CSC 840

Matthew Wishoff

3/20/16

Project 3

Table of Contents

1.0 Abstract 3
2.0 Introduction 3
3.0 Method and software used 4
4.0 Graphs 4
5.0 Analysis 7
5.1 Problem#1 7
5.2 Problem#2 8
5.3 Problem#3 9
5.4 Problem#4 10
5.5 Problem#5 11
6.0 Conclusions 11
7.0 Appendix 12
a.) LOC counter 12
b.) matlab scripts 17
c.) Data 21

1.0 Abstract

Here we will be developing a tool to measure the size of software products and use it to analyze compiler performance. Now what is difficult is defining what the software or program size is. Is it just the number of lines in a program? The measurement of the programmer's effort? There are a lot of different approaches to this problem, and most of them are still considered correct. So it is very important to define early on what our way of measuring the size of software or a program is.

For the purposes of this paper we will be breaking the program up into two parts physical lines of code, and logical lines of code. Using these two forms I can breakdown a program into only executable lines of code, or the entirety of the program. Logical lines of code ignore simple things like comments, and lines that may only have one bracket on them. Also with a combination of these two elements we may be able to determine the effort put forth by the programmer to complete the program.

2.0 Introduction

With a large project such as this one it's important to develop a tool to stream line the process, so that it can be duplicated later on in experiments. Therefore, we will be developing a line of code counter for physical lines of code, and logical lines of code. We will also be looking at relationships between logical lines of code, physical lines of code, and the size of the program (M). Another interesting thing we will be looking at is the compilation time of programs, and how it relates to the logical lines of code. For this the program sizes would need to be sufficiently large to have as little error as possible.

Now to define what we mean when we say logical line of code. To calculate all logical lines we will say it is the main program, plus all the functions that it calls. Additionally we will add all semi colons, data definitions, if statements, while loops, switch statements, and for loops. Adding all these together will total all the logical lines of codes for the entire program. This will exclude things such as brackets on their own line, and other miscellaneous book keeping actions for the compiler that don't add logical value to the program.

Physical lines of code are very easy to calculate in respect to logical lines of code. To calculate physical lines of code any line with any content on it is counted as a physical line of code. This includes comments as it takes programmer effort to write these lines as well. So basically the way we count physical lines of code is just adding together all the new lines, until you reach the end of the program. This tallies together all the lines, except for the purposes of my program I have chosen to ignore lines that have zero content on them. In other words lines that are blank are not included in my calculation of physical lines of code.

3.0 Method and software used

To calculate the Logical lines of code, and the Physical lines of code I wrote a program that would count both of these in O(n) time. I wrote the program to specifically parse through C++ programs, and output the results to a file. For the results the program outputs the number of while, if, switch, for, semi-colons, and data definitions in the bench maker program. I wrote my program using the dev C++ IDE. Because it is very light weight without a lot of bells and whistles. Matlab used for the processing of data, and the output of graphs adjusted to visually look excellent. Also as a way of knowing the general ball park of how accurate my program is I used notepad++ to find the number of semi-colons in the bench maker program, for loops, while loops, switch statements, and data definitions. This allowed me to better track my errors, and to note how egregious they are or how minimal they are.

