Differences in the Definition and Calculation of the LOC Metric in Free Tools*

István Siket
Department of Software Engineering
University of Szeged, Hungary
siket@inf.u-szeged.hu

Árpád Beszédes
Department of Software Engineering
University of Szeged, Hungary
beszedes@inf.u-szeged.hu

John Taylor
FrontEndART Software Ltd.
Szeged, Hungary
john.taylor@frontendart.com

Abstract

The software metric *LOC (Lines of Code)* is probably one of the most controversial metrics in software engineering practice. It is relatively easy to calculate, understand and use by the different stakeholders for a variety of purposes; LOC is the most frequently applied measure in software estimation, quality assurance and many other fields. Yet, there is a high level of variability in the definition and calculation methods of the metric which makes it difficult to use it as a base for important decisions. Furthermore, there are cases when its usage is highly questionable – such as programmer productivity assessment. In this paper, we investigate how LOC is usually defined and calculated by today's free LOC calculator tools. We used a set of tools to compute LOC metrics on a variety of open source systems in order to measure the actual differences between results and investigate the possible causes of the deviation.

1 Introduction

Lines of Code (LOC) is supposed to be the easiest software metric to understand, compute and interpret. The issue with counting code is determining which rules to use for the comparisons to be valid [5]. LOC is generally seen as a measure of system size expressed using the number of lines of its source code as it would appear in a text editor, but the situation is not that simple as we will see shortly. On the other hand, the importance of this metric is clear: it can be used for different purposes including size estimation, productivity assessment and providing a base for other relative measurements such as number of bugs per code lines. Multiple different areas of software engineering benefit from calculating LOC, perhaps most importantly software estimation and quality assurrance [4].

In many cases, this metric is virtually the only one which can be easily explained to non-technical stakeholders such as managers. Consequently, important project decisions are often made based on it, so the reliable measurement and expression of LOC is of utmost importance in any software measurement activity. However, there are several issues with this metric. Some of them are inherent such as problems related to language-dependence, therefore alternative measures such as function

*Technical Report TR-2014-001, ©2014, University of Szeged, Department of Software Engineering. You may freely distribute and make extracts of this document, provided the source is acknowledged.

point analysis are used in certain areas [5]. As Bill Gates puts it, “Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weighs.”

Apart from these drawbacks, we are also interested in other related issues that arise from the diverse definitions, calculation methods and actual interpretations. From the technical point of view, computing LOC is simple, so there are a huge number of tools implementing a plethora of variations of this metric through different programming languages. This results in a fairly confusing situation for any software professional getting in contact with this metric. Consequently, anyone wanting to count program code lines should first establish some kind of a measurement methodology, preferably one standardized within a project or organization [6].

In this study, we investigate how this metric is usually defined and compare existing tools for its calculation. For comparison, we used a set of criteria which includes both general and specific tool properties in relation to the LOC metric. To assess the differences in the actual results provided by the tools, we calculated the LOC metrics for a set of open source programs. Our aim is not to pinpoint the "best" LOC definition or tool but to draw attention to the differences in existing solutions, hence provide guidelines for professionals considering the use of this metric.

2 Definitions of the LOC Metric

There is a controversy among professionals what LOC should actually mean and how it should be defined or measured. Consider, for example, any of the summarization websites cited below such as the Wikipedia page.

The Software Engineering Institute (SEI) of Carnegie-Mellon University has established a framework for the measurement and interpretation of Lines of Code [6]. One of the most important messages of this report is that it is essential for the measurement methodology to be well-established and properly documented – such as using the provided check-list. Capers Jones already discussed how the interpretation of software metrics (including LOC) suffers from precise definitions and calculation methods back in 1994 [3]. Unfortunately, the situation did not change significantly ever since [1, 5, 8].

In this section we collect the most important definitions from various sources including research articles, community entries and tool vendors without the attempt to be comprehensive.

Academic publications

In the research community, the lines of code metric is used for various purposes including quality assessment, defect prediction, productivity measurement, relative process metrics, and so forth. In these applications, the definition based on the physical lines of source code excluding blank lines and comments is most often used (see, for instance, [2, 7, 9]).

