

Finding significant differences in the energy consumption when comparing programming languages and programs

Lukas Koedijk

lukaskoedijk@gmail.com

July 12, 2019, 50 pages

Research supervisor: Dr. A. M. Oprescu, a.m.oprescu@uva.nl
Host/Daily supervisor: D. Stam, stam.dennis@kpmg.nl
Host organisation/Research group: KPMG Advisory N.V., www.kpmg.nl



UNIVERSITEIT VAN AMSTERDAM

FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA

MASTER SOFTWARE ENGINEERING

<http://www.software-engineering-amsterdam.nl>

Abstract

Nowadays it seems like more and more people are concerned with global warming. To help solve this problem we all could try to consume less energy. In the computer science industry this means reducing the energy consumption of a program. This can be done by looking at the hardware or software used. We look at the programming language chosen and how the program is written. The programming languages we investigate are Java, JavaScript, Python, PHP, Ruby, C, C++ and C#. The programs we compare are from *The Computer Language Benchmarks Game* and are divided in groups that solve the same problem. These programs are run on a distributed system called the DAS and their energy consumption is measured using a PDU. We compared the measurements using the Mann Whitney U test with an *alpha*-value of 5% and in some cases we could not prove that there was a difference in the energy consumption. We find that the choice of programming language and the way of writing your program influences the energy consumption. The programming languages that consume the least amount of energy over all the problems are C and C++, these programming languages consume more energy when compilation flags are minimized. We also find that the hardware used to run the programs has an influence on the energy consumption. Furthermore we find that the run time of a programs correlates with the energy consumption, but a shorter run time does not necessarily mean a lower energy consumption.

Contents

1	Introduction	4
1.1	Problem statement	4
1.1.1	Research questions	4
1.1.2	Research method	5
1.2	Contributions	6
1.3	Outline	6
2	Background	7
2.1	Statistics	7
2.1.1	Normal distribution	7
2.1.2	Same distribution	7
2.1.3	Correlation	8
2.1.4	Anomaly detection	8
2.1.5	Clustering	9
3	Experimental set-up	10
3.1	Energy measurement framework	10
3.1.1	Distributed ASCI Supercomputer	10
3.1.2	Approach	10
3.1.3	Set-up	11
3.2	Experimental design	12
3.3	Experimental data preparation	13
3.3.1	Programming languages	13
3.3.2	The computer language benchmarks game	14
3.3.3	Gathering programs	15
3.3.4	Anomalies	15
3.3.5	Hardware	17
3.3.6	Execution time	17
4	Results	20
4.1	Programming Languages	20
22	subsection.4.1.1	
4.2	Programs	22
4.2.1	Code level	22
24	subsection.4.2.2	
5	Discussion	25
5.1	Programming Languages	25
5.2	Programs	26
5.3	Threats to validity	26
6	Related work	28
6.1	Measuring energy consumption	28
6.2	Estimating energy consumption	29
6.3	Finding optimal energy consumption	29
6.4	Hardware	29
6.5	Other	29

7 Conclusion	30
7.1 Future work	30
Bibliography	32
Appendix A Programming languages	34
Appendix B Programs	42
Appendix C Correlation	44
Appendix D Code-level	49

Chapter 1

Introduction

Nowadays it seems like more and more people are concerned with global warming. Global warming is partly the result of the emission of greenhouse gasses during energy generation of conventional energy options [1]. One solution for this problem is to change to green energy generation. Another solution is to decrease the energy consumption, which is not only good for the environment but can also save a lot of money on the energy bill. The energy consumption of communication networks, personal computers and data centers worldwide is increasing every year [2]. This happens at a growth rate of 10%, 5% and 4% respectively [2], therefore it is important to research ways of decreasing the energy consumption. In the field of hardware there is, according to Koomey's law [3], an increase in the number of computations per Joule. However this is not enough, because tasks are designed to need more computations to complete due to the confidence in the improvement of hardware [4]. For this reason we need to look at possibilities in decreasing the energy consumption from a software perspective.

1.1 Problem statement

Decreasing the energy consumption in the computer science industry can be done by looking at the hardware or the software. We found a lot of scientific research focused on decreasing the energy consumption of hardware. There is also some research done about reducing the energy consumption in the software process and the decision making process. Some research is done on the energy consumption of software, but their research goal is mainly to estimate the energy consumption. This thesis aims to investigate good ways of writing code regarding the energy consumption.

1.1.1 Research questions

During this research we answer the following two existence questions:

- (1) **Is there a difference in the energy consumption of software projects in different programming languages that have the same functionality?**
- (2) **Is there a difference in the energy consumption of different software projects (using the same programming language) that have the same functionality?**

To answer these questions we first need to answer the three description and classification questions listed below.

- How can the energy consumption of a software project be measured?
- How do we prove that two programs have the same functionality?
- When is a difference in energy consumption significant?

Based on the results we find when answering the second existence question (2), we look into what this difference is on code level. To do this we need to answer the descriptive/comparative question **What is the difference at code level between software projects that are in the same programming language and have the same functionality, but have a difference in the energy consumption?** This question is only useful when there is a difference in the energy consumption of software projects, in the same programming language, that have the same functionality.

For the first existence question (1) the null hypothesis is that there is a difference in the energy

consumption based on the programming language chosen. The null hypothesis of the second existence question (2) is that there is a difference in the energy consumption of different software projects (using the same programming language) that have the same functionality.

1.1.2 Research method

To answer these questions we derived a controlled experiment, where we run different software projects to measure the energy consumption. Here the projects are the variable input, the energy consumption the output and the environmental variables such as the compiler used and energy consumption measurement method should be constant.

Data used for experiments

For the first research question we need programs that have the same functionality, but are written in different programming languages. Possible resources for such programs are library code, interview code, student assignments and code from competitions solving math problems. We first looked at library code, but found that libraries most of the time do not use the same algorithm. This is a problem because the research question does not target comparing algorithms, but the way a programmer writes code. Therefore we used *The Computer Language Benchmarks Game* [5] as a source for programs that have the same functionality and use the same algorithm for solving the problem at hand. The second research question requires programs that offer the same functionality and are written in the same programming language. Here we also use *The Computer Language Benchmarks Game* as for most problems and programming languages there are multiple implementations.

When proving that all the programs have the same functionality we could have defined properties for the input and output and test if these properties hold for all the programs [6], but *The Computer Language Benchmarks Game* already makes sure that the programs offer the same functionality. This research builds upon these validations.

We use the one-sided Mann Whitney U test to check whether programs consume different amounts of energy. The null hypothesis of this test is that the two distributions are from the same population.

Measuring energy consumption

Measuring the energy consumption of a software project takes into account the energy consumption of CPU, memory and disk [7]. The measurements can be done with a hardware or software approach. A hardware approach is more accurate, but buying hardware is also more expensive [7]. When the hardware is already available and does not need to be bought, then we do not have this disadvantage. This research uses a hardware approach and utilizes the DAS-4 and DAS-5 systems. The DAS is a distributed system which can also be used to do energy measurements [8].

Code level

If there is a difference in the energy consumption of different software projects in the same programming language, we look at the different projects at code level. Here we try to find what is causing a project to have a lower or higher energy consumption. This is done by comparing programs, that consume different amounts of energy, side by side.

1.2 Contributions

Our research makes the following contributions:

1. **An energy consumption measurement set-up.** At the end of this research we have a set-up which we can use to run different kinds of programs on the DAS and measure the energy consumption of these programs. We design this set-up alongside Stephan Kok, who also needs to measure the energy consumption for his thesis. The code for this set-up is available on git¹.
2. **A data set of software programs featuring the same functionality.** During this research we will collect a lot of programs in various programming languages. There will be different groups of programs that will solve the same problem and these programs are a selection from *The Computer Language Benchmarks Game*. All these programs and their compilation methods are available on git¹.
3. **Comparison of programming languages on energy consumption efficiency.** After this research we will have an idea about which programming languages perform better regarding the energy consumption. This comparison is shown in chapter 4.
4. **Preliminary guidelines for writing good software regarding the energy consumption.** We will show some preliminary guidelines about the differences on code level between programs that have a difference in their energy consumption. These findings are shown in chapter 4.

1.3 Outline

In chapter 2 we describe background knowledge that we need to understand the rest of the thesis. Chapter 3 discusses the energy measurement set-up we used, the design of the experiments and the data. The results are shown in chapter 4 and in chapter 5 we discuss these results. In chapter 6 we discuss papers related to our research and lastly we conclude our research in chapter 7.

¹<https://github.com/lukaskoedijk/Green-Software>

Chapter 2

Background

In this chapter we will explain the background information needed to understand the thesis. We do this by investigating the different statistical test that are used.

2.1 Statistics

Not understanding the terminology of statistics may lead to confusion, therefor here are some basic principles in statistics. When performing a statistical test we have a null hypothesis and an alternative hypothesis. Every test has it specific null and alternative hypotheses and the goal of the test is to reject the null hypothesis. Rejecting the null hypothesis is done by looking at the resulting p -value of the test. The p -value is the chance that the value of the statistical test occurs if the null hypothesis is true on a zero to one scale. We therefor reject the null hypothesis if we think this chance is too low, thus below a certain threshold. This threshold is determined beforehand and is called the *alpha*-value, a common value for *alpha* is 5%. When we cannot reject the null hypothesis, the p -value is not below 0.05, it does not necessarily mean that the null hypothesis is true. It could be the case that the test is not powerful enough or the data set used to validate the hypothesis is too small.

2.1.1 Normal distribution

When choosing a statistical test we have to take into consideration the preconditions for using the test. The most common precondition is that the data follows a normal distribution. The Shapiro-Wilk test has more power than the other normality tests [9] and therefore this test was used. The Shapiro-Wilk test calculates a statistic by dividing the summation to the power of two of every point times a coefficient by the summation of every point minus the mean to the power of two [10]. This formula is shown in equation 2.1. The null hypothesis of the Shapiro-Wilk test is that the data is normally distributed. This means that when the null hypothesis gets rejected the data does not follow a normal distribution. For this test we used an *alpha*-value of 1%, because distributions that are close to normally distributed can still be used in some of the statistical tests. When testing if the energy measurements from a single program on a single node follow the normal distribution using the Shapiro-Wilk test we get that not all programs measurements are normally distributed. From the 269 different programs 183 were not normally distributed on node029 and 38 on node028. Therefor we need to choose a statistical test that does not assume the data to be normally distributed.

$$W = \frac{(\sum_{i=1}^n a_i x_{(i)})^2}{\sum_{i=1}^n (x_i - \bar{x})^2} \quad (2.1)$$

2.1.2 Same distribution

When comparing different programs, that are in the same programming language and have the same functionality, we need to find out if there is a significant difference regarding the energy consumption. This can be tested by looking at the two different distributions and testing if they are from the same population. Here the null hypothesis is that the distributions are from the same population. A commonly used test with this null hypothesis is the students t-test. This test however has the preconditions that the two distributions should follow the normal distribution and the variance of the two distributions

should be equal. Our data does not match these preconditions and therefore we could not use this test. An alternative is the Mann Whitney U test [11], which looks at the chance that a random variable from the first distribution is greater than a random variable from the second distribution. When this chance is 50%, then the two different distributions belong to the same population.

There are two versions of the Mann Whitney U test, the one-sided and the two-sided test. For the two-sided Mann Whitney U test the alternative hypothesis is that the distributions are from different populations, but if you want to know the direction of this comparison you need to use the one-sided Mann Whitney test [12]. Which means that the alternative hypothesis for the one-sided Mann Whitney U test is that the first distribution is stochastically larger than the second distribution [12]. Because rejecting the null hypothesis holds more power than not rejecting the null hypothesis, a one-sided Mann Whitney U test was used twice. We tested if the first distribution was stochastically larger than the second distribution and if the second distribution was stochastically larger than the first, with an *alpha*-value of 0.05. If both these tests cannot reject the null hypothesis then we still do not know for certain that the distributions are from the same population.

We also used the two sample Kolmogorov-Smirnov test to find out if two distributions are from the same population. This test also has as its null hypothesis that the two distributions are from the same population and we used an *alpha*-value of 0.05. The Kolmogorov-Smirnov test compares the empirical distribution functions of the two distributions.

2.1.3 Correlation

A commonly known method for calculating the correlation is called the Pearson coefficient. The Pearson coefficient uses the covariance of two variables to calculate its correlation score [13]. For this method the following assumptions should hold, the correlation is linear and the data follows a normal distribution. Our data however does not meet these assumptions for all programs and therefore we needed to look at a different method. The Kendall Tau coefficient calculates a ranked correlation coefficient and does not assume that the data follows a normal distribution [14], therefore this method was used. The Kendall Tau method looks at how many pairs of points follow the same order. For example points (x_i, y_i) and (x_j, y_j) follow the same order if $x_i > x_j$ and $y_i > y_j$ or if $x_i < x_j$ and $y_i < y_j$. Because every pair needs to be checked this could get computational heavy for large number of points, luckily that is not the case in this research project.

The coefficient is in the range of minus one to one, where at zero there is no correlation and at one or minus one there is. But what about the scores in between? According to the standard Guilford scale the range 0 – 0.2 means slight correlation, 0.2 – 0.4 means low correlation, 0.4 – 0.7 means moderate correlation, 0.7 – 0.9 means high correlation and 0.9 – 1 means very high correlation [15]. Because the minus in the coefficient shows which direction the correlation is in, we can use the absolute value to determine in which range of correlation a negative number is.

2.1.4 Anomaly detection

An anomaly, also called outlier, is a point that does not behave like the rest of the points in the results. These anomalies can be detected by using a clustering algorithm like DBSCAN [16]. The DBSCAN algorithm goes through every point and counts how many other points are in a predefined area around the point. If there are more or equal to the minimum amount of points needed in this area then this point is labelled as a core point. Points that are not core points but are in the predefined area around a core point are labelled border points, all the other points are anomalies [17]. This is visualized in figure 2.1, where all red dots are core points, yellow dots border points and blue dots anomalies. The minimum amount of points needed around a point and the predefined area around a point are the input variables of this algorithm, which is the disadvantage of this method.

Input variables

According to Ester *et al.* we can set the minimum amount of points a core point needs in his area to four for two dimensional data [17]. However for the area we need to calculate the radius, also called the *eps* variable. This *eps* is calculated by first calculating the distance to the fourth nearest neighbour for every point. These distances are then sorted and plotted from highest to lowest. The value for *eps* can

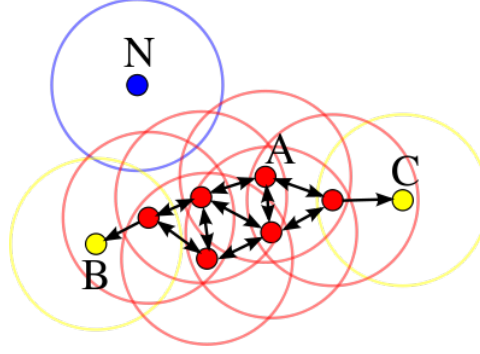


Figure 2.1: The labels according to DBSCAN algorithm with minimum points is four, where the red dots are core points, yellow dots border points and blue dots anomalies. Core points are points that have within a certain distance a minimum amount of points. Border points do not have the minimum amount, but are within reach of a core point. Anomalies do not have the minimum amount of points in a certain area and are not reachable from a core point. (source: <https://en.wikipedia.org/wiki/DBSCAN>)

then be read from the graph looking at the first valley [17]. In our implementation we calculated this *eps* automatically for every single program. To retrieve the first valley we looked at the differences between the slopes around a point. The point with the biggest difference in slopes was the point in the valley and was used as our *eps* variable for that specific program.

