STA2005S - Experimental Design Assignment

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

Saurav Sathnarayan
??University of Cape Town'
sthsau01001@myuct.ac.za

2024-09-14

Abstract

Test

Keywords: key; dictionary; word

1 Introduction

Computation has played 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 of 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 invented for refrigerators).

However, with large computation capability comes complexity in the design of these devices: 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.

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 using the algorithm for each programming language and obtained the following jittered graph.

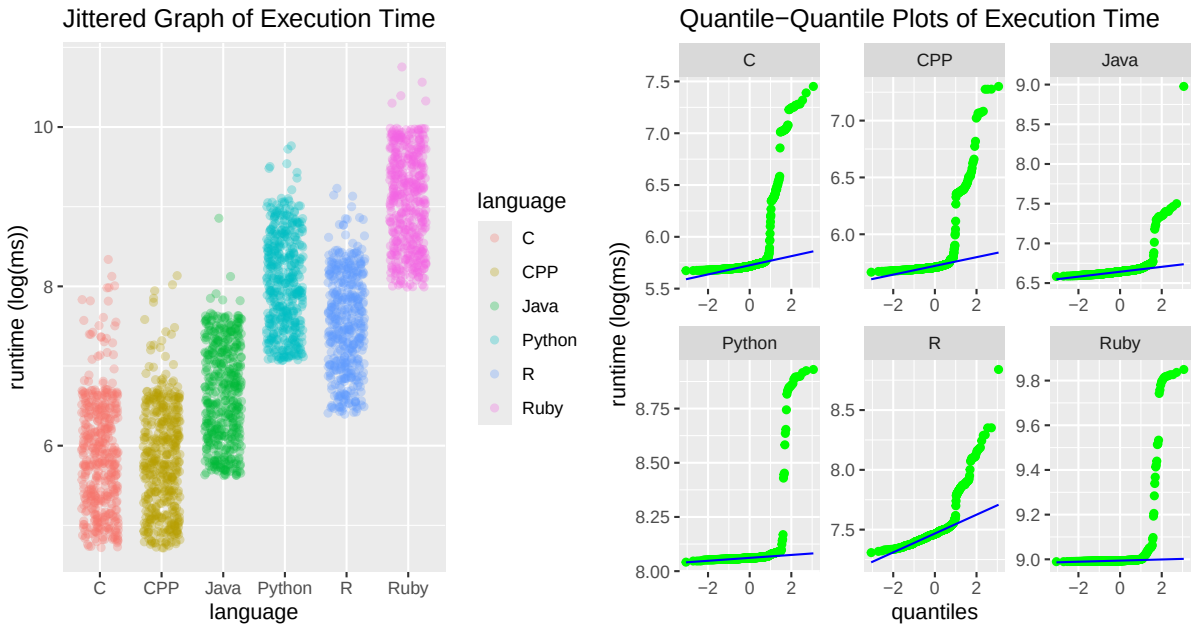

Figure 1: Runtimes of Programming Languages of Interest When Applying Leibniz's Formula up to 100 million terms

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(Q-Q) plots that the execution times are clearly not normally distributed.

2 Reference example

Here are two sample references:. Bibliography will appear at the end of the document.

3 Methods

3.1 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.

3.2 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.

3.3 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 is 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])

3.4 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.

3.5 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.

3.6 Randomisation Procedure

We first ordered 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.

3.7 Planned Comparisons

We planned to conduct pairwise comparisons on all treatments. That is, a total of $\binom{6}{2=15}$ comparisons. It would also serve our objective by comparing the efficiency of compiled languages (C, C++, Java) and interpreted languages (Python, Ruby, R), as the latter are oftentimes easier to program with.