Wikipedia

`http://en.wikipedia.org/wiki/Source_lines_of_code`

- LOC: “The most common definition of physical SLOC is a count of lines in the text of the program’s source code including comment lines. Blank lines are also included unless the lines of code in a section consists of more than 25% blank lines. In this case blank lines in excess of 25% are not counted toward lines of code”

- LLOC: “Logical SLOC attempts to measure the number of executable ‘statements,’ but their specific definitions are tied to specific computer languages (one simple logical SLOC measure for C-like programming languages is the number of statement-terminating semicolons). [It is much easier to create tools that measure physical SLOC, and physical SLOC definitions are easier to explain. However, physical SLOC measures are sensitive to logically irrelevant formatting and style conventions, while logical SLOC is less sensitive to formatting and style conventions. However, SLOC measures are often stated without giving their definition, and logical SLOC can often be significantly different from physical SLOC.]"

Cunningham & Cunningham, Inc.

`http://c2.com/cgi/wiki?LinesOfCode`

- LOC: "Lines of Code, usually referring to non-commentary lines, meaning pure whitespace and lines containing only comments are not included in the metric." (The number of lines of program code is wonderful metric. It's so easy to measure and almost impossible to interpret. It can be used as a measure of complexity or productivity.)

SonarQube tool

`http://docs.codehaus.org/display/SONAR/Metric+definitions`

- Lines: "Number of physical lines (number of carriage returns)."
- Lines of code: "Number of physical lines that contain at least one character which is neither a whitespace or a tabulation or part of a comment."

NDepend tool

`http://www.ndepend.com/Metrics.aspx`

- "NbLinesOfCode: (defined for application, assemblies, namespaces, types, methods) This metric (known as LOC) can be computed only if PDB files are present. NDepend computes this metric directly from the info provided in PDB files. The LOC for a method is equals to the number of sequence point found for this method in the PDB file. A sequence point is used to mark a spot in the IL code that corresponds to a specific location in the original source. More info about sequence points here. Notice that sequence points which correspond to C# braces '{'and '}' are not taken account. Computing the number of lines of code from PDB's sequence points allows to obtain a logical LOC of code instead of a physical LOC (i.e directly computed from source files)."

PC Magazine

`http://www.pcmag.com/encyclopedia/term/46137/lines-of-code`

- Lines of Code: "The statements and instructions that a programmer writes when creating a program. One line of this 'source code' may generate one machine instruction or several depending on the programming language. A line of code in assembly language is typically turned into one machine instruction. In a high-level language such as C++ or Java, one line of code may generate a series of assembly language instructions, resulting in multiple machine instructions.
  Lines of Code Are Not the Same: One line of code in any language may call for the inclusion of a subroutine that can be of any size, so while used to measure the overall complexity of a program, the line of code metric is not absolute. Comparisons can also be misleading if the programs are not written in the same language. For example, 20 lines of code in Visual Basic might require 200 lines of code in assembly language. In addition, a measurement in lines of code says nothing about the quality of the code. A thousand lines of code written by one programmer can be equal to three thousand lines by another."

Eclipse Metrics plugin tool

`http://sourceforge.net/projects/metrics/`
`http://metrics.sourceforge.net/`

- TLOC: "Total lines of code that will counts non-blank and non-comment lines in a compilation unit. usefull for thoses interested in computed KLOC."

SLOC tool

`http://microguru.com/products/sloc/`

- "SLOC Metrics measures the size of your source code based on the Physical Source Lines of Code metric recommended by the Software Engineering Institute at Carnegie Mellon University (CMU/SEI-92-TR-019). Specifically, the source lines that are included in the count are the lines that contain executable statements, declarations, and/or compiler directives. Comments, and blank lines are excluded from the count. When a line or statement contains more than one type, it is classified as the type with the highest precedence. The order of precedence for the types is: executable, declaration, compiler directive, comment and lastly, white space."

