STA2005S - Experimental Design Assignment

Jing Yeh
??University of Cape Town
yhxjin001@myuct.ac.za

Saurav Sathnarayan
??University of Cape Town
yhxjin001@myuct.ac.za

2024-09-13

Abstract

Test

Keywords: key; dictionary; word

1 Introduction

Computation has been playing a major role in human history ever since people began living in cities. The need to calculate taxes motivated the invention of various computing devices that aided such computations, such as the Sumerian abacus, invented in Babylon at around 2500BC[7]. In the 21st century, the capability our digital computing devices have vastly surpassed the capacity of those proto-computers, but so has our need for computational power. Everything in our daily life requires some form of computers: from our phones, cars, to even our refrigerators (side note: initially, Java was actually invented for refrigerators).

However, with large computation capability comes the complexity in the design of these device: to speak plainly, they are damn difficult to use. Computer scientists have therefore invented numerous *programming languages* that allow us to harness the power of these devices more easily.

Eventually, programming languages have become the primary medium for instructing computers to perform our increasingly complex tasks. Understanding which programming languages offer superior execution speed is therefore crucial for developers, especially in domains requiring real-time processing, large-scale data analysis, and other resource-intensive computations. The goal of this experiment is to identify such languages that deliver the fastest execution time.

when calculating a value of π with respect to Leibiniz formula.

$$\sum_{n=0}^{\infty}(-1)^n/(2n+1)$$

This problem will focus on evaluating a selection of popular programming languages, including but not limited to C++, C, R, Python, Java, and Ruby. The evaluation will consider how quickly a value of pi can be calculated by applying leibiniz formula up to 100000000 terms.

1.1 Compiled vs Interpreted Languages

Compiled Language:

In a compiled language, the source code is translated into machine code by a compiler before execution. This machine code, often called an executable, can be run directly by the computer's hardware.

Compiled programs typically run faster since they are already in machine language, which the computer's processor can execute directly.

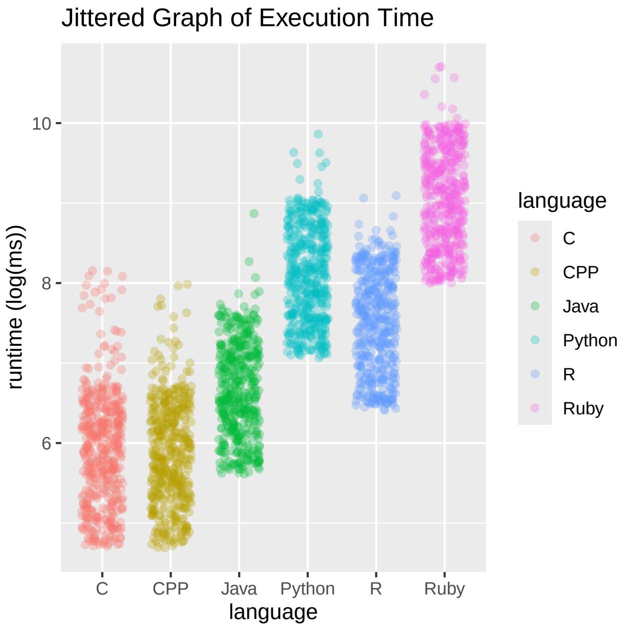
Examples: C, C++, Rust, and Go are examples of compiled languages.

