Turkish Online Journal of Qualitative Inquiry (TOJQI)
Volume 12, Issue 6, July, 2021: 9084 - 9096

Research Article

Tools for analysis of various static software complexities for mat lab code

Neha Bharani[1*], Dr. Abhay Kothari [2]

Abstract

Software code quality, operation, and maintenance are all supported by software metrics. Program metrics such as size, control flow, and data flow metrics all assess different aspects of software complexity. Continual calculation, review, and control are required for these software complexities. Recently, a lot of attention has been dedicated to this difficult issue, because of the commercial value of software projects. In the literature, there are some software metrics and estimation models to measure the complexity of mat lab projects. However, In order to acquire correct findings about software complexity, we must integrate advanced software metrics to the process. This paper reviews the theory of various software complexity metrics and establishes GUI based mat lab tool that calculates a set of complexity metrics such as line of code (LOC), NPATH (NC), McCabb's metrics (MCC), Halstead's Software Science Complexity (HSSC) and Relative System Software Complexity (RSYSC) for mat lab programs to include a wide range of complexity. By identifying new and redefining current measures, we evaluate many software metrics such as software quality, project size/effort, and many more areas. Further, these metrics can be used as inputs in neural networks for more accurate the estimation of software complexity metrics.

Keywords: LOC, NC, MCC, HSSC, RSYSC, Mat lab

1 Introduction

As Software complexity is based on well-known software metrics, it is likely to decrease the time spent and costs incurred in software testing. In the case of software quality, improving the quality of the source code is considered a quantitative way of assessing quality. With regard to calculating values, analyses of source code or the code that the program is written in may be utilized.

The metrics for software complexity have been established to varying degrees. One of the most common complexity metrics developed by McCabe [2] is the cyclomatic complexity metric that indicate the testability and understandability of a program. The software science measures pioneered by Halstead [3] can be used to determine the complexity of software products. The software science measurements include an enhancement of measuring the size of an program by counting lines of code. The number of operators and operands in the program is measured using Halstead's metrics (code). During the calculation of program length and other measures, these operators and operands are considered.

A number of additional techniques were studied, such as Nesting Level [4], Data flow based metrics [5], and the variety of LOC [6], NPATH [7], Function Points [8], Chung's live definition [9] etc

To further describe the design and code implementation metrics in regard to software quality, Kan [10] stated, By understanding the concepts of code Lines of Code (LOC), the software science metrics known as Halstead's LOCs, and the software complexity metric called cyclomatic

complexity, he was able to identify three key metrics for code implementation.
In addition to the metrics related to input/output and structure, there are also several complexity metrics associated with the system's operation (sometimes referred to as fan-in and fan-out). Henry-Kafura [11], for instance, defined complexity as a fan-in and fan-out function to determine the flow of information among various modules. Structural (or inter module) complexity and local (or intra module) with respect to a module) complexity are both significant in architectural design difficulty, according to the study of Card and Agresti [12]. In order to detect the complexity qualities, one may use a specific function for the amount of I/O variables and fan-out of the modules, all of which make up the design.
Additionally, a few researchers, such as [13] and [14], describe some hybrid versions of this metric. Unlike Card D Glass, which defined structural, data, and total complexity metrics, which can be Computed on a module and system level, Card Glass, on the other hand, defined structural, data, and total complexity metrics, which can be computed on a module and system level.
In order to minimize the number of bugs introduced into the product in the early stages of development, it is preferable to apply techniques that can identify bugs as soon as possible rather than employ costly product recalls [15]. A large amount of research has been conducted to create automated analysis methods and tools that are used to perform quality assessment while the code is still in development. The metrics defined for the source code and models are not publicly available because these tools are commercial. Therefore, reproduction of the complexity evaluation is difficult. These tools demonstrate the source code complexity for software projects for the programming languages c, c++, and Java. A complexity analysis tool is not yet available for use with mat lab project with lots of static measures. So, there is a need for well-defined and easy to use complexity metrics for mat lab programs.
In this paper we describe and implement a GUI based mat lab tool called complexity Measure that identifies some of the well known static software measures such as LOC, NPATH, MCC, and HSSC, as well as RSYSC structure metrics for capturing interactions between modules. We have found these metrics to be particularly useful in systems that are layered with subsystems.
Further, these metrics can be used as inputs in neural networks for more accurate the estimation of software complexity metrics.

