STA2005S - Experimental Design Assignment

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

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

2024-09-13

Abstract

Test

Keywords: key; dictionary; word

1 Introduction

The goal of this experiment is to identify the programming language that delivers the fastest execution time when calculating a value of π with respect to Leibiniz formula.

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

With the increasing demand for high-performance applications, understanding which programming languages offer superior speed in terms of execution is crucial for developers, especially in domains requiring real-time processing, large-scale data analysis, and resource-intensive computations.
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.

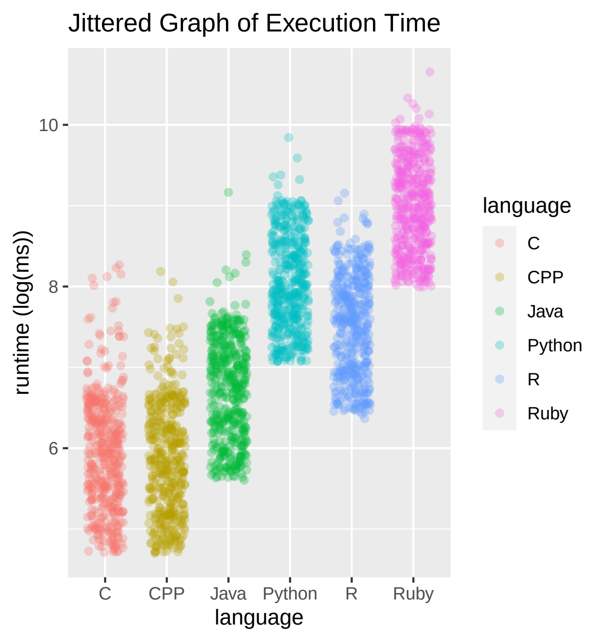
Interpreted Language:
In an interpreted language, the source code is executed line-by-line by an interpreter at runtime. The interpreter reads the code, translates it into machine code, and executes it on the fly.
Interpreted programs generally run slower than compiled ones because the translation happens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

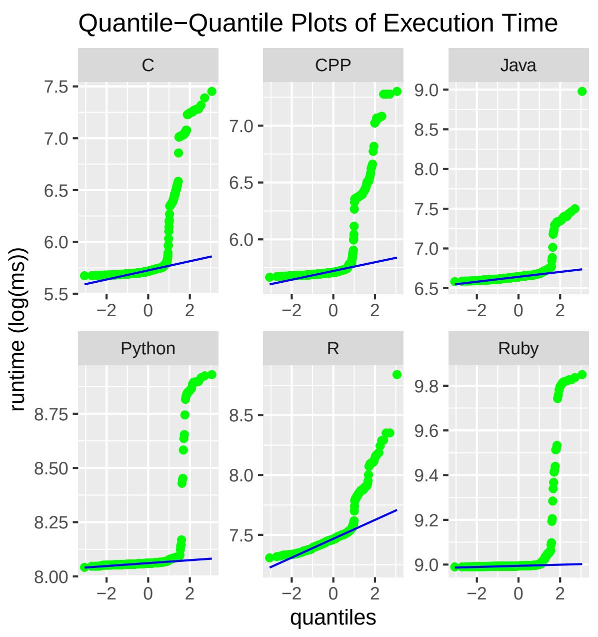
Key Differences: Compiled languages require a compilation step that produces an executable, while interpreted languages are executed directly by an interpreter.
Compiled languages tend to have better performance due to the pre-compiled nature of the code, whereas interpreted languages are more flexible 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 using the algorithm for each programming language and obtained the following jittered graph.





We can observe that C and C++ seem to be the fastest languages, though further analyses still need to be performed. We can also see from the Quantile-Quantile(QQ) 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 5 different hardware setups available as blocks. 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, 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 Sources of Variation

Treatments: We have 6 treatment factors, which are the programming languages that the algorithm is applied 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 and 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 on various hardware setups seem to follow the same order:

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

Whist 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.

3.4 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.

3.5 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 take as a single observation.

3.6 Randomisation Procedure

We first order from 1 to 6 for the computers belonging to each block. 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 number.