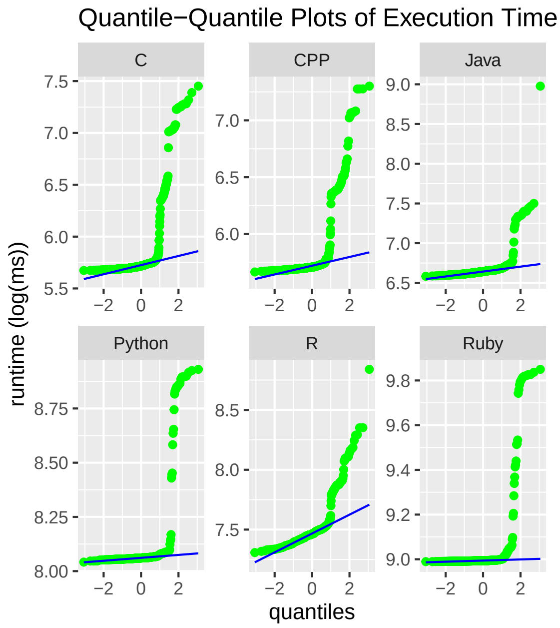
Interpreted Language:
In an interpreted language, the source code is executed line-by-line by an interpreter at runtime. The interpreter reads the code, translates it into machine code, and executes it on the fly.
Interpreted programs generally run slower than compiled ones because the translation happens during execution.
Examples: Python, JavaScript, Ruby, and PHP are examples of interpreted languages.

Key Differences: Compiled languages require a compilation step that produces an executable, while interpreted languages are executed directly by an interpreter.
Compiled languages tend to have better performance due to the pre-compiled nature of the code, whereas interpreted languages are more flexible but slower due to the runtime translation.
Some languages, like Java, use a combination of both techniques, where the code is first compiled into an intermediate form (bytecode) and then interpreted just-in-time (JIT) at runtime.

1.2 A Priori Analysis

Since existing literature on the execution times of programming languages when applying Leibniz's formula is limited, we performed an a priori test to gauge the execution time for the programming languages we planned on experimenting with. We performed 500 approximations of pi ,using the algorithm described in the Method section, for each programming language. We obtained the following jittered graph.





We can observe that C and C++ seem to be the fastest languages, though further analyses are needed. We can also see from the Quantile-Quantile(QQ) plots that the execution times are clearly not normally distributed.

2 Methods

2.1 Setting

This study was mostly conducted at the University of Cape Town, utilising the computers available on campus. We found that there are only 5 different hardware setups available. Thus, to supplement the range of our hardware setups, we also borrowed machines of 2 more hardware setups from our friends.

2.2 Approximation of π

The number π, the ubiquitous and equally mysterious irrational number, has been fascinating the humankind since time immemorial. Mathematicians from 4000 years ago to the present time have devised various methods attempting to get closer to the true value of π. One such method is using Leibniz's formula:

$$4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}...\right)=\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

Leibniz, whom the formula is named after, proved that the series above eventually converges to π. That is:

$$\pi=4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}...\right)=\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

We applied this algorithm in 6 programming languages, including 3 compiled languages: C, C++, Java, and 3 interpreted languages: Python, R, Ruby, up to a billion terms.

2.3 Sources of Variation

Treatments: We have 6 treatment factors, which are the programming languages we applied algorithm to. Each treatment has one level (applying the formula up to 100×10^6 terms). We selected this particular level because it is the largest, practical number of terms we could apply with our hardware setups (For some setups, it may take up to 4 hours to arrive at a single observation), and fewer terms imply larger relative measurement error [7]. We cannot include more levels because in the existing literature, most studies of such kind choose to run all languages on the same machine. However, since we would like to avoid pseudo replication as much as possible we use one machine per observation. The downside of this approach is that we do not have sufficient machines to perform more than one levels.

Blocks: From our a priori analysis, we noticed that the execution times of the 6 programming languages we tested seem to follow the same order on various hardware setups:

$$t(C) \approx t(C++) < t(Java) < t(R) < t(Python) < t(Ruby) \tag{1}$$

Whilst the exact runtimes on machines of the same hardware setups tend to not vary much. This motivates us to block for various hardware setups. We also ensured that the machines are all operating on the same operating system, as we had later found out in the pilot experiment.

2.4 Experimental Units:

As mentioned earlier, we would like to avoid pseudo replication as much as possible. Therefore, we deviated from the tradition of running all programming languages on the same machine, and test only one language per machine. Our experimental units are therefore the individual machines we ran each test on.

