Zipf-Mandelbrot Law and Software Science

Hongyu ZHANG

School of Software, Tsinghua University

Beijing 100084, China

PACS. 89.75.Da – Systems obeying scaling laws.

PACS. 89.75.-k – Complex Systems.

Abstract. – Zipf's law, or more generally, the Zipf-Mandelbrot law, is an empirical law behind many natural and social phenomena. We discover that the distribution of lexical tokens in modern computer programs follows the Zipf-Mandelbrot law. Based on this discovery, we revise the length equation in Halstead's software science theory. Our Zipf-Mandelbrot law based equation can better model the relationship between vocabulary and length of programs. The results are obtained through empirical analysis of real-world software systems. We believe our study reveals the statistical regularities behind computer programming.

Introduction. – Zipf's law [21] is an empirical law obtained from statistical natural language analysis. During the study of natural languages, Zipf discovered that a small number of words (e.g., "a", "the") in a text occur very often and many words occur rarely, and the frequency of use of word is roughly inversely proportional to its rank. If we sort the words by their frequency of occurrences (most frequent gets rank 1 and least frequent gets last rank), we have the following Zipf's law:

$$f \approx \frac{C}{r^a} \qquad (1)$$

where f is the frequency of a word, r is its rank, a is close to 1, and C is a constant. Zipf's law is quite accurate except for very high ranks. Mandelbrot [12] proposed a more general form of the law, the Zipf-Mandelbrot law:

$$f = \frac{C}{(r+b)^a} \qquad (2)$$

where a new constant b is added to the denominator. A reasonably small value of b will lower the frequencies of the first few ranks. The original Zipf law is a special case of Zipf-Mandelbrot law when $b = 0$. It is observed that Mandelbrot's modification provides a better fit of data from natural language text (including data with very high ranks) [5].

Although originated in natural language study, Zipf's law and its general form Zipf-Mandelbrot law are widely observed in many other disciplines including physics, biology and economics. For example, firm sizes [3], city sizes [21], and website hits [1] are all reported to follow Zipf distribution. It is also observed that the distributions of scientific citations [18] and metabolic fluxes [4] can be described by the Zipf-Mandelbrot law.

Like natural language texts, computer programs also consist of words (tokens) that are separated by the delimiters. Like natural languages, programming languages also define rules (grammar) for composing the words. After all, computer programs and natural language text are all human artifacts. This motivated us to discover if the Zipf's law also holds for computer programs. We conduct experiments on a number of real-world software systems and the results confirm our hypothesis - that the distribution of token occurrences in programs follows the Zipf's law, or more precisely, the Zipf-Mandelbrot law.

Having understood the nature of token distribution, we re-examine the software science theory (also called "software physics" or "thermodynamics of algorithms") [9]. The software science theory is the first analytical "laws" for computer software [15] that is based on direct measurement of tokens in programs. In this research, we derive a new software science length equation based on the Zipf-Mandelbrot law, and show that the new equation can better describe the relationship between program vocabulary and program length.

Zipf's Law in Software. – In this study, we explore if the Zipf's law exists in computer programs. A modern computer program usually consists of white spaces, comments and tokens. The tokens could be: a) *identifiers* that are defined by are programmers for naming packages, classes, interfaces, methods, variables and constants; b) *keywords* that are reserved words with pre-defined meanings; c) *literals* that represent constant values of a primitive data type; d) *operators* that denote arithmetic/relational/logical operations; and e) *separators* that separate two tokens. All these lexical tokens can be parsed by the language compilers. In this research, we consider a token as a *word* if it is an identifier, a keyword, a literal, or an operator. We define a *Vocabulary* of a program as the set of distinct words in the program.

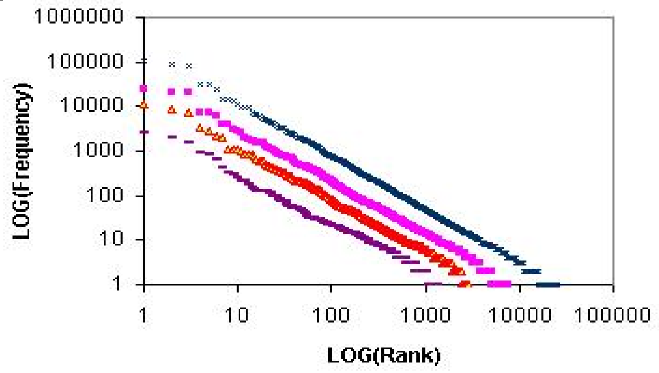
The data used in our analysis is collected from eighteen Java, C, and C++ software systems of different sizes (Table 1). These systems include graphical packages, editors, web servers, class libraries, and utilities. Their source code is available for public, which makes the validation of results possible.