These two parameters have a big influence on how the clusters are divided and which points are labelled as an anomaly. When the minimum amount of points gets smaller more clusters will be formed and when it gets larger less clusters will be formed, assuming *eps* stays the same. When *eps* gets smaller more points will be labelled as anomalies and when *eps* gets larger less points will be labelled as anomalies, assuming the minimum points parameter stays the same.

2.1.5 Clustering

When trying to find clusters in our data, we were searching for a specific amount of clusters. We used the k-means clustering algorithm, because this algorithm works best when the amount of clusters are known from the start. This algorithm begins by randomly assigning means for the amount of clusters specified [18]. Then all the points are passed and divided into a cluster based on which mean has the shortest Euclidean distance. After this, the mean of the clusters is calculated and these will be the new means. This will iterate till a local maximum is found or the maximum amount of iterations has passed. The maximum amount of iterations was left to the default of 300 from the *sklearn* python packages.

Chapter 3

Experimental set-up

In this chapter we discuss how the experiments were set-up. We do this by first discussing the framework created for measuring the energy consumption. After that we discuss how we design/execute the experiments. Lastly we discuss the data used in our experiments.

3.1 Energy measurement framework

When doing energy measurements it is important to not only measure the energy consumption of the CPU, but also of the memory and disk [7]. You can use a hardware method to measure the energy consumption or use a software method to estimate the energy consumption. Using a hardware method is more accurate, but also more expensive [7]. This research uses a hardware approach and utilizes the DAS-4 and DAS-5 systems. In this section we discuss the approach used for measuring the energy. We first look into the *Distributed ASCI Supercomputer* and how to measure the energy consumption. After that we will look at the options and the issues this energy measurement approach has. Finally we show an overview of the set-up used to measure the energy consumption.

3.1.1 Distributed ASCI Supercomputer

The Distributed ASCI Supercomputer (DAS) is used for scientific experiments and was designed by the Advance School for Computing and Imaging (ASCI) [8]. There are a total of six cluster that are located at various locations, namely the *Universiteit van Amsterdam* (UvA), *Vrije Universiteit* (VU), *Universiteit Leiden* (UL), *Technische Universiteit Delft* (TUD), *ASTRON* and *The MultimediaN Consortium* (UvA-MN). Each cluster has different nodes and hardware available, we used the cluster of the VU. Each cluster has a head node where all the users connect to. Here the users can reserve nodes and add jobs to the queue. There are multiple releases of the DAS, currently only DAS-4 and DAS-5 are in use. On the VU DAS-5 cluster we have a total of six nodes available for measuring the energy consumption, because they are connected to a Power Distribution Unit (PDU). This PDU is from Racktivity and has an error margin of 1%. To connect to this PDU we need to make a connection from the DAS-4 through the use of the Simple Network Management Protocol (SNMP). These six node each have different hardware specifications and therefore it is important to distinguish between the nodes. The nodes we used are *node28* and *node29*, where the hardware specifications for *node28* are a GPU node with an Nvidia Tesla K20 (with 6 GB onboard memory), an Xeon Phi and a michost and *node29* has a GPU node with an Nvidia GTX980 (with 4 GB onboard memory) and an TitanX-Pascal.

3.1.2 Approach

There were two kind of energy measurement values we could retrieve from the PDU, the current power (Watt) and the energy consumption (kWh) from when the node was plugged-in till the moment this value is retrieved. Both methods have a disadvantage when applied. When using the current power one needs to retrieve the power constantly and some accuracy is lost because we do not know what happens with the power draw in between two measure points. The method of measuring the energy consumption has the problem of showing a number that is too large, the numbers are in kWh with a low precision. Thus the lowest decimal shows Watt per hour. We found that for small programs it is not sufficient to only measure in Watt per hour, because of the short run times. We tested this with three programs: an

idle program where the only command was *sleep*, and two programs that calculate the 10000th prime number, one recursively and one optimized. The idle program was tested ones with *sleep* eight minutes and twice with *sleep* 62 seconds to match the run times of the two different programs solving the 10000th prime number problem. The results of this test are shown in table 3.1. Here we see that the difference of 0.001 kWh is a large difference when working with numbers that are of scale 0.005 kWh. There is not a clear difference between primes optimized and the *sleep* that takes close to the same amount of seconds as the primes optimized. Because of these results we chose the power measurement method.

	Idle	Prime	PrimeOpt
Time	8m0.002s	7m58.443s	1m2.394s
Energy	0.033 kWh	0.037 kWh	0.005 kWh
Time	1m2.002s	7m59.611s	1m2.220s
Energy	0.004 kWh	0.037 kWh	0.005 kWh
Time	1m2.002s	7m58.503s	1m2.235s
Energy	0.005 kWh	0.036 kWh	0.004 kWh

Table 3.1: A test done using the energy measurement in kWh with three different programs. In the first row we tested the idle with *sleep* eight minutes and in the second and third row with *sleep* 62 seconds. The *Prime* and *PrimeOpt* calculate the 10000th prime number, but the *PrimeOpt* does this in an optimized way.

When using the power measurement method we retrieve a large number of measurement points, where each point has a timestamp and a power value. An example of a measurement and these points are shown in figure 3.1. To calculate the total energy consumed during this program we need to calculate the surface beneath the graph. To do this, we calculate the surface between two points and add all surfaces together. When two points do not have the same value we choose the average between the two to use for the surface. For this it is important to have a small interval between measurement points such that the error marge is as small as possible. When looking at the different nodes we see that they all have a different idle power state. To help compare between the nodes and reduce the difference between the measurement moments, we removed the idle state from the energy consumption. To do this we measured the idle state one minute before the first program and one minute after the last program of a single run of all the programs. The average power was then calculated and subtracted from every measurement. These calculations resulted in the following equation 3.1.

$$E = \sum (t_{n+1} - t_n) * \frac{(p_n - p_{idle}) + (p_{n+1} - p_{idle})}{2} \quad (3.1)$$

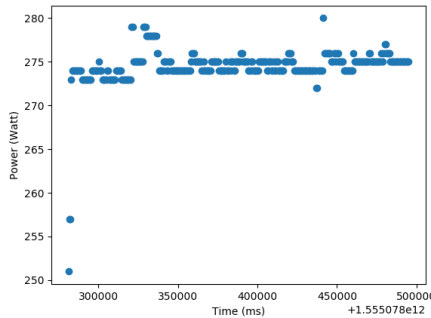


Figure 3.1: An example measurement of the *PrimeOpt* program. Every dots is a measurement from the PDU, with a power value and a timestamp.

3.1.3 Set-up

An overview of the energy measurement set-up is shown in figure 3.2 and we created this set-up together with Stephan Kok. Here we can see that from the DAS-5 a job is sent to one of the energy measurement

nodes. This is done by using the *prun* command and specifying which node to use. The job we send to this node is a bash script that runs all the programs in our data set. In this job we sleep for ten seconds in between the measurements of the programs to give the node time to go back to its idle state. After these ten seconds the measure script on the DAS-4 is started and then the program we want to measure the energy consumption of starts to run on the node. Immediately after the program ends the measure script is stopped. This measure script on the DAS-4 constantly sends a *snmp* message to the PDU and writes the values to an output file. Due to the other nodes constantly being occupied and the long run time, every program was able to run 27 times on *node28* and 22 times on *node29*.

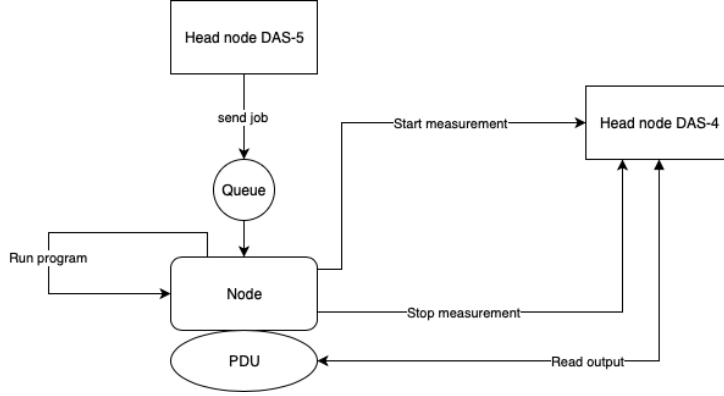


Figure 3.2: The structural overview of how we used the DAS to measure the energy consumption. From the head node of the DAS5 we send jobs to the queue of a certain node we want to measure on. This job will then run on that node when its available. In the job we send a message to the DAS4 to starts its energy measurement, then we run the programs and after that we send a message to the DAS4 to stop the energy measurement. The DAS4 retrieves the energy measurements from the PDU via constant *snmp* requests.

Our script on the DAS-4 uses *snmpwalk* to retrieve the values from the PDU. The time it takes to retrieve the data from the PDU using *snmpwalk* is not constant. This means that we need to take into account that we have some loss of information and in some cases even too few measure points. When there are less than 30 energy measurements for a program, that specific program will run again until it has enough measure points. In table 3.2 the average time between two measure points is shown alongside the standard deviation, the maximum and minimum time difference. Here we see that the average time between two measurements is fast, but the largest time period between two measurements is long.

Statistic	Value
Mean	0.102
Sdv	0.059
Max	11.9
Min	0.067

Table 3.2: Some statistics of the time between two energy measurement in seconds.

3.2 Experimental design

In the end we were left with a large data set of 202 programs with 67 programs that had to run twice, once with and once without flags. Running all these programs takes about eight hours to complete. Unfortunately we were only allowed to use a node on the DAS for 30 minutes to run one script during working hours. This means that we had to run it at night and in the weekend, which limited the amount of data points that were measured. Of course running programs more often would give more accurate results, but this was just too time consuming. We were able to run every single program 27 time on *node28* and 22 times on *node29*.

Some programs were too fast to get a good amount of energy measurement points. To solve this problem those programs were run for multiple times during one measurement and then afterwards their

energy consumption was divided by the amount of times the program was run. Another problem that occurred was that retrieving a measure point takes a variable amount of time. This caused some programs to still not have a good amount of energy measurement points. Therefor we decided to set a minimum amount of 30 measurement points needed in order for the energy measurement to be used.

3.3 Experimental data preparation

To answer the existence questions we need to find programs that have the same functionality. For existence question (1) we need programs that are written in different programming languages and for existence question (2) we need programs that are written in the same programming language. After we find the programs we need to test if they give the correct output. In this section we first look at which programming languages we are going to investigate during this research. After that we look at the source used for collecting the programs followed by a discussion on the programs we found. Then we look at the anomalies we found, after which we discuss the influence the hardware has on the energy consumption. Lastly we look at the correlation between the execution time and the energy consumption.

3.3.1 Programming languages

To find data we first need to decide which programming languages we should choose for testing. We did this by looking into what the most popular and most used programming languages are. For this question it depends on who you ask what answer you get. Therefor four sources were used to determine which programming language to use. The four sources are Git-hub, Indeed, TIOBE and PYPL.

Git-hub is a version control system where multiple programmers collaborate within a project¹. They looked at the amount of pull requests made for that language [19]. The thought was that the language that programmers work a lot with on Git-hub is the most popular, but this is based on only the public repositories. In the first quarter of 2019 the top ten programming languages according to this method are JavaScript, Python, Java, Go, C++, Ruby, PHP, TypeScript, C# and C.

Indeed is a job search site². They looked at the percentage of jobs with a programming language in their name in the tech software category [20]. Thus the more job offering for a programming language the more popular that programming language is. The problem with this method is that job offerings do not show how many people work with a programming language, but only which programming language has a shortage in programmers. Based on data from September 2018 the top ten programming languages according to this method are Java, JavaScript, HTML, Python, C#, C++, XML, Ruby, PHP and Perl.

TIOBE is a software quality company³. They looked at the amount of hits they got when searching "[Language] programming" on a lot of different search engines [21]. There are rules which a search engine needs to comply with for it to be used in the calculations and they also look at what type of hits they find to determine whether or not to use it in the calculations. The pitfall of this method is the favoritism for complex languages. When a language is more difficult to understand, more tutorial pages are needed and more questions about this language will be asked. As of April 2019 the top 10 programming languages according to this method are Java, C, C++, Python, Visual Basic .NET, C#, JavaScript, SQL, PHP and Assembly language.

PYPL index stands for the PopularitY Programming Language index⁴. They look at how many times a language tutorial is searched [22]. This method also has favoritism for complex languages, where programmers need to use the tutorials more often because of the difficulty. Based on data form April 2019 the top ten programming languages according to this method are Python, Java, JavaScript, C#, PHP, C/C++, R, Objective-C, Swift and Matlab.

When languages are in all the top tens of the four sources they were labelled as popular and these are the languages included in this research, thus the languages Java, JavaScript, Python, C#, C++ and

¹<https://github.com>

²<https://www.indeed.nl>

³<https://www.tiobe.com>

⁴<http://pypl.github.io/PYPL.html>

Rank	Git-hub	Indeed	TIOBE	PYPL
1	JavaScript	Java	Java	Python
2	Python	JavaScript	C	Java
3	Java	HTML	C++	JavaScript
4	Go	Python	Python	C#
5	C++	C#	Visual Basic .NET	PHP
6	Ruby	C++	C#	C & C++
7	PHP	XML	JavaScript	R
8	TypeScript	Ruby	SQL	Objective-C
9	C#	PHP	PHP	Swift
10	C	Perl	Assembly	Matlab

Table 3.3: The popularity ranks of the programming languages according to four different sources.

PHP. We also choose to investigate C and Ruby. C because it was in three of the four top tens and it is well-established and known for a high level of control over available resources, thus having a potential impact on the energy consumption. Ruby was in two of the top tens, but also 13th according to TIOBE and 12th in the PYPL index. Thus Ruby was close to be in all the top tens and therefor also chosen to be investigated.

3.3.2 The computer language benchmarks game

The Computer Language Benchmarks Game, also known as *The Great Computer Language Shootout*, compares different programs and languages based on their run time, memory usage, zipped program size and CPU usage [5]. The benchmark features ten different problems with a variety of programs from different programming languages. Everyone can submit a program if it holds to the two requirements: follow the same algorithm and result in the correct output. This is important because we want to compare the way of writing a program and the difference in programming language, but not the difference in algorithm used. For every program featured, the correct output and required steps to compile the program are listed.

The energy consumption of a program is the sum of the CPU, memory and disk energy consumption [7]. To get a good view on what influences the energy consumption, our problems need to be diverse when it comes to these three categories. The problems that we looked into are called Binarytrees, Fannkuchredux, Fasta, Mandelbrot, Nbody, Revcomp and Spectralnorm. An extensive explanation of the problems can be found on the website of *The Computer Language Benchmarks Game* [5]. Below a short description of the problem is listed alongside witch category the problem is in. An overview of category and problem is shown in table 3.4.

- Binarytrees, for this problem a lot of trees are allocated and deallocated in memory and thus is this a memory intensive task.
- Fannkuchredux, for this problem a lot of calculations are don on all permutations and is thus CPU intensive.
- Fasta, for this problem large DNA sequence are created and saved and is therefore memory intensive.
- Mandelbrot, for this problem a large bitmap is saved and thus is it memory intensive.
- Nbody, for this problem a model of the orbit of Jovian planets is crated and is thus CPU intensive.
- Revcomp, for this problem DNA sequences are read line by line, transformed and written to an output file. Therefor the Revcomp problem is disk intensive.
- Spectralnorm, for this problem a lot of calculations are done on a large matrix and is thus CPU intensive.

Category	Problems
CPU	Fannkuchredux, Nbody, Spectralnorm
Memory	Binarytrees, Fasta, Mandelbrot
Disk	Revcomp

Table 3.4: The task intensive categories and the problems that are in that category.