3.8 Pilot Experiment

We followed this direction and performed an pilot study to obtain the following execution times of programming languages on 4 hardware setups

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

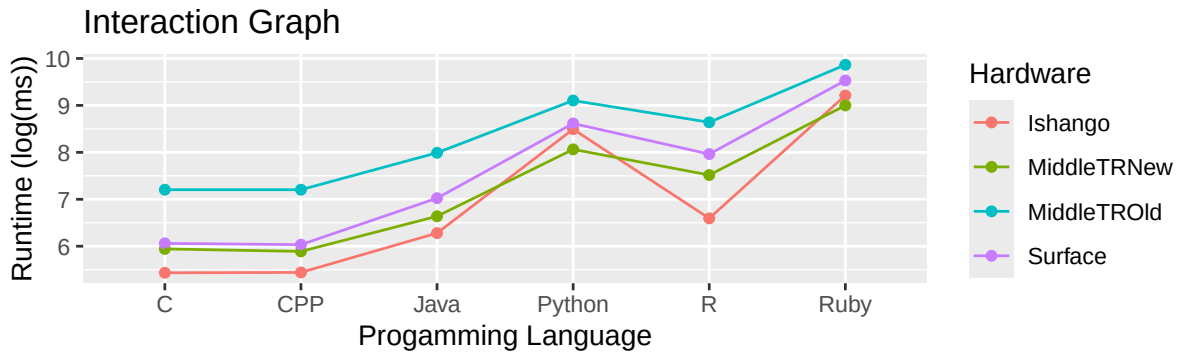

Figure 2: Interaction Graph of Programming Languages and Blocks

From the data collected, we observed that the results collected from Ishango do not follow the general trends established by the other three setups. Firstly, 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. ## 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:

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

Below is the layout of the design

PC Spec	**Treatment**					
Ishango	Java	C	Python	Ruby	C++	R
MidddleTROld	R	C++	Python	C	Ruby	Java
MiddleTRNew	Python	R	Java	C	C++	Ruby
ScilabB	Python	Java	Ruby	R	C	C++
ASUS (i7-5500u)	C	C++	R	Ruby	Python	Java
ScilabD	C	Python	Java	R	Ruby	C++
LT	C	Java	R	C++	Ruby	Python
HP (i5-7200)	R	Python	Ruby	C++	Java	C

Figure 3: Diagram of the Design

4 Results

We performed the experiment described above on 7 different hardware setups and applied all 6 treatments. Detailed tables for data and for each hardware setup 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	65.2112	13.0422	285.9459	< 0.0001
Hardware	6	6.5717	1.0953	24.0136	< 0.0001
Residuals	30	1.3683	0.0456		

