Reading Beside the Lines: Indentation as a Proxy for Complexity Metrics

Abram Hindle
University of Waterloo
Waterloo, Ontario
Canada
ahindle@cs.uwaterloo.ca

Michael W. Godfrey
University of Waterloo
Waterloo, Ontario
Canada
migod@cs.uwaterloo.ca

Richard C. Holt
University of Waterloo
Waterloo, Ontario
Canada
holt@cs.uwaterloo.ca

ABSTRACT

Assessing the maintainability of revisions to source code is often poorly supported by tools. Traditional maintainability metrics are commonly defined in terms of complexity metrics (such as McCabe or Halstead) and lines of code (LOC). However, these approaches are problematic when applied to code fragments written in multiple languages: special parsers are required which may not support the language or dialect used; analysis tools may not be able to understand code fragments; and since so many deltas comprise of only a few lines of code, LOC is a poor discriminator for complexity if used in isolation. In this paper, we propose and evaluate the use of statistical moments of the depth of indentation as a proxy for traditional complexity metrics. This approach is largely language independent and is fast and easy to compute accurately on code fragments. We have extensively evaluated our approach against the entire CVS histories of the 278 of the most popular and most active SourceForge projects, and we found that our results are linearly correlated and rank-correlated with traditional measures of complexity, suggesting strongly that measuring indentation is a cheap and accurate proxy for code complexity of revisions.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures*

Keywords

Whitespace, indentation, maintainability

1. INTRODUCTION

Assessing the maintainability of large evolving software projects presents many technical challenges. Such systems are often heterogeneous: they contain many subcomponents written in multiple languages, and are stored using a variety of repository mechanisms. However, maintainability metrics are commonly language dependent, and computing them requires tools that typically assume access to the full definitions of the software entities.

This paper focuses on assessing the maintainability of code revisions: that is, code fragments that represent the difference between old and new versions of software entities. Consequently, we seek approaches that are language neutral and do not require complete compilable sources.

We have taken this approach because revisions are the currency of ongoing development. Developers and managers typically want to understand what has happened to the code base since the last revision. They want assurances that new code will implement the desired enhancement or bug fix, and will not break the existing system. Existing analysis techniques and tools commonly assume access to compilable source-code. By contrast we seek ways of reliably and efficiently analyzing arbitrary code fragments — not necessarily compilable – representing revisions to source code. If we can measure or estimate the complexity of source code changes we can better rank the maintainability of these changed code fragments, such as revisions in a source control repository. In turn, we could help maintainers identify complex and error prone patches; we could help managers reason about the maintainability of branch merges.

We have looked at how maintainability metrics, such as the maintainability index [14], are composed. Often these metrics incorporate complexity (McCabe's Cyclomatic Complexity [11] (MCC) and Halstead's Complexity metrics [8]) and lines of code (LOC) into their measurements. If we cannot measure complexity because we lack a parser for a particular language or tool, we could use a proxy for complexity. LOC is generally considered such a proxy but it is already used within the maintainability index. As well, LOC is not useful to measure revisions, especially if we have many changes of 4 or less lines, how do we know which are actually complex? Thus we need a proxy of complexity which is easy to calculate, does not require grammar-level semantics of the language and is relevant across many languages. In this paper we propose such a proxy: the statistical moments of indentation.

Variance and standard deviation of indentation and the summation of indentation should be good indicators of the complexity of source code. If the indentation of a code block varies, this implies there are changes at multiple levels of scoping. We assume that properly indented programs reveal their structure simply by indentation. For imperative and procedural code indentation indicates functions and control structures such as conditionals and loops. For OO languages such as C++ and Java, indentation can indicate encapsulation and depth of encapsulation via classes, subclasses, and methods. Whereas for more functional languages such as OCaml, Scheme and Lisp indentation indicates new scope, new closures, new functions, and new expressions.

Multiple branches in source code imply a larger variance or standard deviation of indentation, thus statistical moments of indentation (the set of summary statistics about the indentation) serves as a proxy to McCabe's Cyclomatic Complexity (MCC) as MCC counts branching paths in code. The summation of indentation proxies LOC and complexity in the sense that the deeper the code and the more lines the more complex the code. Most code is shallow (0 to 2 levels of indentation (LIL) deep), thus deep code is rare (in section 3.2 we show this fact).

How regular is indentation? In section 3.2 we provide an analysis showing indentation to be very regular across all languages. Many programmers use indentation to aid the readability of source code. Some programming languages like Python, require indentation to indicate and maintain scope. The best practices for the majority of languages used today — C, C++, Java, Perl, Python, Ruby, PHP, etc. — dictate that software should be properly indented so that it can be made more readable [4, 13].

What we propose, and show, is that statistical moments of indentation (average (AVG), median (MED), variance (VAR), standard deviation (STD), sum of indented lines (SUM)) are correlated with complexity metrics like McCabe Cyclomatic Complexity and Halstead Complexity. As well, we show these statistical moments of indentation can be combined in a linear manner which correlates with the associated complexity measures.

Our contributions in this paper include:

- An empirical survey of indentation of popular OSS projects found on Source-Forge.
- Evidence that there is a correlation between statistical moments of indentation and complexity, thus implying a relationship between statistical moments of indentation and maintainability.

1.1 Previous Work

Indentation is often promoted for helping program readability [13] and defining structure [15] in code. It can be used to trace a program's flow [10] and has been shown to be generally helpful for program comprehension [12].

Indentation is often used as a maintainability metric and for software visualization to provide an overview of the program structure [5]. Gorla et al. [7] uses inappropriate indentation as a code quality metric. Coleman et al. [3] measures horizontal spacing (indented/non-indented) of source code. [1] compares the indentation characters versus the non-indentation characters. Other uses of indentation include plagiarism detection [2].

For complexity there are many metrics, two popular metrics are McCabe's Cyclomatic Complexity (MCC) and Halstead's complexity metrics [11] [8]. We are interested in these two complexity metrics because many studies like that of Oman et al. [14] use these metrics in calculations of maintainability metrics.