Different tokens may have different frequencies of occurrences. For example, the keyword "int" may occur more often than "long", and the operator "=" may occur more often than "%". For each software system, we collect the tokens appeared in the source code and sort them according to their frequencies (i.e., the number of occurrences). Tokens that have the same frequency are assigned with different ranks. We then plot the frequencies against ranks on a log-log diagram. The results for some systems are shown in Figure 1. We can see that the distribution curves are close to straight lines, exhibiting Zipf's behavior (except for very high ranks).

Further analysis shows that the data can be better described by the Mandelbrot's modification of Zipf's law. To obtain the Zipf-Mandelbrot law parameters, we adopt the method proposed in [13]. Firstly we assume that the distribution follows the Zipf law (i.e., b is 0), and calculate C and a by performing least-squares based linear regression analysis. We then increase b in small steps (e.g., 0.1) until the goodness of fit of Equation (2) applied to the first few ranks (those that are considerably below the straight line) stops improving. As an example, Figure 2 shows the distribution of tokens in the jEdit system, with Zipf and Zipf-Mandelbrot regressions curves. The regression parameters are shown in Table 2. We use R^2 and Standard Error of the Estimate (S_e) to evaluate the goodness of fit. The Zipf law results in R^2 value 0.852 and S_e value 1367.6, while the Zipf-Mandelbrot law results in R^2 value 0.959 and S_e value 135.2. Therefore, the Zipf-Mandelbrot law fits the data better, especially for the very high ranks.

Table 1. The studied software systems[1]

Software	Lines of Code	Description
Jena	121K	A framework for writing Semantic Web applications
Jakarta-Tomcat	161K	The Apache Tomcat servlet container
Ant	105K	The Apache build tool
Swing	152K	The Swing package in JDK5.0
jEdit	88K	A program text editor
Jetty	44K	A Java HTTP server and servlet container
jHotdraw	15K	A Java GUI framework
DrJava	59K	A Java IDE tool
Protégé	69K	An ontology editor and knowledge framework
Cocoon	76K	A web development framework
JavaCC	14K	A Java compiler-compiler
JUnit	4K	A framework for Java unit testing
Bash	89K	The GNU Bourne Again Shell
Apache	75K	The Apache Web Server
Make	18K	The Unix make utility
CppUnit	11K	A C++ Unit Test Library
NPP	77K	Notepad++, a C++ source editor
MFC	110K	Microsoft Foundation Classes C++ library



*The systems shown are, from top to bottom: Tomcat, Jetty, jHotdraw and jUnit.

Fig. 1: The distribution of tokens in programs

[1] The websites of the studied systems are, in the same order of references: jena.sourceforge.net, jakarta.apache.org/tomcat, ant.apache.org, java.sun.com, www.jedit.org, jetty.mortbay.org/jetty, www.jhotdraw.org, drjava.org, protege.stanford.edu, cocoon.apache.org, javacc.dev.java.net, junit.org, www.gnu.org/software/bash, www.apache.org, sir.unl.edu, cppunit.sourceforge.net, notepad-plus.sourceforge.net, msdn2.microsoft.com.

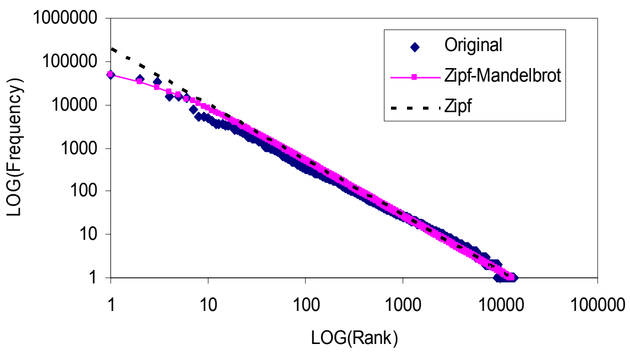


Fig. 2: The distribution of tokens in jEdit

Table 3 shows the Zipf-Mandelbrot parameters for all studied systems, the power-law exponent *a* ranges from 1.14 to 1.30. The R^2 values range from 0.902 to 0.977, indicating good fitness of the Zipf-Mandelbrot law.

Table 2. The parameters of Zipf and Zipf-Mandelbrot laws for jEdit.

	C	*b*	*a*	R^2	S_e
Zipf	201488	-	1.29	0.852	1367.6
Zipf-Mandelbrot	201488	2.0	1.29	0.959	135.2

Table 3. The parameters of Zipf-Mandelbrot law for all studied systems.

