The Attraction of Complexity

Carlo Bottiglieri

December 10, 2017

1 Introduction

How is complexity distributed through a codebase? Does this distribution present similarities across different projects? And, especially, are we more likely to change complex code or simple code?

This summer, towards the end of my article about Technical Interest[1], I found that I wanted to write something like: "not only working on complex pieces of code is expensive, but we are also more likely to work on those complex pieces than on the simple ones". This rings true to my ears, and everyone I talked with on the topic of complexity reasons that, the more complex a piece of code is, the more logic it contains; since the goal of a system modification is to change its logic, we are more likely to end up touching a logic-rich part than a low complexity part. But is this reasoning backed by reality?

I could think of only one way to know: go through all the changes in the history of multiple projects and calculate the complexity of the functions containing each change and then group data together to see if it makes any sense.

2 Collecting the Data

It all starts with git, of course: I created a node.js script that scans the history of a GitHub project and checks out the files changed by every commit, putting them in a folder named with the commit id and writing a file: `changes.csv`, which lists the filepath and line number of every change.

Then I wrote a script in SonarJS that reads `changes.csv`, parses every file mentioned therein and calculates the Cognitive Complexity of the function directly containing the line listed in the change. It stores the filepath, the function name, its complexity and its size in a new file: `stats.csv`. The same script also calculates the complexity of all functions currently (`checkout HEAD`) in the project, it stores the aggregated size of all functions with the same complexity, in a file: `complexity_distribution.csv`.

[1] https://minnenratta.wordpress.com/2017/09/24/quantifying-the-cost-of-technical-debt/

2.1 The Dataset

The projects I selected are :

- Keystone : https://github.com/keystonejs/keystone
- Prettier : https://github.com/prettier/prettier
- Lighthouse : https://github.com/GoogleChrome/lighthouse
- Restbase : https://github.com/wikimedia/restbase
- Ghost : https://github.com/TryGhost/Ghost

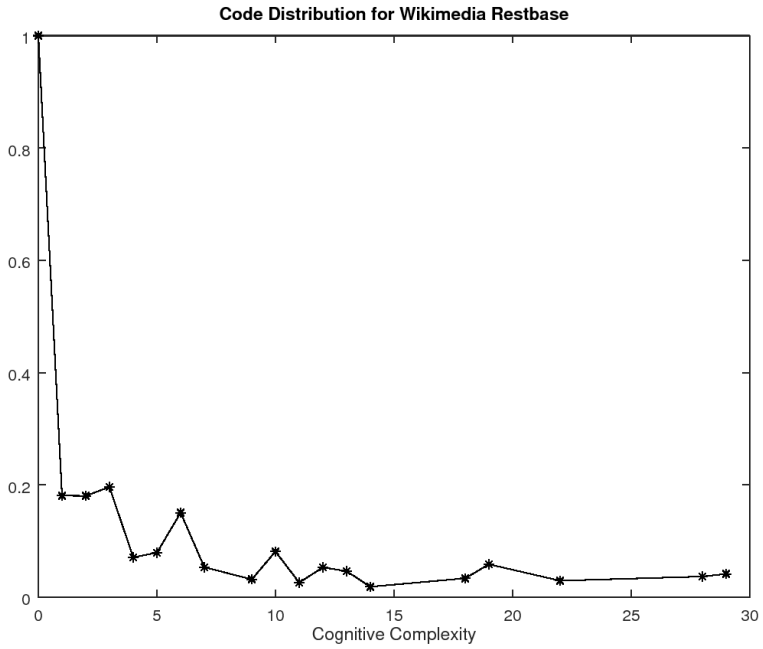
I picked them to get a mix of large, medium and small projects, both libraries and final products. I also wanted to have a large number of changes overall: these five projects together account for roughly 220.000 JS function changes.