4.0 Graphs

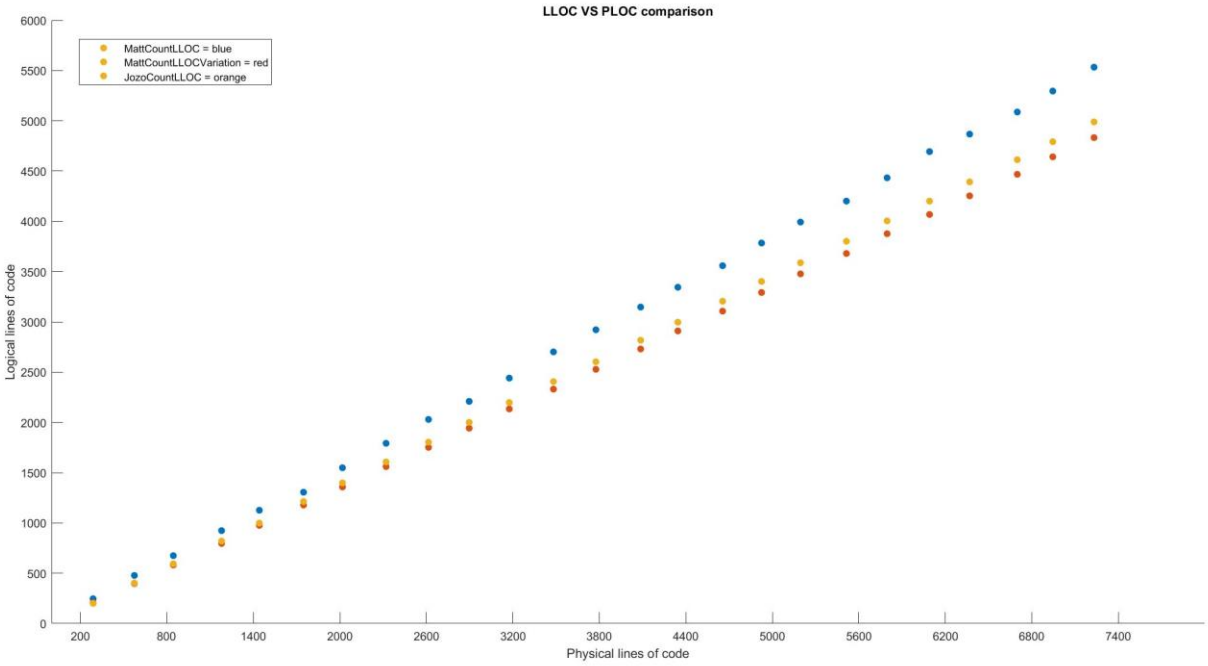

Figure 1

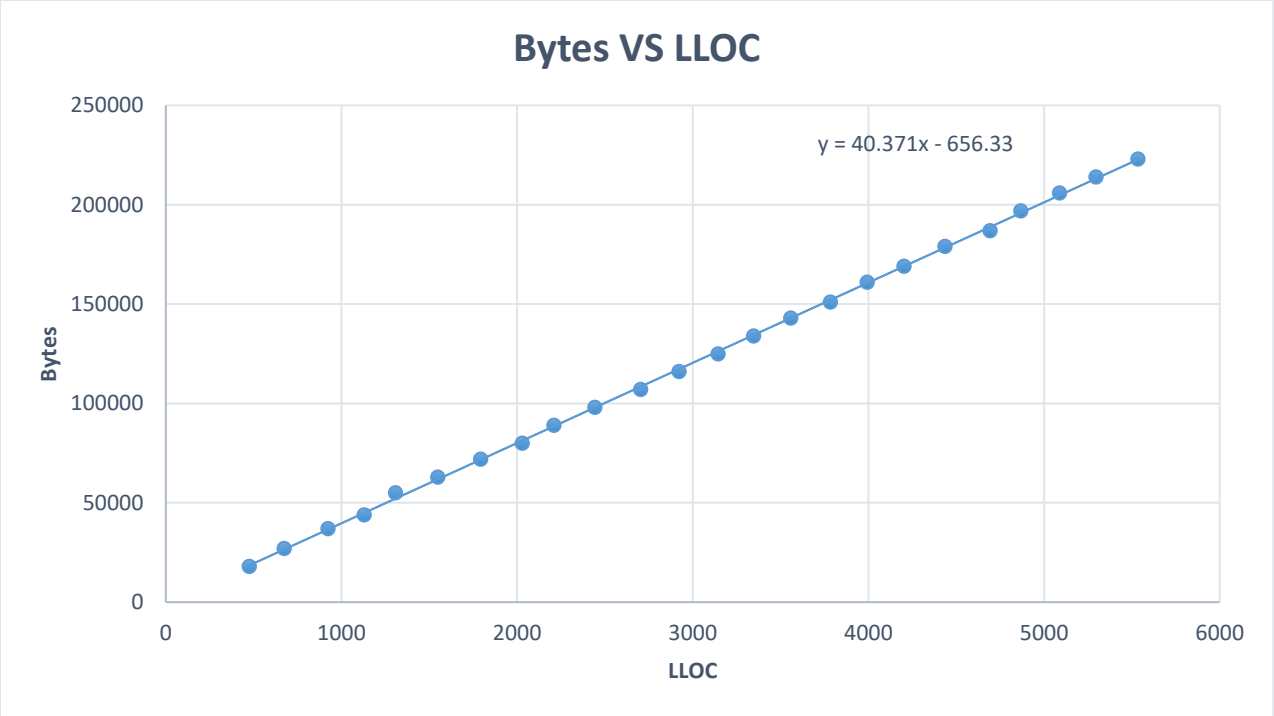

Figure 2

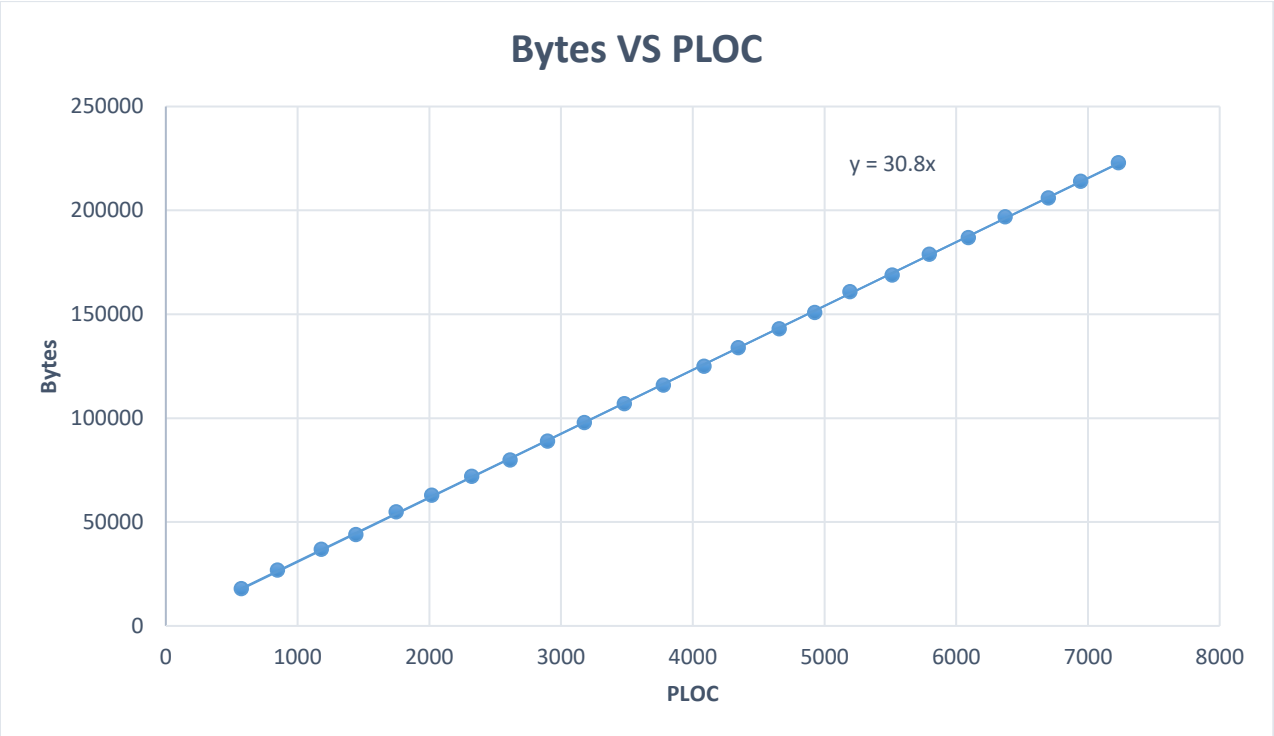

Figure 3

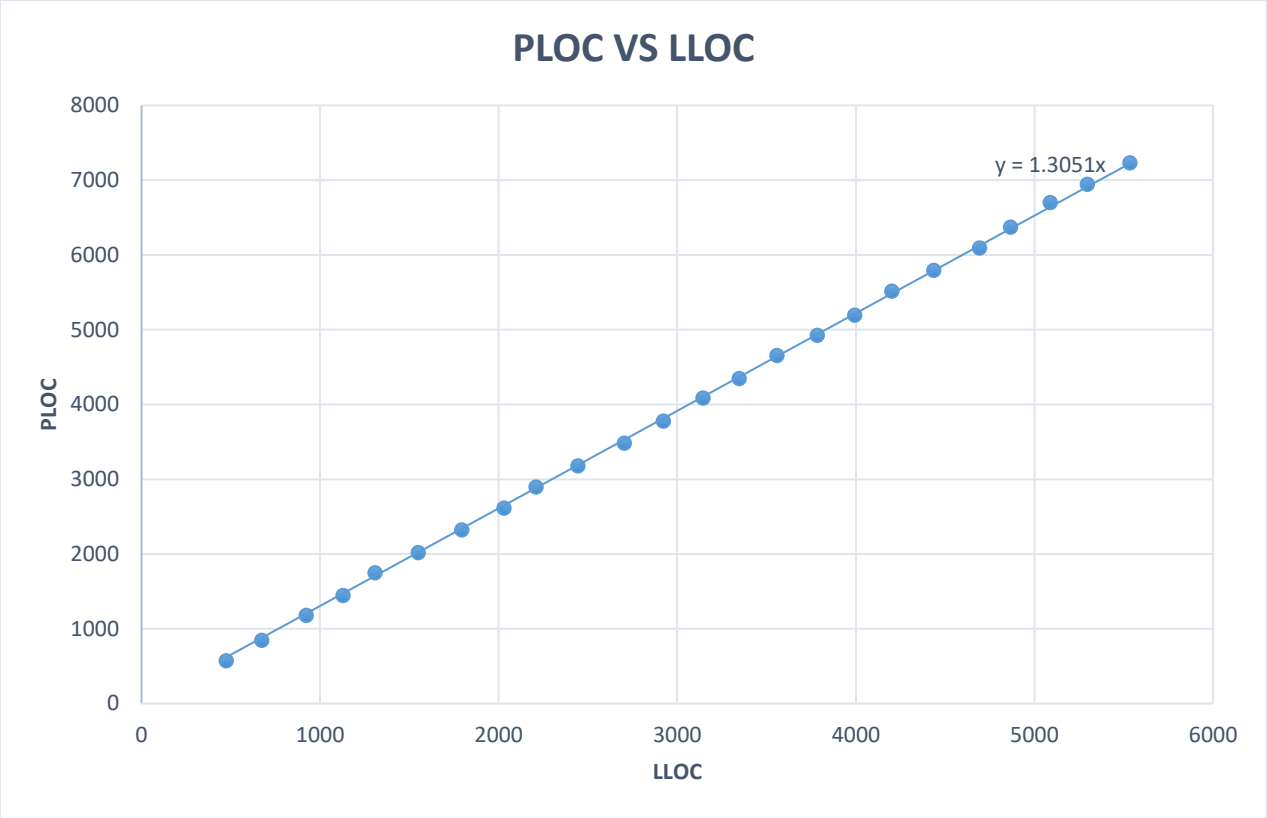

Figure 4

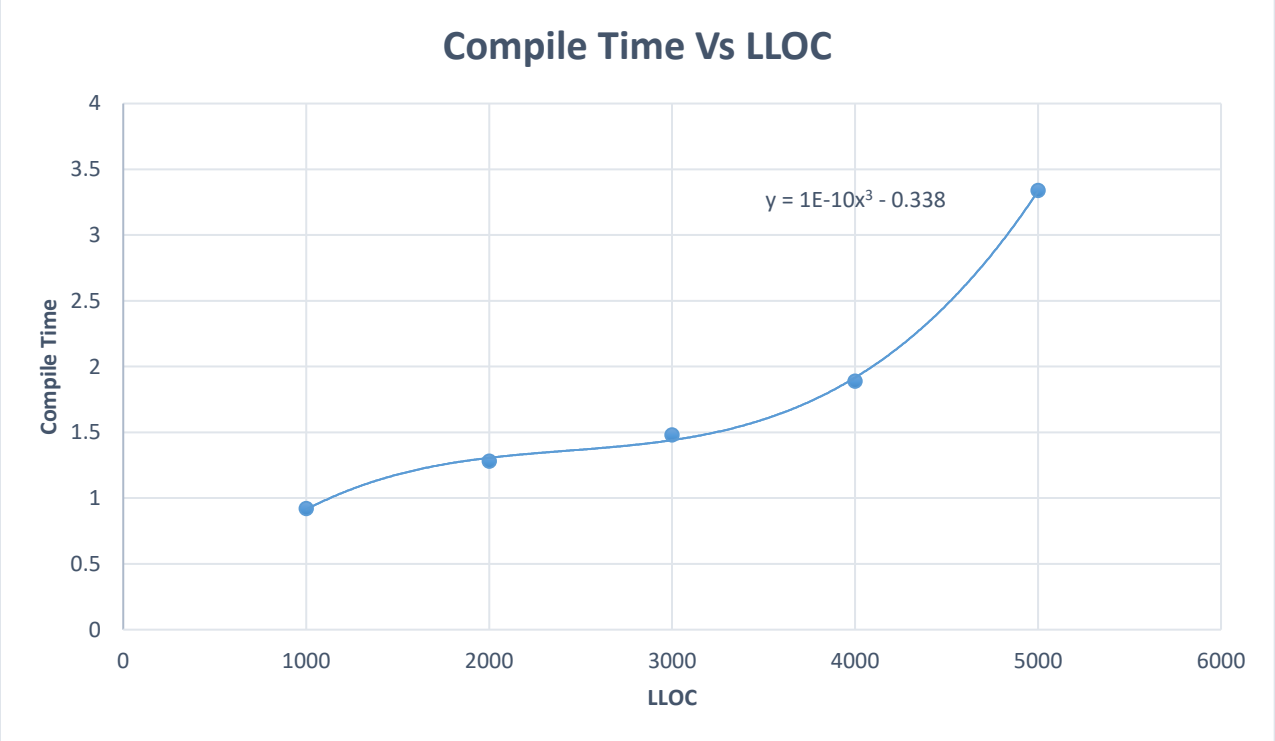

Figure 5

5.0 Analysis

5.1 Problem#1

Developing the LLOC/PLOC tool was a little difficult in my opinion. Because there are two approaches you can take in designing a LLOC/PLOC tool. You can decide to be super sensitive in what you declare as a logical line of code, or you can include too much. Hence, having a very broad definition of what a logical line of code really is. One of the examples I was getting confused about was counting the semi-colons in a for loop

EX: for(; condition ;) { }

I believe it is wrong to count the first semi-colon as it does not denote a logical line of code, but just guarantees that the syntax of the for loop is correct. To furthermore illustrate this point another example worth pointing out is when a semi-colon is on its own line with nothing else.

```
EX:     int I = 0;
        ;
        Int J = 0;
```

I don't believe it is correct to count this semi-colon either as it does not denote anything of value other. Yes the program will still compile, yes the compiler will still see this semi-colon, but if the compiler is smart nothing of value will come from this semi-colon. Furthermore multiple semi-colons on one line continuously one after the other should not be counted.

EX: ;;;;;;;;;;;;;;;;;;;;;;;;;

This line should not equal 25 logical lines of code. I believe even if we have an idiotic programmer we should still try to give an accurate LLOC count. Given these two examples I believe I make a very good case as why not to count the first semi-colon in the for loop declaration if there is nothing of value before it. However, if there is a variable declaration before the semi-colon in the for loop it should definitely be counted and added to the LLOC.