McCabe Cyclomatic Complexity (MCC) counts branching and control flow in a program, this means counting control flow structures and tokens. In most programming languages the accepted style is to indent code that is within structures like if blocks, for loops, exceptions, etc. This implies that some indentation is analogous to branching code paths and thus similar to MCC.

Other complexity metrics, such as Halstead's complexity metric, measure the number of unique operators and operands. Although each metric measures something different they all seem to be correlated with LOC [9]. We applied MCC and Halstead complexity to source code revisions, which relates to modification-aware change metrics as discussed by German et al. [6].

We consider raw indentation to be the actual preceding white space on each line. Logical indentation is the depth of indentation that the programmers meant to portray. In many cases 4 spaces, 8 spaces, or a tab could all be equivalent to one unit of logical indentation. Logical indentation is the unit in which the depth of indentation is measured, where as raw indentation composes logical indentation.

For both logical and raw indentation we will refer to lines of code as LOC, standard deviation as STD, average as AVG, median as MED, variance as VAR, summation as SUM. For logical indentation we will use LSTD, LAVG, LMED, LVAR, LSUM. For raw indentation we will use ISTD, IAVG, IMED, IVAR, ISUM.

1.2 Motivation

Metrics have prerequisites, that is if you need to meet the requirements of the metric before you apply it. If we want to measure the number of classes in some source code, we need the source code, we need to know how to parse the classes and the source code must be parsable. In our case, source control repositories and revisions, most of the source code we are parsing will not compile because we're looking at small chunks of source code that might not even compile, even if we extracted the whole project. We care about partial source code because in this paper we are operating solely on revisions to source code, rather than full checkouts of source code. Not all code is parseable or complete, many metrics are difficult to apply to incomplete code, or they are inaccurate. Some metrics require an AST of the program, others a concrete syntax tree (CST), while others still such as LOC, need only the raw text of the source code.

Metrics also have properties such as run time, development cost (parsers to parse the source code), as well as granularity: some metrics are only concerned with files, some care about LOC, some are concerned with tokens.

Our metrics exist within a metrics space: indentation measurements cost computationally just a bit more than measuring LOC, but they cost less than tokenizing and parsing source code for metrics such as Halstead and MCC. In this paper we show that indentation metrics can be used as a proxy for complexity.

Table 2 shows where our metrics fit in within the space of other metrics. This table orders metrics by their level of language and semantic awareness. Level of awareness is how much information a metric needs about the system it is measuring, LOC does not need to know what language it is measuring where as Number of Comment Lines needs to know what kind of language it is measuring. Semantic awareness requires more than just knowing what language is being parsed, it needs to understand and know about components of the language such as tokens, conditionals, comments, statements and expressions.

Metrics can vary by their difficulty of implementation and their computational performance, for example LOC can be implemented with a simple character search. Indentation measurements can be implemented using a simple scanner, where as to implement a metric that counts tokens, such as Halstead's complexity metric, requires a tokenizer for that particular language. Using our tool we found that tokenizing took about to 2 to 4 times more time than just counting indentation.

In general, the more information you need about a language and its semantics the more work one needs to put into the development of tools which can operate at that semantic level and often the more computationally intensive these tools become.

The rest of this paper has the following structure: we introduce our methodology in section 2, we provide an overview of the indentation we encountered (section 3), we show how the indentation of diffs relate to complexity metrics of the revisions, we discuss our results, then suggest threats to validity and conclude.

2. METHODOLOGY

Our methodology can be summarized as:

- Choose projects to study and download them. We get a list of the most active and most downloaded projects from Source-Forge. We download the CVS repositories of these projects.
- Process the revisions. For each file, we extract each individual revision and we analyze the indentation of the new code.
- Run complexity metrics. We calculate the complexity metrics for each revision.
- Correlate the indentation measurements and the complexity metrics. We then analyze the results and extract correlations between complexity and the indentation metrics.

2.1 Project Repositories

We downloaded the CVS repositories of the top 250 Most Active Source Forge projects and top 250 Most Popular (downloaded) Source Forge projects (as provided by Source Forge on their website). This resulted in 278 projects since the two groups overlap and not all projects had CVS repositories available at the time.

2.2 Extraction and Measurement

For each revision to C, C++, Java, Perl, PHP, and Python files, we analyzed the new and revised code. If one revision wasn't contiguous we just evaluated the changed code blocks (which we call diff-chunks, see figure 1 for an example diff-chunk). We extracted about 13 million diff-chunks, evaluating only the changed-to code (the new code). We did not measure the initial commit as many files are simply imported and would likely skew the results, as well there was no previous revision, so there was no changed-to data. We measured raw indentation and then calculated the logical indentation as described in section 3.1.

We measured each chunk by its LOC, and then we measured the statistics of raw and logical indentation of the diff-chunk: average (IAVG and LAVG), median (IAVG and LMED), standard deviation (ISTD and LSTD), variance (IVAR and LVAR), and summation of indentation per line (ISUM and LSUM). Also, we counted the frequency of indentation depth to produce histograms. Figure 1 provides an example of our measurement of a diff-chunk.

We also calculated MCC and Halstead Complexity metrics per each diff-chunk. Each metric used a tokenizing strategy so running the metrics on broken code was straightforward. We used the full population of each dataset of diff-chunks from each repository, minus values that were removed because they contained or caused metrics to produce values such as Infinity or NaN (not a number).

Since we were using many languages and partial chunks of source code we had to make our own Halstead and McCabe metrics for C, C++, Java, Perl, Python and PHP. This helped us to maintain consistency across the measurements between languages, as well allowed us to act on the diff-chunks. We had 51GB of CVS repositories and it took about 3 days of runtime to measure each revision of every repository on an Intel Pentium IV; this resulted in 13 million diff chunks.