2.5 Sampling Procedure

Since existing literature tend to suggest that execution times of programming languages are not normally distributed, we perform a priori tests to confirm that none of our languages has normally distributed runtime. This imposed an issue as it prevented us to apply anova models. To address this, we applied the Central Limit Theorem(CLT) to obtain a normal distribution for the average execution times. We ran the program 15 times per sample for each programming language, and repeated the process 30 times. Applying CLT, it is relatively safe to assume the distribution of sample means is approximately normal [2]. If we assume sample means to be normally distributed, the mean of the distribution of sample means is then an unbiased estimator for the true run time of each programming language[2], which we will take as a single observation. (Note: We arrived at the number 15 through trial and error, and 30 from [8])

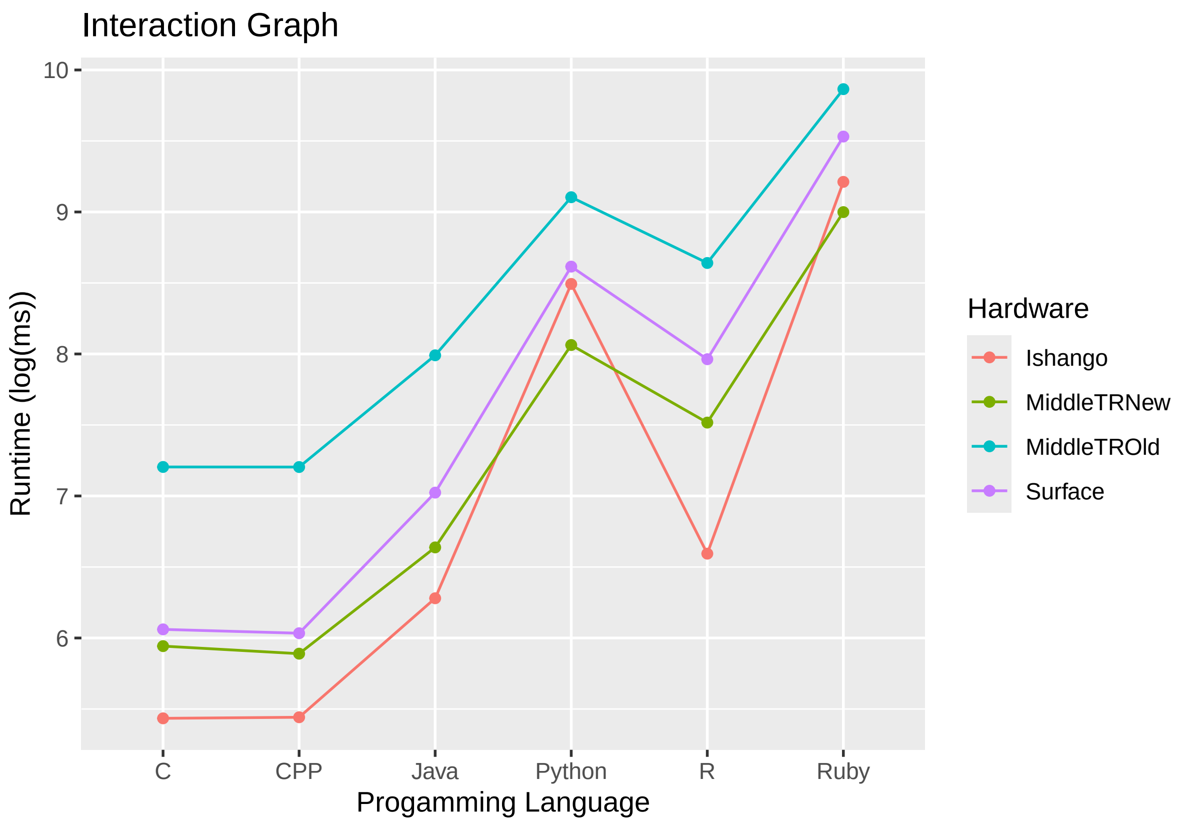
2.6 Randomisation Procedure

We first order the computers belonging to each block from 1 to 6. We then used the random number generator from Python's *random* module to randomly shuffle, and thus producing a permutation of the list, [C, C++, Java, Python, Ruby, R]. The index of each programming language in the permutation would then be paired to the computer with the same assigned number.