3.3.3 Gathering programs

To be able to compare different programs from an energy consumption perspective they need to have the same functionality. A source for programs that have the same functionality is *the computer language benchmark game* [5]. All the programs for the ten problems in our seven programming languages were downloaded. These programs were then tested to see if they could compile, run and result in the correct output. This was all done on the DAS5 to make sure there were no local dependencies. All the programs that did not compile, or resulted in run time errors or incorrect output were excluded from the data set. There were three problems, Knucleotide, Pidigits and Regexredux, that for different programming languages use a partly different algorithm because of different library implementations. Therefore we excluded these problems from the data set. At the time of conducting the research, no working JavaScript implementations of the Mandelbrot problem were available. Nonetheless this problem will still be used and we will just have an empty spot in our results.

Before running all the programs, some needed to be compiled first. The compilation step of languages that have a separated compilation step were not included in the energy measuring. The reason for this is that a finished program is compiled once and then could be used multiple times. This does mean that the languages JavaScript, Python, PHP and Ruby have a bit of a disadvantage, because they don't have a separate compilation step. The compiler can nowadays do a lot of optimizations of the code. During this research we want to see the result of user decisions on the energy consumption. Therefore besides testing the programs with the compilation flags used on the language benchmark game we also tested with as few flags as possible. This means that we removed all the optimization flags except the ones needed for the compilation. Also the compiler version of the different languages is important. To give a good picture of where we stand today we need to use the most recent stable version and also the most commonly used one. Unfortunately it isn't that easy to update the programming language versions on the DAS. Therefore we used the versions that were already on the DAS and these programming language versions are listed alongside the compiler used in table 3.5.

Languages	Compiler	Version
Java	javac	1.8.0_161
JavaScript	node	6.12.3
Python	python	3.4.5
PHP	php	5.4.16
C	gcc	6.3.0
C++	gcc	6.3.0
C#	mcs	5.10.1.20
Ruby	ruby	2.0.0

Table 3.5: All the different compilers and versions of the programming languages used.

3.3.4 Anomalies

We checked for anomalies using the DBSCAN algorithm discussed in section 2.1.4. An example of the results we got from this approach is shown in figure 3.3. There we can see in the top graph the distribution of measurements for a single program on one of the two nodes on the DAS. The red dots are the measurements DBSCAN labelled as an anomaly. In the bottom graph we see from every measurement point its distance to its fourth nearest neighbour. This is sorted and the red dot is the choice of the input variable of DBSCAN called *eps* and is calculated by finding the first valley.

For *node28* there were a total of 345 anomalies from 7263 measurement points, which is roughly 4.8%. For *node29* there were a total of 305 anomalies from 5918 measurement points, which is roughly 5.1%. Even though there are more anomalies found than programs, not all programs yielded anomalies. All the anomalies found were removed from before further processing the results.

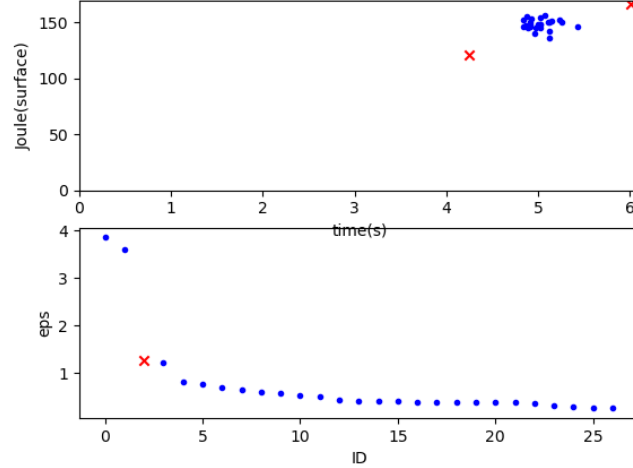


Figure 3.3: In the top graph the distribution of measurements from program java-3 on the Binarytrees problem on *node28* is shown. The bottom graph shows the sorted fourth nearest neighbour graph. The red dots at the top are measurements labelled as an anomaly and at the bottom is the choice for the *eps*-value.

Two clusters

While performing the anomaly detection we saw, for some cases on *node29*, that the resulting measurements for a single program could be divided in two clusters. We looked further into this and found that where we saw two clusters it seemed to follow an ordering based on the moment of measuring. With clusters based on the moment of measuring we mean that in the first cluster all measurements were done before a certain date and in the second cluster after that date. The DBSCAN algorithm divided the measurements of a single program only in seven of 269 cases into two cluster for *node28*, none of these clusters were based on the moment of measuring. For *node29* there were 58 cases of which 53 were cluster based on the moment of measuring. At *node29* for all these 53 programs the null hypothesis that they are from the same distribution can be rejected.

We then, instead of letting a clustering algorithm cluster first and then check if the clusters are based on the moment of measuring, clustered them ourselves based on the moment of measuring. We find that, for *node28* 218 out of 269 programs and for *node29* 41 out of 269 programs, the null hypothesis that the two clusters based on the moment of measuring are from the same population cannot be rejected.

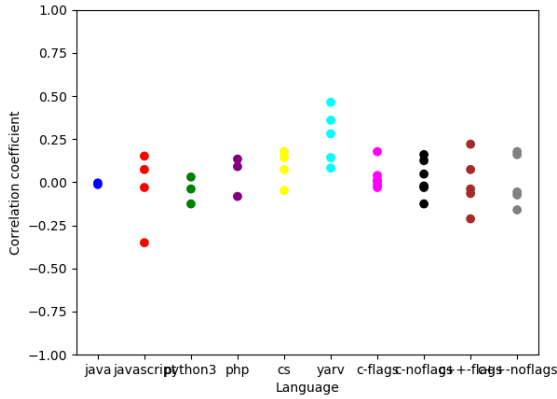
Because of the large amount of programs where the clusters were not from the same population at *node29*, we decided to run another clustering algorithm named k-means as described in section 2.1.5. We set the amount of clusters the k-means algorithm will search for at two, because we wanted to find these two clusters based on the moments of measuring we saw. We then found that there were zero programs that had clusters based on the moment of measuring for *node28* and 63 programs for *node29*.

The 63 programs all rejected the null hypothesis that the two clusters were from the same population. When looking at the programming languages and problems these 63 programs entail, we saw a good representation of the programming languages and different problems. There were no indications of some programming language or problem specific cause for this difference. There still could be some other commonality between these programs, but it is too difficult to find this between the 63 programs. Therefore we left it like this with on *node28* no difference in the moment of measuring and on *node29* only a difference for some of the programs.

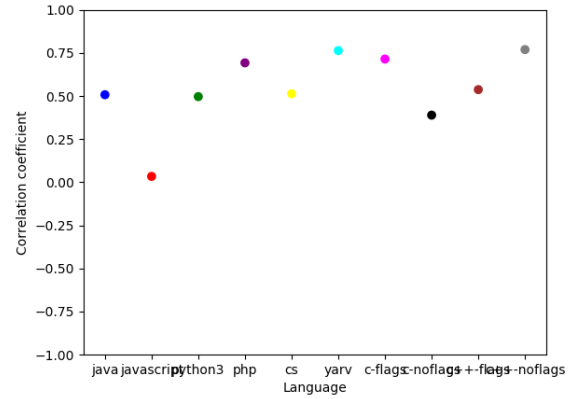
We observe that only a small amount of the energy measurements are labelled as anomalies, 4.8% and 5.1% for *node28* and *node29* respectively. We also see a difference in the behaviour of 63 programs based on the moment of measuring on *node29*.

3.3.5 Hardware

Our measurements were run on two different nodes, which had different hardware specifications. To find out how big of an influence the node is we wanted to compare the measurements between the two nodes for a single program. We first used the one-sided Mann Whitney U test and found that for two programs we could not reject the null hypothesis that they are from the same distribution. For most programs their energy consumption highly depends on which node they were run on. To see if only the height is different, but still follow the same trend we calculated the Kendall correlation scores as described in 2.1.3. This correlation was calculated between the different programs and an example of the results is shown in figure 3.4a. In this figure we have positive and negative numbers, this only shows the direction of the correlation. Thus the correlation number 0.7 and -0.7 have the same correlation strength. All the other graphs of the other problems are shown in appendix C. In all these graphs we see that most scores are low. Because the size of the distribution of the programs are small we decided to check the correlation of all the measurements in one language for one problem. These are also all listed in appendix C and an example is shown here in figure 3.4b. We notice that these Kendall correlation scores are higher.



(a) The Kendall correlation score for every single program that solves the Fasta problem.



(b) The Kendall correlation score for every programming language that solves the Fasta problem.

Figure 3.4

3.3.6 Execution time

In the related work (6) we find some statements about the influence of the execution time on the energy consumption. The research of Verdecchia *et al.* and Acar *et al.* show that the execution time has an influence on the energy consumption, but Pereira *et al.* found that a faster program does not necessarily consume less energy. Because of these statements we wanted to investigate the relationship between the energy consumption and the execution time. We plotted a separate graph for every problem with every measurement from both the nodes, where on the x-axis the execution time is shown and on the y-axis the energy consumption. These graphs are shown in figure 3.5. Looking at these graphs we see that the points seem to follow a line and based on the Kendall tau scores, the execution time and energy consumption have a high correlation according to the Guilford scale. In some of the cases we also see a second line, which shows that a shorter run time does not necessarily result in less energy consumed.

When comparing the measurements from both the different measurement nodes to each other, we find that all but two programs have a difference in their energy consumption based on which node the program runs on. When comparing measurements from specific programs to each other we observe a slight to low correlation and when we compare all programs in one language that solve one problem we

observe a moderate to high correlation.

Finding 1: The hardware has an influence on the energy consumption of a program running on it.

If we look at the correlation between the energy consumption and the run time for every single problem separately, we observe that they all have a high correlation. At three problems we see not one but two trends, which shows that a point with a faster run time does not necessarily consume less energy.

Finding 2: The run time of a program correlates with the energy consumption, but a shorter run time does not always mean that less energy is consumed.

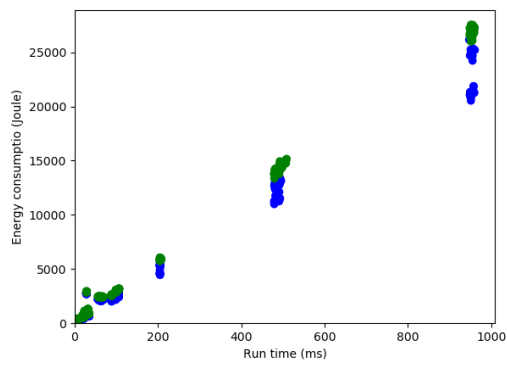
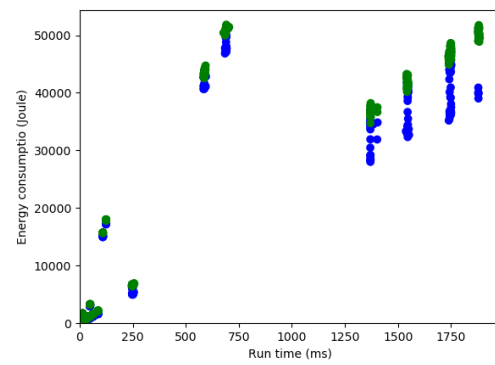
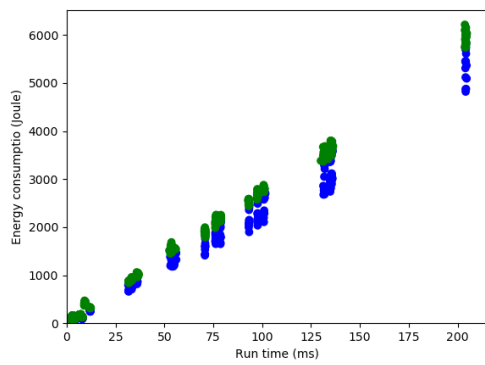
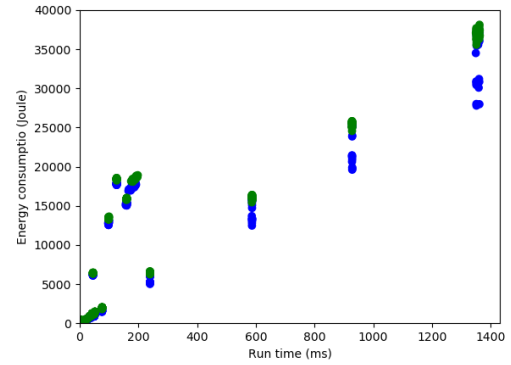
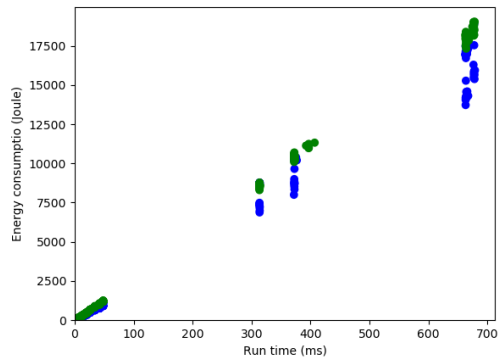
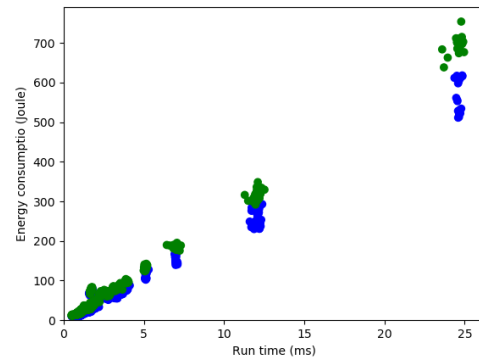
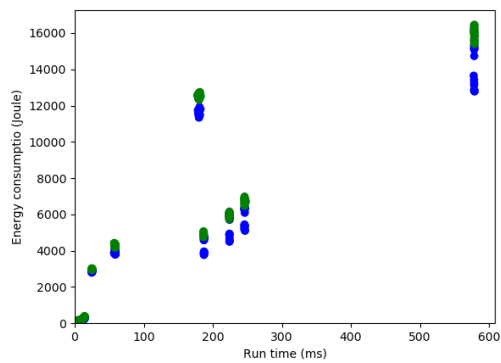

 (a) Binarytrees, $\tau=0.85$

 (b) Fannkuchredux, $\tau=0.72$

 (c) Fasta, $\tau=0.84$

 (d) Mandelbrot, $\tau=0.84$

 (e) Nbody, $\tau=0.89$

 (f) Revcomp, $\tau=0.82$

 (g) Spectralnorm, $\tau=0.85$

Figure 3.5: All the energy measurements and their corresponding run times for every problem. Here the green points are the measurements from *node28* and the blue from *node29*.

Chapter 4

Results

In this chapter we show the results from our experiments. From running all the experiments we got a lot of raw data. This data, alongside all the programs we tested, are on a publicly available Git-hub repository¹. On that Git-hub repository you can also find the programs used to do the measurements, the calculations of various statistical tests and all the intermediate results. The ordering of this chapter will be as followed, first we take a look at the results from the experiments regarding the programming languages. After that we will show the results from the experiments when looking at programs.

4.1 Programming Languages

We now focus on the comparison between programming languages. We select one problem and every measurement of a program that solved the problem. We used these points to create box-plots, an example is shown in figure 4.1. There are a total of seven problems and thus also seven box-plots. The box-plots for all these problems are shown in appendix A. In a box-plot the box is from the first quartile till the third quartile and a line at the median. The dots are measurement points laying outside the fences, which are located at $Q1 - 1.5(Q3 - Q1)$ and $Q3 + 1.5(Q3 - Q1)$. The whiskers show the range of points that are outside the box but inside the fences.