2.3 Analysis

To analyze the results we extracted, we used various statistical tools for comparing distributions of indentation depth and calculating correlations. Our data distributions were usually discrete and positive. The matching distributions often included the Pareto distribution, the Poisson distribution, the Binomial distribution and the Exponential distribution. We also use summary statistics on the count data.

To show a similarity between indentation styles (the kind of indentation used) we compare the distributions of indentation of sets of revisions (indentation per revision per language). We expect that similar indentation distributions suggest similar styles of indentation, coding, indicating scope and similar semantics. For instance C and C++ should be similar since C++ and its syntax was derived from C.

To compare distributions we use the Kologormov Smirnov test. This is a non-parametric test that can handle data that has troublesome non-normal, non-Gaussian distributions like the exponential distribution. It does so by measuring the maximum distance between two cumulative distribution functions. To characterize the indentation of a language we used the distribution of indentation depth. We then used the Kologormov Smirnov test to compare these distributions.

If one measurement is similar or related to another measurement, if it can replace the other, we need to show there is a relationship between them. The easiest way to show a relationship between two variables is to see how well they correlate. We use correlation in this paper to show a relationship between indentation metrics and code complexity metrics, thereby showing that one could potentially replace the other.

To determine correlations between variables we use two kinds of correlations: linear correlation and rank-based correlation. The difference is great: a linear correlation attempts to show the strength of a linear relationship between two or more variables. A rank-based correlation does not rely on a linear relationship, it orders the variables, ranking them from smallest to largest and then correlates those ranks with the rank of the other variable. Thus if the high ranked values for the first variable occur often with low ranked values of the second variable, the rank-based correlation will be negative; if a high rank of one variable frequently corresponds to

```
> void square(int * arr, int n) {
>     int i = 0;
>     for ( i = 0 ; i < n ; i++ ) {
>         arr[i] *= arr[i];
>     }
> }
```

Metric	Raw	Logical
LOC	6	6
AVG	3.33	0.833
MED	4	1
STD	2.75	0.687
VAR	9.07	0.567
SUM	20	5
MCC	2	2
HVOL	142	142
HDIFF	15	15
HEFFORT	2127	2127

Figure 1: An example diff-chunk with corresponding indentation and complexity metrics

a high rank of the second variable the correlation will be positive. Our linear correlation is the Pearson Correlation Coefficient , our rank based correlations are the Spearman-Rho Correlation Coefficient and the Kendall-Tau Correlation Coefficient . All three of these correlations produce values between -1 and 1 where 0.1 to 0.25 indicates a weak positive correlation (0 indicates no correlation), 0.25 to 0.75 indicates a medium positive correlation and 0.75 to 1.0 indicates a strong positive correlation (and vice versa for negative correlations).

If there is truly a linear relationship, as suggested by a linear correlation, we should be able to build a linear model of complexity using indentation. The linear model of indentation should be able to do better then a model composed of only LOC. Thus to further support assertions of linear correlation, we use Least Squares Linear Regression to produce a best fit of coefficients of our statistical moments of indentation to both MCC and Halstead complexity metrics. This method uses an R^2 measure, which indicates the percent of the variation between the model and the data that is accounted for by the model. Larger values of R^2 (0.6 or greater) indicate a good linear fit.

To calculate these correlations we developed our own software in OCaml which parallelized the correlation calculations for Kendall-tau because Kendall-tau correlation has a algorithmic complexity of $O(N^2)$, while Spearman correlation has a complexity of $O(Nlog(N))$. This was a problem because we had 13 million diff-chunks to correlate. Our largest correlation, run on the C language, was on about 4 million diff-chunks. Our correlations took 8 CPU years to calculate (which was collapsed down to a few actual weeks on a cluster).

3. INDENTATION OF REVISIONS

In this section we give an overview of the data we are analyzing. We have the source code repositories of 278 Projects, of which, we evaluate 6 languages (C, C++, Java, PHP, Perl, Python). We characterized the indentation depth distributions of the languages and projects; we related the languages with each other via their distributions.

3.1 Distributions of Indentation Depth

In general for all projects and languages we found that the actual indentation follows a base 4 rule (raw indentation depth is usually divisible by 4, a single logical unit of indentation was 4 spaces). A logical unit of indentation is depth of nesting a programmer wanted to convey; for instance, inside of an if block a programmer probably often means to indent the conditional code 1 more unit of logical indentation, regardless if they use tabs or spaces to achieve that. If tabs are used, they act as a single unit of logical indentation. Tabs are often used to represent an even number of spaces of indentation. One must note, this is not the indentation of a released product, this is the indentation per diff in the CVS repository.

File Type	Dist.	Logical Dist.	Gap Dist.
.c files	Exp	Poisson	Exp
.cpp files	Exp	Binomial	Exp
.h files	Pareto	Poisson	Exp
.java files	Exp	Binomial	Poisson
.php files	Pareto	Exp	Exp
.pl files	Exp	Exp	Exp
.py files	Exp	Poisson	Exp

Table 1: Distributions of file collections

In figure 2 we can see spikes appearing at line numbers which are divisible by 4. Tabs were considered to be 8 characters in depth. The spikes in the plots seem to indicate that the data is composed of 2 distributions, the distribution of the peaks and the distribution between the peaks. We evaluated both distributions, the peak distribution and gap distribution, and for all languages they had either exponential, binomial, Pareto or Poisson distributions (only Java's gap distribution were Poisson). Table 1 records the closest distributions of the indentation of different languages.

In figure 3 we can see that logical indentation follows a set of distributions such as the Exponential, Poisson, and binomial distributions. It should be noted that all 4 distributions, exponential, Pareto, Poisson and Binomial are related in shape and sometimes definition; usually they rely on exponentiation.

These distributions, in table 1, were determined by best fit parameter fitting of distributions using Kologomorov-Smirnov tests.