2.7 Pilot Experiment

We follow this direction and perform an pilot study on 4 differnt hardware setups to obtain the following data

```
## Warning in read.table(file = file, header = header, sep = sep, quote = quote, :
## incomplete final line found by readTableHeader on 'pilotData.csv'
```



From the data collected, we observed that the results collected from Ishango do not follow the general trends established by the other three setups. The hardware setup in Ishango lab is significantly less advance than MiddleTRNew; yet, most programming languages tend to perform better on the Ishango machine. Secondly, to add to the first observation, not all programming languages perform better on the Ishango machine.

After further investigation, we learned that programming languages perform differently on various operating systems [4]. We hypothesised that this is likely the reason for the deviation, though further studies are needed to confirm this (we lack access to machines with the same hardware setup but run on different operating system). Therefore, we added another constraint for selecting suitable machines: the machines must all run on Windows 10, as these machines are the most widely available.

Besides the observation mentioned above, we can also see that there is relatively little difference in execution times on the other hardware setups. This motivates us to employ an Randomised Block Design(RBD)

2.8 Design

We assume that:

$$e_{ij} \sim \mathcal{N}(0, \sigma^2)$$

We use the following anova model for our response variables:

$$Y_{ij} = \mu + \alpha_i + \beta_j + e_{ij}$$

$$i = 1...a$$

$$j = 1...b$$

With the following constraints:

$$\sum_{i=1}^{a} \alpha_i = \sum_{j=1}^{b} \beta_i = 0$$

Where:

μ	overall mean
α_i	effect of i^{th} treatment
β_j	effect of j^{th} block
e_{ij}	random error of the observation

We also assume that each e_{ij} is independent to each other, which allows us to assume that each Y_{ij} is also independent to each other, and are normally distributed. If there are no blocking and treatment effects, then:

$$Y_{ij} \sim \mathcal{N}(\mu, \sigma^2)$$

Otherwise, if there are blocking and treatment effects, then:

$$Y_{ij} \sim \mathcal{N}(\mu + \alpha_i + \beta_j, \sigma^2)$$

3 Results

We performed the experiment described above on 7 different hardware setups and applied all 6 treatments. The table for data and details of each hardware setups can be found in the appendix. The Analysis of Variance (ANOVA) table is shown below:

	Df	Sum sq	Mean sq	F value	Pr(>F)
Language	5	58.1057	11.6211	1032.9348	< 0.0001
Hardware	6	5.5176	0.9196	81.7379	< 0.0001
Residuals	30	0.3375	0.0113		

4 Discussion

You can cross-reference sections and subsections as follows: Section **??** and Section **??**.

Note: the last section in the document will be used as the section title for the bibliography.

5 References

6 Appendix

PC Specificiations

	CPU	MEMORY	OPERATING SYST
Ishango PC	9th Gen Intel® Core™ i3-9100	8,0 GB	Ubuntu 22
MidddleTROld	9th Gen Intel(R) Core(TM) i5-9500 CPU	8,0 GB	Windows
MiddleTRNew	12th Gen Intel(R) Core(TM) i5-13400	16,0 GB	Windows
ScilabB	12th Gen Intel(R) Core(TM) i5-12400	16,0 GB	Windows
Surface	9th Gen Intel(R) Core(TM) i5-8250 CPU	16,0 GB	Windows
ASUS laptop	5th Gen Intel(R) Core(TM) i7-5500U CPU	6,0 GB	Windows

Acknowledgements

This is an acknowledgement.

It consists of two paragraphs.