We can use a neural network of three layers with a single hidden layer and train this network by using distinct training algorithms to determine the accuracy of software complexity.
Here are the following sections that follow. Section II describes the materials and methods needed to put our tool into practice. Section III illustrates the simulation of the proposed mat lab tool. In Section IV, the analysis and evaluation of the system is described. To sum up, we draw the conclusion in section V.

2 Material & Methods

2.1 Line of Code

The simplest measure of software complexity advised by Hatton (1977) is Lines of Code (LOC). It is an indication of how sophisticated the programming is. This metric is very simple to use and measure the number of source instruction required to solve a problem. While counting a number of instructions (source), blank and commenting lines are ignored, but each line is counted

individually. Today's software systems are becoming increasingly sophisticated, and as a result, effective testing approaches are now required. Size qualities are commonly used to describe physical magnitude, bulk etc. Halstead's software science [3] and lines of code are good examples of size metrics. According to M. Halstead, the measurements suggested were referred to as software science.

2.2 Cyclomatic complexity

Cyclomatic complexity, a measure of software complexity, determines the number of linearly independent pathways in a code segment. It is used to calculate the program complexity by using the Control Flow Graph. A graph represents nodes, which are used to represent the smallest part of a program's commands, and edges connect the nodes to indicate that the second command might be immediately following the first.

Cyclomatic complexity, for example, will be 1 if no control flow statement is included in the source code. In general, Cyclomatic complexity is 2 if there is just one path through source code.

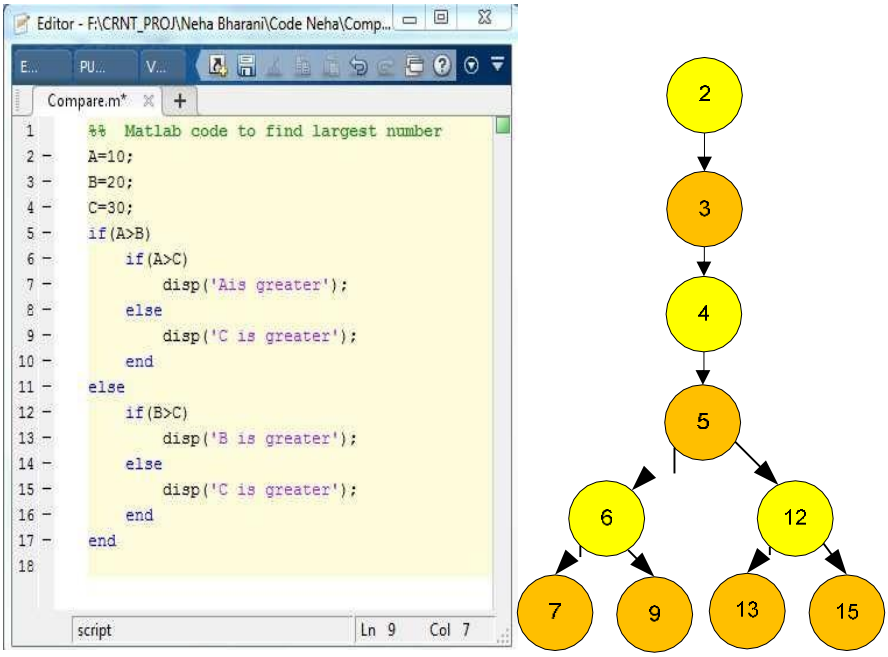
In summary, thus, cyclomatic complexity C is defined as.

$C(G) = E - N + 2P$

Where,

E = the number of edges in the CFG N = the number of nodes in the CFG P = the number of connected components

Let a section of code as well as Control Flow Graph of the corresponding code as show in figure 1.



(b)

Fig. 1. Shows (a) the samples program written in mat lab in and its corresponding control flow graph in (b)

The complexity of the above code is calculated by determining the CFG. Ten nodes are shown on the graph, as well as 9 edges, meaning the graph has a total of 9-10+2 = 1 Cyclomatic complexity. With McCabb's Complexity, there is a problem as it fails to make fine distinctions between conditional statements (control flow structures). Another consideration that was overlooked is the nested level of control flow structures.