3.2 Language Analysis

Java files were the only files to not have an exponential gap distribution. This is apparently because java has many lines of non-even raw indentation with an initial indentation of one are common. Since all methods must be within a class, Java programmers apparently try to save screen space by indenting in only one space for the initial class. Java's logical indentation distribution was closest to a Binomial distribution with a p value of 0.017. Java is binomial because of the tall peak at Logical Indentation Level (LIL) 2.

Header files (.h files) for C and C++ were predictably

Files
Bytes
Characters
Lines
Indentation
Classes and modules
Methods and functions
Expressions and statements
Halstead Complexity
McCabe Cyclomatic Complexity

Table 2: Metric-space of metrics ordered by Language Dependence and Semantic Awareness (least to most)

indented very little. LIL 0 was the most popular followed by LIL 1. LIL 1 was composed of 4 spaces or 1 tab. There were many lines indented by 1 or 2 spaces but there were more lines of LIL 1. According to figure 5, header files have the least similar logical indentation distribution.

Perl's indentation distribution is the closest to C and PHP, although it shares some relation with Python and .c files. This might be because classes in Perl do not require further indentation since they denoted by a `package` keyword. Often, Perl code uses 4 spaces and tabs although sometimes 2 spaces are used. All of the Perl indentation distributions follow an exponential distribution to any other language.

Python's logical indentation distribution is the most similar to Java files. Python is a unique language that uses indentation to indicate scope, that is, indentation has semantics. Python's logical units of indentation were very consistent, either 4 spaces or 1 tab. More lines were indented at LIL 1 or LIL 2 times than at LIL 0. Notably, Python's logical indentation distribution matched with a Poisson distribution.

PHP's indentation was the most similar to C and Perl. PHP stood out because it had some common deep indentations with logical units of 1 tab and 4 spaces. It appears that due to the mixing of HTML and PHP code that the logical indentation units of PHP ends up being mixed between spaces and tabs.

C++ files (.cpp files) were the most similar with .c files and were somewhat similar with Perl files. Perl and C++ define methods similarity so this might have been the case. C++ files had definite pronounced gap heights, 2 spaces was quite common although most files followed a 4 spaces or tabbed indentation. 0 to 2 LILs were common with C++.

C files (.c files) were very similar to C++ files in distribution and style. 2 spaces were common units, although 4 spaces or tabs dominated. C files were more similar, indentation-wise, than C++ files with Perl and PHP indentation.

4. INDENTATION AND COMPLEXITY

In this section we correlate complexity metrics like Halstead complexity and McCabe's Cyclomatic complexity with moments of indentation.

For McCabe's Cyclomatic Complexity we measure the MCC and the number of return statements. The Halstead metric is a set of measurements of tokens: length (HLEN), vocabulary (HVOCAB), volume (HVOL), difficulty (HDIFF) and effort (HEFFORT). We correlated these metrics against the indentation metrics for raw indentation and logical indentation. Our metrics were the statistical moments of raw and logical indentation: LOC, IAVG and LAVG, IMED and LMED, ISTD and LSTD, IVAR and LVAR, ISUM and LSUM.

4.1 Measures and Correlation

Our observation was that the AVG and MED did not correlate well with any of the complexity metrics for both linear correlation (Pearson) and rank-based correlation (Spearman and Kendall).

LOC, SUM, STD, and VAR had medium strength (0.4 to 0.6) rank based correlations and small linear correlations (0.2 to 0.4) with the complexity measures such as Halstead Difficulty (HDIFF) and MCC. For MCC, LOC had a linear correlation of 0.75 and a rank-based correlation of 0.41 to 0.45. For HDIFF, LOC had rank and linear based correlation of 0.49 to 0.55.

Halstead has count-based metrics such as Halstead length, Halstead vocabulary and Halstead volume (these linearly correlated well with LOC and SUM). Halstead difficulty and Halstead Effort try to estimate complexity based on the number of unique operands and operators versus the total number of operands and operators. Halstead Effort is supposed to model the time it took to code that source code, which correlates best with LOC in most cases.

4.2 Complexity and Language

In general, rank based correlations showed that SUM and STD correlated better with complexity than LOC did. For linear correlations LOC usually faired better than SUM. Figures 6 and 7 depict the correlation coefficients of SUM and STD. The Halstead length metrics all correlated best with LOC, both with linear and rank-based correlations.

The C files had low scores for Pearson correlation, with MCC correlating better with SUM than LOC. Rank based correlations confirmed that LOC was correlated with complexity measures but also that STD and VAR were important. Kendall correlation coefficients were lower than Spearman coefficients. Both Spearman and Kendall correlation of STD (Spearman 0.48, Kendall 0.44) were more correlated with MCC than LOC (Spearman 0.43, Kendall 0.39).

For C++, SUM correlated linearly with MCC (0.79) more than LOC (0.73). Although with rank based correlation STD and VAR of indentation were equally correlated with MCC (0.45)

For .h files LOC, SUM, then STD, in descending order, correlated well with HDIFF and MCC. Surprisingly SUM correlated well with the number of returns and complexity of functions and methods in .h files.

For Java LSUM linearly correlated with complexity better than LOC (0.77 versus 0.76). For rank based measures STD and VAR had medium correlations with MCC and HDIFF (0.43 – 0.45).

For PHP, rank based correlations of MCC with STD and SUM were better correlated than LOC. For linearly correlation both SUM and LOC were correlated to complexity.

Python files were interesting as their linear correlation between LOC and complexity was relatively low (0.64 and 0.49). STD had a medium linear correlation with HDIFF (0.39).