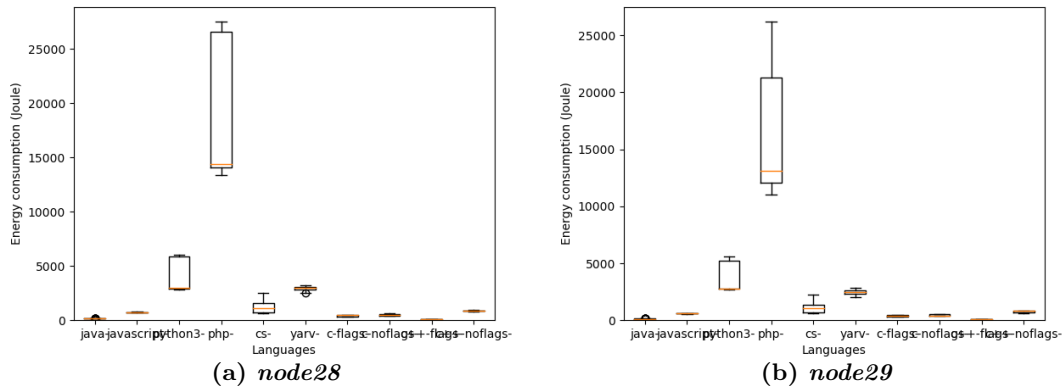


Figure 4.1: The box plots of the different programs in various programming languages for the Binarytrees problem.

In the box-plot we cannot always clearly see which programming language is using less energy. To find out what the relationship is between two programming languages we used twice the one-sided Mann Whitney U test. We calculated this for every language with every other language and because of the fact that if one is smaller the other should be larger this table is inverted with respect to the diagonal zero's, thus a $+$ changes into a $-$. Table 4.1 shows the relationship between programming languages for the Nbody problem. In that table a $+$ means that the programming language on the row is performing

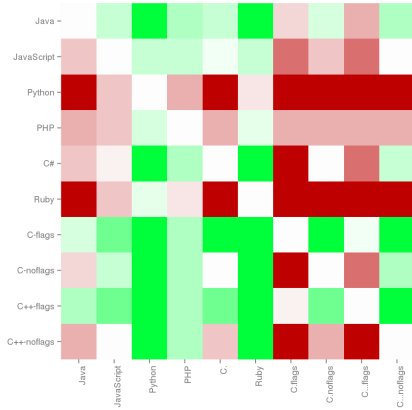
¹<https://github.com/lukaskoedijk/Green-Software>

better than the programming language on the column, i.e. the programming language on the row consumes less energy. The - means the opposite, a 0 means equal and *unknown* means that both one-sided Mann Whitney U tests could not be rejected. All the tables are shown in appendix A for *node28* and *node29*.

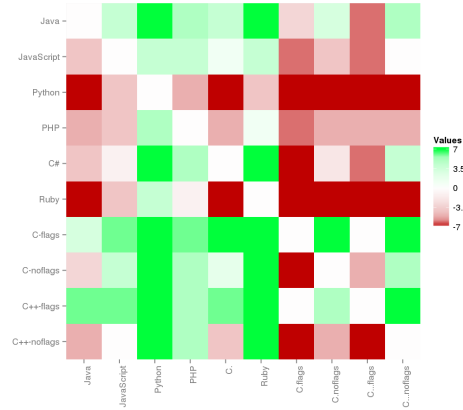
	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	-	+	-	+
JavaScript	-	0	+	+	+	+	-	+	-	+
Python	-	-	0	-	-	-	-	-	-	-
PHP	-	-	+	0	-	-	-	-	-	-
C#	-	-	+	+	0	+	-	+	-	+
Ruby	-	-	+	+	-	0	-	-	-	-
C-flags	+	+	+	+	+	+	0	+	Unknown	+
C-noflags	-	-	+	+	-	+	-	0	-	+
C++-flags	+	+	+	+	+	+	Unknown	+	0	+
C++-noflags	-	-	+	+	-	+	-	-	-	0

Table 4.1: The comparison of the different languages for the Nbody problem on *node28*. A + means that the language on the row consumes less energy than the language on the column, the opposite for -, and the *Unknown* means that we could not reject the null hypothesis.

Using these tables we can see which language is performing better compared to others, but only for a single problem at a time. To give a total overview of problems and programming languages we calculated a score for every combination. One point was rewarded when there was a plus, one point was subtracted when there was a minus and nothing was added or subtracted in the case of a zero or unknown. With the use of these scores we created a heatmap, where green means a high score and red a low. This heatmap is shown in figure 4.2. Here the programming languages with a lot of green in their rows (or a lot of red in their columns), are performing better than most programming languages. We also wanted to see for which programming language combination *unknown* occurred the most. For this we also created a heatmap where a one was added for every *unknown* found. This resulted in figure 4.3.

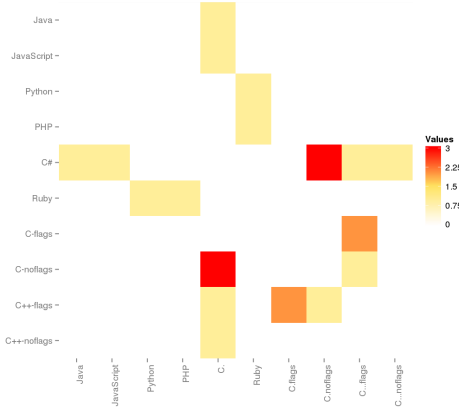


(a) The heatmap of the relationship between the programming languages for all problems measured on *node28* combined. A green box means that the language on the row has a lower energy consumption than the language in the column over all the programs, red is the reverse.

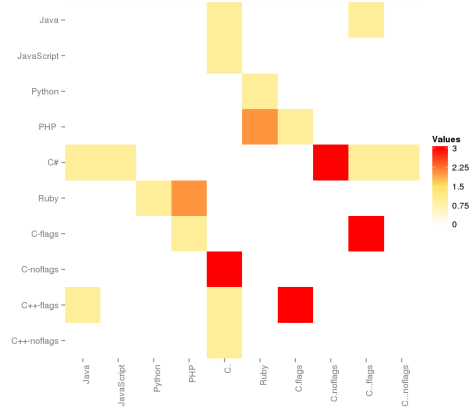


(b) The heatmap of the relationship between the programming languages for all problems measured on *node29* combined. A green box means that the language on the row has a lower energy consumption than the language in the column over all the programs, red is the reverse.

Figure 4.2



(a) The heatmap of the amount of times we could not reject the null hypothesis for the relationship between the programming languages for all problems measured on *node28*.



(b) The heatmap of the amount of times we could not reject the null hypothesis for the relationship between the programming languages for all problems measured on *node29*.

Figure 4.3

4.1.1 Existence question (1)

For the first existence question (1), we need to find if there is a difference in the energy consumption of different programming languages that solve the same problem. In our results we notice that there are 45 programming language combinations and that most have a difference, but in 12 and 15 out of 45 combinations on *node28* and *node29* respectively we could not prove their different.

4.2 Programs

When analyzing the energy consumption of programs written in the same programming language and solving the same problem, we use a box-plot. An example of such a box plot is shown in figure 4.4. Here we cannot always see which program is performing better regarding the energy consumption. Therefore we also used the one-sided Mann Whitney U test twice here. From all possible program combinations we only compare the program combination if the programs are written in the same programming languages and solve the same problem. For some combination we were not able to reject both the Mann Whitney U tests. All these program combinations and their p -values are listed in table B.1 for *node28* and in table B.2 for *node29*. There were a total of 24 program combinations on *node28* and 59 on *node29* where the null hypotheses could not be rejected.

We tried to also use Kolmogorov-Smirnov test to see if we would get fewer cases of uncertainty. This resulted in a total of 33 program combinations for *node28* and 61 for *node29*. However when we look at the overlap we see that there are only 24 program combinations on *node28* and 50 for *node29*. Thus this second test removed the uncertainty of nine program combinations on *node29*.

4.2.1 Code level

Now that we know that the energy consumption of most programs is different, we can investigate this difference on code level. When comparing the different programs we found the differences shown in table 4.2. Here we find for most comparisons multiple differences. If there are multiple differences found we cannot be sure that a specific difference has an influence on the energy consumption and if we only see one difference that does not mean that there are no further. Another problem with multiple differences is that the difference is relative. One way of writing code can be better than a second way of writing code, but worse than a third regarding the energy consumption. When looking at table 4.2 we see that multithreading is a lot of times a difference, but it occurs in both the good and bad programs. This can also be said for the difference between a while and for-loop. We also see that using a library instead of implementing it yourself is beneficial regarding the energy consumption.

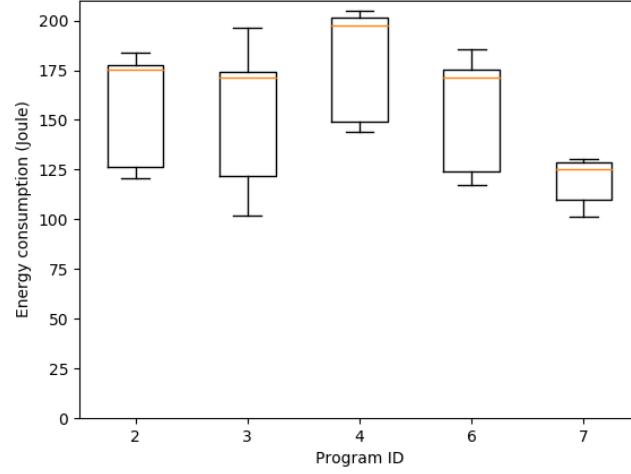


Figure 4.4: The box-plot for the programming language Java comparing the measurements of different programs that solve the Binarytrees problem and are measured on *node28*.

It is really hard to exactly point out what causes the programs to be different regarding their energy consumption, a better approach would be to make a small change in one program when you have two exactly the same programs. To give an example of how this is done we did this for two small changes. The first change we chose was the difference between while and for-loop, as the results show that both occurred in the good and bad sections. For the second change we choose the difference between an if-statement with condition and body on different lines and an if-statement with condition and body on the same line, because we expected this to not have an effect. These four programs are shown in D. When we measure the energy consumption 30 times each for these four programs we get the results shown in figure 4.5. Using the Mann Whitney U test we find that there is a significant difference between the *for-loop* program and the *while-loop* program, but we could not reject the null hypothesis when comparing the *sameline* and *diffline* programs.

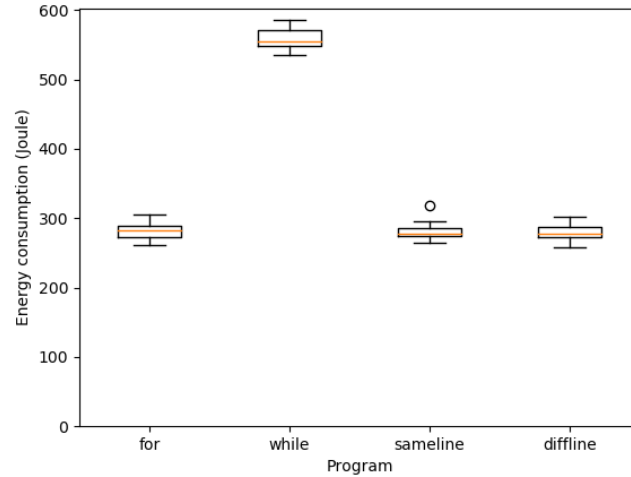


Figure 4.5: The box-plot comparing two programs that have a small difference on code level when solving the 10000th prime number problem. The measurements were done on *node28* and the programs were measured 30 times each.

Better program	Worse program	Better code-level	Worse code-level
python3-1.problem0	python3-2.problem0	Body below the condition of if statement	Condition and body of if statement on same line
		More functions and main function	Less functions and no main function
		Uses map	Doesn't use map
python3-4.problem1	python3-1.problem1	Doesn't use classes	Uses classes
		Uses libraries, math.factorial, itertools.islice and itertools.starmap	Doesn't use libraries
		Uses while-loop	Uses for-loop
python3-3.problem2	python3-2.problem2	Uses no additional function	Uses additional function
		Uses library math.sqrt	Doesn't use library math.sqrt
python3-2.problem4	python3-1.problem4	Less depth in main var	More depth in main var
python3-6.problem5	python3-4.problem5	Uses library itertools.starmap	Doesn't use library itertools.starmap
		Reading data from stdin.buffer	Reading data from stdin
python3-5.problem6	python3-6.problem6	More separate functions	Less separate functions
		Separate for-loop and list comprehension	Nested list comprehension
		Multithreading	Not multithreading
php-2.problem0	php-3.problem0	Uses classes	Doesn't use classes
php-3.problem1	php-1.problem1	Uses for-loop	Uses while true
php-4.problem2	php-2.problem2	Less bigger functions	More smaller functions
		Uses for-each-loop	Use for-loop
		Uses output buffer and implicit flushing	Doesn't use output buffer and implicit flushing
php-3.problem3	php-1.problem3	Doesn't use output buffer and implicit flushing	Uses output buffer and implicit flushing
		Multithreading	Not multithreading
php-1.problem5	php-3.problem5	Uses echo	Uses fwrite
		Not multithreading	Multithreading
php-3.problem6	php-2.problem6	Multithreading	Not multithreading
yarv-4.problem0	yarv-3.problem0	Uses return-statement in body of if-statement	Uses return-statement and then if-statement on same line
		Doesn't use garbage collection	Uses garbage collection
		Uses .each do	Uses while
yarv-1.problem1	yarv-2.problem1	Assigns values to var in condition of if and while statements	Doesn't assign values to var in condition of if and while statements
		Uses while	Uses upto
yarv-3.problem2	yarv-2.problem2	Uses classes	Doesn't use classes
yarv-6.problem3	yarv-1.problem3	Uses classes and functions	Doesn't use classes
		Multithreading	Not multithreading
yarv-3.problem5	yarv-2.problem5	Multithreading	Not multithreading
		Read input via: \$stdin.each_line('>')	Read input via: STDIN.each
yarv-5.problem6	yarv-1.problem6	Uses .each do	Uses for-loop
		Multithreading	Not multithreading

Table 4.2: The differences we found when looking at two programs that had a difference in their energy consumption, solved the same problem and were written in the same programming language. The better program consumes less energy than the worse programs.

4.2.2 Existence question (2)

For the second existence question (2), we need to find if there is a difference in the energy consumption of programs that solve the same problem and are written in the same programming language. In our results we notice that in most cases when we compare programs we find a difference, but in 24 and 50 cases on *node28* and *node29* respectively we cannot prove their different.

Chapter 5

Discussion

In this chapter we discuss the various results from the different experiments. First we will look at the results we got from comparing programming languages to each other. Then we will discuss the results from comparing programs to each other. Finally we will discuss some additional results we found.

5.1 Programming Languages

Comparing the programming languages to each other showed that there was a difference in their energy consumption for most cases, for 12 and 15 out of 45 combinations of programming languages on *node28* and *node29* respectively we could not prove their different. The combination of programming languages for which we could not prove a difference the most amount of times was C-noflags with C on *node28* and C-noflags with C and C++-flags with C-flags on *node29*.

Finding 3: The choice of programming language has an influence on the energy consumption of a program.

When looking at which programming language preforms the best regarding the energy consumption for every problem, we observe that C-flags and C++-flags are the best.

Finding 4: Programming languages C and C++ perform the best regarding the energy consumption.