Just looking at this one small example of should you not count something versus whether you should count something can be very ambiguous. This made developing a LLOC/PLOC tool quite challenging, and even after much thought and analysis my tool is not 100 percent accurate. If anything it gives me great appreciation for the people who write compilers, as that must be some of the most tedious code written that is used by all programmers.

5.2 Problem#2

Now I will be testing the LLOC/PLOC program that I created against 25 bench maker programs. I have a few qualms with how the bench maker is giving out the amount of logical lines of code that it calculates. Under much deliberation with Hari, we believe that the bench maker is only counting semi-colons for its total logical lines of code. Given the formula in the assignment sheet:

LLOC = NPU + NSC + NEQ + Nif + Nswitch + Nwhile + Nfor

I feel that the bench maker is giving us a false number, more importantly it's giving too many logical lines of code if you calculate the logical lines using this formula. For example putting the program into notepad++ and searching for just semi-colons of a program returned the number 201. 201 was also the number of logical lines counted by the bench maker program. With this in mind you've maxed out the formula as everything else has to be 0 for LLOC to equal NSC. So therefore, I think the bench maker program is returning a slightly inaccurate count of how many logical lines are actually in the program.

In figure 1 the blue plot points are counting every semi-colon with control structures, and data definitions. The red plot points are not counting every semi-colon, it only counts semi-colons that terminate logical statements as defined earlier in this paper. Finally the orange is the bench maker's count of how many logical lines are in the program. In figure 1 we see that there are effectively 3 different ways we are counting LLOC, but what do they all have in common? We see as the physical lines of code increases, the logical lines of code also increase. With this I believe we can come to the assumption that PLOC will always be greater than LLOC for reasonably well written programs. Earlier in the report I mentioned how we should not count silly semi-colons, this is one way that this fits the logic that PLOC should be greater than LLOC. As a bunch of semi colons on one line is one physical line of code, and could either be 0 or the number of semi-colons for LLOC. It should be zero so that we can maintain this paradigm, as well as keeping a good LLOC count.