Understand tool

`http://scitools.com/documents/metricsList.php`

- Lines of Code (Include Inactive): "Number of lines containing source code, including inactive regions."
- Physical Lines: "Number of physical lines."
- Source Lines of Code: "Number of lines containing source code. [aka LOC]: The number of lines that contain source code. Note that a line can contain source and a comment and thus count towards multiple metrics. For Classes this is the sum of the CountLineCode for the member functions of the class."

CMT++ Complexity Measures tool

`http://www.verifysoft.com/en_linesofcode_metrics.html`

- LOCphy: "number of physical lines"
- LOCpro: "number of program lines (declarations, definitions, directives, and code)"
- LOCbl: "number of blank lines (a blank line inside a comment block is considered to be a comment line)"
- LOCcom: "number of comment lines"

Conclusion

It appears that there is no common interpretation of the LOC metric. There are even surprising definitions such as Wikipedia's LLOC which refers to actual syntactic program elements rather than code lines. Similarly, non-empty lines, declarations and comments are interpreted differently, for example in the case of "invisible" characters such as block delimiting brackets or compiler directives.

Another issue is the lack of standardized notation for the different variants of the LOC metric as seen above. This concerns comment lines, logical lines, physical lines, etc. For this study, we use two common types of the LOC metric in our measurements: physical lines of code and logical lines of code (details given below).

3 Assessment Criteria

3.1 Tool Properties

First we compare LOC calculation tools based on a set of general criteria – secondly according to various properties related to the metric itself. These criteria are listed in Table 1.

General properties	*Note*
Version information	Actual version assessed
Language(s)	Supported languages
License	End user license type
Operating system(s)	Supported platforms
User interface	Type of the UI and subjective assessment
Extra features	E.g., is it part of a more complete metric suite?
LOC-related properties	*Note*
Other metrics	Does the tool compute additional metrics as well?
LOC types	What types of LOC are calculated?
PLOC name	How does the tool refer to the PLOC-type metric?
LLOC name	How does the tool refer to the LLOC-type metric?

Table 1: Comparison criteria

3.2 Metrics compared

We chose to use the two basic LOC types in the measurements: physical and logical lines of code. As there are many variations to the names and precise interpretation of these metrics, we will use these two general definitions, which will be applicable to most of the concrete situations:

1. Physical lines of code – hereinafter referred to as *PLOC* – means the actual lines as appearing in a text editor. This means that all lines are counted included empty ones and whitespaces. For a project, this practically means the total lines for all source files included in the measurement.
2. Logical lines of code – hereinafter referred to as *LLOC* – means the typical definition for logical lines, i.e. commented and whitespace lines are excluded.

Note, that the tools may use different denominations for these metrics and may implement different variations of the metrics as we will present in the next section.

3.3 Language

In the present study, we concentrated on calculating the lines of Java programs. In case of other languages – such as C/C++ – other complications may occur such as inclusion of preprocessor related code and common includes.

4 LOC Calculation Tools

There are numerous tools capable of calculating LOC metrics, both free and commercial. Many of them provide the feature as part of a more complex functionality (e.g. general metric calculation tools), while there are specialized ones as well. A (not necessarily complete) list can be found at `http://www.locmetrics.com/alternatives.html`. For this study, we selected a set of popular, freely available LOC calculator tools for measuring Java code; some of these tools are actually open source. The following list is a summarization of the examined tools (the descriptions are cited from the respective project's websites):