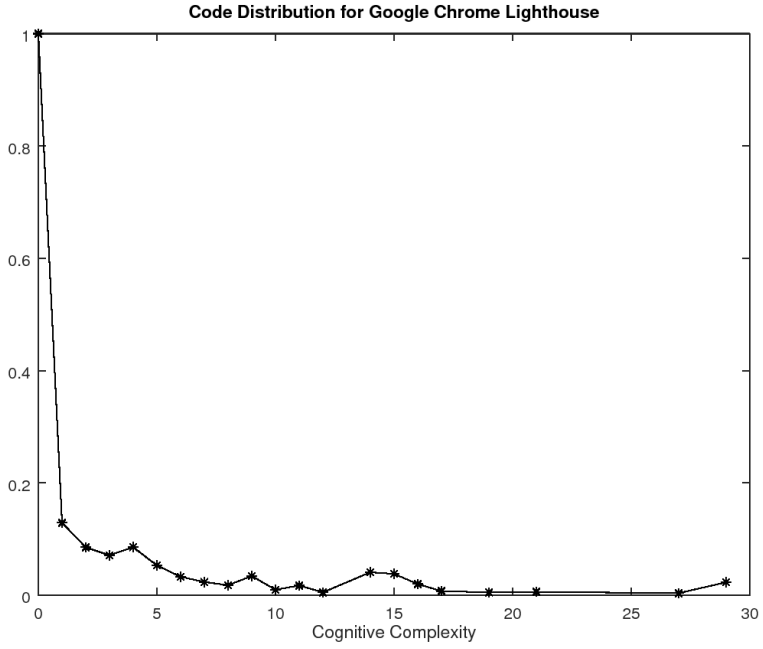
2.2 Why only JavaScript

Since I work on SonarJS most of the time, it's easy for me to customize it as I need and thus focusing on JavaScript projects spared me a lot of effort. This leaves an interesting area of investigation wide open: how these findings transpose, if at all, to code-bases written in languages with different features and domains of application?

3 Project Code Distribution

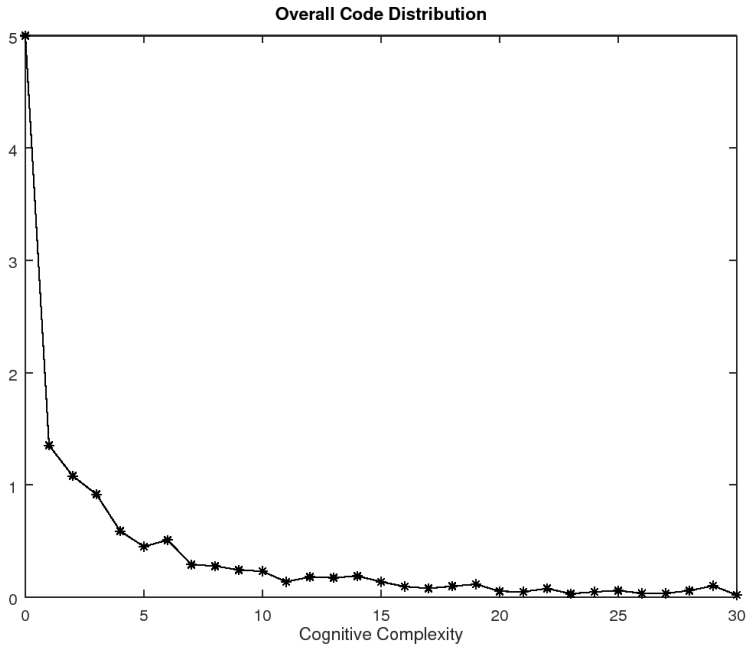
I collected the overall code distribution (the data found in `complexity_distribution.csv`) mostly because I needed it for the change frequency normalizations later on, but a first interesting initial finding is that, looking at how code is distributed across complexity at a given point in time, the distributions are quite similar across very diverse projects. Here's a couple examples and then the aggregation of all five under study. The amount of code in the projects' graphs is normalized to the largest value, so all normalized values are within 1 and 0. I calculated absolute values by counting the number of expressions/statements and, depending on the project, the value 1 can represent tens of thousands or hundreds of thousands of those.





