**A dynamic metric to measure the time required to execute mentally a program**

**Abstract**

An important part of the maintenance cost of a software is due to the time needed to debug it. It is probable that this time is strongly correlated to the time needed to understand the source code. As a complex code is *a priori* more difficult to understand than a simpler one, we may think that the existing complexity metrics would help us to predict the time needed to fix the bugs of a given program.

However, several scientific studies recently demonstrated that all these metrics are (at best) weakly correlated with the time needed for an approaching task: find the returned value of a given function. These results may lead us to doubt the validity of these metrics. At a minimum, it is probable that they will not help us to predict the time to debug a program.

In this article, we will first give some reasons which could explain why this correlation is so weak. Second, we will define a new metric which is much better correlated with the results of the experiments. To this end, we will introduce the concept of dynamic metric.

Finally, we will demonstrate that a dynamic metric may predict with high precision the average time that developers will need to find the values returned by given functions (for the same experiment). For a different experiment, we also demonstrated that the same metric is able to do similar predictions, although with lower precision.

These promising results encourage us to continue to work in this direction, hoping to be able in the near future to predict the average time needed to debug a real program.

**1. Introduction**

For now, most of the complexity metrics are statically analyzing the code. They simply “read it”, calculate a complexity score for each line, and then return the sum of these scores. We will see why this approach may be relevant to calculate the global complexity of code snippets, but not to estimate the time T needed to mentally execute a given program.

Then, we will define the concept of dynamic metric, and use the results published by Janet Siegmund et al. in 2012 to demonstrate how a dynamic metric may be strongly correlated with T. In mathematical terms, we will define a metric *dyn* which gives a score linearly dependent on T with a Pearson coefficient near to 1. For example, if dyn(f) = 2 \* dyn(g), we will predict that the average time to find the returned value of *f* will be approximately equal to twice the average time to find the returned value of *g* (i.e. T(f) = 2 \* T(g)).

**2. Prior and related work**

During the last decades, dozens of metrics were defined in the aim to measure the complexity of the code. Some of them are very simple: count the number of lines of code (LOC), count the number of identifiers, etc. Others are more sophisticated, like Halstead metrics or McCabe metric (also called “cyclomatic complexity”).

In 2017, the software [SonarQube] defined an extension of the McCabe metric in the aim to measure the cognitive complexity of the code, *i.e.* the difficulty for a human being to understand it.

Later, in 2020, the software [Genese Cpx] defined a new metric which fills multiple gaps of the SonarQube’s metric, and thus probably returns more relevant results.

Which of these metrics are really correlated with the difficulty to understand the code ? Many scientific works tried to reply to this question.

In 2017, Scalabrino et al. [1] reached a surprising result: none of the existing metrics seemed to be really correlated with the code comprehension.

 In 2020, Wyrich et al. [2] demonstrated that the SonarQube metric is the only one which probably has a significant correlation with cognitive complexity.

In 2021, Peitek et al. [3] extended Siegmund’s work [4] by using Functional Magnetic Resonance Imaging (FMRI) in the aim to measure the intensity of the activation of the Broca’s areas of developers which were asked to find the returned value of given functions. Like Scalabrino et al., they found no or weak correlation between 41 metrics and the measure of the activation of the Broca’s areas. Even the SonarQube’s metric was not (or weakly) correlated with their measures.

Is it so surprising ? Maybe not. For example, the aim of SonarQube and Genese Cpx metrics is to give a score which represents the difficulty to understand the code, with a definition of the word “*understand*” which could be “*understand the specs of the function, and verify if its implementation corresponds to its specs*”.  This is very different from “*find the returned value of a given function*”.

Consequently, there is *a priori* no reason to find a correlation between these metrics and the time needed to mentally execute a program.

**3. The actual metrics**

During the last decades, dozens of metrics were defined in the aim to measure the complexity of the code. Some of them are very simple: count the number of lines of code (LOC), count the number of identifiers, etc. Others are more sophisticated, like Halstead metrics or McCabe metric (also called “cyclomatic complexity”).

Later, SonarQube and Genese Cpx extended the McCabe metric in the aim to explicitly measure the cognitive complexity of the code, *i.e.* the difficulty for a human being to understand it. Hereafter, we will call this kind of metrics the cognition metrics.