2.3 NPATH complexity

Control structure of a program is used to calculate control flow complexity metrics. The control flow measure, NPATH, developed by Nejmeh [9], counts the number of acyclic execution paths.
NPATH complexity (NP) is determined as shown in table 1.

Table 1. NPATH complexity measures

Structure	Complexity expression
if ([expr]) { [if-range] }	NP(if-range) + 1 + NP(expr)
if ([expr]) { [if-range] } else { [else-range] }	NP(if-range) + NP(else-range) + NP(expr)
while ([expr]) { [while-range] }	NP(while-range) + NP(expr) + 1
do { [do-range] } while ([expr])	NP(do-range) + NP(expr) + 1
for([expr1]; [expr2]; [expr3]) { [for-range] }	NP(for-range) + NP(expr1) + NP(expr2) + NP(expr3) + 1
switch ([expr]) { case : [case-range] default: [default-range] }	S(i=1:i=n)NP(case-range[i]) + NP(default-range) + NP(expr)
[expr1] ? [expr2] : [expr3]	NP(expr1) + NP(expr2) + NP(expr3) + 2
goto label	1
break	1
Expressions	Number of && and || operators in expression. No operators - 0
continue	1
return	1
Statement (even sequential statements)	1
Empty block {}	1
Function call	1
Function(Method) declaration or Block	P(i=1:i=N)NP(Statement[i])

The shortcomings of McCabe's measure, which fails to distinguish between different types of control flowand nesting levels control structures, are addressed by NPATH, a gauge of software complexity.

2.4 Halstead Complexity

Halstead complexity is based on a novel method of calculating program size that includes counting lines of code. Halstead's measurements are calculated in two phases. The first stage is to count the number of operators and operands in the program; the second step is to count each occurrence of the number of operators and operands (code). The aforementioned operators and operands are considered for determining program length, vocabulary, volume, prospective volume, predicted program length, difficulty, and effort.

The following are the fundamental definitions for these tokens:

$n1$ □ *number of unique operator's*$n2$ □ *number of unique operands*
$N1$ □ *Total occurrences of operators*$N2$ □ *Total occurrences of operands*

The identification of operators and operands depends on the programming language. Halstead provides a number of software qualities based on these concepts of operators and operands. Table 2 shows the measures.

Table 2. The measures of Halstead

Size of Vocabulary	$n = n_1 + n_2$
Program Length	$N = N_1 + N_2$
Program Volume	$V = N \log_2 (n)$
Potential Volume	$V^* = (2 + n_2) \log_2 (2 + n_2)$
Program Difficulties	$D = V / V^*$
Program Effort	$E = (n_1 N_2 N \log_2 (n)) / (2n_2)$
Programming Times(Seconds)	$T = E / 18$

One of the significant flaws of this complexity is that it does not account for control flow complexity,which is difficult to compute quickly.

2.5 Information Flow Complexity

We applied the Henry-Kafura's metrics to calculate the information flow complexity.

$$hkCMX = size * (fan + in * fan + out)^2$$

Where *hkCMX* the information flow complexity of a subsystem, *size* is the number of contained blocks including subsystem blocks, $fan + in$ and $fan + out$ represent the number of different incoming and different outgoing links of a subsystem, respectively.

2.6 System complexity

According to Card and Agresti, (whose term is) system complexity is defined as a measure of the complexity inside procedures as well as the complexity between them. The complexity of a system designis defined by how many procedures are called, how many parameters are passed, and how much data is used.

System complexity was originally meant to be used during design time. Even before the implementation isdone, you can use it to evaluate the difficulty of building a designed system. To calculate system complexity, we can use the source code as well.

External structural complexity (SC) and internal data complexity (DC) are two distinct characteristics of system complexity.

2.6.1 Structural complexity

Our definitions start with the following:

$$SC = \frac{\sum_{i=1}^{n} f_{out}^2 (i)}{n}$$

Where f_{out} is a structural fan-out and it is equivalent to number of other procedures called by the procedure. $f_{out}(i)$ is fan-out of sub-function *i* and *n* is a number of function in the system.

As you can see, a procedure that calls a large number of other procedures has a relatively high structural complexity. This interaction with other procedures is why SC is thought of as the external complexity.