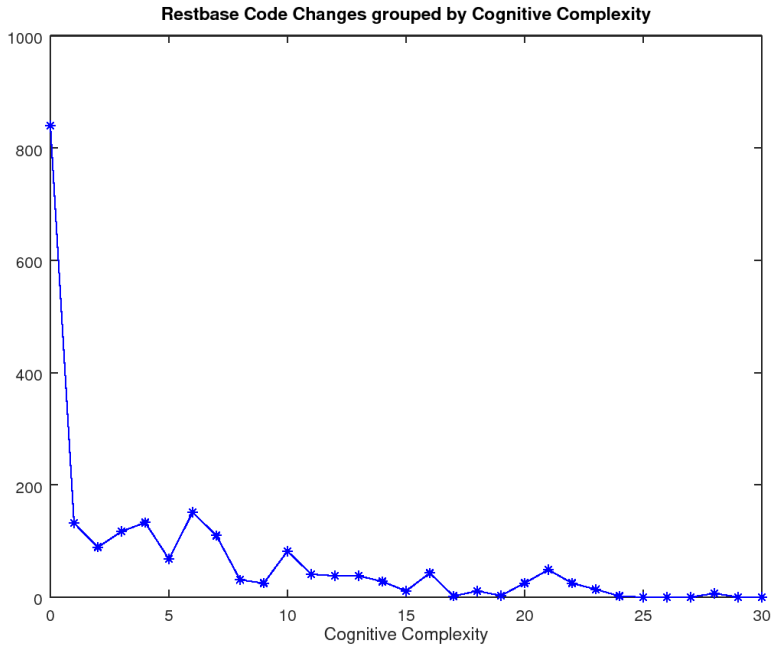
For this and any other aggregations across the five target projects, don't

forget that I add normalized values, this way the relative of each project has no impact.