As we can see, there are many different kinds of “complexity”. That’s why the expression “*the complexity of the code*”, without any precisions, is a non-sense. We should always specify which kind of complexity we are talking about, and eventually specify if this complexity should be in relation with something happening in the real world.

For example, the metric “number of lines of code” (LOC) simply counts the number of lines of code snippets, and is not intended to measure something else. There is nothing in the real world that this metric is supposed to measure.

Inversely, SonarQube and Genese Cpx were developed in the aim to estimate something existing in the real world: the difficulty for a human being to understand a code snippet. That’s why, unlike LOC, we could experimentally demonstrate if these metrics are correctly correlated with what they claim to measure.

Unfortunately, as far as I know, no experiments  were ever organised in the aim to measure the difficulty of comprehension of the code, defined as “*the time needed to understand the specs of the code snippet, and to verify if the implementation of the code corresponds to its specs*”. Until we have achieved experiments specifically designed to measure these two elements, it will be impossible to affirm that these metrics are correctly measuring what they claim to measure.

**4. Static metrics and mental processes**

**4.1 Loops**

Most of the actual metrics analyze statically the code snippets. This simply means that they define some rules to calculate the complexity of each line of code, and then return the sum of these values.

Example 4.1.a

Une image contenant texte

Description générée automatiquement

The complexity of the function *f* is :

Cpx(f) = Cpx(line1) + Cpx(line2) + ... + Cpx(line7)

This score represents the complexity of the function in its globality. For cognition metrics like SonarQube or Genese Cpx, this score should be correlated with the time needed for a human being to understand what *f* is doing. This score depends only on the implementation of *f*.

Remark:

Please note that the lack of comments and the absence of semantics do not let us know what f *should* do.

First, assume that we ask some developers to find the value returned by *f* for the input arr = [2, 3]. Now, assume that instead of [2, 3], we would say that arr = [11, 23, -5, 42, 17, 128, 97, -79]. What will happen ?

It is highly probable that the average time to find the returned value will be much higher than with arr = [2, 3] (and with more errors). Thus, for the same developers and the same function, the results of the experiment will be completely different by simply changing the value of the input.

Consequently, why should static metrics be correlated with the results of this kind of experiment ? Their goal is to provide one and only one complexity score for a given function, not multiple scores depending on the values of the inputs.

It is an important reason which explains why we should not expect to find a correlation, and thus to be able to verify the validity of static metrics with this kind of experiment.

**4.2 Conditionals**

Let’s look at the function *g* :

Example 4.2.aUne image contenant texte

Description générée automatiquement

For a cognition metric like SonarQube or Genese Cpx, the complexity of *g* should be very high, because of the complexity of the *else* case.

However, assume that we do the same experience as above, with a = 1. The average time to find the returned value will be very short, because the developers don’t need to read the code which is in the *else* case. Inversely, if a = -1, it is probable that the developers will need a very long time to find the result.

That's why, like in the example 4.1.a with loops, there is no reason to expect that a static metric will be correlated with the results of this kind of experiment. We should not expect any kind of correlation between static metrics and results of experiments asking developers to mentally execute a program.

**4.3 The illusion of correlation**

In the chapter above, we saw that we should not expect to find a correlation between the scores provided by static metrics and the results of experiments depending on the mental execution of code snippets. However, some previous studies seem to demonstrate the opposite. It seems that, for particular experiments, some static metrics are really correlated (weakly or strongly) with the results of these experiments. Why ?

In reality, multiple biases exist which are distorting the conclusions. For example, let’s use the LOC metric, and let’s use the set of code snippets below:

Une image contenant texte

Description générée automatiquement

 The complexity scores of *u*, *v*, and *w* are :

LOC(u) = 4

LOC(v) = 5

LOC(w) = 6

Let’s note *T(f(a))* the average time needed by developers to find the value of a given function f for a given input *a*. It is highly probable that we will observe that T(u(a)) < T(v(a)) < T(w(a)).

Consequently, we could deduce from this result that there is a positive correlation between the LOC metric and the time needed to find the returned value of a given function.

However, this positive correlation is probably an illusion: it happens because we chose the same value *a* for the functions *u*, *v* and *w*. To demonstrate this affirmation, let’s imagine that we used a *=* 1296,985 for *u*, 257 for *v*, and 1 forw; with this choice, we would probably have obtained the inverse result : T(u(a)) > T(v(a)) > T(w(a)). Thus, our conclusion would be inverted too.