Because we notice that there is a difference between programming languages that have a pre-compilation phase and those that do not, we also looked at the programming languages without a pre-compilation phase separately. Here we observe that JavaScript performs better than the other programming languages without a pre-compilation phase. Which let us conclude finding 5.

Finding 5: From the programming languages that do not have a pre-compilation phase JavaScript performs the best regarding the energy consumption.

We expected the compilation flags to play a part in the energy consumption of C and C++. The results indicate that the programs written in C and C++ compiled with compilation flags perform better regarding the energy consumption than those obtained without compilation flags.

Finding 6: Programs in the language C and C++ consume more energy when the amount of compilation flags is minimized.

For the first existence question (1), we need to find if there is a difference in the energy consumption of different programming languages that solve the same problem. We find that the energy consumption is influenced by the choice of programming language (finding 3). Thus we find that there is a difference in the energy consumption of different programming languages that solve the same problem.

5.2 Programs

We observe that when comparing programs that solve the same problem and are written in the same programming language most have a difference in their energy consumption, for 24 and 50 programs on *node28* and *node29* respectively we cannot prove they are different.

Finding 7: The way a program is written influences the energy consumption of that program.

When comparing two programs, where we introduced a small difference, we found that a for-loop use less energy than a while-loop. We also compared the difference between having the body and condition of an if-statement on the same or different lines. Here we could not reject the null-hypothesis and thus it looks like the placement of the body does not matter.

Finding 8: A for-loop consumes less energy than a while-loop.

Finding 9: Regarding the energy consumption it does not matter if the body of an if-statement is on the same or on a different line as the condition.

For the second existence question (2), we need to find if there is a difference in the energy consumption of programs that solve the same problem and are written in the same programming language. We find that how a program is written influences the energy consumption of that program (finding 7). Thus we find that there is a difference in the energy consumption of programs that solve the same problem and are written in the same programming language.

5.3 Threats to validity

We used programs listed on *the computer language benchmark game* which gave us a large amount of programs with the same functionality. These programs could be submitted by everyone and were checked to make sure they use the same algorithm and have the same output. For some programming languages there were more submissions than others, there were cases that for a problem there was only one submission for a programming language. When we have only a few programs in a single programming language that solve one problem we cannot be certain that these few programs represent the whole language for this problem, maybe we were just lucky/unlucky and got the best/worst performing programs in that programming language confirm the energy consumption. This uncertainty could be decreased by finding more programs and measuring those for the energy consumption.

The languages that normally have a pre-compilation phase have an advantage because the energy consumption of this pre-compilation was not measured. This approach was chosen because we found that this was a programming language feature and we wanted to mirror the real world use of the programming languages as close as possible. But there are ways of pre-compiling the other languages or improving the following runs by caching. This could improve the energy consumption of the programming languages that do not have a pre-compilation phase, resulting in a different comparison between the languages.

Before the first program and after the last program of a single run we measured the idle state. This idle state was then used to subtract from the measurements to be able to compare the two different nodes. The time between the before and after idle measurement is eight hours. Maybe the idle state fluctuates so much that we were better of calculating the idle state before and after every single program. However we did not choose this option because it would increase the run time even more.

Because of the long run time and the restrictions of only being able to measure outside of working hours we measured each program 27 times on *node28* and 22 times on *node29*. After this for most programs we decreased this number for they were detected as anomalies. This sample size could be on the small side, giving less accurate results.

For *node29* we saw that 63 programs had a division in their measurements based on at which point

in time the measurement was done. This could mean that something could have changed at *node29* during the measurements, which would influence our findings. We did not run into the same problem at *node28*.

Chapter 6

Related work

In table 6.1 we show information of related research to be able to compare the approaches. We found a lot of research that is about reducing the energy consumption of a specific piece of hardware, for example scheduling on a multi-core processor. There is also some research about reducing the energy consumption at a software level and during the decision making process. Some research is done on the energy consumption of software, but their research goal mainly was to estimate the energy consumption. Our research aims to compare the energy consumption of different programming languages and ways of writing code.

Papers	Type of Research	Unit of Analysis	Goal
[4]	Controlled Experiment	Deployment strategies, releases and use case scenarios	Finding optimal energy consumption
[24]	Case Study & Controlled Experiment	HPC bag of task applications	Finding optimal energy consumption
[7]	Case Study	Small functions	Estimating energy consumption
[25]	Case Study & Controlled Experiment	Small functions	Estimating energy consumption
[26]	Case Study	Task on complex micro-architectures	Estimating worst-case energy consumption
[27]	Controlled experiment	I/O application tasks	Estimating energy consumption of I/O tasks
[28]	Empirical Study	Six commonly used refactorings	Finding impact of refactorings on energy consumption
[23]	Controlled experiment	Programming languages	Rank programming languages based on speed, memory usage and energy consumption
[29]	Case Study	Multi-core processor scheduling	Efficient workload partitioning
[30]	Case Study	Cache storage management algorithms	Power aware cache management
[31]	Case Study	Java applications	Framework to automate decision-making support regarding energy consumption
[32]	Case Study	Software process	Two level green software model

Table 6.1: Overview of related research

6.1 Measuring energy consumption

Using the energy consumption we compared programs and programming languages. Sahin *et al.* looked at the impact refactorings have on the energy consumption [28]. They found that refactorings can influence the energy consumption, but the influence of one refactoring is not necessarily the same when used for different programs. Pereira *et al.* tries to find a connection between the speed, memory usage and energy consumption of a programming language [23]. They do this by choosing the fastest implementation of a programming languages, but this might not give a clear view on how a programming languages performs regarding the energy consumption. From our results we see that a faster program does not necessarily have a lower energy consumption. Also we see for example in figure A.3a that the PHP programs with the lowest energy consumption has a lower energy consumption than JavaScript, Python and Ruby, but the PHP program with highest energy consumption is also higher than JavaScript, Python and Ruby.

6.2 Estimating energy consumption

We used a hardware approach for measuring the energy consumption, but there are also some papers that estimate the energy consumption using a software approach. Acar *et al.* implementation is called *Tool to Estimate Energy Consumption* (TEEC) and looks at the CPU, memory and disk [7]. They compare their results to measurements done with a hardware method, but they do not tell how accurate their approach is. Liqat *et al.* estimates the energy consumption by looking at instruction set simulation level [25]. They compare their estimation using small programs to a mathematical equation. Jayaseelan *et al.* estimates the worst-case energy consumption of a task on complex micro-architectures [26]. They test their result against a commonly used benchmark and they find that their values have at most 33% difference with the benchmark. Ferreira da Silva *et al.* looks at estimating the energy consumption of I/O heavy applications [27]. Their estimation algorithm is based on another algorithm that only looks at the CPU utilization and values gotten from a test using a hardware approach.

6.3 Finding optimal energy consumption

There are some papers that try to find the optimal energy consumption given some parameters. Verdecchia *et al.* did this by looking at the impact of releases and deployment strategies [4]. They saw that both the releases and deployment strategies impacted the energy consumption. Therefore they concluded that there is no absolute optimal option for releases and deployment strategies with respect to energy consumption. They also found that the execution time had a bigger impact on the energy consumption than the power consumption, because of the low variability in power consumption. This could be different from this research because they do not subtract the idle power from a measurement.

Filip *et al.* looked at a way of calculating the trade-off between energy consumption and performance with the scheduling of bags of task applications in High Performance Computing (HPC) [24]. A hill climbing algorithm was used and this algorithm uses 12% less resources than an exhausted search. They used a similar energy measurement set-up, also using a PDU and the DAS. There was a minimum of four measure points per application what makes us believe that our minimum of 30 measure points is sufficient.

6.4 Hardware

We looked at improving the software to decrease the energy consumption, but there are also method of decreasing the energy consumption by improving the hardware or hardware management. Zakarya *et al.* proposes an algorithm to makes sure all cores in a multi-core processor have the same workload [29]. Zhu *et al.* tries different algorithms for storage cache management [30].

6.5 Other

There is also other research done regarding the energy consumption that does not fit into the previous groups. Manotas *et al.* implements a framework that automatically optimizes the code of a program given a list of possibilities [31]. All the possibilities are run and measured for the energy consumption and they were able to reduce the energy consumption of a program by 17%. Because the framework needs possible changes beforehand, we do not know if the resulting code is the most energy efficient version.

Instead of decreasing the energy consumption of programs we could also try to decrease the energy consumption of software development. Mahmoud *et al.* makes a two level green software model [32]. The first level is about making the software process more energy efficient. This new process is a hybrid of the sequential, iterative, and agile development processes. The second level is about the role software tools can have on improving the energy efficiency of software.

Chapter 7

Conclusion

We looked at the impact the choice of programming language and way of coding has on the energy consumption. This was measured using a PDU and a distributed node system called DAS. These measurements were compared using the one sided Mann Whitney U test twice and in most cases we could reject one of the two null hypotheses.

When looking into the first research question (1) **Is there a difference in the energy consumption of software projects in different programming languages that have the same functionality?** we find that we cannot make a conclusion for all the programming language combinations in every problem. In the most cases we find a difference, but this difference could not be proven for every programming language combination for every problem. However for some problems on a specific node we can. There is a difference in the energy consumption of different programming languages that solve the problem Binarytrees, Fannkuchredux and Fasta on *node28* and Binarytrees for *node29*.

For the second research question (2) **Is there a difference in the energy consumption of different software projects (using the same programming language) that have the same functionality?** we find that we cannot make a conclusion for all the projects. In most cases there is a difference in software projects that solve the same problem and are written in the same programming language, but not for all of the software projects combinations. We did find that there seemed to be a good representation of programming languages and problems for the cases where we could not prove they were different.

7.1 Future work

In future research we would like to further investigate the impact of the difference on code level on the energy consumption. This could be done by having two identical programs and then slightly change one of them. Measure them both for the energy consumption, proof that they are different and show which one consumes less energy. When this is done for a lot of different small changes we could get a clearer view on how to write code in such a way that we consume less energy.

We would also like to investigate what happens to the energy consumption when we add a pre-compilation phase to the programming languages that normally do not have this phase. It is also interesting to investigate what happens when we do other optimizations like caching of programs.

From our current result we cannot determine if programs written in C are consuming less energy than in C++, or the other way around. Further research could look at these two programming languages and try to determine which one is performing the best regarding the energy consumption.

We also found two trends in some of the problem graphs comparing time with the energy consumption. We first thought that maybe language was the cause, but this did not seem to be true. That is why it would be interesting to research why some problems have two trends.

Acknowledgements

I want to thank my supervisor from the UvA dr. Ana-Maria Oprescu, who has helped me throughout the research by giving feedback. The weekly meetings were both informative as enjoyable. A thank you to my supervisor from KPMG Dennis Stam, who has helped a lot by listening to me, looking at my results and proof reading my thesis. I also would like to thank the whole KPMG Digital Enablement team who have made me feel at home and made my time at the company enjoyable. Thanks to the other interns and their supervisors for listening and asking questions to me during the monthly sessions.

Furthermore I want to thank my little sister Mirjam Koedijk for reading through the thesis and giving pointers on what could be improved.

I also want to thank Stephan Kok for the good collaborating with the energy measurement set-up. Finally I would like to thank drs. Kees Verstoep who has helped me with any question I had about the DAS.

Bibliography

- [1] S. H. Schneider, “The greenhouse effect: Science and policy”, *Science*, vol. 243, no. 4892, pp. 771–781, 1989.
- [2] W. Van Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet, and P. Demeester, “Trends in worldwide ict electricity consumption from 2007 to 2012”, *Computer Communications*, vol. 50, pp. 64–76, 2014.
- [3] J. Koomey, S. Berard, M. Sanchez, and H. Wong, “Implications of historical trends in the electrical efficiency of computing”, *IEEE Annals of the History of Computing*, vol. 33, no. 3, pp. 46–54, 2010.
- [4] R. Verdecchia, G. Procaccianti, I. Malavolta, P. Lago, and J. Koedijk, “Estimating energy impact of software releases and deployment strategies: The kpmg case study”, in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, IEEE Press, 2017, pp. 257–266. DOI: 10.1109/ESEM.2017.39.
- [5] I. Gouy, *The computer language benchmarks game*, Accessed: 26-04-2019. [Online]. Available: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>.
- [6] T. Mens and T. Tourwé, “A survey of software refactoring”, *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.
- [7] H. Acar, G. Alptekin, J.-P. Gelas, and P. Ghodous, “The impact of source code in software on power consumption”, *International Journal of Electronic Business Management*, vol. 14, pp. 42–52, 2016.
- [8] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, “A medium-scale distributed system for computer science research: Infrastructure for the long term”, *Computer*, vol. 49, no. 5, pp. 54–63, 2016.
- [9] N. M. Razali, Y. B. Wah, *et al.*, “Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests”, *Journal of statistical modeling and analytics*, vol. 2, no. 1, pp. 21–33, 2011.
- [10] S. S. Shapiro and M. B. Wilk, “An analysis of variance test for normality (complete samples)”, *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.
- [11] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other”, *The annals of mathematical statistics*, pp. 50–60, 1947.
- [12] N. Nachar *et al.*, “The mann-whitney u: A test for assessing whether two independent samples come from the same distribution”, *Tutorials in quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.
- [13] E. S. Pearson, “Some notes on sampling tests with two variables”, *Biometrika*, pp. 337–360, 1929.
- [14] M. G. Kendall, “A new measure of rank correlation”, *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.
- [15] J. P. Guilford, “Fundamental statistics in psychology and education”, 1942.
- [16] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey”, *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.
- [17] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, “A density-based algorithm for discovering clusters in large spatial databases with noise.”, in *Kdd*, vol. 96, 1996, pp. 226–231.
- [18] J. A. Hartigan, “Clustering algorithms”, 1975.
- [19] C. Zapponi, *Githut 2.0 - a small place to discover languages in github*, Accessed: 16-04-2019, Apr. 2019. [Online]. Available: https://madnight.github.io/githut/#/pull_requests/2019/1.

- [20] A. D. Rayome, *The 10 most in-demand programming languages of 2018*, Accessed: 16-04-2019, Sep. 2018. [Online]. Available: <https://www.techrepublic.com/article/the-10-most-in-demand-programming-languages-of-2018/>.
- [21] TIOBE, *TIOBE index for april 2019*, Accessed: 16-04-2019, Apr. 2019. [Online]. Available: <https://www.tiobe.com/tiobe-index/>.
- [22] P. Carbonnelle, *PYPL popularity of programming language*, Accessed: 16-04-2019, Apr. 2019. [Online]. Available: <http://pypl.github.io/PYPL.html>.
- [23] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, “Energy efficiency across programming languages: How do energy, time, and memory relate?”, in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ACM, 2017, pp. 256–267.
- [24] A. V. Filip, A.-M. Oprescu, S. Costache, and T. Kielmann, “E-bats: Energy-aware scheduling for bag-of-task applications in hpc clusters”, *Parallel Processing Letters*, vol. 25, no. 03, p. 1 541 005, 2015.
- [25] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder, “Energy consumption analysis of programs based on xmos isa-level models”, in *International Symposium on Logic-Based Program Synthesis and Transformation*, Springer, 2013, pp. 72–90.
- [26] R. Jayaseelan, T. Mitra, and X. Li, “Estimating the worst-case energy consumption of embedded software”, in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, IEEE, 2006, pp. 81–90.
- [27] R. Ferreira da Silva, A.-C. Orgerie, H. Casanova, R. Tanaka, E. Deelman, and F. Suter, “Accurately simulating energy consumption of I/O-intensive scientific workflows”, in *2019 International Conference on Computational Science (ICCS)*, 2019.
- [28] C. Sahin, L. Pollock, and J. Clause, “How do code refactorings affect energy usage?”, in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ACM, 2014, p. 36.
- [29] M. Zakarya, N. Dilawar, M. A. Khattak, and M. Hayat, “Energy efficient workload balancing algorithm for real-time tasks over multi-core”, *World Applied Sciences Journal*, vol. 22, no. 10, pp. 1431–1439, 2013.
- [30] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao, “Reducing energy consumption of disk storage using power-aware cache management”, in *Software, IEE Proceedings-*, IEEE, 2004, pp. 118–118.
- [31] I. Manotas, L. Pollock, and J. Clause, “Seeds: A software engineer’s energy-optimization decision support framework”, in *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014, pp. 503–514.
- [32] S. S. Mahmoud and I. Ahmad, “A green model for sustainable software engineering”, *International Journal of Software Engineering and Its Applications*, vol. 7, no. 4, pp. 55–74, 2013.