2.6.2 DC Data complexity

Data complexity (the local or internal complexity) for a procedure is defined by the following equation:

$$DC = \frac{V(i)}{(f_{out}(i)+1).n}$$

Where $V(i)$ defines number of input/output variables for a procedure *i*.

The more data the procedure reads and writes, the higher data complexity it has. On the other hand, the more other procedures it calls ($S(f_{out})$), the lower the data complexity, as parts of the complex data processing is likely to have been delegated to the other procedures.

Now that we can calculate SC and DC for one procedure, let's calculate the complexity of the entire system.

Total system complexity

$SYSC = Sum(SC, DC)$

Relative system complexity

$RSYSC = avg(SC, DC)$

The relative system complexity is the more interesting measure. It measures the average complexity of procedures. It is a normalized measure for the entire system and it does not depend on the system size. It thus allows for design complexity evaluation among different systems.

2.6.3 Minimizing the relative system complexity RSYSC

It is to be noted that minimizing DC may result in smaller procedures but more calls between them, leading into an increase in SC.

Originally, Card & Agresti investigated 8 old systems (the newest one was from 1981) and found out the following values:

Measure	Value range
SC/proc	11.8–24.6
DC/proc	4.9–12.1
RSYSC	22.6–32.8

Good RSYSC ≤ 25.3, poor RSYSC ≥ 26.5

2.7 Neural Networks

Neural Networks (NNs) are an effective technique for estimating processes. The NN model used in this study is made up of three layers of neurons: input, hidden, and output. The input layer of this approach is made up of the number of matrices generated by system. The association weights are adjusted in this of network to lessen the error between the actual and calculable values of the system variables. System quality (SYSC), also known as style quality, is a composite live of quality inside and between procedures. It assesses the effectiveness of a system's style in terms of method calls, parameter passing, and data usage. Originally, system quality was a design-time metric.

Neural networks are very good at modeling digital circuits. The Neural Network representation can be utilized in a variety of situations where digital circuit behaviors must be portrayed as a computer software solution to a specific problem.

2.7.1 Digital Circuit Consideration

Arithmetic functions, including as addition, subtraction, multiplication, and division, are performed using a variety of integrated circuits. Here, we'll look at the Half Adder's digital circuit as well as its Neural Network counterpart. The Half Adder is a fundamental arithmetic circuit. To understand how it works, consider the addition of two one-bit words, which results in two bits of data, the sum bit and the carry bit. [16] We're looking at the generalized scenario where adding two bits of data always produces a sum and a carry.

When we look at the Half Adder block diagram, we can see that the inputs A and B produce two outputs, the Sum and Carry. The truth table reflects the conditionality of the sum and carry, as shown in table 3. Figure 2 shows a block diagram. This block diagram can also be turned to a circuit diagram to help you understand it more clearly and technically.

Table 3. True Table of Half adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

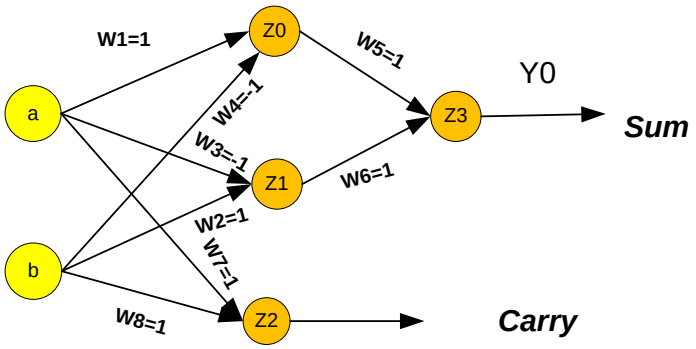


Fig. 2. Neural Network implementation of Half adder

3 Simulation

We developed a complexity analysis tool that comprise of LOC, NC, MCC, HSSC and RSYSC that automatically measure the complexity metrics defined in Section 2. It is a sophisticated tool that measures modularity metrics of mat lab functions/scripts as well as function's dependency on other programs/scripts [11]. The complexity analysis tool reads mat lab file with the standard structural format and generates the metrics with the list of subsystems and the respective complexity metrics. Our tool also measures the system or project complexity by passing the main program into the tool.

For implementation of complexity analysis tool, we have implemented the following program written in mat lab language.

Npath.m: It identifies the Npath software complexity of the input mat lab file. For simplicity we have omitted the instructions path calculations.