If the values of the inputs were always chosen randomly, we would not be misled : we would clearly see that no correlation exists. Unfortunately, we can’t randomly choose the inputs, because of the nature of the experiments : we can’t give inputs which induce calculations which are too complex to be done manually by the developers. Each code snippet must not be too long or too hard to understand. For this reason, each line must have a low number of identifiers to remember, the operations between the different identifiers must be simple enough, and when the function contains a loop, we must choose an input which implies to execute the loop only a few times. Similarly, these experiments will never contain a long and complex *else* block if the developers will never read it (if the execution process never enters in this block).

Consequently, it is highly probable that the main part of the correlations found in previous studies is due to the multiple biases related to the choice of the functions and the inputs.

**5. Dynamic metrics**

**5.1 Definition**

In the previous chapter, we explained why we should not expect to find a correlation between static metrics and the results of experiments consisting of asking developers to mentally execute a program. However, we can expect it with dynamic metrics.

In this article, we call ***dynamic metric*** a metric which only depends on the mental process that a developer needs to mentally execute a program.

Example 5.1.a

Une image contenant texte, écran

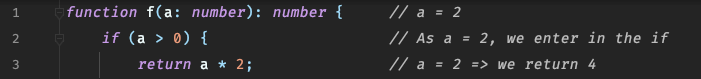
Description générée automatiquement

We saw sooner that for a static metric, we simply have

Cpx(f) = Cpx(line 1) + Cpx(line 2) + ... + Cpx(line 9)

Now, assume that we perform an experiment where we ask developers to find the value returned by *f* for a = 2. What will be their mental process to find the solution?

Let’s try to represent it by writing the only lines which are mentally executed :



For a dynamic metric, we will have

Cpx(f) = Cpx(line 1) + Cpx(line 2) +Cpx(line 3)

Caution: the complexity of each line depends on the value of a ! For example, with a good dynamic metric, we should have Cpx(f) lower for a = 2 than for a = 8997, because if a = 8997 the multiplication in line 3 takes much more time to be mentally calculated.

**5.2 Loops and dynamic metrics**

Now, let’s look at the usage of dynamic metrics with loops. If the developer needs to read 2 times the content of the loop, a good dynamic metric should usually find a lower value than if the developer needs to read it 15 times.

Example 5.2.a

Une image contenant texte

Description générée automatiquement

Assume that arr = [2, 3]. Let’s try to watch in details the mental process of the developer which tries to find the value returned by *f* :

Une image contenant texte

Description générée automatiquement

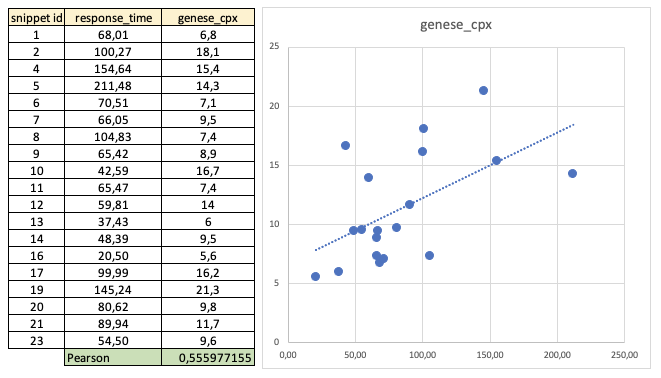
As we can see, the developer needs to read two times the loop, and thus to “manually execute” two times its content. A good dynamic metric should take it into account.

**6. Dynamic metric construction**

**6.1 The Siegmund’s experiment**

In this chapter, we will try to construct a dynamic metric which would be strongly correlated with the results of a real experiment. To this end, we will use the datasets published in open-source by Siegmund et al. with their article published in 2012 [4]. In their experiment, they asked 41 second-year students from a software-engineering course at the University of Passau to find the values returned by 23 different functions, and they noted the time they needed to do this task (it was not the aim goal of their experience, but it is what we need for now).