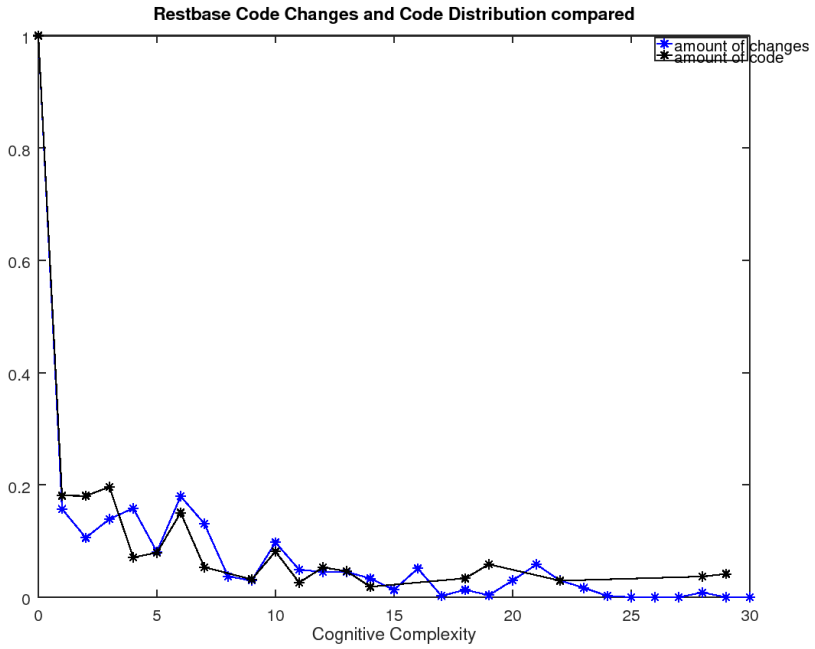


4 Complexity and Change Frequency

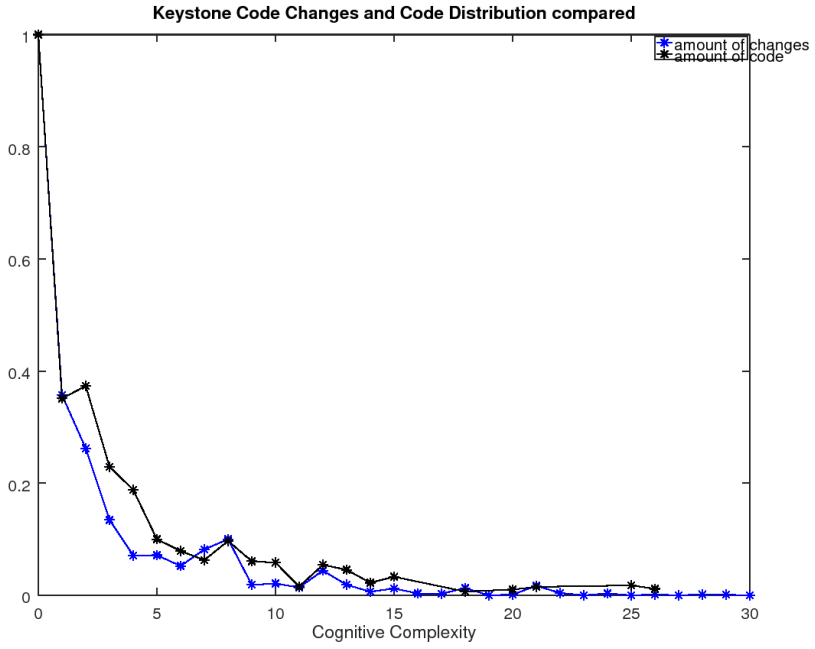
We are now getting closer to the core of the matter, let's see how many times code is changed over the history of a project depending on the code complexity. Here's Restbase:



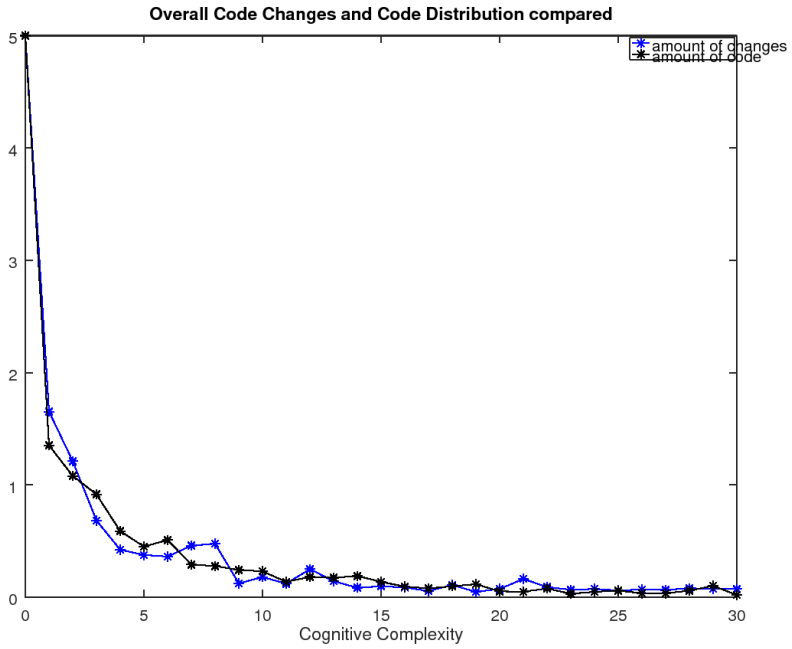
The vast majority of changes is happening in low complexity code. It would seem that there's an inverse relationship between the likelihood of change and the complexity of code; but this diagram is also strikingly similar to Restbase's general code distribution. If I normalize the number of changes and plot them together with the code distribution, the similarity is obvious:



The same is true for the other projects, for instance, Keystone:



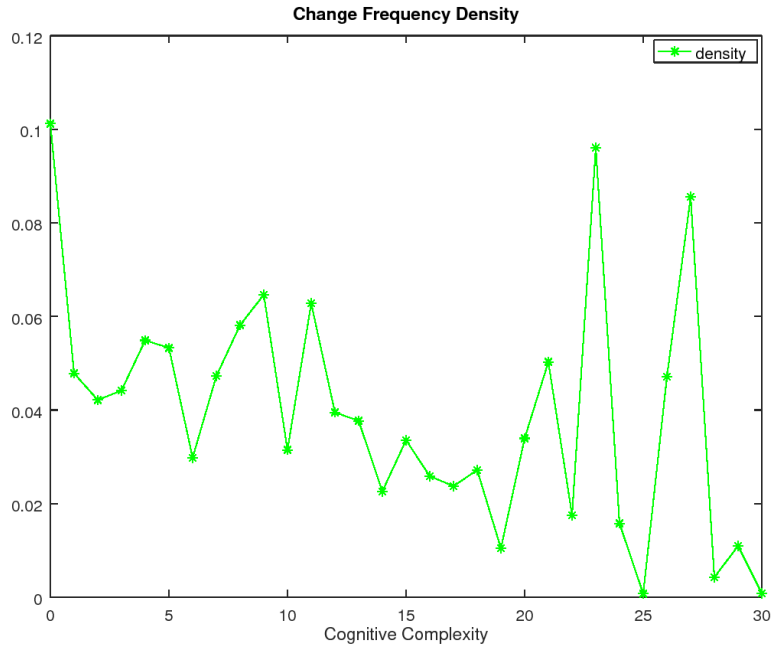
And all projects combined:



This makes sense: if a project contains code, that code has to come from changes, more code, more changes. I can't stop here though, the sheer mass of code and the changes that were required to initially write it might be hiding the true trends of change frequency over complexity.

5 Change Frequency Density

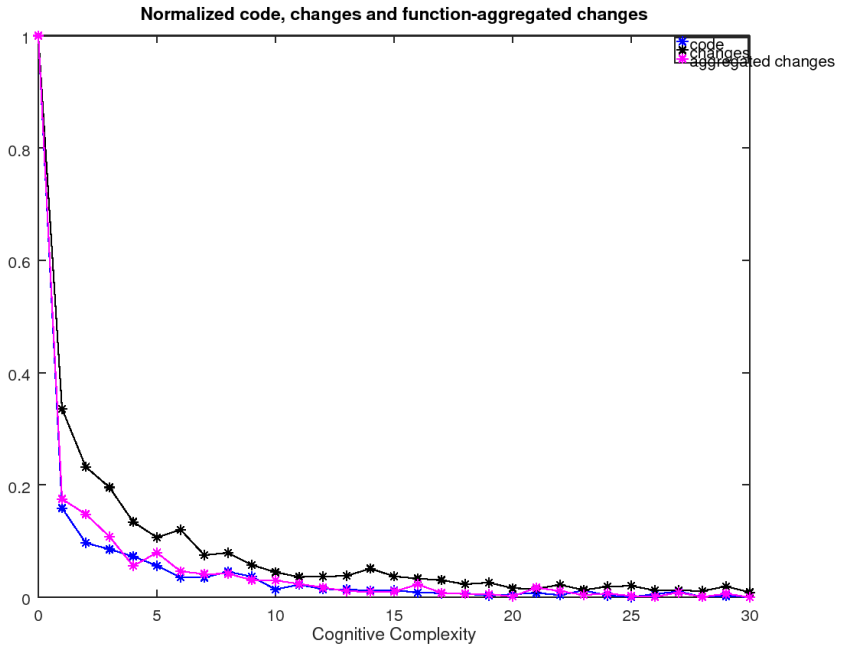
If I normalize the number of changes over the amount of code I am calculating how frequently a piece of code changes on average. This I call the change frequency density, or change frequency per expression. This measure can be interpreted also as how likely is a piece of code to change depending on the complexity of the function that contains it.