For Perl, STD was more correlated linearly with HDIFF than with LOC (0.47 versus 0.42), although LOC strongly linearly correlated with MCC (0.75). For rank based correlations STD is correlated best for MCC (Spearman 0.52, and Kendall

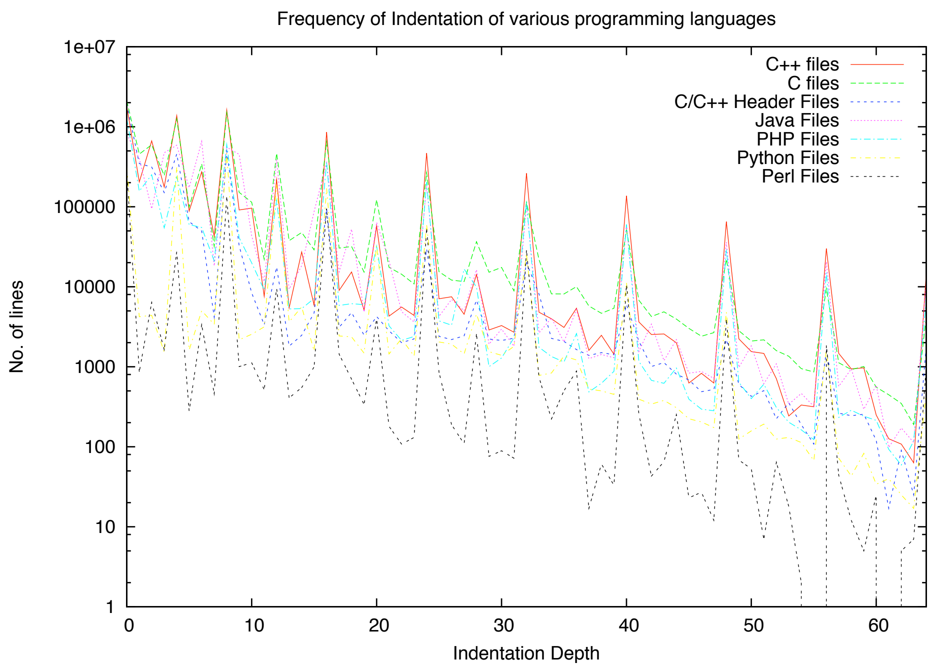

Figure 2: Frequency of Physical Indentation of various languages (Log Scale)

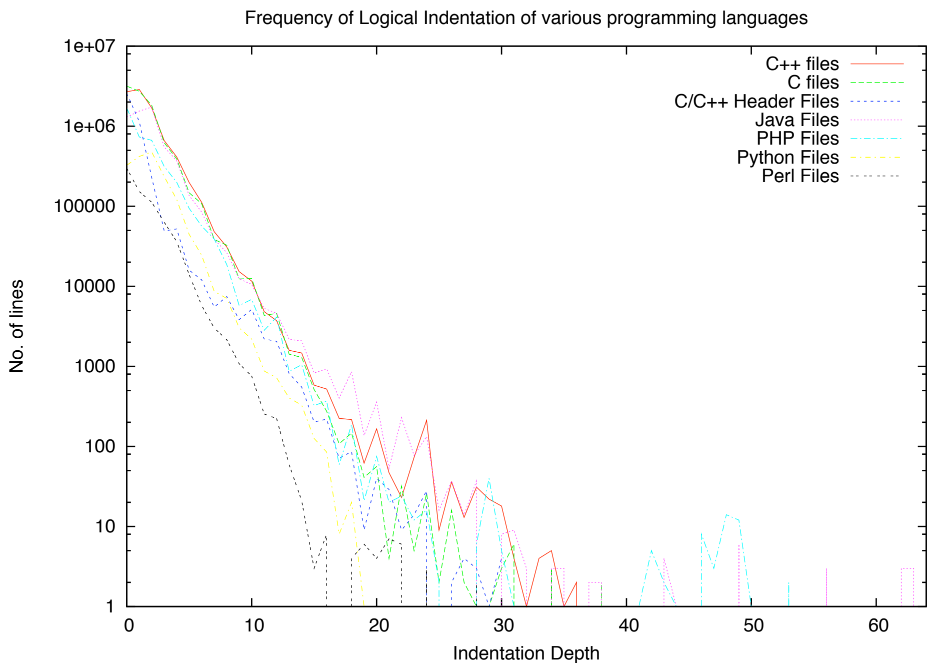

Figure 3: Frequency of Physical Logical Indentation of various languages (Log Scale)

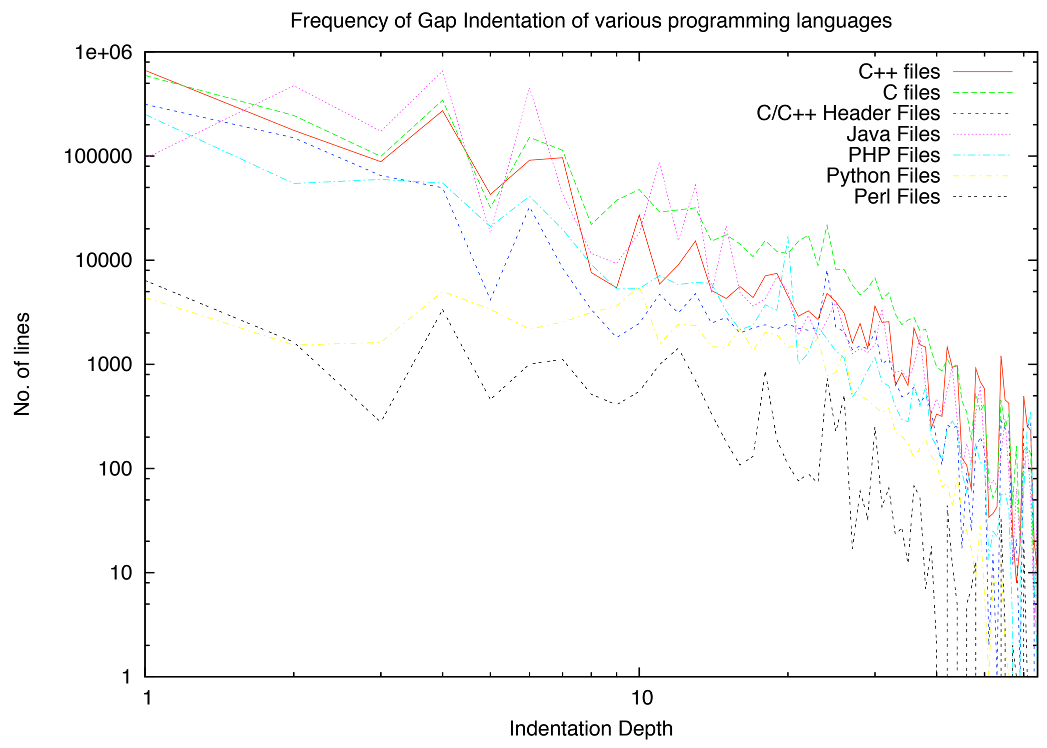

Figure 4: Frequency of Gap Indentation (non-logical units of indentation) of various languages (Log Scale)