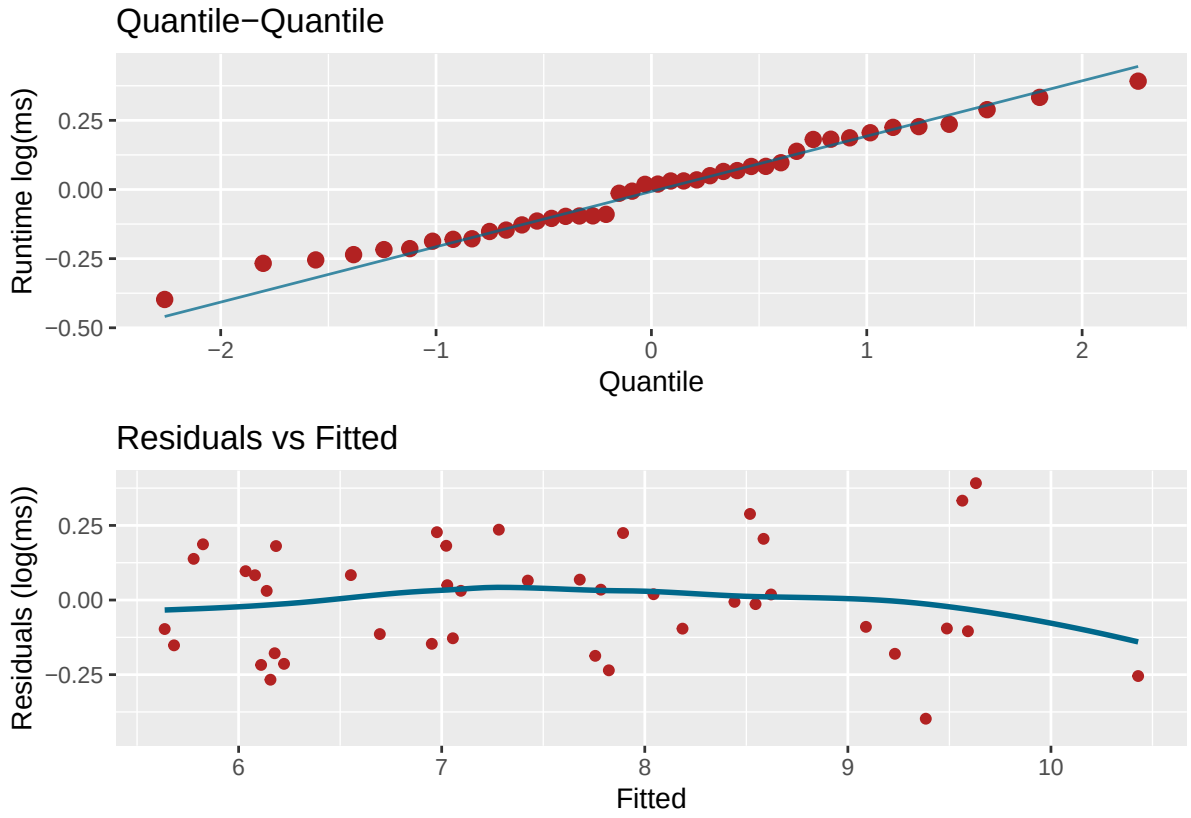

Figure 4: The ANOVA table and Relavent Plots

4.1 Verifying Model

We verified our model by observing the Fitted vs Residuals graph: the residuals seem to spread out haphazardly, with no obvious patterns to be discerned. This suggests homoscedasticity [7]. Also, the Quantile-Quantile graph offers a fairly good fit, hinting that residuals are normally distributed.

We further verified our assumptions by performing Shapiro Wilk test on the residuals. We obtained a fairly large p-value of 0.7521, indicating that we have little evidence for

residuals not being normally distributed. Further, we also got that the mean of the residuals is approximately zero ($< 10^{-15}$), indicating that it is sensible to assume $e_{ij} \sim \mathcal{N}(0, \sigma^2)$

4.2 Pairwise Comparisons

C	CPP	Java	Python	Ruby	R
2.7381	2.72	3.118	3.7273	4.198	3.4101

Given that we know our response variables are normally distributed, we used Tukey's Method to construct 95% confidence intervals that compare the execution time of every possible pair of our 6 programming languages. Sine we know the estimate of the standard deviation of residuals, we can estimate the standard error (SE) of our treatment means, and then find Tukey's Honestly Significant Difference(HSD):

$$\begin{aligned} SE &= \frac{s}{\sqrt{b}} \\ &= \frac{0.0456}{\sqrt{13}} \end{aligned}$$

$$\begin{aligned} HSD &= q_{6,30}^{0.05} \frac{s}{\sqrt{b}} \\ &= 4.3015 \times \frac{\sqrt{0.0456}}{\sqrt{13}} \\ &= 0.2535 \end{aligned}$$

DIFFERENCE	C	CPP	Java	Python	Ruby	R
C		0.0182	0.3799	0.9891	1.4598	0.6719
CPP			0.3981	1.0073	1.4780	0.6901
Java				0.6092	1.0799	0.2921
Python					0.4707	-0.3172
Ruby						-0.7879
R						

Figure 5: Results of Tukey's Honestly Significance Difference Test. Values Greater than HSD are Highlighted in Blue

We can see, with 95% confidence, that from the table of Tukey's HSD tests that C and C++ have the fastest execution times than others when performing the algorithm up to 100×10^6 terms. The results can be summarised as follows:

$$t(C) \approx t(C++) < t(Java) < t(R) < t(Python) < t(Ruby)$$

The p-values for relatively less significant results (p-value > 0.005) are given below:
- R vs Python: 0.0173
- R vs Java: 0.0080