Appendix A

Programming languages

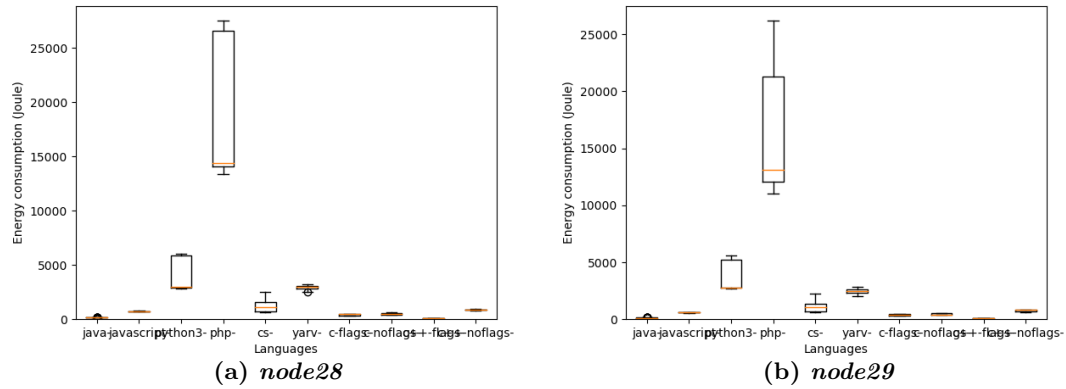


Figure A.1: The box plots of the different programs in various programming languages for the Binarytrees problem.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	+	+	-	+
JavaScript	-	0	+	+	+	+	-	-	-	+
Python	-	-	0	+	-	-	-	-	-	-
PHP	-	-	-	0	-	-	-	-	-	-
C#	-	-	+	+	0	+	-	-	-	-
Ruby	-	-	+	+	-	0	-	-	-	-
C-flags	-	+	+	+	+	+	0	+	-	+
C-noflags	-	+	+	+	+	+	-	0	-	+
C++-flags	+	+	+	+	+	+	+	+	0	+
C++-noflags	-	-	+	+	+	+	-	-	-	0

Table A.1: The comparison of the different languages for the Binarytrees problem on *node28*. A + means that the language on the row consumes less energy than the language on the column, the opposite for -.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	+	+	-	+
JavaScript	-	0	+	+	+	+	-	-	-	+
Python	-	-	0	+	-	-	-	-	-	-
PHP	-	-	-	0	-	-	-	-	-	-
C#	-	-	+	+	0	+	-	-	-	-
Ruby	-	-	+	+	-	0	-	-	-	-
C-flags	-	+	+	+	+	+	0	+	-	+
C-noflags	-	+	+	+	+	+	-	0	-	+
C++-flags	+	+	+	+	+	+	+	+	0	+
C++-noflags	-	-	+	+	+	+	-	-	-	0

Table A.2: The comparison of the different languages for the Binarytrees problem on *node29*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$.

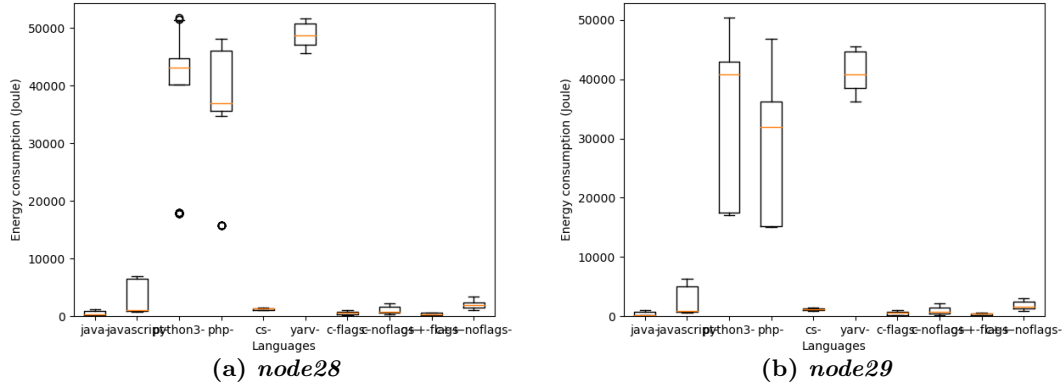


Figure A.2: The box plots of the different programs in various programming languages for the Fannkuchredux problem.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	+	+	+	+
JavaScript	-	0	+	+	+	+	-	-	-	+
Python	-	-	0	-	-	+	-	-	-	-
PHP	-	-	+	0	-	+	-	-	-	-
C#	-	-	+	+	0	+	-	-	-	+
Ruby	-	-	-	-	-	0	-	-	-	-
C-flags	-	+	+	+	+	+	0	+	-	+
C-noflags	-	+	+	+	+	+	-	0	-	+
C++-flags	-	+	+	+	+	+	+	+	0	+
C++-noflags	-	-	+	+	-	+	-	-	-	0

Table A.3: The comparison of the different languages for the Fannkuchredux problem on *node28*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	+	+	Unknown	+
JavaScript	-	0	+	+	+	+	-	-	-	+
Python	-	-	0	-	-	Unknown	-	-	-	-
PHP	-	-	+	0	-	+	-	-	-	-
C#	-	-	+	+	0	+	-	-	-	+
Ruby	-	-	Unknown	-	-	0	-	-	-	-
C-flags	-	+	+	+	+	+	0	+	-	+
C-noflags	-	+	+	+	+	+	-	0	-	+
C++-flags	Unknown	+	+	+	+	+	+	+	0	+
C++-noflags	-	-	+	+	-	+	-	-	-	0

Table A.4: The comparison of the different languages for the Fannkuchredux problem on *node29*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$.

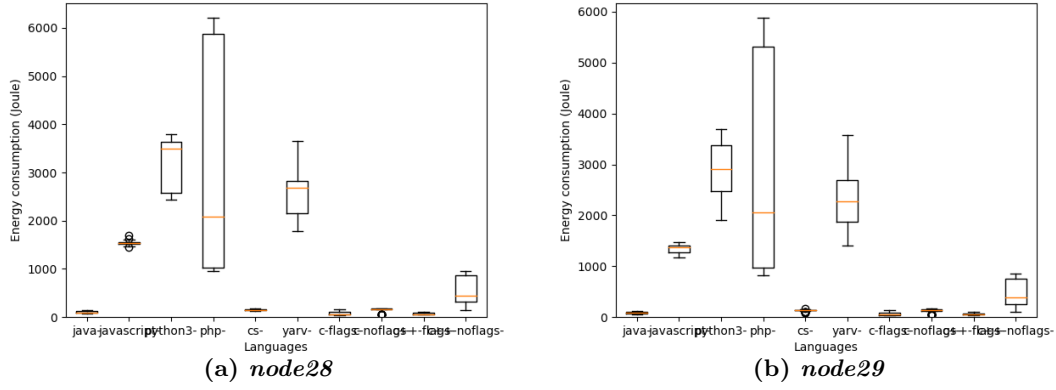


Figure A.3: The box plots of the different programs in various programming languages for the Fasta problem.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	-	+	-	+
JavaScript	-	0	+	+	-	+	-	-	-	-
Python	-	-	0	-	-	-	-	-	-	-
PHP	-	-	+	0	-	+	-	-	-	-
C#	-	+	+	+	0	+	-	+	-	+
Ruby	-	-	+	-	-	0	-	-	-	-
C-flags	+	+	+	+	+	+	0	+	+	+
C-noflags	-	+	+	+	-	+	-	0	-	+
C++-flags	+	+	+	+	+	+	-	+	0	+
C++-noflags	-	+	+	+	-	+	-	-	-	0

Table A.5: The comparison of the different languages for the Fasta problem on *node28*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	-	+	-	+
JavaScript	-	0	+	+	-	+	-	-	-	-
Python	-	-	0	-	-	-	-	-	-	-
PHP	-	-	+	0	-	Unknown	-	-	-	-
C#	-	+	+	+	0	+	-	Unknown	-	+
Ruby	-	-	+	Unknown	-	0	-	-	-	-
C-flags	+	+	+	+	+	+	0	+	Unknown	+
C-noflags	-	+	+	+	Unknown	+	-	0	-	+
C++-flags	+	+	+	+	+	+	Unknown	+	0	+
C++-noflags	-	+	+	+	-	+	-	-	-	0

Table A.6: The comparison of the different languages for the Fasta problem on *node29*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$, and the *Unknown* means that we could not reject the null hypothesis.

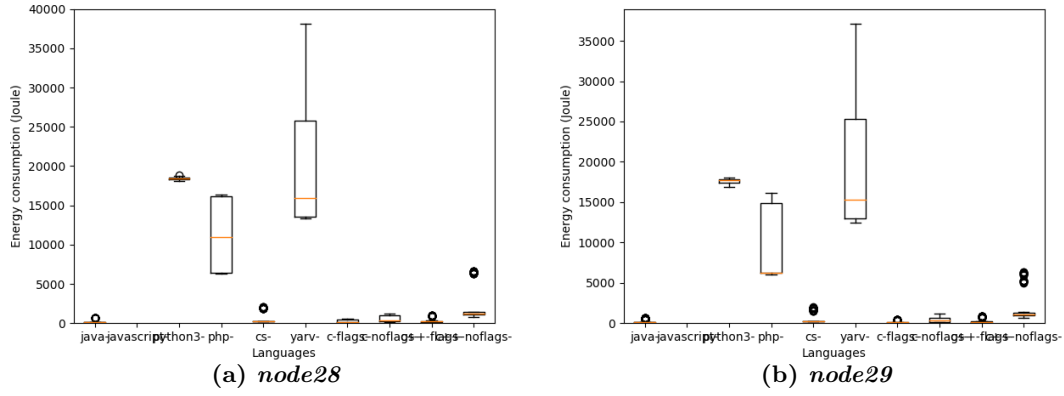


Figure A.4: The box plots of the different programs in various programming languages for the Mandelbrot.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0		+	+	+	+	-	+	-	+
JavaScript										
Python	-		0	-	-	Unknown	-	-	-	-
PHP	-		+	0	-	+	-	-	-	-
C#	-		+	+	0	+	-	Unknown	-	+
Ruby	-		Unknown	-	-	0	-	-	-	-
C-flags	+		+	+	+	+	0	+	Unknown	+
C-noflags	-		+	+	Unknown	+	-	0	-	+
C++-flags	+		+	+	+	+	Unknown	+	0	+
C++-noflags	-		+	+	-	+	-	-	-	0

Table A.7: The comparison of the different languages for the Mandelbrot problem on *node28*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$, and the *Unknown* means that we could not reject the null hypothesis.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0		+	+	+	+	-	+	-	+
JavaScript										
Python	-		0	-	-	-	-	-	-	-
PHP	-		+	0	-	+	-	-	-	-
C#	-		+	+	0	+	-	Unknown	-	+
Ruby	-		+	-	-	0	-	-	-	-
C-flags	+		+	+	+	+	0	+	Unknown	+
C-noflags	-		+	+	Unknown	+	-	0	-	+
C++-flags	+		+	+	+	+	Unknown	+	0	+
C++-noflags	-		+	+	-	+	-	-	-	0

Table A.8: The comparison of the different languages for the Mandelbrot problem on *node29*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$, and the *Unknown* means that we could not reject the null hypothesis.

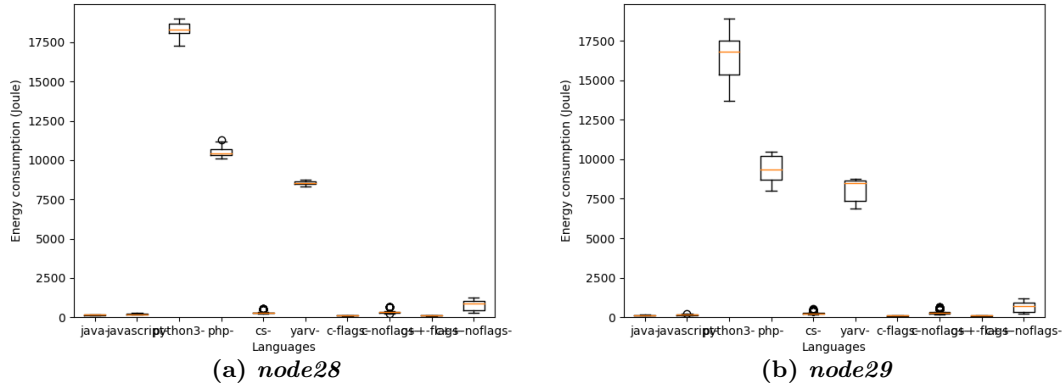


Figure A.5: The box plots of the different programs in various programming languages for the Nbody problem.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	-	+	-	+
JavaScript	-	0	+	+	+	+	-	+	-	+
Python	-	-	0	-	-	-	-	-	-	-
PHP	-	-	+	0	-	-	-	-	-	-
C#	-	-	+	+	0	+	-	+	-	+
Ruby	-	-	+	+	-	0	-	-	-	-
C-flags	+	+	+	+	+	+	0	+	Unknown	+
C-noflags	-	-	+	+	-	+	-	0	-	+
C++-flags	+	+	+	+	+	+	Unknown	+	0	+
C++-noflags	-	-	+	+	-	+	-	-	-	0

Table A.9: The comparison of the different languages for the Nbody problem on *node28*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$, and the *Unknown* means that we could not reject the null hypothesis.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	+	+	+	-	+	-	+
JavaScript	-	0	+	+	+	+	-	+	-	+
Python	-	-	0	-	-	-	-	-	-	-
PHP	-	-	+	0	-	-	-	-	-	-
C#	-	-	+	+	0	+	-	+	-	+
Ruby	-	-	+	+	-	0	-	-	-	-
C-flags	+	+	+	+	+	+	0	+	Unknown	+
C-noflags	-	-	+	+	-	+	-	0	-	+
C++-flags	+	+	+	+	+	+	Unknown	+	0	+
C++-noflags	-	-	+	+	-	+	-	-	-	0

Table A.10: The comparison of the different languages for the Nbody problem on *node29*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$, and the *Unknown* means that we could not reject the null hypothesis.