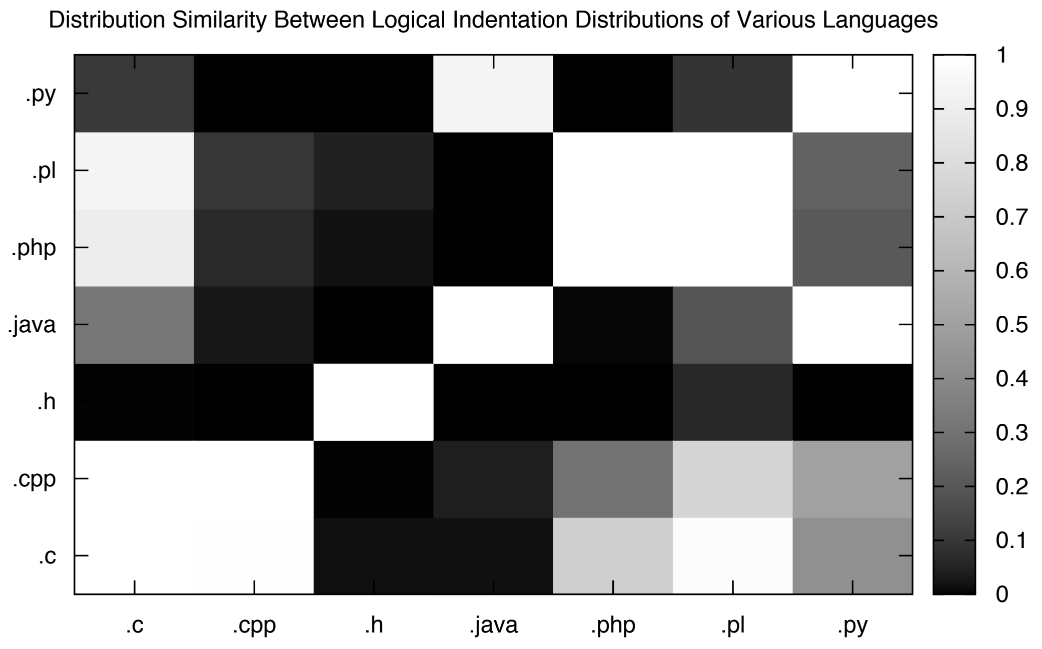

Figure 5: Logical Indentation Distribution Similarity between Languages (P-value 1.0 indicates they are very similar, Probability 0.05 or less suggests that there is only a 5% or less chance the distributions are similar)

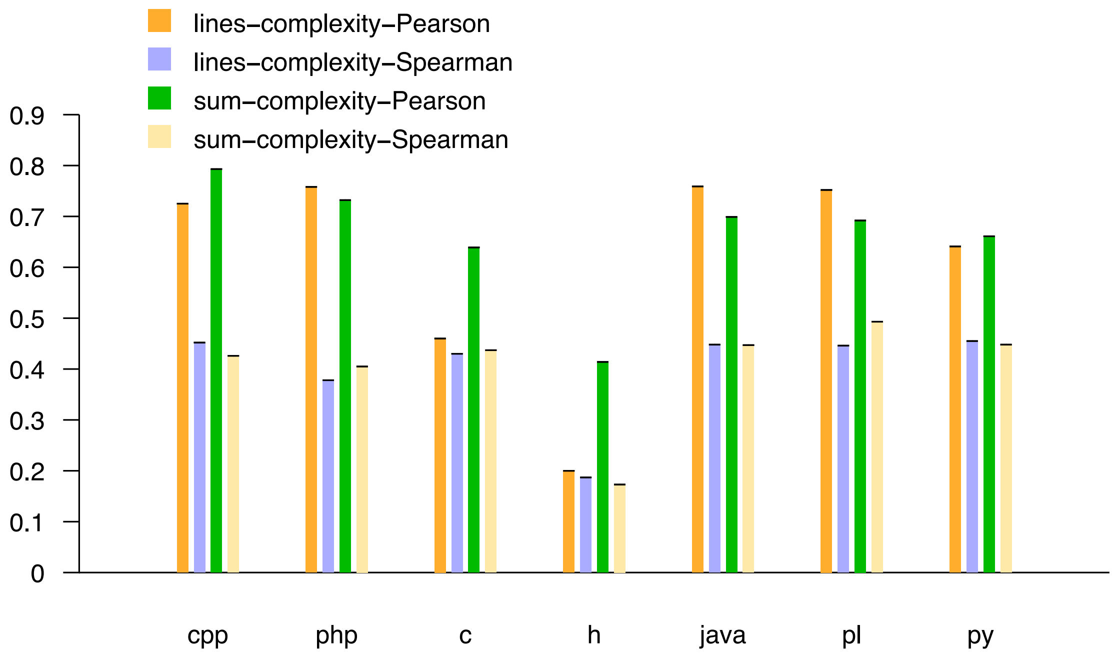

Figure 6: Correlation of Complexity to Summation of Indentation and Complexity to LOC