4.3 Compiled vs Interpreted

We were also interested in comparing the execution times of compiled languages and interpreted languages. The contrast, L, can be stated as follows:

$$L = \frac{1}{3}(\mu_C + \mu_{C++} + \mu_{Java}) - \frac{1}{3}(\mu_{Python} + \mu_{Ruby} + \mu_R)$$

We can estimate the values relevant to the contrast as follows:

$$\begin{aligned}
\hat{L} &= \frac{1}{3}(\bar{y}_C + \bar{y}_{C++} + \bar{y}_{Java}) - \frac{1}{3}(\bar{y}_{Python} + \bar{y}_{Ruby} + \bar{y}_R) \\
&= -0.9198 \\
\widehat{Var(\hat{L})} &= s^2 \sum_{i=1}^{6} \frac{{h_i}^2}{n} \\
&= 0.0456 \sum_{i=1}^{6} \frac{\frac{1}{9}}{7} \\
&= 0.004343 \\
SE(\hat{L}) &= \sqrt{\widehat{Var(\hat{L})}} \\
&= \sqrt{0.004343} \\
&= 0.0659
\end{aligned}$$

We first note that $t^{30}_{0.025} = 2.0422$. We will then construct a 95% confidence interval as follows:

$$\hat{L} \pm SE(\hat{L}) \times t^{30}_{0.025}$$
$$-0.9198 \pm 0.0659 \times 2.0422$$

Thus, the 95% Confidence interval is as follows:

$$95\%C.I. = (-1.0544, -0.7852)$$

The mean log execution time of compiled languages is estimated to be -1.05 log(ms) to -0.79 log(ms) shorter than the mean log execution time of interpreted languages at 95% confidence. We therefore have strong evidence to suggests that compiled languages have shorter execution times when applying Leibniz's formula up to a very large term. # References

5 Appendix

PC Specificiations

% latex table generated in R 4.4.1 by xtable 1.8-4 package % Sat Sep 14 16:46:43 2024

	PC	CPU	RAM	OS
1	Ishango PC	9th Gen Intel® Core™ i3-9100	8.0 GB	Ubuntu 22.04
2	MidddleTROld	9th Gen Intel(R) Core(TM) i5-9500 CPU	8.0 GB	Windows 10
3	MiddleTRNew	12th Gen Intel(R) Core(TM) i5-13400	16.0 GB	Windows 10
4	ScilabB	12th Gen Intel(R) Core(TM) i5-12400	16.0 GB	Windows 10
5	Surface	9th Gen Intel(R) Core(TM) i5-8250 CPU	16.0 GB	Windows 10
6	ASUS (i7-5500u)	5th Gen Intel(R) Core(TM) i7-5500U CPU	6.0 GB	Windows 10
7	ScilabD	10th Gen Intel(R) Core(TM) i5-10500	16.0 GB	Windows 10
8	LT	9th Gen Intel(R) Core(TM) i5-9400f	8.0 GB	Windows 10
9	HP (i5-7200)	7th Gen Intel(R) Core(TM) i5-7200	8.0 GB	Windows 10

Table 3: Table of Pcs used and their respective specifications

% latex table generated in R 4.4.1 by xtable 1.8-4 package % Sat Sep 14 16:46:43 2024

	Hardware	C	CPP	Java	Python	Ruby	R
1	MiddleTROld	3.13	3.13	3.53	3.90	4.42	3.75
2	MiddleTRNew	2.40	2.41	2.88	3.50	3.91	3.26
3	ScilabB	2.61	2.57	2.86	3.51	3.93	3.25
4	ASUS(i5)	2.61	2.61	3.09	3.82	4.35	3.30
5	HP(i5-7200)	2.56	2.56	3.07	3.82	4.30	3.29
6	ScilabD	2.68	2.66	2.96	3.66	4.08	3.37
7	LT	2.76	2.68	3.01	3.71	4.12	3.40

Table 4: Table of data used for analysis

Acknowledgements

This is an acknowledgement.

It consists of two paragraphs.