5.3 Problem#3

Figures 2, 3, and 4 show the relationships between the size of the programs in bytes, LLOC, and PLOC. You can see how in all three of the graphs you get linearly increasing regression lines. You can also see that as the program has more physical lines of code, you get a larger and larger byte size. Which makes sense the more you put into a file the larger it should be. In figure 2 you can see how the byte size for amount of LLOC increases similarly to the byte size for PLOC did. Which again makes sense, as you are adding lines of code the program will get larger. In figure 4 we compare LLOC to PLOC, and find once again a positively increasing regression line. I believe this is due to arithmetic expressions being terminated with a semi-colon. Because in this case when it's being terminated by a semi-colon it adds to both LLOC and PLOC. So for very heavy arithmetic expressions or general statements terminated with a semi-colon you are going to get a graph such as figure 4 that has a high correlation between LLOC and PLOC. Inversely if you had a program with tons of white space added in between lines the correlation between LLOC and PLOC would be much less. This is due to the fact that white space is not added to LLOC, but is added to PLOC. This allows the PLOC count to grow much faster than the LLOC count and would therefore be much greater for programs with lots of whitespace in them.

5.4 Problem#4

Using the graphs from problem #3 which are figures 2, 3, and 4 we can solve equations that may help us predict the number of bytes that a program will take up.

(1) M = 40.371 * LLOC – 656.33
(2) M = 30.8 * PLOC
(3) LLOC = 1.30 * PLOC

Next we will be using these equations to estimate the number of bytes and LLOC in a program, just by knowing the LLOC or PLOC of the program. To do this we will be using a non-trivial program that has a substantial size, the 25th program that was made from bench maker should suffice as that program has PLOC = 7232, LLOC = 55.