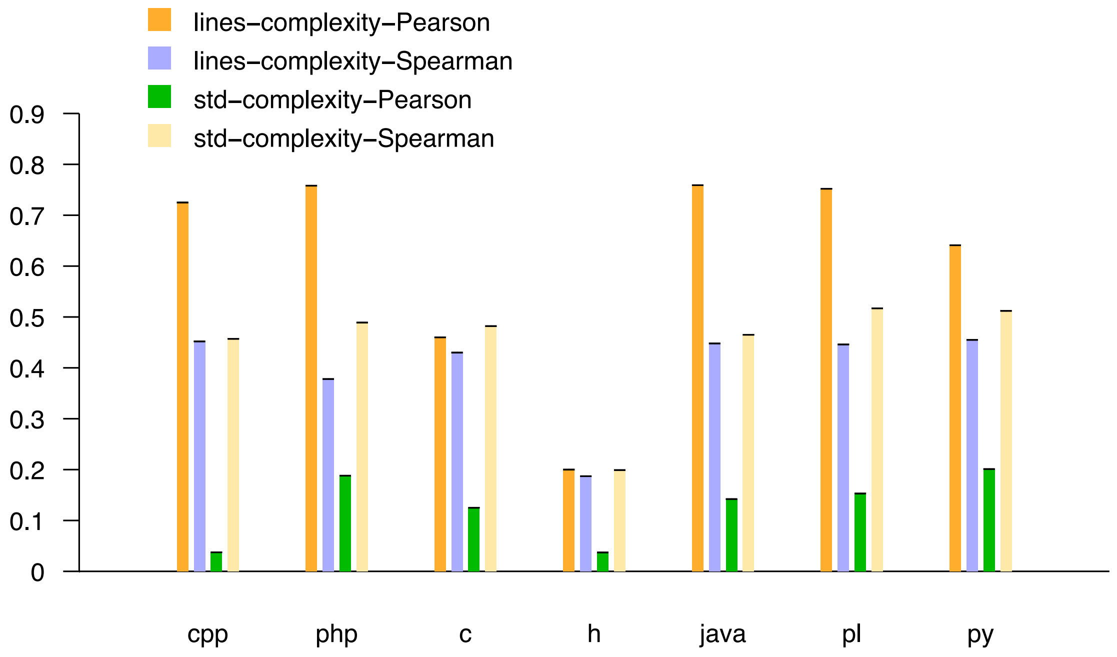

Figure 7: Correlation of Complexity to Standard Deviation of Indentation and Complexity to LOC

Coefficient	Value
-0.03	IAVG
0.05	IMED
-0.05	ISTD
4.81e-05	IVAR
3.93e-05	ISUM
0.16	LMED
0.28	LSTD
-0.10	LVAR
0.08	LSUM

Table 3: Coefficients for the linear model of complexity. This model had a R^2 of 0.385

0.47) and SUM correlated best with HDIFF (Spearman 0.47 and Kendall 0.44).

Thus for all the languages we have shown strong and medium linear correlations between MCC and HDIFF with LOC and SUM. We have also shown for all languages studied, there was medium strength linear and rank-based correlations between complexity, HDIFF and STD and VAR.

4.3 Complexity Correlation per Project

For most projects, LSUM and ISUM had a greater linear correlation for MCC than LOC, although LOC was just above SUM for HDIFF. For rank based correlation STD and SUM were better correlated than LOC for complexity, but LOC was better correlated for HDIFF.

Some projects had relatively strong linear (0.55 to 0.67) correlations between complexity and STD, such projects included: Sodipodi, Bittorrent, Dynapi, Aureal, PHPnuke. Some projects which did not linearly correlate (0.01 to 0.07): CrystalSpace, sed, jedit, BOOST. For rank based correlations, Bastille-Linux, Unikey, Sodipodi and OpenCV were above 0.67 (Bittorrent, Dynapi, Aureal, and PHPNuke were all above 0.5).

4.4 Linear Combinations

To further show the linear relationship between indentation metrics and complexity metrics we tried to linearly model each of MCC and Halstead difficulty with our moments of indentation. Our model is:

$$c = \alpha_0\beta_0 + \alpha_1\beta_1 ... \alpha_{n-1}\beta_{n-1} + \alpha_n\beta_n$$

where c is MCC or HDIFF and α_0 through α_n are the coefficients of the indentation metrics that enumerated as β_0 through β_n where n is the number of indentation metrics. In the model shown in table 3, we do not use LOC as we want to see if the linear relationship still holds without LOC. Without SUM and withoutLOC most of the R^2 values are very low. For MCC, LOC does not improve the R^2 much, it increases from 0.385 to 0.388; this implies that our indentation metrics provide most of the information that LOC provides.

Halstead difficulty had worse results than McCabe's Cyclomatic complexity (R^2 of $0.20-0.22$) and Halstead Effort had an R^2 below 0.041. Halstead Volume and Length had the highest R^2 values of 0.6 and 0.5. This suggests that the important Halstead metrics such as difficulty were harder to model linearly than MCC.

We can see there is some linear relationship between statistical moments of indentation and complexity, although there is a lot of variation unaccounted for in these models. LOC on its own does not fare well against most of measures Halstead volume (R^2 of 0.59) , Halstead length (R^2 of 0.51), number of returns (R^2 of 0.38) and MCC (R^2 of 0.29). We can see that by including indentation metrics in our model we do gain information and accuracy within our linear models. We have shown that there is value in measuring indentation as well as LOC as we can model complexity more accurately with indentation and LOC combined.

5. DISCUSSION

We can see from the results there is some correlation between our indentation measures and traditional complexity measures (McCabe's and Halstead's). There is some linear correlation and there is some rank based correlation but it is not overly strong. This suggests that our statistical moments of indentation can be used as proxies for complexity; the larger our measurements the more complex the code, particularly the larger the standard deviation and the larger the summation of indentation the more complex that block of code is.

Standard deviation of indentation seems to be a good proxy for complexity because one could argue that the more change in indentation depth, the more complex the statements are in that code block. A large standard deviation in indentation could indicate many if blocks or many expressions within a diff-chunk, which would correlate well with Halstead's complexity metrics.