cloc	Counts blank lines, comment lines, and physical lines of source code in many programming languages. (`http://cloc.sourceforge.net`)
CodeAnalyzer	A tool for basic software source metrics. It can calculate these metrics across multiple source trees as one coherent "Code Set." (`http://www.codeanalyzer.teel.ws`)
Analytix	CodePro Analytix is the premier Java software testing tool for Eclipse developers who are concerned about improving software quality and reducing developments costs and schedules. The Java software audit features of the tool make it an indispensable assistant to the developer in reducing errors as the code is being developed and keeping coding practices in line with organizational guidelines. (`https://developers.google.com/java-dev-tools/codepro/doc`)
LOCCounter	Counts lines of code, including comment and blank lines. Interactive GUI, or command line. End user may add new file types by editing config file. Supports any source language that uses characters to delimit comments, such as // or /* */. (`http://sourceforge.net/projects/loccounter/`)
LocMetrics	This tool counts total lines of code (LOC), blank lines of code (BLOC), comment lines of code (CLOC), lines with both code and comments (C&SLOC), logical source lines of code (SLOC-L), McCabe VG complexity (MVG), and number of comment words (CWORDS). Physical executable source lines of code (SLOC-P) is calculated as the total lines of source code minus blank lines and comment lines. Counts are calculated on a per file basis and accumulated for the entire project. (`http://www.locmetrics.com/`)
Metrix++	Metrix++ is a tool to collect and analyse code metrics. Any metric is useless if it is not used. Metrix++ offers ease of introduction and integration with a variety of application use cases: monitoring trends, enforcing trends and automated assistance to review against standards. (`http://metrixplusplus.sourceforge.net`)
SLOCCount	A set of tools for counting physical Source Lines of Code (SLOC) in a large number of languages of a potentially large set of programs. SLOCCount runs on GNU/Linux, FreeBSD, Apple Mac OS X, Windows, and hopefully on other systems too. To run on Windows, you have to install Cygwin first to create a Unix-like environment for SLOCCount. (`http://www.dwheeler.com/sloccount/`)
SonarQube	An open platform to manage code quality. As such, it covers the 7 axes of code quality: architecture and design, comments, complexity, coding rules, duplications, potential bugs, unit tests. (`http://www.sonarqube.org/`)
SourceMeter	An innovative tool built for the precise static source code analysis of Java projects. This tool makes it possible to find the weak spots of a system under development from the source code itself without the need of simulating live conditions. Calculates more than 60 types of source

code metrics at component, file, package, class, and method levels. (`http://sourcemeter.com/`)

SourceMonitor The freeware program SourceMonitor lets you see inside your software source code to find out how much code you have and to identify the relative complexity of your modules. For example, you can use SourceMonitor to identify the code that is most likely to contain defects and thus warrants formal review. (`http://www.campwoodsw.com/sourcemonitor.html`)

UCC The Unified Code Counter (UCC) is a comprehensive software lines of code counter produced by the USC Center for Systems and Software Engineering. It is available to the general public as open source code and can be compiled with any ANSI standard C++ compiler. (`http://sunset.usc.edu/research/CODECOUNT/`)

In Table 2, we list some general properties of the examined tools based on the criteria from Table 1.

Tool	Version	Language	License	Op. system	Interface	Extra features
cloc	1.62	145	GPL v2	Win/Linux	cmd	Comment
CodeAnalyzer	0.7.0	5	GPL v2	Win/Linux	GUI	Comment
CodePro Analytix	7.1.0.r37	1	Google TOS	Win/Linux	GUI	Metrics, clones, code coverage, etc.
LOCCounter	2011-8-27	General[1]	BSD 3-cl	Win	GUI	
LocMetrics	2007 oct.	4	n.a.	Win	GUI/cmd	Metrics, comment
Metrix++	1.3.168	3	GPL	Win/Linux	cmd	Metrics
SLOCCount	2.26	27	GPL	Win/Linux	cmd	
SonarQube	4.4	20	LGPL v3	Win/Linux	GUI/cmd	Metrics, clones, coding rules, etc.
SourceMeter	6.0	1	EUAL	Win/Linux	GUI/cmd	Metrics, clones, coding rules
SourceMonitor	3.5.0.306	6	EULA	Win	GUI	Metrics
UCC	2013.04B	30	USC-CSSE	Win/Linux	cmd	Metrics

Table 2: General LOC tool features

The other aspect of the investigated tools is how they interpret the various LOC metrics, so we list LOC-related properties separately in Table 3 (see criteria in Table 1). An important detail is the actual metric we used in the measurements for PLOC and LLOC, which are given in the last two columns of the table. Note, that there are cases where the actual names used by the tool do not reflect the type of metric being computed.