(1) M = (40.371 * 4833) – 656.33
    M = 194456.713

(2) M = 30.8 * 7232
    M = 222745.6

(3) LLOC = 1.3 * 7232
    LLOC = 9401.6

Now we will calculate the error of these formulas to see how well we did with
Error percentage = abs(((approximate value – exact value)) / exact value) * 100

(1) Error = abs((194456.713 -223000) / 223000) * 100 = 12.8%
(2) Error = abs((222745.6- 223000) / 223000) * 100 = 0.114%
(3) Error = abs((9401.6 – 4992) / 4992) * 100 = 88.3%

After calculating the errors we see that the first equation isn't that bad, it gives us a result very close to what we are wanting with an error of 12.8% which is acceptable. With the second formula we see that we have an error of 0.114% which is fantastic which means our formula was spot on, and couldn't have performed any better! The last formula was rather disappointing with an unacceptable error of 88.3%. I think this may be due to an error in calculating the formula as an LLOC of 9401.6 for approximated is very far from where it should be. I think in formula three the value that you multiply PLOC by definitely needs to be less than 1 to get a reasonable error for this formula. If I did this experiment again that would for sure be one of the things I check for to make sure the results are more accurate.

5.5 Problem#5

Now we take a look at compilation time of large programs in the scope of LLOC. We gather the formula:

T = T0 + T1 * LLOC^q, for q greater than or equal to 1.

I found that a q of 3 or greater gave a good polynomial fit so that all the points in the graph are hit by it. You can see this in figure 5 above. Specifically we get the equation $T = 1E\text{-}10x^3 - 0.338$. To get this equation I plugged 5 data points into Excel with varying LLOC so that I could get a good curve, and then adjusted the polynomial so that all the points are intersected by the curve. Some conclusions we can draw from this are that as programs become more complex with more and more lines of code, the compilation time will increase. So it makes sense to make smaller programs so you don't have to compile huge amounts of code you know works over and over again when making changes.

6.0 Conclusions

There are many different ways you can write a program that counts logical lines of code. Some are more complex than others, and may be more accurate due to added rules to get an accurate count on specific edge cases; such as, the semi colon situation that I talked about at length earlier in this paper. However, this all depends on what you wish to call a logical line of code, so results will vary from person to person on what they wish to include in their definition.

I found this tool to be very useful in determining how much you've actually done in a program. For example a program with a high PLOC count and low LLOC count will probably not have a lot of statements that do much of anything. While something that has a low PLOC count and a high LLOC count will be very dense, and could be quite confusing as a small number of lines are packed with all the logical statements.

Furthermore I thought it was really interesting that I was able to predict the number of bytes in a program very accurately just by knowing how many physical lines of code are in the file. I could see this being very useful 10 or 20 years ago when hard drive space was very valuable, and you had to be careful in how you used it. However, in today's world I think the benefit of such a formula has gone down, as it has become far easier to create disk space and lots of it. Hence, allowing programmers to not have to deal with such minimal things as how many bytes your program takes up, and can it fit on your 10mb hard drive.

Later I think it would be a fun side project, to try and correct some of the errors my program was having that lead to an 89% error when calculating the predicted logical lines of code. I feel that test is something that you could use as a bench mark that could help you more accurately see how well your program is performing.