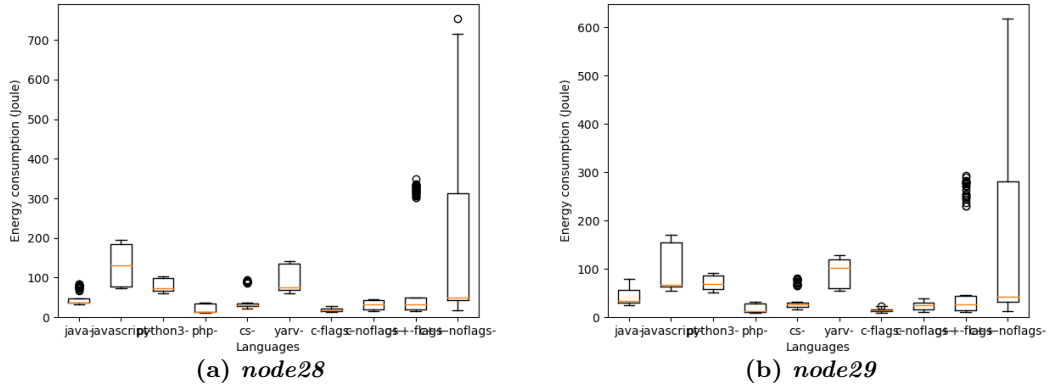


Figure A.6: The box plots of the different programs in various programming languages for the Revcomp problem.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	-	-	+	-	-	-	+
JavaScript	-	0	-	-	-	-	-	-	-	-
Python	-	+	0	-	-	+	-	-	-	-
PHP	+	+	+	0	+	+	+	+	+	+
C#	+	+	+	-	0	+	-	Unknown	Unknown	+
Ruby	-	+	-	-	-	0	-	-	-	-
C-flags	+	+	+	-	+	+	0	+	+	+
C-noflags	+	+	+	-	Unknown	+	-	0	Unknown	+
C++-flags	+	+	+	-	Unknown	+	-	Unknown	0	+
C++-noflags	-	+	+	-	-	+	-	-	-	0

Table A.11: The comparison of the different languages for the Revcomp problem on *node28*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$, and the *Unknown* means that we could not reject the null hypothesis.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	+	+	-	-	+	-	-	-	+
JavaScript	-	0	-	-	-	-	-	-	-	-
Python	-	+	0	-	-	+	-	-	-	-
PHP	+	+	+	0	+	+	Unknown	+	+	+
C#	+	+	+	-	0	+	-	-	Unknown	+
Ruby	-	+	-	-	-	0	-	-	-	-
C-flags	+	+	+	Unknown	+	+	0	+	+	+
C-noflags	+	+	+	-	+	+	-	0	+	+
C++-flags	+	+	+	-	Unknown	+	-	-	0	+
C++-noflags	-	+	+	-	-	+	-	-	-	0

Table A.12: The comparison of the different languages for the Revcomp problem on *node29*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$, and the *Unknown* means that we could not reject the null hypothesis.

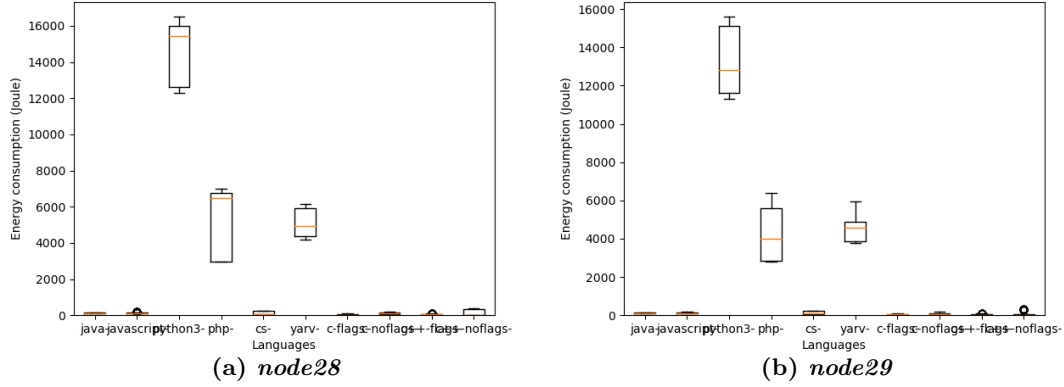


Figure A.7: The box plots of the different programs in various programming languages for the Spectralnorm problem.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	-	+	+	Unknown	+	-	-	-	-
JavaScript	+	0	+	+	Unknown	+	-	-	-	-
Python	-	-	0	-	-	-	-	-	-	-
PHP	-	-	+	0	-	Unknown	-	-	-	-
C#	Unknown	Unknown	+	+	0	+	-	Unknown	-	Unknown
Ruby	-	-	+	Unknown	-	0	-	-	-	-
C-flags	+	+	+	+	+	+	0	+	+	+
C-noflags	+	+	+	+	Unknown	+	-	0	-	-
C++-flags	+	+	+	+	+	+	-	+	0	+
C++-noflags	+	+	+	+	Unknown	+	-	+	-	0

Table A.13: The comparison of the different languages for the Spectralnorm problem on *node28*. A $+$ means that the language on the row consumes less energy than the language on the column, the opposite for $-$, and the *Unknown* means that we could not reject the null hypothesis.

	Java	JavaScript	Python	PHP	C#	Ruby	C-flags	C-noflags	C++-flags	C++-noflags
Java	0	-	+	+	Unknown	+	-	-	-	-
JavaScript	+	0	+	+	Unknown	+	-	-	-	-
Python	-	-	0	-	-	-	-	-	-	-
PHP	-	-	+	0	-	Unknown	-	-	-	-
C#	Unknown	Unknown	+	+	0	+	-	Unknown	-	Unknown
Ruby	-	-	+	Unknown	-	0	-	-	-	-
C-flags	+	+	+	+	+	+	0	+	+	+
C-noflags	+	+	+	+	Unknown	+	-	0	-	-
C++-flags	+	+	+	+	+	+	-	+	0	+
C++-noflags	+	+	+	+	Unknown	+	-	+	-	0

Table A.14: The comparison of the different languages for the Spectralnorm problem on *node29*. A *+* means that the language on the row consumes less energy than the language on the column, the opposite for *-*, and the *Unknown* means that we could not reject the null hypothesis.

Appendix B

Programs

Program 1	Program 2	p-less	p-greater
java-3.problem0	java-6.problem0	0.266	0.740
javascript-1.problem2	javascript-2.problem2	0.829	0.176
javascript-1.problem2	javascript-3.problem2	0.413	0.594
javascript-2.problem2	javascript-3.problem2	0.197	0.808
javascript-1.problem6	javascript-3.problem6	0.532	0.475
javascript-1.problem6	javascript-5.problem6	0.272	0.734
javascript-3.problem6	javascript-5.problem6	0.243	0.763
python3-2.problem3	python3-5.problem3	0.488	0.520
cs-3.problem3	cs-4.problem3	0.088	0.915
cs-3.problem4	cs-5.problem4	0.622	0.385
cs-4.problem4	cs-6.problem4	0.493	0.515
c-noflags-1.problem2	c-noflags-2.problem2	0.882	0.122
c-flags-2.problem4	c-flags-3.problem4	0.230	0.776
c-noflags-1.problem4	c-noflags-6.problem4	0.218	0.787
c-flags-3.problem5	c-flags-6.problem5	0.175	0.830
c-noflags-4.problem5	c-noflags-5.problem5	0.090	0.912
c++-flags-1.problem0	c++-flags-8.problem0	0.571	0.436
c++-noflags-1.problem0	c++-noflags-3.problem0	0.883	0.121
c++-noflags-1.problem0	c++-noflags-8.problem0	0.874	0.130
c++-noflags-3.problem0	c++-noflags-8.problem0	0.453	0.555
c++-flags-1.problem2	c++-flags-2.problem2	0.317	0.690
c++-flags-3.problem4	c++-flags-8.problem4	0.168	0.837
c++-flags-4.problem4	c++-flags-6.problem4	0.225	0.781
c++-noflags-5.problem6	c++-noflags-6.problem6	0.374	0.633

Table B.1: The program combinations from *node28* where the null hypothesis that they are from the same distribution could not be reject for the Mann Whitney U one-sided test less and bigger.

Program 1	Program 2	p-less	p-greater	K-S test
java-2.problem0	java-3.problem0	0.937	0.066	Unknown
java-2.problem0	java-6.problem0	0.820	0.186	Unknown
java-3.problem0	java-6.problem0	0.164	0.842	Unknown
java-2.problem3	java-4.problem3	0.514	0.495	Unknown
java-2.problem4	java-3.problem4	0.864	0.142	Unknown
java-4.problem5	java-5.problem5	0.732	0.276	Unknown
java-5.problem5	java-6.problem5	0.915	0.089	Unknown
javascript-1.problem2	javascript-2.problem2	0.174	0.832	Unknown
javascript-1.problem2	javascript-3.problem2	0.106	0.898	Unknown
javascript-2.problem2	javascript-3.problem2	0.305	0.704	Unknown
javascript-3.problem2	javascript-4.problem2	0.120	0.884	Unknown
javascript-1.problem4	javascript-4.problem4	0.060	0.943	Unknown
javascript-1.problem4	javascript-5.problem4	0.053	0.950	Unknown
javascript-4.problem4	javascript-5.problem4	0.401	0.609	Unknown
javascript-1.problem6	javascript-3.problem6	0.658	0.352	Unknown
javascript-1.problem6	javascript-5.problem6	0.628	0.382	Unknown
javascript-3.problem6	javascript-5.problem6	0.495	0.516	Unknown
cs-1.problem1	cs-3.problem1	0.609	0.400	Rejected
cs-1.problem1	cs-4.problem1	0.606	0.404	Rejected
cs-2.problem1	cs-5.problem1	0.793	0.214	Rejected
cs-2.problem1	cs-6.problem1	0.909	0.095	Rejected
cs-1.problem3	cs-4.problem3	0.756	0.252	Unknown
cs-3.problem3	cs-4.problem3	0.372	0.638	Unknown
cs-3.problem3	cs-6.problem3	0.877	0.128	Rejected
cs-4.problem3	cs-6.problem3	0.855	0.151	Unknown
cs-2.problem4	cs-3.problem4	0.952	0.050	Unknown
cs-3.problem4	cs-5.problem4	0.882	0.123	Unknown
cs-4.problem4	cs-6.problem4	0.430	0.580	Unknown
cs-4.problem4	cs-8.problem4	0.945	0.058	Unknown
cs-6.problem4	cs-8.problem4	0.952	0.050	Unknown
cs-1.problem5	cs-5.problem5	0.907	0.097	Unknown
cs-2.problem5	cs-5.problem5	0.157	0.849	Unknown
yarv-1.problem1	yarv-2.problem1	0.825	0.190	Unknown
c-flags-2.problem2	c-flags-5.problem2	0.292	0.716	Rejected
c-noflags-1.problem2	c-noflags-2.problem2	0.525	0.485	Unknown
c-noflags-1.problem2	c-noflags-5.problem2	0.200	0.806	Unknown
c-noflags-1.problem2	c-noflags-7.problem2	0.582	0.427	Unknown
c-noflags-2.problem2	c-noflags-5.problem2	0.224	0.783	Unknown
c-noflags-2.problem2	c-noflags-7.problem2	0.690	0.319	Unknown
c-noflags-5.problem2	c-noflags-7.problem2	0.861	0.145	Unknown
c-flags-2.problem4	c-flags-3.problem4	0.657	0.352	Unknown
c-flags-2.problem4	c-flags-6.problem4	0.470	0.540	Unknown
c-flags-3.problem4	c-flags-6.problem4	0.311	0.698	Unknown
c-noflags-1.problem4	c-noflags-6.problem4	0.410	0.599	Unknown
c-flags-3.problem5	c-flags-6.problem5	0.278	0.730	Unknown
c-noflags-4.problem5	c-noflags-5.problem5	0.300	0.708	Unknown
c-noflags-4.problem6	c-noflags-5.problem6	0.401	0.609	Rejected
c++-noflags-1.problem0	c++-noflags-3.problem0	0.340	0.669	Unknown
c++-noflags-1.problem0	c++-noflags-8.problem0	0.212	0.795	Unknown
c++-noflags-3.problem0	c++-noflags-8.problem0	0.398	0.612	Unknown
c++-flags-4.problem1	c++-flags-6.problem1	0.244	0.763	Rejected
c++-flags-1.problem2	c++-flags-2.problem2	0.676	0.333	Unknown
c++-flags-1.problem2	c++-flags-6.problem2	0.903	0.102	Rejected
c++-flags-2.problem3	c++-flags-5.problem3	0.790	0.217	Unknown
c++-noflags-8.problem3	c++-noflags-9.problem3	0.272	0.736	Unknown
c++-flags-3.problem4	c++-flags-8.problem4	0.642	0.367	Unknown
c++-flags-4.problem4	c++-flags-6.problem4	0.473	0.538	Unknown
c++-noflags-1.problem4	c++-noflags-6.problem4	0.277	0.731	Unknown
c++-noflags-7.problem4	c++-noflags-8.problem4	0.910	0.094	Unknown

Table B.2: The program combinations from *node29* where the null hypothesis that they are from the same distribution could not be reject for the Mann Whitney U one-sided test less and bigger.

Appendix C

Correlation

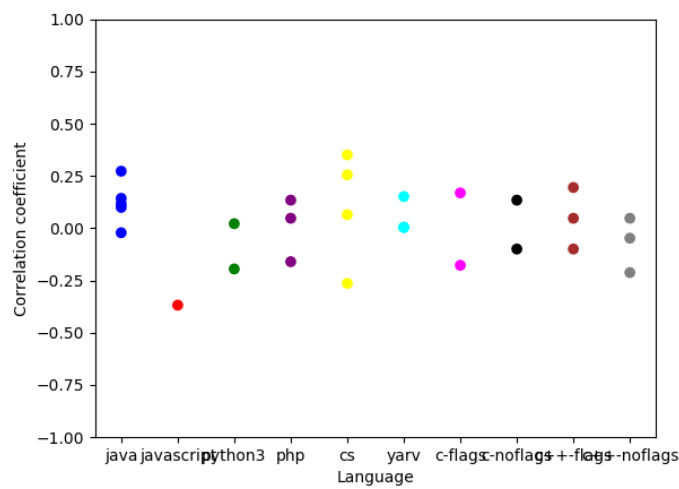


Figure C.1: The Kendall correlation score for every single program that solves the Binarytrees problem.

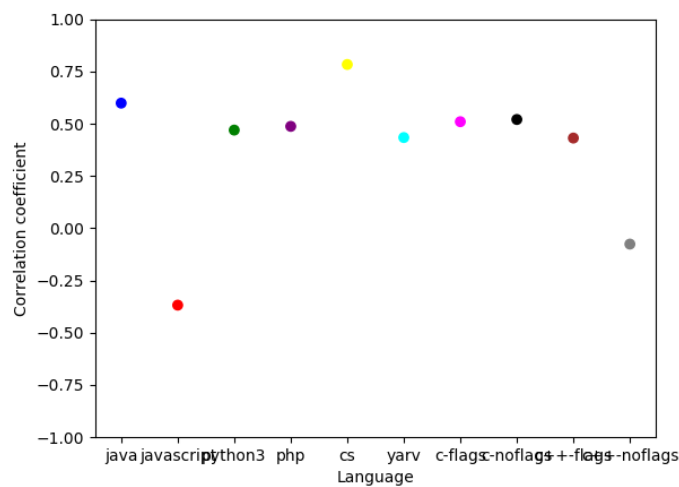


Figure C.2: The Kendall correlation score for every programming language that solves the Binarytrees problem.