Tool	Other metrics	LOC types	PLOC name	LLOC name
cloc	No	Comment, Blank	files+blank+comment+code	code
CodeAnalyzer	No	Comment, Blank	Total Lines	Code Lines
CodePro Analytix	Yes	Comment	Number of Lines	Lines of Code
LOCCounter	No	Comment, Blank	Lines	Source LOC
LocMetrics	Yes	Blank, Comment, etc.	Lines of Code	Executable Physical
Metrix++	Yes	Comment	-	std.code.lines:total
SLOCCount	Yes	-	-	Total Physical Source Lines of Code
SonarQube	Yes	Comment	Lines	Lines of code
SourceMeter	Yes	Comment	LOC	LLOC
SourceMonitor	Yes	Comments	Lines	-
UCC	Yes	Blank, Comment	Total Physical Lines	Physical SLOC

Table 3: LOC-related properties of the tools

5 Measurements

5.1 Subject Programs

We selected a set of open source Java programs for analysis. The programs vary in size from medium (several tens of thousand lines) to large (over million lines) systems and are from various domains. The main characteristics of the systems are given in Table 4.

[1]Supports languages that use characters to delimit comments, such as // or /* */.

Name	Version	Homepage	Java files
Ant	1.9.4	`http://ant.apache.org`	1,233
ArgoUML	0.34	`http://argouml.tigris.org`	1,922
Cewolf	1.2.4	`http://cewolf.sourceforge.net/new/`	134
Hibernate	4.3.6	`http://hibernate.org`	7,273
Liferay	6.2	`http://www.liferay.com/`	10,118
PMD	5.1.2	`http://pmd.sourceforge.net`	1,115
Tomcat	8.0.9	`http://tomcat.apache.org`	2,037
Xerces	1.4.4	`http://xerces.apache.org/xerces-j`	546

Table 4: Analyzed Systems

5.2 Results

5.2.1 LOC counts

Tables 5 and 6 list the primary measurement results: the actual PLOC and LLOC metrics computed by the tools, respectively.

PLOC	Ant	ArgoUML	Cewolf	Hibernate	Liferay	PMD	Tomcat	Xerces
cloc	267,063	391,874	16,563	842,410	3,624,085	104,023	495,190	178,923
CodeAnalyzer	265,830	389,952	16,429	835,137	3,613,967	102,909	493,153	178,377
CodePro An.	265,788	389,952	16,429	832,072	3,612,853	102,909	492,610	178,377
LOCCounter	265,830	389,952	16,429	835,137	3,613,967	102,909	493,153	178,337
LocMetrics	267,061	391,837	16,563	841,867	3,613,969	103,974	495,054	178,922
Metrix++	-	-	-	-	-	-	-	-
SLOCCount	-	-	-	-	-	-	-	-
SonarQube	266,815[2]	391,837	16,563	-	3,613,969	103,969	494,823	175,412
SourceMeter	267,061	391,837	16,563	841,968	3,613,969	103,975	495,054	178,922
SourceMonitor	264,190	387,173	16,429	832,263	3,606,843	102,908	487,341	178,377
UCC	265,613	389,993	16,431	833,831	3,611,915	102,896	492,642	178,377

Table 5: PLOC comparison

LLOC	Ant	ArgoUML	Cewolf	Hibernate	Liferay	PMD	Tomcat	Xerces
cloc	135,225	195,363	8,476	506,597	1,985,827	68,374	275,379	85,543
CodeAnalyzer	135,066	195,496	8,476	506,577	1,985,127	68,079	274,672	83,776
CodePro An.	135,830	195,670	8,476	504,858	1,986,168	68,420	275,822	85,698
LOCCounter	135,778	195,672	8,475	506,582	1,986,904	68,418	276,211	85,600
LocMetrics	135,837	195,670	8,476	506,599	1,986,904	68,420	276,209	85,698
Metrix++	135,888	195,764	8,476	507,748	1,990,364	68,534	276,552	85,724
SLOCCount	135,715	195,670	8,476	506,481	1,986,904	68,420	276,081	85,698
SonarQube	135,733	195,670	8,476	-	1,986,904	68,415	276,033	83,699
SourceMeter	135,836	195,670	8,476	506,487	1,986,901	68,407	276,209	85,697
SourceMonitor	-	-	-	-	-	-	-	-
UCC	135,699	195,711	8,478	505,479	1,984,852	68,408	275,752	85,698