First, let’s see what is the correlation between these results and the Genese Cpx metric :



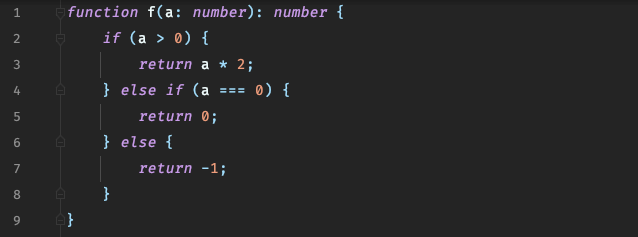
As we can see, there is a clear correlation between the time needed to give the correct response, and the complexity score given by Genese Cpx. The Pearson coefficient is approximately equal to 0,556, which is not so bad (the linear correlation is strong when this coefficient is near to 1, and weak when it is near to 0). However, as we said in chapter 4.3, this correlation is probably due to multiple biases.

Now, let’s try to define a dynamic metric which would be much better correlated to the results of this experiment. With this new metric, we would be able to predict approximately the time needed to find the returned value of any similar function (in the same experiment).

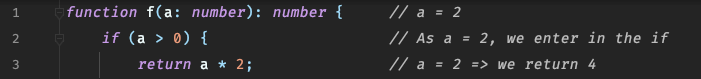
**6.2 If / else**

First, we will define a dynamic metric *dyn* estimating the complexity of each line by counting the number of identifiers used in an operation located in this line. In other words, we will assign a  weight w = 1 to each of these identifiers.

Let’s remember the example 5.1.a :



Assume that the developers are asked to find the value returned by *f* for a = 2. The lines which are mentally executed are :



Now, let’s calculate the complexity of *f* for our dynamic metric *dyn* :

Une image contenant texte

Description générée automatiquement

We find the complexity dyn(f) = 2.

**6.3 Loops**

Now, assume that we perform an experiment where we ask developers to find the value returned by the function of the example 5.2.a for arr = [1, 2, 4, 5, 7, 8].

Une image contenant texte

Description générée automatiquementWhat will be the mental process of the developers to find the solution?

Une image contenant texte

Description générée automatiquement

Let’s calculate dyn(f) (please note that we count 0 for assignments) :

Une image contenant texte

Description générée automatiquement

We find a complexity equal to 18, which seems to be too high for a function which is simply adding the numbers of a given array. Where is the problem ?

Intuitively, when we enter the loop for the first time, we read it normally. The second time, we remember what we must do (an addition). And the third time (maybe sooner), we understand that this function simply adds the numbers of a given array. Thus, we stop to read the loop, and we simply mentally add all the other elements of the array, without reading the last lines.

More generally, we may suppose that each time that we read the same line, we need less effort to understand it. Step after step, we need less time to mentally execute the same line.

To take this fact into account, we will add a new rule: each time that we will read the same line, we will multiply its complexity score by a coefficient *c* between 0 and 1.

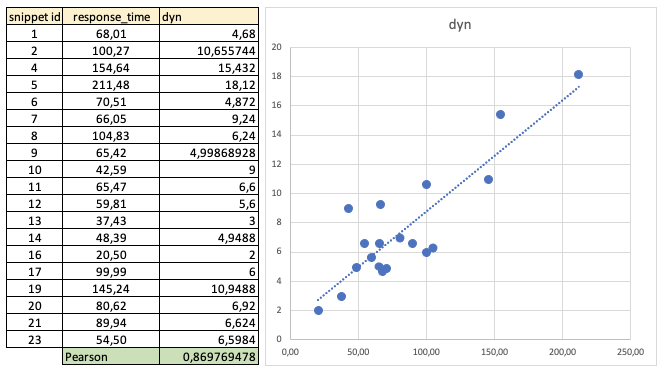
Let’s try with c = 0.5 :

Une image contenant texte

Description générée automatiquement

With c = 0.5, we find a complexity equal to 5.90625, which seems to be more reasonable.

Now, let’s apply our new metric to the results of the Siegmund’s experiment :



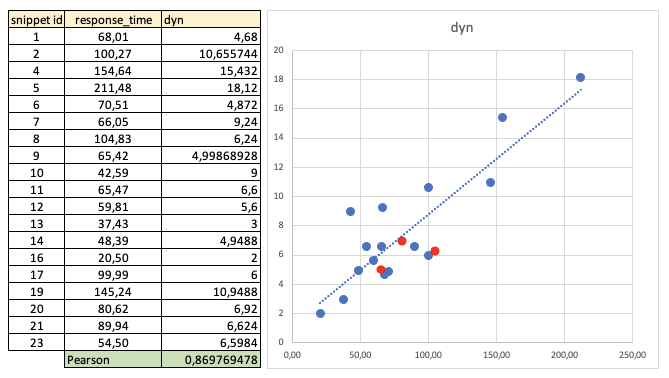
As we can see, the linear correlation is already much better than with the static metric of Genese Cpx (and even more with the other metrics that we tried). The Pearson coefficient was approximately equal to 0.556, and now to 0.87.