7.0 Appendix

a.) LOC counter

```
// By: Matthew Wishoff
// Date: 3/21/16
// Class: CSC 840
// Description:

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <chrono>
#include <regex>

bool oneLiners(char s[]);

using namespace std;

//Coutning LLOC?
//NSC = Number of semi colons
//NEQ = Number of Equals signs
//LLOC += (NSC + NEQ); // NSC = 1

//1. We count logical lines of code and denote their number as LLOC.
//2. Each logical statement counts as one, regardless its complexity or level of nesting.
//3. All comments are ignored: in C++, both single line [//] and multiple line [/*...*/].
//4. The logical statement counting formula (based on token analysis) is the following:

        // LLOC = NPU // main program plus all functions
        // + NSC // in the whole program, except comments
        // + NEQ // data definitions only (constructor-initializer)
        // + Nif // all if statements
        // + Nswitch // all switch statements
        //+ Nwhile // all while statements
        // + Nfor // all for statements

int main(int argc, char**argv)
{
        string fileName;

        int PLOC = 0;
        int LLOC = 0;
        int ifCnt = 0;
        int whileCnt = 0;
        int switchCnt = 0;
```

```
        int forCnt = 0;
        int semiColonCnt = 0;
        int equalSignCnt = 0;
        int comment = 0;
        int function =0;
        string currentLine;



        // Read in C++ file
        cout << "Enter the file name: ";
        cin >> fileName;
        ifstream C_File;
        C_File.open(fileName);

        regex Function("int(.*)");
//      regex Function("^\\s*[\\w_][\\w\\d_]*\\s*.*\\s*[\\w_][\\w\\d_]*\\s*\(.*\\)$");
//      regex oneLiner("([^;]*(;))([^;]*(;))([^;]*(;))");
//      regex oneLiner("^([^;]?;){3}.*$");
        regex oneLiner(".*for(.*;.*;.+;+)");
//      regex oneLiner("for\([^;]*?;[^;]*?;[^)]*?) ;{3}");
//      regex oneLiner("for*\([^;]*?;[^;]*?;[^)]*?)\\ [^;];"); // Regex for a forloop
        regex If("if");  //Need to make the definition more robust!
        regex For("for"); //Need to make the definition more robust!
        regex While("while"); //Need to make the definition more robust!
        regex Switch("switch"); //Need to make the definition more robust!

        //Currently ignores +=, -=, *=, /= expressions.
        regex Equals("[^+*-/]="); //Need to make the definition more robust!
        //(";|;*|;+") <--- Accepts everything that is passed in???
        regex SemiColon(";"); //Need to make the definition more robust!

        //Do this last, has least impact on the program.
        regex Comments("//"); //Need to make the definition more robust!
        regex MultiCommentStart("(/*)*");
//      regex MultiCommentEnd("(*/)*"); //Breaks regex whyyyy?

        // Go line by line counting lines of code.
        while(!C_File.eof())
        {
//              cout << "**********Next line**********" << endl;

                //take a line from the program
                getline(C_File, currentLine);
                ++PLOC;
                cout << currentLine << endl;

//              cout << "**********current line**********" << endl;
```

```
//
	**********************************************************************************
*******************************
//		if we see a /* we see that a multi line comment has started.
		if(regex_match(currentLine, MultiCommentStart))
		{
//			//we continue to go through the file until we see a */ signifying the comment
has ended.
//			//or the end of file is reached
			while(regex_match(currentLine, MultiCommentEnd) && !C_File.eof())
			{
				getline(C_File, currentLine);
			}
		}
//	Probably need to make sure that if a multi line comment is at the end of a file this logic does not
break it.

//
	**********************************************************************************
*******************************

		//wont work if comments are on the same line?
		if(regex_match(currentLine, Comments))
		{
			//Do nothing
		}
		else //incriment appropriate counters.
		{
			if(regex_search(currentLine, If))
			{
				LLOC++;
				ifCnt++;
//				cout << "If" << endl;
			}
			else if(regex_search(currentLine, For))
			{
				LLOC++;
				forCnt++;
//				cout << currentLine << endl;
					if(regex_search(currentLine, oneLiner))
					{
						LLOC++;
//						cout <<"HAlLALULHA" << endl;
					}
//				cout << "For" << endl;
			}
			else if(regex_search(currentLine, While))
```

```
                        {
                                LLOC++;
                                whileCnt++;
//                              cout << "While" << endl;
                        }
                        else if(regex_search(currentLine, Switch))
                        {
                                LLOC++;
                                switchCnt++;
//                              cout << "Switch" << endl;
                        }
                        else if(regex_match(currentLine, Function))
                        {
                                LLOC++;
                                function++;
//                              cout << "Function" << endl;
                        }
                        else if(regex_search(currentLine, Equals))
                        {
                                LLOC++;
                                equalSignCnt++;
//                              cout << "Equals" << endl;
                        }

                        if(regex_search(currentLine, SemiColon))
                        {
                                LLOC++;
                                semiColonCnt++;
//                              cout << "Semi-Colon" << endl;
                        }

                        //If it is a line of something incriment PLOC
                }

        }

        C_File.close();

        freopen ("BMProgram25.txt","w",stdout);

        cout << "================================" << endl;
        cout << fileName << endl;
        cout << "PLOC Count: " << PLOC << endl;
        cout << "LLOC Count: " << LLOC << endl;
        cout << "If statement Count: " << ifCnt << endl;
        cout << "for loop Count: " << forCnt << endl;
        cout << "while loop Count: " << whileCnt << endl;
        cout << "switch statement Count: " << switchCnt << endl;
```

```
	cout << "function statement Count: " << function << endl;
	cout << "Semi Colon Count: " << semiColonCnt << endl;
	cout << "Equal sign Count " << equalSignCnt << " - - (Data def only)" << endl;
	cout << "================================" << endl;

	fclose (stdout);

	return 0;
}
```