Table 6: LLOC comparison

From this data, several things can be observed. First, it is apparent that some of the tools are only capable of calculating one of the two metric types (Metrix++, SLOCCount, SoureMonitor). Secondly, we were unable to calculate either LOC metrics of Hibernate with SonarQube. This was because Hibernate contains a lot of Java source files which SonarQube could not analyze because they hold incomplete or incorrect example code for users to easily understand the usage (e.g. the code contains '...' between the important parts or the file contains some Java source code without a class or method).

There were notable differences between the collected results. One of the typical reasons for deviation was that some of the tools define LOC differently than others. For example, some tools calculate PLOC by counting 'carriage return' characters while others count the actual number of lines as would

[2]Since SonarQube analyzes all Java source files in the Ant directory but some of them were incorrect or incomplete Java files we had to make some minor modifications to the code in order for it to be analyzable by this tool.

appear in a text editor. This divergence can be observed in the case of Cewolf where the difference between the two typical PLOC values (16,563 and 16,429) is 134 – which is exactly the number of source files present (see Table 7). In the case of Cewolf almost all LLOC values were correct with the exception of two results which were slightly off. After examining these cases we found what was responsible for the gap. It was either empty lines containing only semicolons which were not counted or lines ending in double semicolon which were counted twice.

Another possible reason behind the fluctuation can be that most of the tools applied structural source code analysis (instead of lexical) because they calculate other metrics as well. However, some files contained invalid Java code which the tool could not understand, so these files were skipped. In general, the tools provided information about analysis problems and reported the files that were ignored.

The actual line counts are relatively similar, however there are very few cases where the data produced by different tools coincide. In case of physical lines, CodeAnalyzer and LOCCounter produce the same results, while SonarQube and SourceMeter give similar values as well. The situation is a bit different in the case of logical lines: only LocMetrics, SLOCCount, SonarQube and SourceMeter conincide in a few cases, the other results are diverse.

5.2.2 Relative differences for programs

The average difference can be observed better in relative terms, as listed in tables 7 and 8 for the two metrics.

PLOC	Ant	ArgoUML	Cewolf	Hibernate	Liferay	PMD	Tomcat	Xerces
Files	1,233	1,922	134	7,273	10,118	1,115	2,037	546
Avg	266,139.0	390,489.7	16,488.8	836,835.6	3,613,948.6	103,385.8	493,224.4	178,224.9
Min	264,190	387,173	16,429	832,072	3,606,843	102,896	487,341	175,412
Max	267,063	391,874	16,563	842,410	3,624,085	104,023	495,190	178,923
Diff	2,873	4,701	134	10,338	17,242	1,127	7,849	3,511
%	1.1%	1.2%	0.8%	1.2%	0.5%	1.1%	1.6%	2.0%

Table 7: PLOC system statistics

LLOC	Ant	ArgoUML	Cewolf	Hibernate	Liferay	PMD	Tomcat	Xerces
Files	1,233	1,922	134	7,273	10,118	1,115	2,037	546
Avg	135,660.7	195,635.6	8,476.1	506,378.7	1,986,685.5	68,389.5	275,892.0	85,283.1
Min	135,066	195,363	8,475	504,858	1,984,852	68,079	274,672	83,699
Max	135,888	195,764	8,478	507,748	1,990,364	68,534	276,552	85,724
Diff	822	401	3	2,890	5,512	455	15,963	2,025
%	0.6%	1.9%	0.0%	0.6%	0.3%	0.7%	0.7%	2.4%

Table 8: LLOC system statistics

Here, the minimum and maximum values, as well as the maximum differences are provided over the examined tools. The differences are given in absolute values as well as in percentages relative to the minimal metric values. The average differences were around 0.5–2% for both metrics, which we think is an acceptable level of deviation.