	C	*b*	*a*	R^2
Jena	205869	1.2	1.22	0.954
Tomcat	347049	2.1	1.28	0.927
Ant	108555	0.5	1.17	0.934
Swing	360767	3.0	1.28	0.939
jEdit	201488	2.0	1.29	0.959
Jetty	108435	2.0	1.30	0.948
jHotdraw	30268	1.8	1.26	0.957
DrJava	109911	1.0	1.22	0.948
Protégé	138347	1.6	1.25	0.967
Cocoon	202283	1.5	1.30	0.969
JavaCC	17718	0.3	1.20	0.902
JUnit	5151	1.0	1.16	0.977
Bash	102092	0.9	1.20	0.969
Apache	115884	1.5	1.22	0.957
Make	38957	1.5	1.29	0.973
CppUnit	17943	2.0	1.21	0.974
NPP	435008	0.4	1.14	0.962
MFC	225265	4.0	1.24	0.965

Software Science. – Software science [9] is a theory proposed by late Maurice Halstead of Purdue University to measure various properties of programs. Software science theory attempts to provide a theoretical foundation to computer software. It considers a program as a series of tokens that is classified as either "operators" or "operands". The four basic measurements in software science are as follows:

n_1 = The number of unique operators
n_2 = The number of unique operands
N_1 = The total number of occurrences of operators
N_2 = The total number of occurrences of operands

The operators include keywords, arithmetic/relational/logical operators, and separators. The operands include all program variables and constants (literals). Figure 3 shows an example of a simple Algol program for calculating the greatest common divisor of two numbers (Euclid's Algorithm) [9]. Its operators and operands are shown in Table 4.

```
        IF (A=0)
LAST: BEGIN GCD := B; RETURN END;
        IF (B=0)
        BEGIN GCD := A; RETURN END;
HERE: G := A/B; R:= A-B*G;
        IF (R=0) GO TO LAST;
        A := B; B := R; GO TO HERE
```

Fig. 3: A simple Algol program

Operator	Number of occurrences	Operand	Number of occurrences
;	9	B	6
:=	6	A	5
( ) or BEGIN..END	5	O	3
IF	3	R	3
=	3	G	2
/	1	GCD	2
-	1		
*	1		
GOTO HERE	1		
GOTO LAST	1		
$n_1 = 10$	$N_1 = 31$	$n_2 = 6$	$N_2 = 21$

Table 4. Counting the number of operators and operands in Fig. 3

Halstead also defines metrics for measuring various program properties (such as program volume, language level, programming effort, etc.) based on these basic measurements. Among these properties, the program *vocabulary* `n` and the program *length* `N` are of particular importance as they provide foundations to other metrics. They are defined as follows:

$$n = n_1 + n_2$$

$$N = N_1 + N_2$$

Halstead hypothesizes that the program length N can be estimated as follows:

$$N\hat{} = n_1 * \log_2 n_1 + n_2 * \log_2 n_2 \quad (3)$$

As a basis of software science theory, the length equation claims that the program length is a function of the vocabulary. It tries to establish a link between the counts of unique operators/operands and the count of their occurrences.

Since its official publication in 1977, software science has drawn widespread attention from the computer science community, and has grown from measuring the algorithms to measuring entire software systems. A large number of research reports and papers can be found in the literature (for example, a special issue of IEEE transactions on software engineering in March 1979). Some reported experimental supporting evidences [8, 11, 19] while others raised serious questions about its validity [10, 17]. Despite the disagreements, software science has influenced many software engineering methods. For example, the well-known "function points" [2] software size estimation method has a strong theoretical support based on Halstead's software science. As one of the earliest software metrics, software science is still introduced in recent textbooks (e.g., [14], [15], [20]) and implemented by many today's commercial software metric tools.

Revised Software Science Length Equation. – For each of the systems in Table 1, we count the number of tokens and the numbers of unique operators and operands in all programs. We find that the length equation (3) does not hold for modern software systems. Table 5 shows the evaluation results for all the systems studied. We use MRE (Magnitude of Relative Error) to measure the accuracy of estimation, which is computed as (|N^-N|/N). The MRE values reported by software science estimation all fall beyond 25%, indicating large estimation derivations.

Based on the Zipf-Mandelbrot law, we could derive a new length equation as follows:

$$N\hat{} = \int_1^n \frac{C}{(r+b)^a} dr = C\left[\frac{(r+b)^{1-a}}{(1-a)}\right]_1^n$$

$$= C\frac{(n+b)^{1-a} - (1+b)^{1-a}}{(1-a)} \quad (4)$$

Let *p = 1-a*, *q = 1+b*, and because *n>>b*, the Equation (4) can be transformed into:

$$N\hat{} = C\frac{n^p - q^p}{p} \quad (5)$$

Equation (5) provides a new way of modeling "vocabulary-length" relationship in the software science theory. The evaluation results of equation (5) are shown in Table 5. All MRE values are below 25%, showing that we can estimate the length of a software system based on its vocabulary with good accuracy.