b.) matlab scripts

```
% number of programs created.
step = [1, 2, 3, 4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14 ,15, 16, 17, 18, 19, 20, 21 ,22 ,23 ,24 ,25]

%Jozos Benchmark program
RequestedLLOC = [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000, 2200, 2400, 2600, 2800,
3000, 3200, 3400, 3600, 3800, 4000, 4200, 4400, 4600, 4800, 5000]

jozoCountLLOC = [201, 404, 592, 817, 999,  1213, 1399, 1605, 1805, 1999, 2199, 2405, 2606, 2815, 3000,
3208, 3400, 3588, 3801, 4003, 4202, 4393, 4614, 4794, 4992]

jozoCountPLOC = [284, 569, 842,1176, 1439, 1744, 2013, 2317, 2609, 2893, 3173, 3477, 3772, 4081,
4341, 4651, 4922, 5190, 5510, 5791, 6087, 6376, 6697, 6941, 7228]

%Data set that counts every semi-colon
mattCountLLOC = [248, 475, 673, 923, 1129, 1307, 1549, 1793, 2030, 2209, 2443, 2704, 2922, 3145,
3346, 3558, 3784, 3994, 4202, 4436, 4692, 4867, 5089, 5296, 5534]

mattCountPLOC = [288, 573, 846, 1180, 1443, 1748, 2017, 2321, 2613, 2897, 3177, 3481, 3776, 4085,
4345, 4655, 4926, 5194,5514, 5795, 6091, 6370, 6700, 6945, 7232]

%Data set that doesn't count every semi-colon.
%Leaves out useless ones in forloop syntax.
mattCountLLOCVariation = [204, 398, 580, 798, 974, 1180, 1360, 1559, 1753, 1941, 2133, 2333, 2527,
2730, 2908, 3110, 3295, 3476, 3683, 3877, 4070, 4254, 4469, 4641, 4833]

dontCountAllSemiColons = [126 ,261 ,402 ,560 ,685 ,828 ,984 ,1113 ,1241 ,1414 ,1535 ,1651 ,1795 ,1955
,2083 ,2253 ,2371 ,2498 ,2678 ,2803 ,2913 ,3079, 3263,3372 ,3494]

%Data structure count in programs
IfCount = [7, 14, 22, 31, 37, 46, 54, 61, 68, 76, 85, 92, 100, 107, 116, 122, 131, 138, 147, 156, 161, 170,
178, 186, 194]

forCount = [16, 25, 31, 38, 44, 52, 58, 65, 71, 77, 85, 91, 98, 104, 111, 117, 124, 131, 137, 145, 151, 158,
164, 172, 178]

whileCount = [7, 16, 26, 35, 42, 52, 61, 70, 78, 86, 97, 106, 114, 122, 132, 140, 148, 158, 166, 178, 184,
194, 202, 212, 220]

switchCount = [2, 4, 6, 9, 11, 13, 15, 18, 20, 22, 24, 26, 29, 31, 33, 35, 37, 39, 42, 44, 46, 49, 51, 53, 55]

functionStatementCount = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45,
47, 49, 51]
```

```
CountAllSemiColons = [173, 343, 502, 694, 851, 1031, 1188, 1364, 1537, 1703, 1868, 2047, 2217, 2399,
2552, 2734, 2895, 3053, 3236, 3403, 3578, 3736, 3929, 4076, 4246]

dataDef = [24, 43, 48, 69, 89, 111, 100, 133, 166, 147, 176, 226, 239, 249, 260, 260, 290, 307, 298, 324,
378, 357, 354, 376, 412]

%Number of bytes per each program
bytes = [10000, 18000, 27000, 37000, 44000, 55000, 63000, 72000, 80000, 89000, 98000, 107000,
116000, 125000, 134000, 143000, 151000, 161000, 169000, 179000, 187000, 197000, 206000, 214000,
223000]

figure
plot(step, mattCountLLOC, 'blue', step, mattCountLLOCVariation, 'green', step, jozoCountLLOC, 'black');
title('LLOC comparison');
xlabel('Number of programs 1 is small, 25 is large')
ylabel('Logical lines of code');
legend('MattCountLLOC', 'MattCountLLOCVariation', 'JozoCountLLOC');
ax = gca;
ax.XTick = 1:1:25;
ax.YTick = 0:500:6000;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure
plot(step, CountAllSemiColons, 'red', step, dontCountAllSemiColons, 'yellow');
title('Counting semi-colons comparison');
xlabel('Number of programs 1 is small, 25 is large')
ylabel('Logical lines of code');
legend( 'countAllSemiColons', 'dontCountAllSemiColons');
ax = gca;
ax.XTick = 1:1:25;
ax.YTick = 0:500:6000;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure
scatter(step, CountAllSemiColons, 'filled');
hold on;
scatter(step, dontCountAllSemiColons, 'filled');
title('Counting semi-colons comparison');
xlabel('Number of programs 1 is small, 25 is large');
ylabel('Logical lines of code');
legend('countAllSemiColons', 'dontCountAllSemiColons');
ax = gca;
ax.XTick = 1:1:25;
ax.YTick = 0:500:6000;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
figure
scatter(step, mattCountLLOC, 'filled');
hold on;
scatter(step, mattCountLLOCVariation, 'filled');
hold on;
scatter(step, jozoCountLLOC, 'filled');
hold on;
title('LLOC comparison');
xlabel('Number of programs 1 is small, 25 is large')
ylabel('Logical lines of code');
legend('MattCountLLOC = blue', 'MattCountLLOCVariation = red', 'JozoCountLLOC = orange');
ax = gca;
ax.XTick = 1:1:25;
ax.YTick = 0:500:6000;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure
plot(step, IfCount, step, forCount, step, switchCount, step, functionStatementCount, step, dataDef, step,
whileCount);
title('comparison of frequency of control structures')
xlabel('Number of programs 1 is small, 25 is large')
ylabel('Times the control strucutre appears');
legend('IfCount', 'forCount', 'switchCount', 'functionStatementCount', 'dataDefCount', 'whileCount');
ax = gca;
ax.XTick = 1:1:25;
ax.YTick = 0:100:500;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure
scatter(mattCountPLOC, mattCountLLOC, 'filled');
hold on;
scatter(mattCountPLOC, mattCountLLOCVariation, 'filled');
hold on;
scatter(mattCountPLOC, jozoCountLLOC, 'filled');
hold on;
title('LLOC VS PLOC comparison');
xlabel('Physical lines of code')
ylabel('Logical lines of code');
legend('MattCountLLOC = blue', 'MattCountLLOCVariation = red', 'JozoCountLLOC = orange');
ax = gca;
ax.XTick = 200:600:7400;
ax.YTick = 0:500:6000;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%Good but needs a linear regression line.

calc = bytes\mattCountLLOC;
size(calc)
size(bytes)
regLine = calc*calc2;
%calc = mattCountLLOC\bytes;
%regLine = calc*mattCountLLOC;


figure
scatter(mattCountLLOC, bytes, 'filled');
hold on;
plot(mattCountLLOC, regLine);
title('Bytes VS MyLLOC');
xlabel('Logical lines of code')
ylabel('Bytes');
ax = gca;
%ax.XTick = 200:600:7400;
ax.YTick = 0:10000:250000;
ax.YAxis.Exponent = 0;
```

c.) Data

```
 ================================
BM1BMPrograms1.cpp
PLOC Count: 288
LLOC Count: 204
If statement Count: 7
for loop Count: 16
while loop Count: 7
switch statement Count: 2
function statement Count: 7
Semi Colon Count: 126
Equal sign Count 23 - - (Data def only)
================================
================================
BM1BMPrograms2.cpp
PLOC Count: 573
LLOC Count: 398
If statement Count: 14
for loop Count: 25
while loop Count: 16
switch statement Count: 4
function statement Count: 11
Semi Colon Count: 261
Equal sign Count 42 - - (Data def only)
================================
================================
BM1BMPrograms3.cpp
PLOC Count: 846
LLOC Count: 580
If statement Count: 22
for loop Count: 31
while loop Count: 26
switch statement Count: 6
function statement Count: 15
Semi Colon Count: 402
Equal sign Count 47 - - (Data def only)
================================
================================
BM1BMPrograms4.cpp
PLOC Count: 1180
LLOC Count: 798
If statement Count: 31
for loop Count: 38
while loop Count: 35
switch statement Count: 9
function statement Count: 19
Semi Colon Count: 560
Equal sign Count 68 - - (Data def only)
================================
```