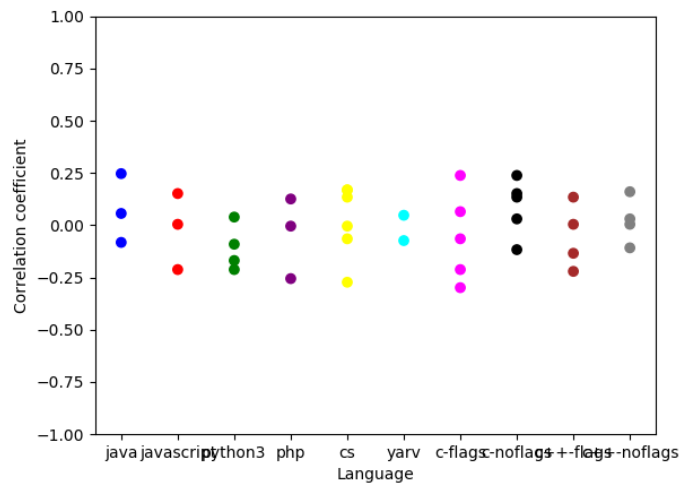


Figure C.3: The Kendall correlation score for every single program that solves the Fannkuchredux problem.

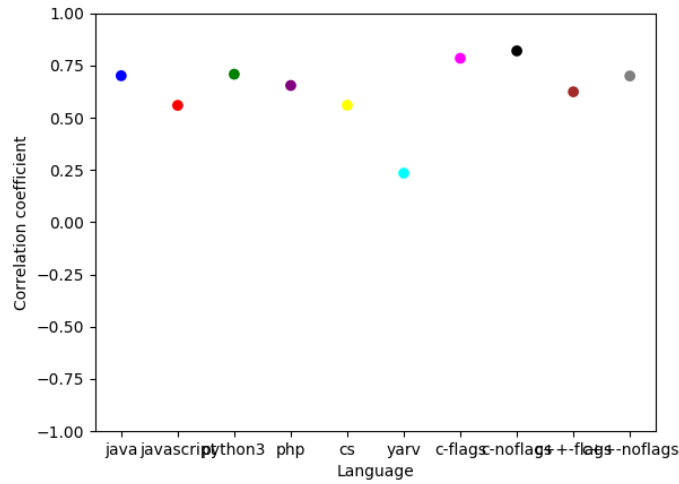


Figure C.4: The Kendall correlation score for every programming language that solves the Fannkuchredux problem.

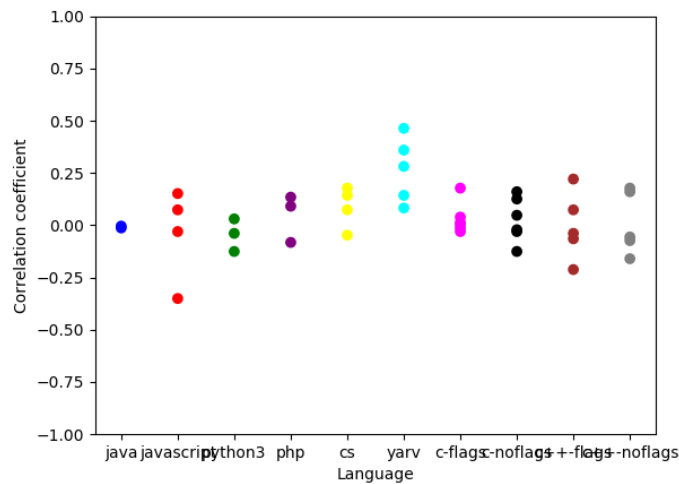


Figure C.5: The Kendall correlation score for every single program that solves the Fasta problem.

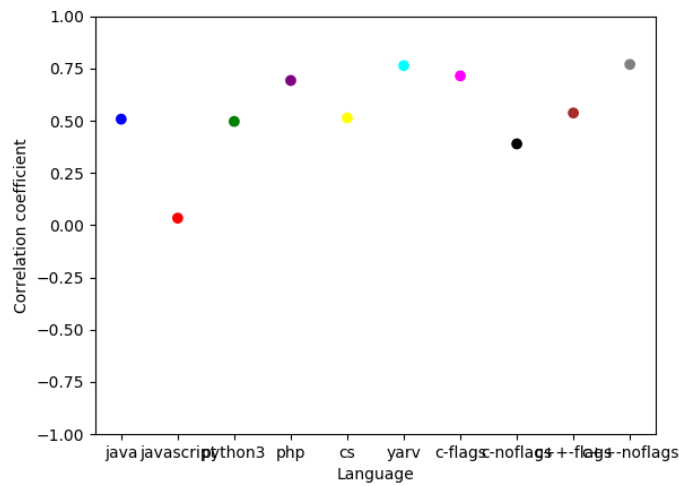


Figure C.6: The Kendall correlation score for every programming language that solves the Fasta problem.

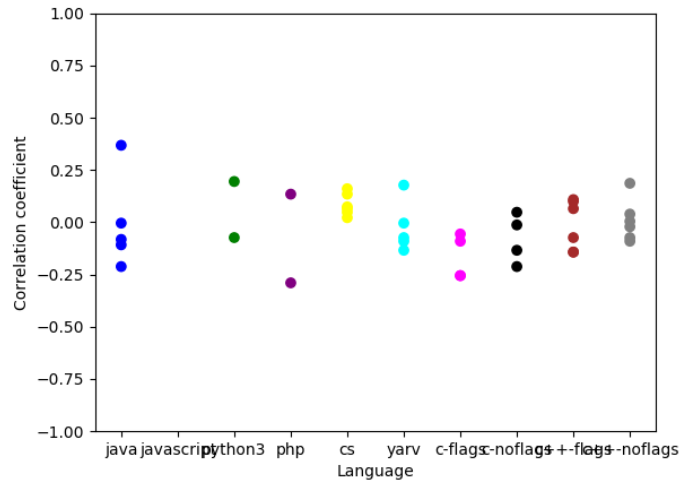


Figure C.7: The Kendall correlation score for every single program that solves the Mandelbrot problem.

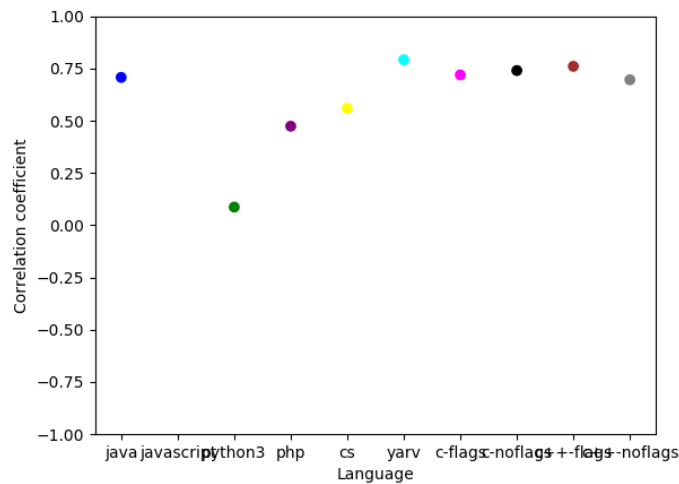


Figure C.8: The Kendall correlation score for every programming language that solves the Mandelbrot problem.

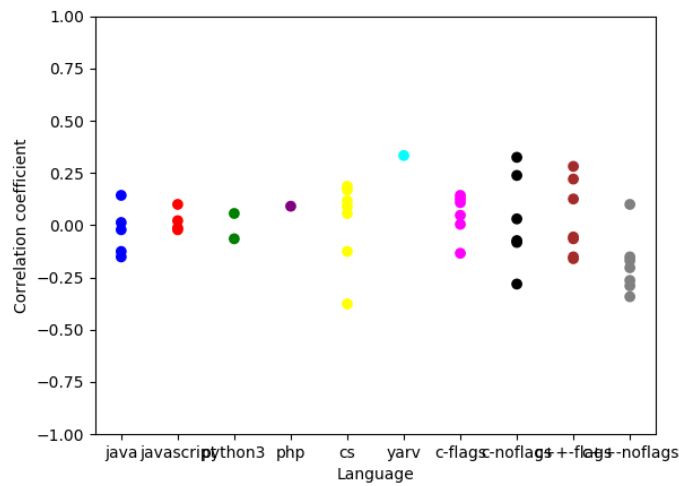


Figure C.9: The Kendall correlation score for every single program that solves the Nbody problem.

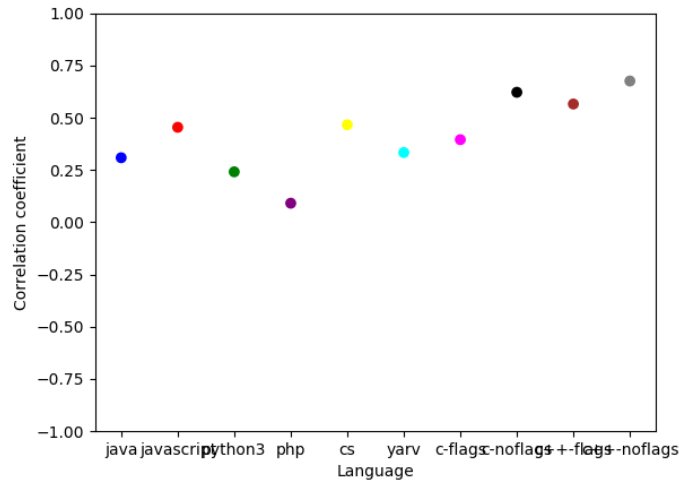


Figure C.10: The Kendall correlation score for every programming language that solves the Nbody problem.

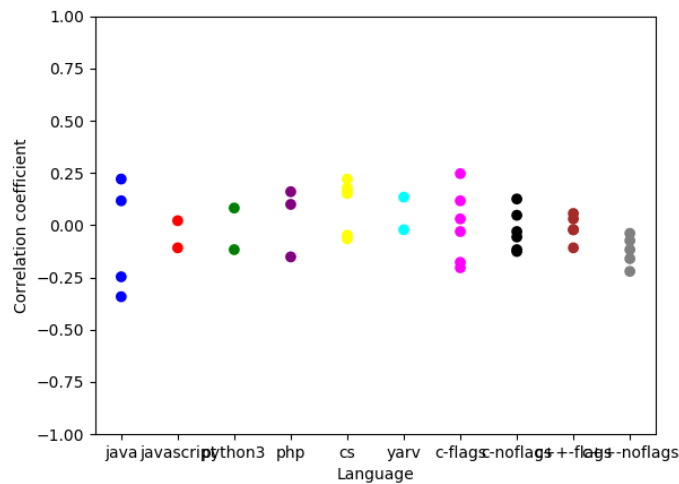


Figure C.11: The Kendall correlation score for every single program that solves the Revcomp problem.

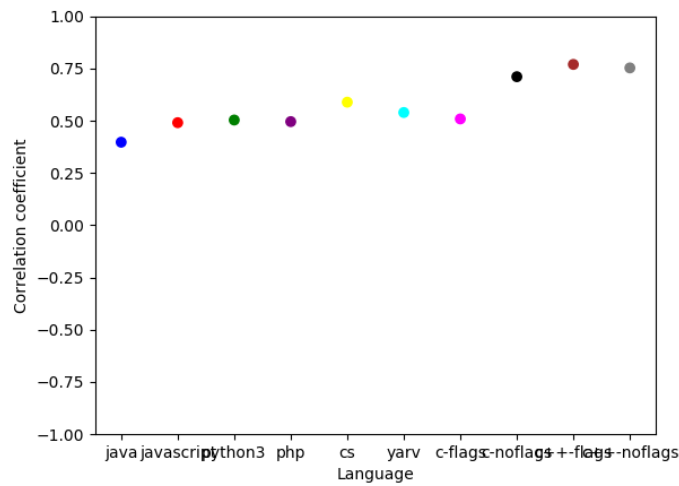


Figure C.12: The Kendall correlation score for every programming language that solves the Revcomp problem.

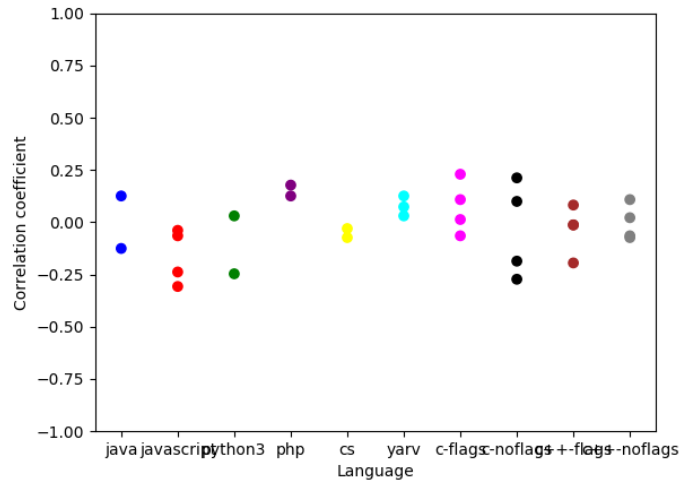


Figure C.13: The Kendall correlation score for every single program that solves the Spectralnorm problem.

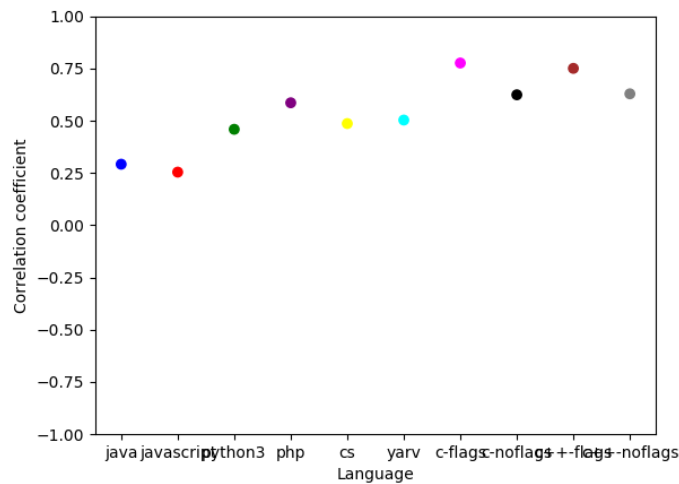


Figure C.14: The Kendall correlation score for every programming language that solves the Spectralnorm problem.

Appendix D

Code-level

```
import sys

n = int(sys.argv[1])

lastPrime = 2
prime = [lastPrime]

while len(prime) < n:
    lastPrime += 1
    possiblePrime = True
    i = 0
    while i < len(prime):
        if lastPrime % prime[i] == 0:
            possiblePrime = False
            break
        i += 1
    if possiblePrime:
        prime.append(lastPrime)

print(prime)
```

Listing D.1: The program that solves the n th prime number using a while loop.

```
import sys

n = int(sys.argv[1])

lastPrime = 2
prime = [lastPrime]

while len(prime) < n:
    lastPrime += 1
    possiblePrime = True
    for i in range(len(prime)):
        if lastPrime % prime[i] == 0:
            possiblePrime = False
            break
    if possiblePrime:
        prime.append(lastPrime)

print(prime)
```

Listing D.2: The program that solves the n th prime number using a for loop.

```
import sys

n = int(sys.argv[1])

lastPrime = 2
prime = [lastPrime]

while len(prime) < n:
    lastPrime += 1
```

```
possiblePrime = True
for i in range(len(prime)):
    if lastPrime % prime[i] == 0: possiblePrime = False; break
    if possiblePrime: prime.append(lastPrime)

print(prime)
```

Listing D.3: The program that solves the n th prime number using if-statements where the body and the condition are on the same line.

```
import sys

n = int(sys.argv[1])

lastPrime = 2
prime = [lastPrime]

while len(prime) < n:
    lastPrime += 1
    possiblePrime = True
    for i in range(len(prime)):
        if lastPrime % prime[i] == 0:
            possiblePrime = False
            break
    if possiblePrime:
        prime.append(lastPrime)

print(prime)
```

Listing D.4: The program that solves the n th prime number using if-statements where the body and the condition are on a different line.