After successive tries, we optimized the value of the coefficient *c*. The best approximation of *c* in this experiment is approximately equal to 0.4.

**6.4 Modulos**

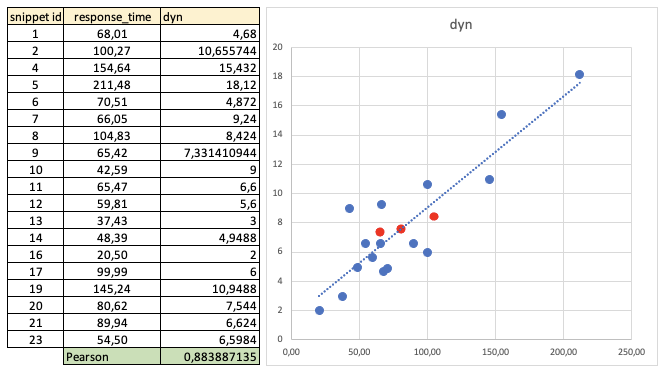
When we tried to reproduce the experiment by mentally executing the functions, we noticed that some operators were more difficult to use than others. For example, the operator “modulo” (%) is clearly more difficult to use than the operators + or \*. Our dynamic metric didn’t take this fact into account. Consequently, we expected that the functions containing the modulo operator have a score abnormally low in our calculations.

We verified this intuition by colouring in red the dots corresponding to the 3 functions containing modulos :



The supposition seemed to be correct: the 3 dots are below the linear regression line.

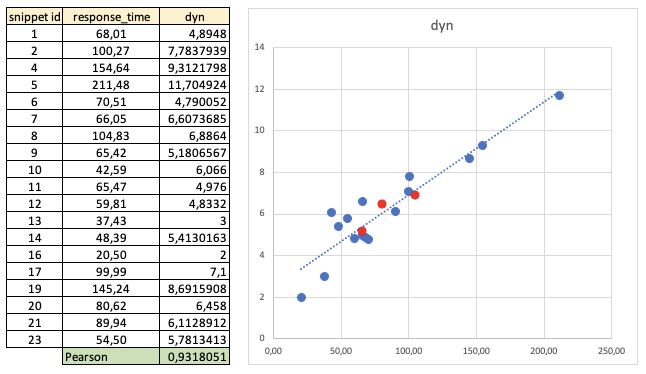
That’s why we decided to improve the performance of our metric by adding a weight to the modulos. After successive approximations, the best value for this weight was equal to 1.4.



The correlation is now a little better : the Pearson coefficient grows from 0.87 to 0.884.

**6.5 Reuse of same values for same variables**

Finally, we remarked that when we used multiple times the same variable which was keeping the same value, we needed each time less effort to remember it. That’s why, like for loops, we used a coefficient *c* decreasing the weight of the variables each time that we read them (when they were keeping the same value). The best approximation of *c* seemed to be approximately equal to 0.7.



Now, the correlation is excellent, with a Pearson coefficient approximately equal to 0.932.

**7. Application of the dynamic metric to another experiment**

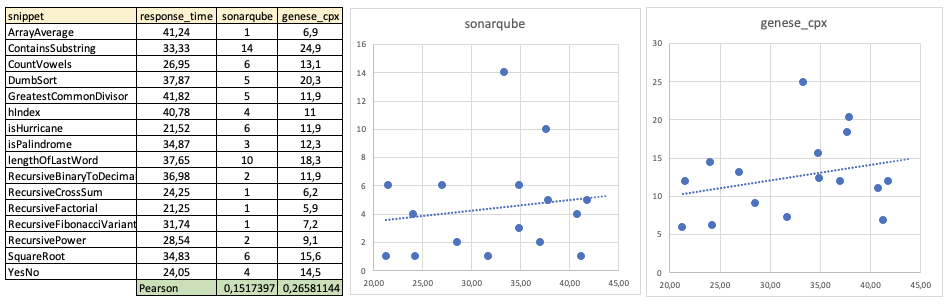
**7.1 The Peitek’s experiment**

In the previous chapter, we intuitively detected some parameters which seemed to have a significant influence on the time needed by the developers to execute some classic mental processes. In the aim to be able to predict the average time needed by developers to execute some mental processes, we decided to use our dynamic metric with the results of another experiment. This metric is defined by 4 rules :