==========
BM1BMPrograms5.cpp
PLOC Count: 1443
LLOC Count: 974
If statement Count: 37
for loop Count: 44
while loop Count: 42
switch statement Count: 11
function statement Count: 23
Semi Colon Count: 685
Equal sign Count 88 - - (Data def only)
==========
==========
BM1BMPrograms6.cpp
PLOC Count: 1748
LLOC Count: 1180
If statement Count: 46
for loop Count: 52
while loop Count: 52
switch statement Count: 13
function statement Count: 27
Semi Colon Count: 828
Equal sign Count 110 - - (Data def only)
==========
==========
BM1BMPrograms7.cpp
PLOC Count: 2017
LLOC Count: 1360
If statement Count: 54
for loop Count: 58
while loop Count: 61
switch statement Count: 15
function statement Count: 31
Semi Colon Count: 984
Equal sign Count 99 - - (Data def only)
==========
==========
BM1BMPrograms8.cpp
PLOC Count: 2321
LLOC Count: 1559
If statement Count: 61
for loop Count: 65
while loop Count: 70
switch statement Count: 18
function statement Count: 35
Semi Colon Count: 1113
Equal sign Count 132 - - (Data def only)
==========
==========
BM1BMPrograms9.cpp
PLOC Count: 2613
LLOC Count: 1753
If statement Count: 68

for loop Count: 71
while loop Count: 78
switch statement Count: 20
function statement Count: 39
Semi Colon Count: 1241
Equal sign Count 165 - - (Data def only)
==========
==========
BM1BMPrograms10.cpp
PLOC Count: 2897
LLOC Count: 1941
If statement Count: 76
for loop Count: 77
while loop Count: 86
switch statement Count: 22
function statement Count: 43
Semi Colon Count: 1414
Equal sign Count 146 - - (Data def only)
==========
==========
BM1BMPrograms11.cpp
PLOC Count: 3177
LLOC Count: 2133
If statement Count: 85
for loop Count: 85
while loop Count: 97
switch statement Count: 24
function statement Count: 47
Semi Colon Count: 1535
Equal sign Count 175 - - (Data def only)
==========
==========
BM1BMPrograms12.cpp
PLOC Count: 3481
LLOC Count: 2333
If statement Count: 92
for loop Count: 91
while loop Count: 106
switch statement Count: 26
function statement Count: 51
Semi Colon Count: 1651
Equal sign Count 225 - - (Data def only)
==========
==========
BM1BMPrograms13.cpp
PLOC Count: 3776
LLOC Count: 2527
If statement Count: 100
for loop Count: 98
while loop Count: 114
switch statement Count: 29
function statement Count: 55
Semi Colon Count: 1795

Equal sign Count 238 - - (Data def only)
==========
==========
BM1BMPrograms14.cpp
PLOC Count: 4085
LLOC Count: 2730
If statement Count: 107
for loop Count: 104
while loop Count: 122
switch statement Count: 31
function statement Count: 59
Semi Colon Count: 1955
Equal sign Count 248 - - (Data def only)
==========
==========
BM1BMPrograms15.cpp
PLOC Count: 4345
LLOC Count: 2908
If statement Count: 116
for loop Count: 111
while loop Count: 132
switch statement Count: 33
function statement Count: 63
Semi Colon Count: 2083
Equal sign Count 259 - - (Data def only)
==========
==========
BM1BMPrograms16.cpp
PLOC Count: 4655
LLOC Count: 3110
If statement Count: 122
for loop Count: 117
while loop Count: 140
switch statement Count: 35
function statement Count: 67
Semi Colon Count: 2253
Equal sign Count 259 - - (Data def only)
==========
==========
BM1BMPrograms17.cpp
PLOC Count: 4926
LLOC Count: 3295
If statement Count: 131
for loop Count: 124
while loop Count: 148
switch statement Count: 37
function statement Count: 71
Semi Colon Count: 2371
Equal sign Count 289 - - (Data def only)
==========
==========
BM1BMPrograms18.cpp
PLOC Count: 5194

LLOC Count: 3476
If statement Count: 138
for loop Count: 131
while loop Count: 158
switch statement Count: 39
function statement Count: 75
Semi Colon Count: 2498
Equal sign Count 306 - - (Data def only)
==========
==========
BM1BMPrograms19.cpp
PLOC Count: 5514
LLOC Count: 3683
If statement Count: 147
for loop Count: 137
while loop Count: 166
switch statement Count: 42
function statement Count: 79
Semi Colon Count: 2678
Equal sign Count 297 - - (Data def only)
==========
==========
BM1BMPrograms20.cpp
PLOC Count: 5795
LLOC Count: 3877
If statement Count: 156
for loop Count: 145
while loop Count: 178
switch statement Count: 44
function statement Count: 83
Semi Colon Count: 2803
Equal sign Count 323 - - (Data def only)
==========
==========
BM1BMPrograms21.cpp
PLOC Count: 6091
LLOC Count: 4070
If statement Count: 161
for loop Count: 151
while loop Count: 184
switch statement Count: 46
function statement Count: 87
Semi Colon Count: 2913
Equal sign Count 377 - - (Data def only)
==========
==========
BM1BMPrograms22.cpp
PLOC Count: 6370
LLOC Count: 4254
If statement Count: 170
for loop Count: 158
while loop Count: 194
switch statement Count: 49

function statement Count: 90
Semi Colon Count: 3079
Equal sign Count 356 - - (Data def only)
==========
==========
BM1BMPrograms23.cpp
PLOC Count: 6700
LLOC Count: 4469
If statement Count: 178
for loop Count: 164
while loop Count: 202
switch statement Count: 51
function statement Count: 94
Semi Colon Count: 3263
Equal sign Count 353 - - (Data def only)
==========
==========
BM1BMPrograms24.cpp
PLOC Count: 6945
LLOC Count: 4641
If statement Count: 186
for loop Count: 172
while loop Count: 212
switch statement Count: 53
function statement Count: 99
Semi Colon Count: 3372
Equal sign Count 375 - - (Data def only)
==========
==========
BM1BMPrograms25.cpp
PLOC Count: 7232
LLOC Count: 4833
If statement Count: 194
for loop Count: 178
while loop Count: 220
switch statement Count: 55
function statement Count: 103
Semi Colon Count: 3494
Equal sign Count 411 - - (Data def only)
==========