Cyclometic Complexity.m: It determines the mcCabb's software complexity of input mat lab program/scripts

halstead.m: it determines the has stead's software complexity of input mat lab program/scripts. *SystemComplexity.m:* This modules determines the inter dependency with other functions and gives theRSYSC complexity. It uses the fdep mat lab function in order to calculate system complexity.

Token.m: Token modules are the class definition of the program tokens.

settings.m : This modules check the required setting for tokenizer.

*Tokenize.m:*Tokenize module divides the whole program into smallest units called tokens with its type likekeyword, identifier, constant, etc.

The system operation begins with the default position of the MAT LAB. System starts by running the mainprogram i.e. complexity Measure.m , by typing the file name at command window of MAT LAB or by simply click on run command in MAT LAB show in Figure 3.

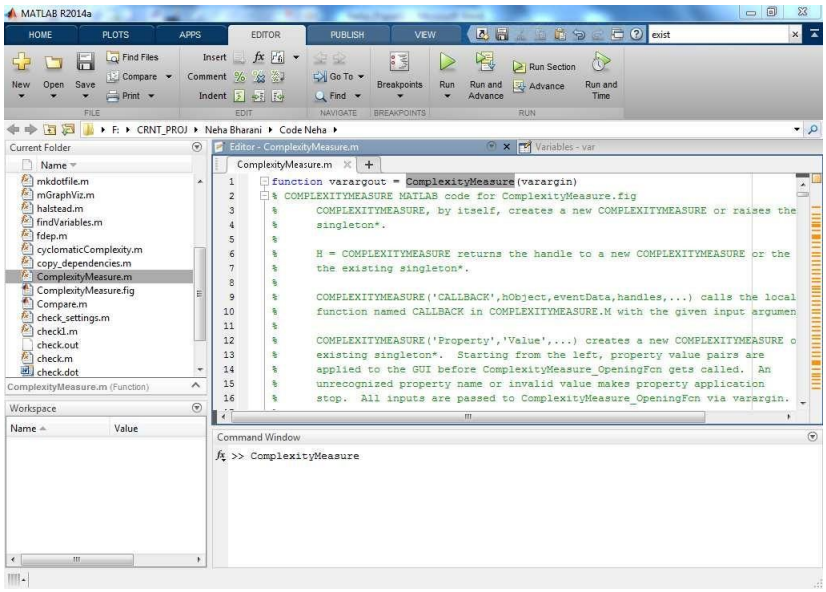


Fig. 3. Shows default position of the mat lab tool

Figure 4 shows the default position of the proposed Complexity Measure tool. The complexity of each mat lab program or script is determined in the mat lab module complexity panel. It includes the measure of LOC, cylometic complexity Npath complexity and halstead complexity. In this module when we click on the specific button it will give the respective complexity of input mat lab function/scripts. We have another panel in the system complexity panel, called system complexity consisting of check dependency, show dependency and calculate complexity. When we click on the check dependency button, it will bring all the required MAT LAB files that are needed to run input files and tools (required to run) as well. Show dependency buttons generates the metrics with the list of all subsystems and shows the dependency on each other. Calculate Complexity button calculates the fan-in

fan-out and list of input/output variables for each functions and then calculates structural complexity and data complexity and based on these values determines the RSYSC complexity.