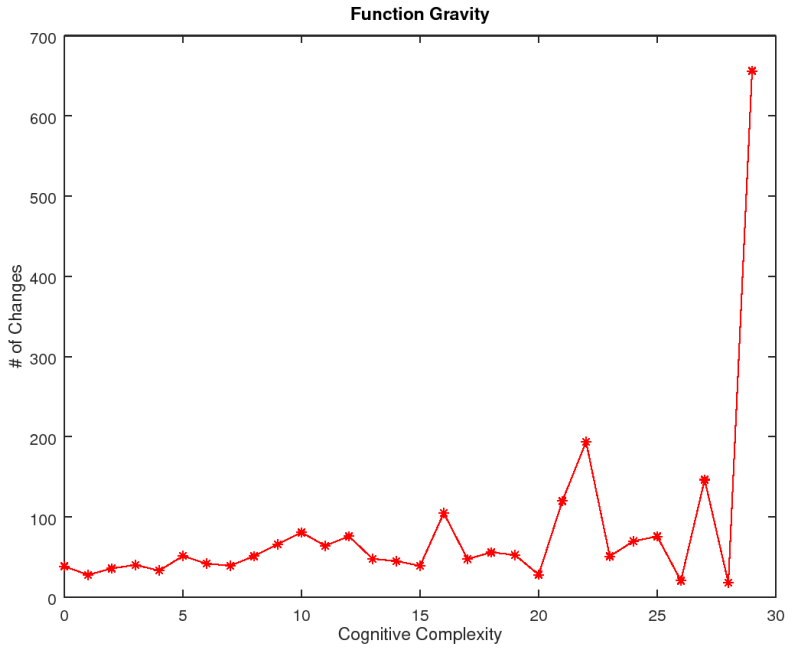
This looks almost like random noise: after an initial peak (one change for every ten expressions) for code at zero-complexity, the number of changes per expression drops and continues to decrease until well after complexity 15, then it starts to jump around randomly.

6 Function Gravity

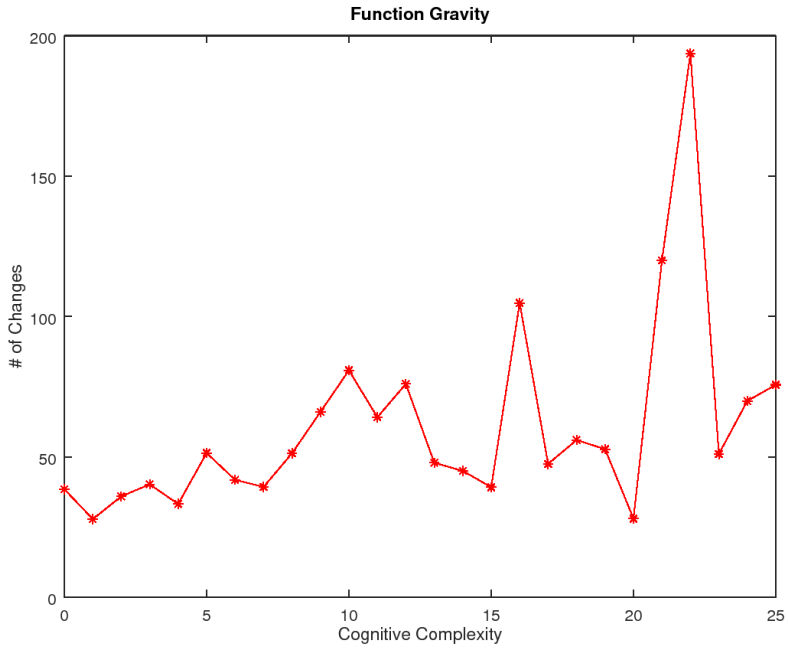
The problem is that complexity is a metric that is function-based and the goal of all this exercise is to find out if a function with a given complexity is more or less likely to change, not if a specific expression is more or less likely to change when it belongs to a function with a given complexity. What I want is to get the number of changes in the history of a function of a given complexity, so I group together all the changes of every function (this is something that I can do because I've collected the fully-defined name of the function enclosing every git change) and then I calculate the average complexity of the function through its history, all the changes of the function are then accounted for at that average complexity. In short, function-aggregated changes.



Looking carefully, the pink plot shows some interesting bumps, especially in the higher complexities, but the overall code distribution still dominates the shape of the data. What if I calculate the aggregated changes divided by the number of functions? That should really show how much a single function aggregates changes regardless of the sheer amount of functions existing at a given complexity. I call this, the 'Function Gravity'.



I suspect the final peak is more of an anomaly due to the extreme rarity of functions with an average complexity above 25, Let's reduce the range to maximum 25, to stay in an area where the data-set is not too thin.



I distinguish two zones of this graph. From zero to complexity 10, the attraction is almost monotonically growing, after 10 there are peaks and valleys, with a similar mean growth, but minima that, while being more than twice the complexity, can have a much lower attraction to change than the functions with average complexity 10.