The other interesting thing is that one would expect the differences in PLOC to be smaller than LLOC because calculating physical lines is much more exact. Surprisingly, this was generally not the case. As extremes, Cewolf, Hibernate and Tomcat showed at least twice as big difference in PLOC than LLOC, and only ArgoUML was better in the case of PLOC.

5.2.3 Differences among the tools

Finally, we calculated the average deviation of the specific tools from the average LOC counts computed from all of the results. As we were unable to declare a "ground truth" for the correct metric values – given it does not exist – we could not determine the absolute precision of the tools. Nevertheless, we were able to compare the examined tools in terms of their relative deviation from the average. Tables 9 and 10 show the related results. Here, in each cell the difference is shown in percentage relative to the average for the respective subject system (the averages are shown in the second rows of tables 7 and 8).

The last column of these tables shows the average of the deviation values per each program. Overall, the deviations range from 0.1% to about 0.5% for both PLOC and LLOC. In the case of physical lines the most precise tools were ‘CodeAnalyzer’ and ‘LOCCounter’, however for calculating logical lines ‘LOCCounter’ and ’SLOCCount’ proved to be closest to the average. Regarding the furthest cases, we could not clearly identify tools that performed worst in this respect.

PLOC	Ant	Argo.	CeWolf	Hibern.	Liferay	PMD	Tomcat	Xerces	Min	Max	Avg
cloc	0.35%	0.35%	0.45%	0.67%	0.28%	0.62%	0.40%	0.39%	0.28%	0.67%	**0.44%**
CodeAnalyzer	0.12%	0.14%	0.36%	0.20%	0.00%	0.46%	0.01%	0.09%	0.00%	0.46%	**0.17%**
CodePro An.	0.13%	0.14%	0.36%	0.57%	0.03%	0.46%	0.12%	0.09%	0.03%	0.57%	**0.24%**
LOCCounter	0.12%	0.14%	0.36%	0.20%	0.00%	0.46%	0.01%	0.06%	0.00%	0.46%	**0.17%**
LocMetrics	0.35%	0.35%	0.45%	0.60%	0.00%	0.57%	0.37%	0.39%	0.00%	0.60%	**0.39%**
Metrix++	-	-	-	-	-	-	-	-	-	-	**-**
SLOCCount	-	-	-	-	-	-	-	-	-	-	**-**
SonarQube	0.25%	0.35%	0.45%	-	0.00%	0.56%	0.32%	1.58%	0.00%	1.58%	**0.50%**
SourceMeter	0.35%	0.35%	0.45%	0.61%	0.00%	0.57%	0.37%	0.39%	0.00%	0.61%	**0.39%**
SourceMonitor	0.73%	0.83%	0.36%	0.55%	0.20%	0.46%	1.19%	0.09%	0.09%	1.19%	**0.55%**
UCC	0.20%	0.13%	0.35%	0.36%	0.06%	0.47%	0.12%	0.09%	0.06%	0.47%	**0.22%**