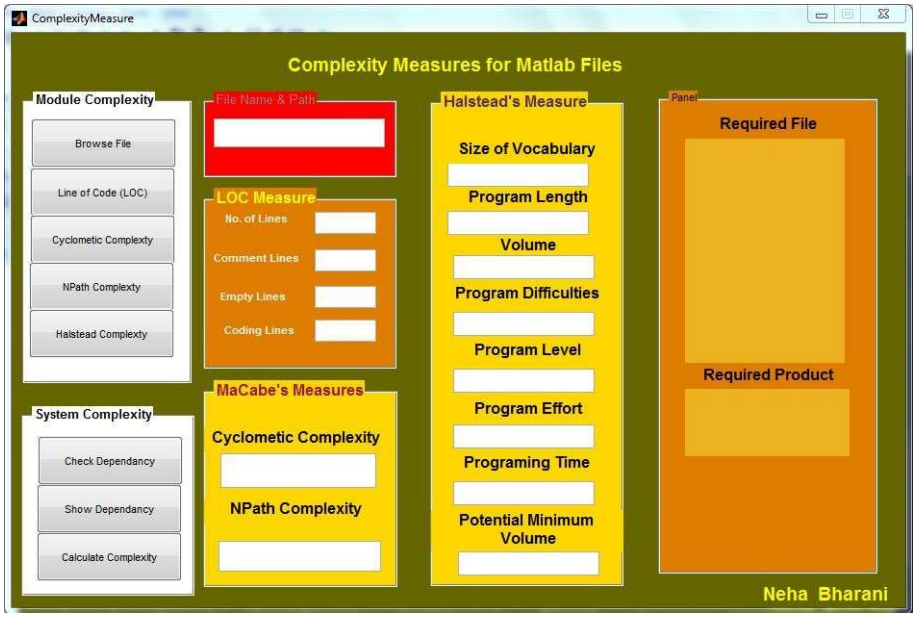


Fig. 4. Shows default position of the proposed Complexity Measure tool

After calculating all the complexity of the halstead .m mat lab file (As example), the result will be shown as given in figure 5.

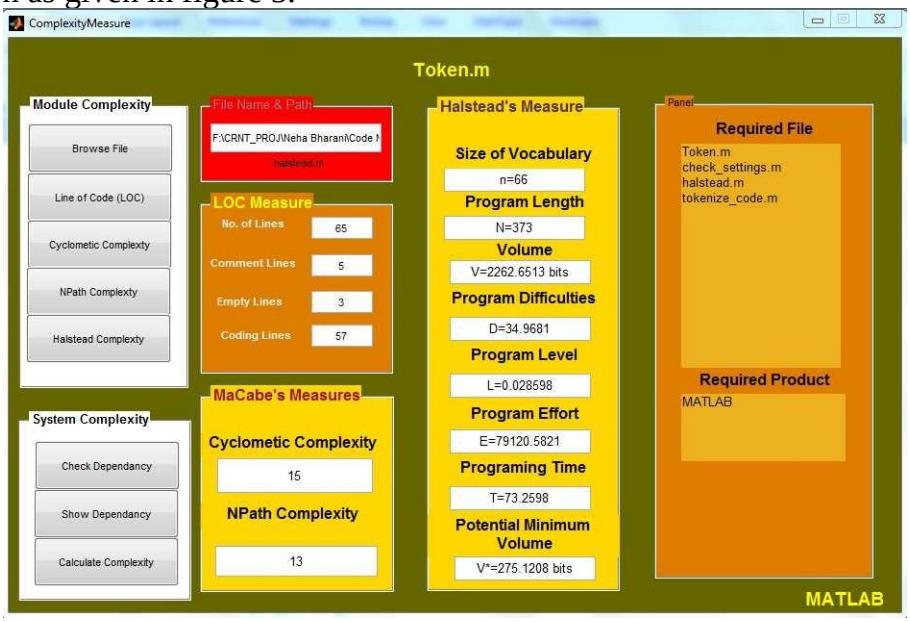


Fig. 5. Result after evaluate the complexity of halstead.m file

4 Evaluation

Static analysis of software complexity metrics, such as size and control flow metrics, form the basis of our analysis. We've researched five program characteristics in the literature and concluded that they all have an impact on program complexity.

In order to quantify complexity, we have calculated the Npath, McCabe, and LOC metrics. And finally, we've used Halstead's software science complexity measures. Next, we calculate the system complexity RSYSC as a whole. The study of metrics connected to program execution is considered the fundamental approach to statistical measurement in the fields of software development and engineering (source code).It covers two aspects that deal with the various dimensions of program design, like the physical dimensions of the program (volume/size) as well as the logical dimensions (organization and control structure).
Instead of measuring directly the number of lines in a program, LOC only executable lines in a program. While counting a number of instructions (source), line used for blank and commenting lines are ignored. The NPATH counts the number of acyclic execution paths that attempt to program optimization. For simplicity, we've only examined the control structures complexity and ignored the instructions' complexity.
Halstead's metrics count the number of operators and operands in the program, and keep track of the number of times each type appears (code). When calculating the length, vocabulary, volume, potential volume, estimated program length, difficulty, and effort, these operators and operands are considered.
For each program P written for implementing our proposed tool are considered to measure out system complexity. The complexity measured by us i.e Lines of Code (LOC), NPATH Complexity (NC), McCabb's complexity (MCC) and Halstead's software science complexity (HSSC) are shown inTable IV.