Table 5: Evaluation of software science length equation

	Vocabulary (n)	**Actual Length (N)**	**Software Science Estimation**		**Zipf-Mandelbrot law based Estimation**	
			N^	**MRE**	**N^**	**MRE**
Jena	26677	781434	391429	49.91%	687340	12.04%
Tomcat	24866	887289	362247	59.17%	830069	6.45%
Ant	22631	576013	326573	43.30%	479908	16.68%
Swing	18451	845361	260694	69.16%	791626	6.36%
jEdit	13846	420897	189742	54.92%	461488	9.64%
Jetty	7511	214092	96050	55.13%	235112	9.82%
jHotdraw	2969	84160	30268	59.89%	74515	11.46%
DrJava	14530	362525	200187	44.78%	368273	1.59%
Protege	12174	406288	164524	59.51%	383116	5.70%
Cocoon	13361	475870	182387	61.67%	473218	0.56%
JavaCC	4183	85034	49798	41.43%	67347	20.79%
JUnit	1311	20120	13196	34.41%	18606	7.53%
Bash	14344	433442	197365	54.47%	373694	13.79%
Apache	15422	408893	213793	47.71%	370625	9.36%
Make	3883	92443	45826	50.43%	90761	1.82%
CppUnit	3016	51903	34349	33.82%	51903	1.0%
NPP	15516	435008	215275	50.51%	356164	18.13%
MFC	18662	525433	264000	49.75%	549264	4.54%

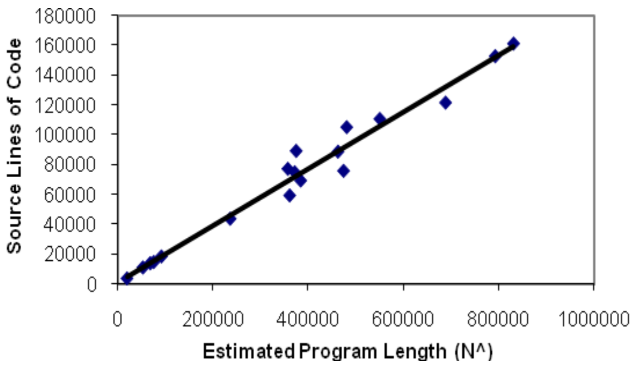


Fig. 4. Correlation between the N^ and SLOC

We also find that there is a strong correlation between the estimated program length N^ and the Source Line of Code (SLOC; line of code excluding blank lines and comments). Figure 4 shows this correlation for all systems studied. Linear regression analysis results in the least-squares line with the following equation:

$$SLOC = 0.19N\hat{} + 1496 \qquad (6)$$

with the R^2 value 0.973. Equation (6) shows that we can estimate SLOC based on N^, and vice versa. SLOC is the most commonly used software size measure in practices. The fact that there is a linear relationship between N^ and SLOC suggests that the N^ could also be a useful software metric.

Discussions. – Halstead gives an algebraic derivation of the length equation in his book ([9], page 9-11). The derivation process involves two major assumptions: 1). A program of length N can be divided into N/n substrings of length n (n is the size of vocabulary), and no duplications of these substrings. Thus divided, the program length has the upper limit $N <= n^{n+1}$; 2). Operators and operands tend to alternate, so the upper limit becomes $N <= n * n1^{n1} * n2^{n2}$. Our study shows that tokens in programs are not uniformly distributed. Different tokens (thus different substrings) have different probability of occurrences and operators and operands do not alternate. Therefore, the above two major assumptions do not hold for programs. As a result, the derived software science length equation (3) does not estimate program length well.

In English text, a word is a sequence of characters bounded by non-alphanumerics (a word may also contain a quote or hyphen). In this study, we treat each lexical token as a "word". We also experimented with different definitions of "word" and obtained similar results. For example, if we only consider identifiers as "words", the distribution of word frequencies still follows Zipf's law. If we consider identifier, keyword and operator as "words" (excluding literals), we could obtain similar results. In our experiment, we consider a string literal as a single token (e.g., "a string" is treated as one token), if we consider a string literal as a composition of tokens that are separated by delimiters (e.g., to treat "a string" as a composition of two tokens "a" and "string"), we still get similar results.