We noticed there was little difference between logical and non-logical indentation in the correlations. This suggests two things: that the relationship between logical and non-logical indentation is for the most part linear (for examples 4 spaces are often 1 logical unit) and that indentation is regular enough that logical indentation does not matter. What it also suggests is the out-lier, non-base 4 indentations, do not affect the results much otherwise there would be significant differences between raw indentation and logical indentation.

Indentation can provide information that a tokenizer could not, indentation can show the scope of expressions whereas a tokenizer is a flattened representation. To get the information that indentation supplies one would have to parse the source code into a tree. Indentation can often do well proxying complexity metrics, it is potentially its own complexity metric. Halstead's metrics do not count scope where as MCC often does, but indentation will capture more scoping semantics than MCC because not every new scope is a new branch in the code.

6. VALIDITY THREATS

Our work potentially suffers from a few threats to validity. The five main categories of the threats are: metric application issues, sampling issues, data cleaning issues, language issues, and development tool issues.

Our measurement of MCC and Halstead Complexity was done on revisions, not on methods, functions, modules, or files. Often these measurements are taken at a semantic level of structural granularity (functions, modules) but we only applied them to diffs-chunks. Although both MCC and Halstead Complexity correlate with LOC, which suggests that the metrics are cumulative thus allowing for measurements of deltas.

We sampled from popular Source-Forge projects, we did not sample across all Source-Forge Projects, we chose only those which had accessible CVS repositories to study. This sample of software might not be representative of many classes of software, for instance large projects might not exist on Source-Forge as often as they have their own websites.

When we ran our metric-suite on the projects we did not analyze any results which have NaN or Infinite as a row entry. This could alter our results as it excludes certain classes of changes which result in these values, usually they are very small changes.

Our choice of languages were the 6 most popular languages in the repository other than XML, Makefiles and shell scripts. These languages are related to each other through a common heritage of C and thus are syntactically similar. Would not one assume that the results per each of these languages would be very similar? Thus can we claim our results are generalizable?

Does the auto-indentation provided to programmers, by linters, pretty printers and IDEs, affect these results? We suspect they would not, since we showed indentation is relatively uniform.

7. CONCLUSIONS

In this paper we tested and confirmed our assertion that indentation was a meaningful proxy for code complexity. It has been suggested by others [9] that LOC correlated with complexity metrics well enough such that complexity metrics weren't needed. We have shown through correlations and linear models that cheap metrics such as indentation, when combined with LOC or alone, can be used to better model and simulate complexity measures than just LOC alone. We showed that for revisions to source code, there were medium and strong linear and rank based correlations of complexity metrics to summation of indentation and standard deviation of indentation. In many cases summation of indentation and standard deviation of indentation did better than LOC, especially with rank based correlation. We found little difference between raw and logical indentation metrics with respect to complexity.

We have provided an overview of indentation with respect to a large body of successful, popular Open Source software, as ranked by Source Forge. We have shown that for our sampling of multiple projects written in multiple languages, that indentation is actually quite regular across languages. We expected common logical units of indentation of 2 spaces to be frequent, but across all of the languages, 4 spaces of indentation or 1 tab of indentation were the most common logical unit. We compared the distributions of indentation per language to each other and found that the indentation of one language was often similar to another. For instance Python and Java had similar indentation styles, while Perl, C and PHP were similar to each other and C and C++ were very similar to each other.

We have shown that indentation is regular and consistent enough to be worth measuring. We demonstrated the value of measuring indentation along side LOC, it can be used as a proxy for complexity, and it is almost as cheap as LOC to calculate. Thus with the knowledge that indentation metrics are generally language agnostic, language unaware and cheap to calculate, we can use them as cheap proxies for complexity and maintainability, especially within the context of software evolution.

7.1 Future Work

Future work should include the investigation other languages as well, particularly those which do not have a shared history with C. Languages such as Smalltalk, LISP, Scheme, Dylan, Ruby are more foreign to C than Java, C++, Perl, PHP and Python.

Does the shape of the indentation in the diff-chunk matter? Are code changes that have bubble shaped indentation more complex than code with flat indentation?

We measured indentation of revisions in this paper, perhaps we should measure code characters per line or tokens per line. We could also compare complete versions of a system before and after a revision rather than just measuring the source code deltas.

Acknowledgements: This work was partially funded by an NSERC PGS D Scholarship. We would like to thank SHARCNET for their infrastructure.

8. REFERENCES

[1] R. E. Berry and B. A. Meekings. A style analysis of c programs. *Commun. ACM*, 28(1):80–88, 1985.

[2] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection, 2004.

[3] D. Coleman, D. Ash, B. Lowther, and P. W. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[4] D. Conway. *Perl best practices*. O'Reilly, Beijing [u.a.], 1. ed. edition, 2005.

[5] R. B. Findler. PLT DrScheme: Programming environment manual. Technical Report PLT-TR2007-3-v371, PLT Scheme Inc., 2007.

[6] D. M. German and A. Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Proceedings of 11th International Software Metrics Symposium (Metrics 2005)*, 2005.

[7] N. Gorla, A. C. Benander, and B. A. Benander. Debugging effort estimation using software metrics. *IEEE Trans. Softw. Eng.*, 16(2):223–231, 1990.

[8] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[9] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a theoretical model for software growth. In *MSR 2007: Proceedings*, page 21, Washington, DC, USA, 2007. IEEE Computer Society.

[10] R. F. Mathis. Flow trace of a structured program. *SIGPLAN Not.*, 10(4):33–37, 1975.

[11] T. J. Mccabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[12] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program indentation and comprehensibility. *Commun. ACM*, 26(11):861–867, 1983.

[13] P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *Commun. ACM*, 33(5):506–520, 1990.

[14] P. W. Oman and J. Hagemeister. Construction and testing of polynomials predicting software maintainability. *J. Syst. Softw.*, 24(3):251–266, 1994.

[15] R. Power, D. Scott, and N. Bouayad-Agha. Document structure. *Comput. Linguist.*, 29(2):211–260, 2003.