* add a weight equal to 1 to each identifier used in mental calculations
* use a coefficient equal to 0.4 in the aim to decrease exponentially the score of the lines which are read multiple times
* add a weight equal to 1.4 for the modulo operators
* use a coefficient equal to 0.7 in the aim to decrease exponentially the score of the variables which are used consecutively with the same value.

Naturally, as these weights and coefficients were optimised for Siegmund’s experiment, it is sure that the correlation will be lower for others. We tried to estimate the validity of our metric by applying it to the results of an experiment realized by Norman Peitek in 2021. Its principle was the same as in Siegmund's experiment, but the functions and the developers (19 participants) were different.

First, let’s look at the correlation of SonarQube and Genese Cpx static metrics with the results of Peitek’s experiment :

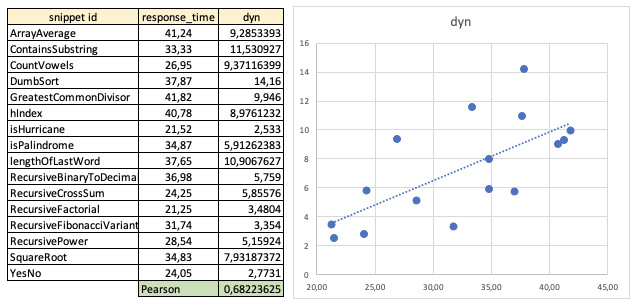


As we can see, the linear correlation of the results with the SonarQube metric is very weak, with a Pearson’s coefficient approximately equal to 0.152. The correlation with Genese Cpx is significantly better (0.266), but we still should consider it as weak.

As we said in chapter 4.3, it is normal to find a weak correlation for static metrics. In Siegmund’s experiment, the correlation was much better, but it was probably the result of multiple biases. In Peitek’s experiment, the functions are maybe less similar between them, or maybe the skills of the participants were less homogeneous. It is difficult to identify which are the most important factors which are explaining this difference.

Since the weak correlation between static metrics and Peitek’s results, we expected to find a weak correlation for our dynamic metric too.

Here are the results :

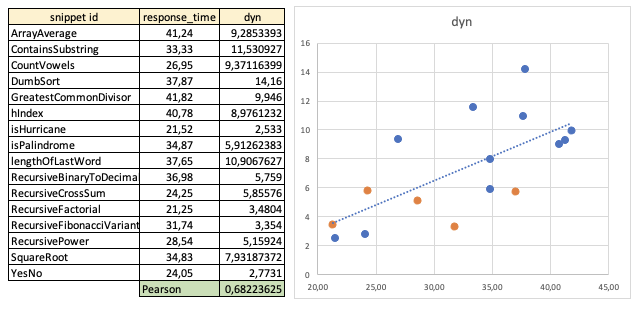


The linear correlation was better than expected, with a Pearson coefficient approximately equal to 0,682. Naturally, it is less than the 0.932 of the previous experiment, but it is still a good correlation, which is much better than every other static metric that we tried during our verifications.

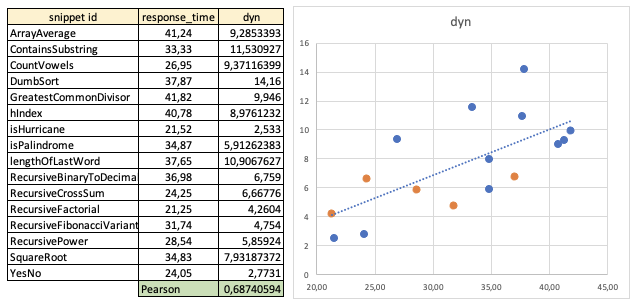
**7.2 Recursion**

We tried to identify which could be the code structures in the functions of the Peitek’s dataset which were absent from the Siegmund’s code snippets. We identified one : the recursions.