Robinson and Torsun [16] analyzed the distribution of keywords and operators in a set of FORTRAN programs written and run in a university environment. They found that some FORTRAN statements (e.g., ASSIGNMENT) occurred more frequently than the others (e.g., STOP) and suggested that this finding could be useful for code optimization. Their results are consistent with our findings. However, they did not derive a formal distribution model from the data. There are also some models for the frequency distribution of operators in PL/I programs, such as the Zweben model [22] and the two variations of the Zipf model (Zipf-A and Zipf-B) [7]. We tested these models using Java programs and found that they did not correlate well with the actual operator distribution. Chen [6] stated that the program length is somewhere between Zipf estimate and Halstead estimate, and proposed to apply Simon-Yule model to merge the two estimates. However, no empirical study is given to show the validity of the approach. In this paper, we analyzed all lexical tokens in real-world software and found that the distribution of tokens follows the Zipf-Mandelbrot law.

Conclusions. – We discovered that the distribution of tokens in modern computer programs follows the Zipf-Mandelbrot law. We then derived a revised software science length equation (Equation 5) based on our findings. Our equation models the relationship between program vocabulary and length, and fits the actual data well. We performed the study through the empirical analysis of eighteen real-world public-domain software systems.

We believe our study reveals the statistical regularities that are behind program construction, and hope our results could contribute to a deep understanding of the nature of computer programming. Our results also suggest that there is a possible connection between programming languages and natural languages, and that the research on programming languages could learn from statistical analysis of natural languages.

REFERENCES

[1] Adamic, L.A. and Huberman, B.A., The Nature of Markets in the World Wide Web, *Quarterly Journal of Electronic Commerce*, vol. 1, pp. 5-12, 2000.

[2] Albrecht, A.J. and Gaffney, J., Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, *IEEE Transactions on Software Engineering*, vol. 9, no.6, 1983.

[3] Alex, R.L., Zipf Distribution of US Firm Sizes, *Science*, 293, pp. 118-120, 2001.

[4] Almaas E., Kovács B., Vicsek T., Oltvai Z.N., Barabási A.L., Global organization of metabolic fluxes in the bacterium Escherichia coli, *Nature*, 427(6977):839-43, 2004.

[5] Baeza-Yates, R. and Ribeiro-Neto, B., *Modern Information Retrieval*, ACM Press, 1999.

[6] Chen, Y., Zipf's law in natural languages, programming languages, and command languages: the Simon-Yule approach, *International Journal of Systems Science*, 22(11), 1991.

[7] Elshoff, J. L., A study of the structural composition of PL/I programs. *SIGPLAN Notice* 13, 6, pp. 29-37, 1978.

[8] Firzsimmons, A. and Love, T., A Review and Evaluation of Software Science, *ACM Computing Surveys*, Vol. 10, No. 1, March 1978.

[9] Halstead, M. H., *Elements of Software Science*, Elsevier, 1977.

[10] Hamer, P.G. and Frewin, D., M.H.Halstead's Software Science – A Critical Examination, in *Proc. 6th International Conference on Software Engineering*, Japan, pp. 197-206, 1982.

[11] Li, H. and Cheung, W., An Empirical Study of Software Metrics, *IEEE Transactions on Software Engineering*, vol. 13, no. 6, 1987.

[12] Mandelbrot, B., An information theory of the statistical structure of language, in Jackson, W., (ed) *Communication Theory*, New York. Academic Press, pp. 503–512, 1953.

[13] Baroni, M., Distributions in text. In Anke Lüdeling & Merja Kytö (eds), *Corpus linguistics: An international handbook*, Berlin: Mouton de Gruyter, 2005.

[14] Peters, J. and Pedrycz, W., *Software Engineering: An Engineering Approach*, Wiley, 1999.

[15] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, sixth edition, McGraw-Hill, 2005.

[16] Robinson, S. and Torsun, I., An Empirical Analysis of FORTRAN programs, *The Computer Journal*, 19(1):56-62, 1976.

[17] Shen, V., Conte, S. and Dunsmore, H., Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support, *IEEE Transactions on Software Engineering*, vol. 9, no.2, 1983.

[18] Silagadze, Z., Citations and the Zipf-Mandelbrot's law, *Complex Systems*, vol. 11, 1997.

[19] Smith, C., A software science analysis of programming size, *Proc. ACM 1980 annual conference*, ACM Press, 1980.

[20] Van Vliet, H., *Software Engineering: Principles and Practice*, Wiley, New York, 2000.

[21] Zipf, G. K., *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Cambridge, MA, 1949.

[22] Zweben, S. H., A Study of the Physical Structure of Algorithms, *IEEE Transactions on Software Engineering*, vol. 3, pp.250-258, 1977.