Table 9: PLOC tool statistics

LLOC	Ant	Argo.	CeWolf	Hibern.	Liferay	PMD	Tomcat	Xerces	Min	Max	Avg
cloc	0.32%	0.14%	0.00%	0.04%	0.04%	0.02%	0.19%	0.30%	0.00%	0.32%	**0.13%**
CodeAnalyzer	0.44%	0.07%	0.00%	0.04%	0.08%	0.45%	0.44%	1.77%	0.00%	1.77%	**0.41%**
CodePro An.	0.12%	0.02%	0.00%	0.30%	0.03%	0.04%	0.03%	0.49%	0.00%	0.49%	**0.13%**
LOCCounter	0.09%	0.02%	0.01%	0.04%	0.01%	0.04%	0.12%	0.37%	0.01%	0.37%	**0.09%**
LocMetrics	0.13%	0.02%	0.00%	0.04%	0.01%	0.04%	0.11%	0.49%	0.00%	0.49%	**0.11%**
Metrix++	0.17%	0.07%	0.00%	0.27%	0.19%	0.21%	0.24%	0.52%	0.00%	0.52%	**0.21%**
SLOCCount	0.04%	0.02%	0.00%	0.02%	0.01%	0.04%	0.07%	0.49%	0.00%	0.49%	**0.09%**
SonarQube	0.05%	0.02%	0.00%	-	0.01%	0.04%	0.05%	1.86%	0.00%	1.86%	**0.29%**
SourceMeter	0.13%	0.02%	0.00%	0.02%	0.01%	0.03%	0.11%	0.49%	0.00%	0.49%	**0.10%**
SourceMonitor	-	-	-	-	-	-	-	-	-	-	**-**
UCC	0.03%	0.04%	0.02%	0.18%	0.09%	0.03%	0.05%	0.49%	0.02%	0.49%	**0.12%**

Table 10: LLOC tool statistics

5.3 Discussion

We wanted to find the causes of the major differences between the results so we manually investigated several notable data points, and came to the following typical reasons:

- Some tools could not deal with lines of syntactically incorrect code. This usually affected both types of metrics calculated.
- The calculation of the metric was different than we expected based on the tool documentation, which could be generally attributed to unexpected, undocumented or perhaps defective behaviour of the tool. This affected one or both of the metrics.
- There were differences in the definitions and interpretation of the metrics such as the case of ‘carriage return’ characters vs. number of lines (affects PLOC), or the ignorance of empty semicolons (affects LLOC).
- Handling of non-Java files or file parts of the project. If a tool analyzes a non-Java file, the errors encountered are handled differently (e.g skipping some parts of the code or ignoring the whole Java file). This issue regards also various special code such as tests, which were analyzed by some tools and skipped by others.

6 Conclusions

The results are overall not surprising. We were certain that there will be differences among the calculated values, yet the amount was unknown. In general, we think that the relative differences of 1–2% are not over-the-top. However, in absolute terms there were significant differences of several thousand lines occasionally, which could cause confusion in certain uses.

A surprising find was that even counting physical lines is not a straightforward task, so one may not expect exactly the same results from different tools.

LOC will continue to be one of the most frequently used metrics in various fields of software engineering, so we encourage practitioners to carefully plan their measurement and the usage of tools. The most important steps are to make sure that the actual definition of the LOC concepts are clear from the beginning and they are *consistent* within the organization or project.

If solely LOC calculation is required, it is probably not that important which tool to use for the calculation as long as the users are aware of how the metrics are defined and calculated by the tool. However, if the tool should be used for other purposes as well – such as calculating different metrics or integrating it to more complex measurement process –, other aspects of the available solutions must be taken into account as well.

References

[1] Norman E. Fenton and Martin Neil. Software metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)*, pages 357–370. ACM, 2000.

[2] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. In *IEEE Transactions on Software Engineering*, volume 31, pages 897–910. IEEE Computer Society, October 2005.

[3] Capers Jones. Software metrics: Good, bad and missing. *Computer*, 27(9):98–100, September 1994.

[4] Stephen H. Kan. *Metrics and Models in Software Quality Engineering, 2nd Edition*. Addison-Wesley Professional, 2002.

[5] Linda M. Laird and M. Carol Brennan. *Software Measurement and Estimation*. John Wiley & Sons, Inc, 2006.

[6] Robert E. Park. Software size measurement: a framework for counting source statements. Technical Report CMU/SEI-92-TR-020, Software Engineering Institute, Carnegie Mellon University, September 1992.

[7] Jarett Rosenberg. Some misconceptions about lines of code. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 137–142. IEEE Computer Society, November 1997.

[8] Linda G. Wallace and Steven D. Sheetz. The adoption of software measures: A technology acceptance model (TAM) perspective. *Information & Management*, 51(2):249–259, 2014.

[9] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *IEEE International Conference on Software Maintenance (ICSM'09)*, pages 274–283. IEEE Computer Society, September 2009.