This means that using the complexity of a function is a relatively good way to target the refactoring effort, functions with complexity 8 and 12 today are likely to accumulate much more changes, as they get to higher complexities, than functions with complexity 0 to 6 would (remember the graph shows a function's average complexity over its life).

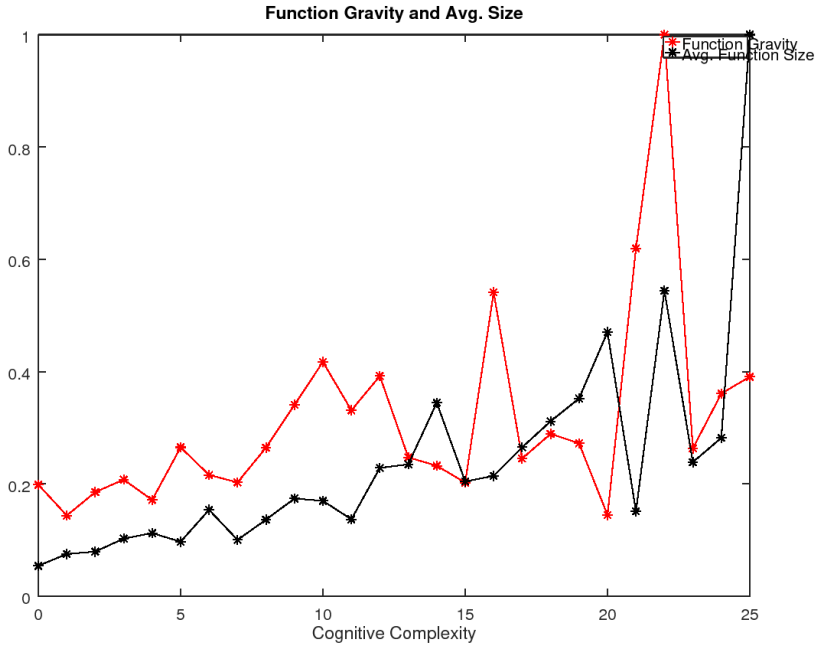
This is no longer true after complexity 12 though: while targeting a function that today has complexity 16 might allow you to refactor a piece of code that will change a lot, the vast oscillations in this part of the graph make this an hazardous bet, you can win big, but you can also spend days to refactor a complicated function that in the future will not change more frequently than a 0-complexity getter.

Can we get a better metric to spot functions which change very frequently?

7 Change Potential

Function that have historically received many changes should be larger on average. But does that hold true? I believe the macroscopic approach has now

reached its limits and to proceed further I should start studying specific functions and see if there are any indicators to their change-frequency future, but still, a final set of graphs:



This seems to confirm that between 0 and 10 favoring the refactoring of higher complexity functions over lower complexity functions is a safe bet, especially in the range 7 - 10. It also shows the random nature of the code changes after complexity 15: it becomes possible to both have lower average sizes with proportionally much larger change histories and large functions which change very rarely. This might be because very complex code impacts developer behavior: for instance, it's not unusual for developers to group multiple changes to especially complex functions to avoid having to re-learn them multiple times. It's in those places, at high complexity, where functions changes and function sizes diverge strongly, where I would like to abandon the aggregate view and look at specific cases, but that will have to wait.

8 Conclusions

There seems to be a correlation between function complexity and number of changes, beyond what mere function size would suggest, at least between complexity 0 and 10. After complexity 10 things start to change, after complexity 15 the appearance of relatively small functions with lots of changes and large, high-complexity functions with few changes, makes for some great opportunities and big risks.

9 Side Notes

While I was writing this article I happened to think of the open-closed principle. OCP is notoriously hard to measure and in a discussion on the XP mailing list years ago I even went so far as to say that it's impossible to measure (or apply) OCP up front, it can only be used to evaluate the soundness of a system's modularity after the fact. From this point of view, the distribution of changes over complexity might be seen as a metric to measure how well OCP is respected: if existing complexity is most often the site of new changes, OCP is not being respected, if new changes tend to happen mostly in low complexity zones of code instead, we are implementing new features without modifying existing logic, and OCP is respected.