Intuitively, the recursivity is a non-trivial operation, which takes time to understand and to use correctly. We coloured in orange the dots corresponding to the functions containing recursions, and we expected that these dots should be located (in average) below the linear regression line.



As we expected, the orange dots are, on average, slightly below the line. Thus, like for modulo operators, we added a weight to the recursions. After multiple tries, the best weight seemed to be near 0.5.



As we can see, the correlation is slightly better, but not significantly. Maybe because of the heterogeneity of the code snippets ? For now, it is impossible for us to conclude.

**7.3 Synthesis of the two experiments**

We can now provide a synthesis of the results of the two experiments :

Une image contenant table

Description générée automatiquement

On average, for the two experiments, our dynamic metric has a Pearson coefficient which is approximately equal to 0.82, which may be considered as a good linear correlation.

**8. Conclusion**

The lectures of the publications of Scalabrino, Wyrich and Peitek inspired us many questions. Why, for most of the known metrics, the correlation with the code comprehension seems to be so weak, or nonexistent ? Why, for some experiments, the correlation seems to be much better ? The first goal of this article was to reply to these two questions.

To that end, we saw that some of the experiments used in some studies consist of asking developers to mentally execute some programs. Then, we gave in chapters 4.1 and 4.2 two reasons which may explain why we should not expect to find a correlation between the scores of static complexity metrics and the results of this kind of experiment. In chapter 4.3, we gave other reasons which could explain why, in some experiments, some of these metrics seem to be correlated with their results, and why we should be careful with the interpretation of this apparent correlation.

Later, in chapter 5, we imagined a new kind of metric : the dynamic metrics. Our main goal was to be able to predict with good precision the average time that, for the same experiment, the developers will need to mentally execute other (but similar) functions.

To that end, we reused the results published by Siegmund in 2012 and Peitek in 2021. By adjusting different weights and coefficients, we were able to define a dynamic metric with a very good correlation with the results of Siegmund’s experiment, and a good correlation with Peitek’s experiment.

We estimate that our dynamic metric is precise enough to make predictions with a correct error margin for similar experiments. However, we think that we need to add more parameters and to better optimize our weights and coefficients to be able to make predictions with a low error margin.

**9. Future work**

As we said in conclusion, we may try first to optimize our weights and coefficients to improve the correlation between our dynamic metric and the results of similar experiments. In this aim, we would like to use other datasets from other studies. We hope to avoid more biases with this work, and to be able to define a dynamic metric which would be able to make correct predictions for most of the past and future similar experiments.

Then, we would like to extend our metric to real projects, in the aim to be able to predict the average time needed by developers to debug the code, which is one of the most fastidious (and costly) tasks for developers. To this end, we will need to analyze the results of more sophisticated experiments, with more complex problematics. For example, we would like to analyze the results of experiments with functions using objects instead of primitives.

Finally, we would like to mix static and dynamic metrics, in the aim to develop a software which would be able to provide relevant information about the difficulty to debug a program.

**10. Acknowledgments**

Thanks to Janet Siegmund, Simone Scalabrino, Martin Wyrich and Norman Peitek for their amazing job, and for publishing their data in open-source. They inspired my work.

Thanks too to Chouki Tibermacine, Thibaud Thedon and all the [LIRMM] team (University Montpellier II), which encouraged me to work on this subject.

If you have any question or comment about this article, I will reply with pleasure. The datasets are available [here].

[1] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, *« Automatically Assessing Code Understandability: How Far Are We ? »* in Proc. Int’l Conf. Automated Software Engineering (ASE). IEEE, 2017, pp. 417–427.

[2] M. M. Baron, M. Wyrich, and S. Wagner, *« An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability »* in Proc. Int’l Symp. Empirical Software Engineering and Measurement (ESEM), 2020, p. 1-12.

[3] N. Peitek, S. Apel, C. Parnin, A. Brechmann, J. Siegmund, *« Program Comprehension and Code Complexity Metrics: An fMRI Study »*, in Proc. Int’l Conf. Software Engineering (ICSE) IEEE, 2021, pp 524-536

[4] J. Siegmund, A. Brechmann, S. Apel, C. Kastner, J. Liebig, T. Leich, G. Saake, *« Toward Measuring Program Comprehension with Functional Magnetic Resonance Imaging »*, in Proc. of the ACM SIGSOFT, 20th International Symposium on the Foundations of Software Engineering