Table 4. Calculation of the static complexity measures for proposed tool

Measures	**Parameters**	Complexity	Npath.m	cyclometicC	halstead.m	SystemComplexity.m	Token.m	Setting.m	Tokenize.m
LOC	**TotalLine**	538	53	39	65	40	44	92	340
	Empty	237	6	10	5	4	10	28	105
	Comment	128	6	4	3	8	8	10	21
	CodeLine	173	41	25	57	28	26	54	214
Npath	**NP**	14	10	6	13	4	4	2	54
MCC	**C(G)**	5	12	4	15	2	2	1	38
	n1	6	19	15	19	6	11	8	26
	n2	24	29	30	43	14	13	40	139
	N1	16	136	88	200	10	53	106	602
	N2	35	115	75	173	22	48	130	570
	Vocabulary	29	48	45	62	20	24	48	165
	Length	51	251	163	373	32	101	236	1172

HSSC	**Volume**	250.25	1409.3	900.34	2229.5	140.5542	469.02	1325.1	8643.5
	Difficult	3.6458	37.6724	18.75	38.2209	4.7143	20.3077	13	53.3094
	Effort	912.3750	53091	16881	85215	662.6125	9524.9	17226	460780
	Time	0.8448	49.1587	15.6309	78.9023	0.6135	8.8194	15.9499	426.6499
	Potential Vol.	122.2114	153.5801	160	247.13	64	58.6034	226.4773	1006.7

Figure 6 shows an example of modular structure of proposed tool that includes the control flow from onemodule to another and dependency on each other.

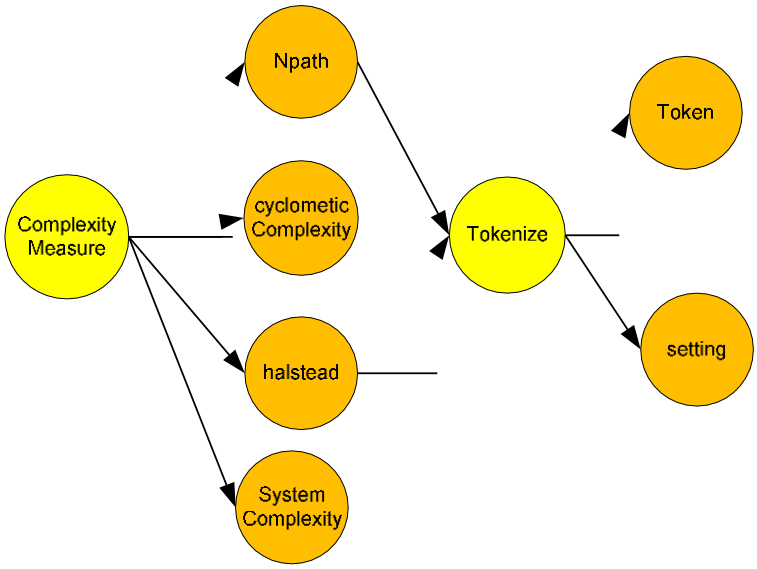


Fig. 6. Example of proposed tools control flow and dependency graph

From the above graph we can calculate the system complexity that can help to determine the fan-in and fan-out measures for each module. Table shows the modules system complexity measure. The fan-in of a module M is the number of local flows that terminate at M. similarly, the fan-out of a module M is the numbers of local flows that emanate from M. High information flow complexity values indicate highly coupled components. These modules need to be looked at in terms of fan-in and fan-out to see how to reduce the complexity level.

