstructive Cost MOdel, COCOMO, was developed to better estimate software cost, effort, and time to development. The formula for COCOMO 81 is $SM = aS^b \times EAF$, where SM is the effort measured in staff months, S is thousands of lines of code estimated to complete the project, EAF is the effort adjustment factor, and a and b are constants to be determined. The determination of the value of EAF is fully explained by Boehm [1]. As explained in section 4, we will assume that $EAF = 1$.

The values of a and b depend upon the mode of the software project. For our purposes, we will assume the project is 'embedded.' Using this mode, Boehm computes $a = 3.6$ and $b = 1.20$. Our choice of mode is arbitrary. However, it is important to choose the same mode consistently throughout the experiment to achieve comparable results.

For COCOMO II, the formula is $SM_2 = 2.94S^{(0.91+W)}EM$, where SM_2 is the effort measured in staff months, S is the predicted size of the project measured lines of code, W is an adjustment to the exponent of S, and EM is the effort multiplier. The determination of the values of W and EM is fully explained in the Model Definition Manual [2].

The major deficiency in both models is that the formulas require that the software project's size be measured in lines of code. The paper written by Albrecht and Gaffney [3] concludes that function points are a superior metric in measuring project size compared to lines of code. However, if the project's size is measured in function points, the function points must be converted to an equivalent number of lines of code.

We graphically depict how the current COCOMO model works in Figure 1.

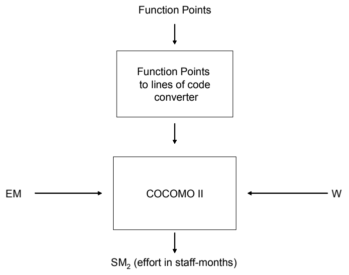

Function Points
Function Points to lines of code converter
EM
COCOMO II
W
SM_2 (effort in staff-months)

Figure 1. COCOMO II as it currently stands with the conversion of function points to lines of code.

To convert function points to lines of code, a function point index is used. A function point index is a ratio of the number of lines of code per function point. The indices are different for different programming languages. Thayer [5] provides such a table of indices; however, his table is not in agreement with the table used in the Model Definition Manual [2].

These differences are no small matter. In their paper, Musilek et al. [4] demonstrate that the COCOMO II model is most sensitive to errors in the size variable, S. So, if the function point index is in error, the estimate of the project effort, SM, can also be in error. Since the cost of the software project is a function of the software project effort, the projected cost of the software can also be in error.

3. Our Proposal—f^2 COCOMO

Our alternative model is depicted below in Figure 2.

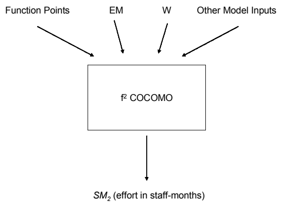


Figure 2. Our proposed model f^2 COCOMO, which takes function points as a direct input.

In our proposed model, function points, EM, W, and possibly other model inputs are taken as a direct input. The function point to lines of code converter is not used. The box labeled 'f^2 COCOMO' is the mathematical formula, which computes SM_2 from function points, EM, W, and possibly other model inputs.

4. Experimental Results

In our experiments, we considered basic COCOMO 81. Basic COCOMO 81 assumes that $EAF = 1$. We chose basic COCOMO for two reasons. First, we wanted to make the analysis more tractable by having fewer parameters. Second, we knew that we ultimately wanted to use Boehm's original data, which was used to find the formula for COCOMO 81—not COCOMO II. To be consistent in our experiment, COCOMO 81's mathematical formula was the appropriate choice.

The experiment involved re-calibrating COCOMO. Re-calibration means re-computing the values of the parameters a and b in $SM = aS^b \times EAF$.

The experiment was conducted in two phases. In the first phase, we wanted to check that the Gauss-Newton method was appropriate for computing the parameters a and b of the equation $SM = aS^b \times EAF$. We chose a small set of 15 projects found in Conte et al [6]. We found that the Gauss-Newton method was appropriate.

In the second phase of our experiment, we turned to Boehm's data set [1]. His data set consists of 63 software projects. We divided his data by programming language and recalibrated COCOMO for each programming language. We were successful except in two cases. The reasons for failure are explained in section 4.2.

Roughly speaking, the Gauss-Newton method begins by assuming a form of the solution and taking an initial guess of the parameters a and b. Iteratively, the method computes the 'goodness-of-fit' of the current parameters, and computes new and better parameters. As it iterates, the method (hopefully) converges upon the 'correct' solution. Of course, there will always be differences between the model's predicted values and the actual values unless the assumed form of the solution can entirely explain the relationship between the independent and dependent variables.

4.1 US Army Data

The results of the first phase of our experiment are shown in Figure 3. The o's represent data points; the solid line, the recalibrated model; and the dashed line, Boehm's embedded model. Through re-calibration using the Gauss-Newton method, we reduced the sum of the absolute values of the error by 1.3% from 9,722 staff-months to 9,594 staff-months. The Gauss-Newton method determined that $a = 0.9239$ and $b = 1.4064$ were better constants. We concluded that the Gauss-Newton method is appropriate.

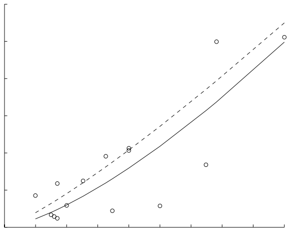


Figure 3. Results of our analysis of the US Army Data.