Table 5. Module System Complexity measures

Module	fin	$fout$	f^2	$\sum_{n} f^2(i)$	v variable	$DC = \frac{V}{\quad}$

	$fan \square in$	$fan \square out$	$fan \square out)^2$ out	$SC \square$ $i \square 1$ out n		(i) $(fout (i) \square 1).n$
ComplexityMeasure	0	5	25	2.77	21	0.39
Npath	1	1	1	0.12	11	0.62
SystemComplexity	1	0	0	0	9	1.00
Token	1	2	4	0.45	6	0.23
setting	1	0	0	0	3	0.34
cyclometicComplexity	1	0	0	0	12	1.34
halstead	1	1	1	0.12	15	0.84
Tokenize	2	0	0	0	47	5.23
Sum	**8**	**9**	**31**	**3.46**	**124**	**9.99**

$RSYSC \square avg(SC, DC)$ $RSYSC \square avg(3.46, 9.99)$

$RSYSC \square 6.72$

To know whether the entire system has a high-quality project, look at the value of RSYSC. If the value of RSYSC is less than 25.3, then the entire system has a good quality project. The tool we're developing has less system complexity, so we can use that as our claim.

5 Conclusion

Software complexity metrics are frequently used to assess the quality of software development and are an important component of the SDLC. The volume, control, and data-based complexity of today's software systems necessitate the use of effective testing techniques. Static analysis could lead to a reduction in software development costs, while also improving software testing effectiveness and software quality. Thepaper discusses software complexity metrics such as LOC, NPATH (NC), McCabb's metrics (MCC), and Halstead's Software Science Complexity (HSSC) for mat lab program, and introduces a mat lab GUI-based tool for calculating these various complexity measures. Additionally, In addition, it helps to determine the dependency on other mat lab files and it also shows the RSYSC complexity of the system. We also assessed the efficiency of tools in relation to RSYSC, and discovered that our proposed tools have a lower value than the threshold, implying that our system achieves less complex structures. We will strive to construct a static complexity analysis suggestions system that will help developers keep bugs to a minimum while their product is being built, and we will use these metrics as inputs in neural networks for more accurate findings. We will use a three-layer neural network with a single hidden layer that will be trained using different training algorithms to determine the accuracy of software complexity. We will also use the neural network to implement addition, multiplication, and various arithmetic operations to determine the complexity of software.

References

1. W. Harrison, K. Magel, R. Kluczny, and A. Dekok, Applying Software Complexity Metrics to Program Maintenance Compute, vol. 15, pp. 65-79, 1982.
2. T. J. McCabe. A complexity measure. IEEE Transactions on Software Engineering, (4):308{320, 1976.
3. M. Halstead. Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA, 1977.
4. W. Harrision and L. I. Magel, "A complexity based on nesting level," Sigplan Notice, vol. 16, no. 3, 1981.
5. A. Fitzsimmons and T. Love, "A review and evaluation of software science," Computing Survey, vol. 10, no. 1,
   March 1978.
6. S. D. Conte, H. E Dunsmore, and V. Y. Shen, "Software engineering metrics and models," Benjamin/Cummings Publishing Company, Inc., 1986.
7. B. A. Nejmeh, "NPATH: A measure of execution path complexity and its applications," Comm. of the ACM, vol. 31, no. 2, pp. 188-210, February 1988.
8. E. E. Millis, "Software metrics," SEI Curriculam Module SEI- CM. vol. 12, no. 2.1, Dec, 1988.
9. C. M Chung and M. G Yang, "A software maintainability measurement," Proceedings of the 1988 Science, Engineering and Tech. Houston, Texas, pp. V12-16.
10. S. Kan. Metrics and Models in Software Quality Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.
11. Henry, S. & Kafura, D. (1981). Software structure metrics based on information flow. IEEE transaction of software engineering, SE-7(5), 510-518.
12. D.N. Card, W.W. Agresti, Measuring software design complexity, Journal of Systems and Software, Volume 8, Issue 3, 1988, Pages 185-197.
13. S. Henry and C. Selig. Predicting source-code complexity at the design stage. IEEE Software, 7(2):36{44, Mar. 1990.
14. M. Shepperd and D. Ince. Derivation and Validation of Software Metrics. International Series of Monographs on Computer Science. Clarendon Press, 1993.
15. Y. Dajsuren, A. Serebrenik, R.G.M. Huisman, and M.G.J. van den Brand. A quality framework for evaluating automotive architecture. In the FISITA World Automotive Congress, pages 1-7. FISITA, 2014.

16. D. P. Sharma, Professor, Dept. of Computer Science, Dean, Sciences.St. Joseph's Hyderabad, India, College, Neural Network Simulation of Digital Circuits, *International Journal of Computer Applications (0975 – 8887)*
   Volume 79 – No6, October 2013