4.2 Boehm's Data Set

Confident that our method was feasible, we turned our attention to Boehm's data set [1]. His data set is the data he used originally to derive the COCOMO 81 model. It consists of 63 software projects written in seven different programming languages.

We began this phase of our experiment by separating the data by programming language. Next, for each programming language, we recalibrated a and b using the Gauss-Newton method. Our results are given in Table 1.

Table 1. Recalibration Results

Language	Number of Projects	a	b
Fortran	24	46.4489	0.7116
Machine Languages	20	10.0019	1.1338
Cobol	5	76.6506	0.5635
Jovial	5	0.0066	2.4187
PL\I	4	no results	no results
Higher Languages	3	3.1613	1.1927
Pascal	2	no results	no results

Next, we computed the sum of the absolute values of the errors: first for Boehm's embedded model and second for the re-calibrated model. In every case, other than machine languages, we found that the error was reduced under calibration. For the machine languages, the error increased by 2.3%—a negligible amount.

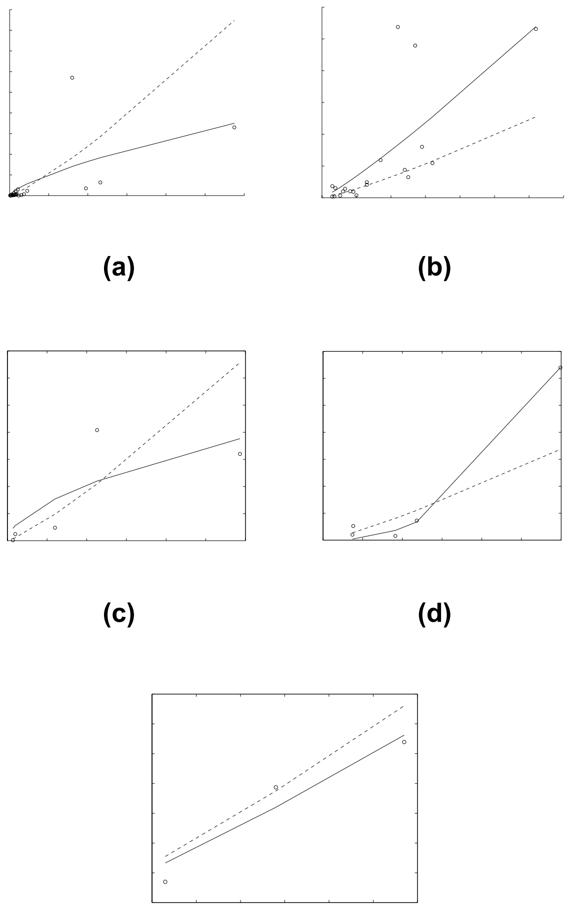
For the PL\I language, we found that the Gauss-Newton method did not converge. So, no results are available. For the Pascal language, there were not enough data points. The error results are given in Table 2.

Table 2. Errors under re-calibration

Language	Error not Calibrated	Error Calibrated	Change
Fortran	2,947.1	2,080.5	29%
Machine Languages	2,804.4	2,868.9	(2.3%)
Cobol	3,021.3	2,130.3	29%
Jovial	4,377.6	894.0	79%
PL\I	no results	no results	no

			results
Higher Languages	219.1	153.3	30%
Pascal	no results	no results	no results

Figure 4 presents the results of re-calibration graphically. The o's represent data points; the solid line represents the re-calibrated model; and the dashed line represents Boehm's embedded model.


(a)

(b)

(c)

(d)

(e)

Figure 4. The o's are data points; the solid line, the re-calibrated model; and the dashed line Boehm's embedded model (a) Fortran; (b) Machine Languages; (c) Cobol; (d) Jovial; and (e) Higher Languages.

5. Conclusions

By experimentation, we see that software project data can be analyzed on a programming language basis. The different programming languages are reflected in the constants *a* and *b*.

This suggests that f^2 COCOMO is quite workable. We regret that Boehm's data [1] does not include function point information. Otherwise, we would have continued our analysis to show what, if any, relationship exists between function points and project effort.

In this paper, we show that a model can be derived for each programming language by simply separating the data on a programming language basis. Suppose we have project data which relates function points to project effort. If that data is separated by programming language and analyzed, then a function point model, f^2 COCOMO, can be derived for each programming language.

Further research needs to be conducted to improve COCOMO II so that it can take function points as a direct input.

6. References

[1] Boehm, B.W: Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ (1981)

[2] Center for Software Engineering, Computer Science Department, University of Southern California: Overall Model Definition. (2003) ftp://ftp.use.edu./pub/ soft_engineering/ COCOMOII/cocomo99.0/modelman.pdf Accessed November 19, 2003.

[3] Albrecht, A.J., Gaffney, J.E.: Software function, sources lines of code, and development effort prediction: A software science validation. In IEEE Transactions on Software engineering, SE—9(6):639 – 652, November 1983.

[4] Thayer, R.H.: Software Engineering Project Management. 2E. Edwards Brothers Inc. (2003)

[5] Musilek, P., Pedrycz, W., Sun, N., Succi, G.: On the sensitivity of COCOMO II software cost estimation model. In IEEE Symposium of Software Metrics. (2002) 13 – 20

[6] Conte, S.D., Dunsmore, H.E., Shen, V.: Software Engineering Metrics and Models. Benjamin/Cummings Pub., Menlo Park (1986)