4 diagram

% latex table generated in R 4.3.1 by xtable 1.8-4 package % Fri Sep 13 19:05:55 2024

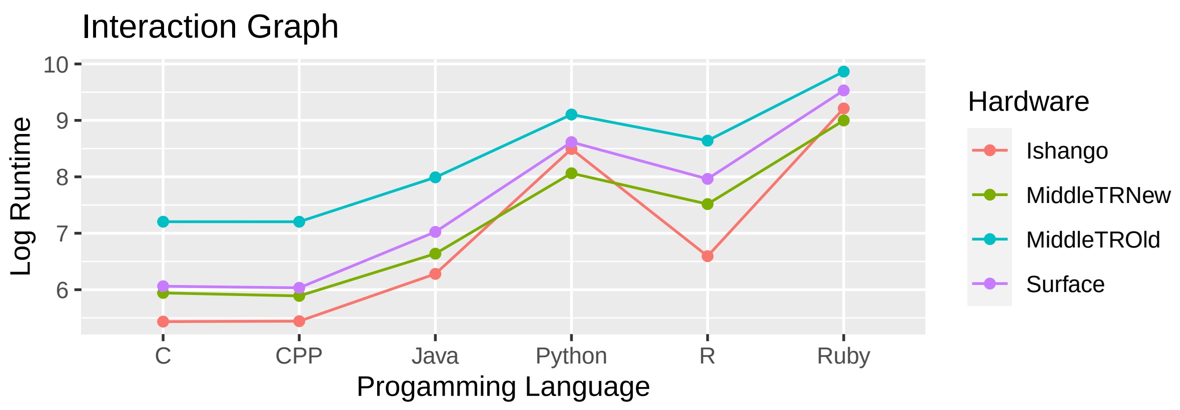
	PCSpec	Treatment
1	Ishango PC	java c python ruby c++ r
2	MidddleTROld	r c++ python c ruby java
3	MiddleTRNew	python r java c c++ ruby
4	ScilabB	python java ruby r c c++
5	ASUS (i7-5500u)	c c++ r ruby python java
6	ScilabD	c python java r ruby c++
7	LT	c java r c++ ruby python
8	HP (i5-7200)	r python ruby c++ java c

Table 1: placeholder

4.1 Pilot Experiment

We follow this direction and perform an pilot study 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'
```



	Hardware	C	CPP	Java	Python	Ruby	R
1	Ishango	229.1316	230.8817	533.6732	4881.330	10015.010	730.6641
2	MiddleTROld	1345.2420	1344.6080	2953.2610	8989.950	19240.590	5657.5140
3	MiddleTRNew	381.0000	361.2894	763.4666	3174.553	8095.145	1838.7730
4	Surface	428.6748	417.2392	1122.8945	5514.829	13780.423	2873.1987

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

4.2 A subsection

A numbered list:

1) First point
2) Second point
    - Subpoint

A bullet list:

- First point
- Second point

5 Results

computers of different specifications are harder to come by, we will only use 3 different hardware setup for this pilot study.

You can reference this figure as follows: Fig. **??**.

```
plot(1:5, pch = 19, main = "Some data", xlab = "Distance (cm)", ylab = "Time (hours)"
```

Reference to second figure: Fig. 1

5.1 Generate a table using `xtable`

```
df <- data.frame(ID = 1:3, code = letters[1:3])

# Creates tables that follow OUP guidelines using xtable
library(xtable)
print(xtable(df, caption = "This is the table caption", label = "tab:tab1"),
  comment = FALSE
)
```

	ID	code
1	1	a
2	2	b
3	3	c

Table 2: This is the table caption

You can reference this table as follows: Table 2.

5.2 Generate a table using `kable`

```
df <- data.frame(ID = 1:3, code = letters[1:3])

# kable can alse be used for creating tables
knitr::kable(df,
  caption = "This is the table caption", format = "latex",
  booktabs = TRUE, label = "tab2"
)
```

You can reference this table as follows: Table 3.

6 Discussion

You can cross-reference sections and subsections as follows: Section **??** and Section 4.2.

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

7 References

Table 3: This is the table caption

ID	code
1	a
2	b
3	c

8 Appendix

PC Specificiations

% latex table generated in R 4.3.1 by xtable 1.8-4 package % Fri Sep 13 19:05:56 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 4: Table of Pcs used and their respective specifications

Acknowledgements

This is an acknowledgement.

It consists of two paragraphs.

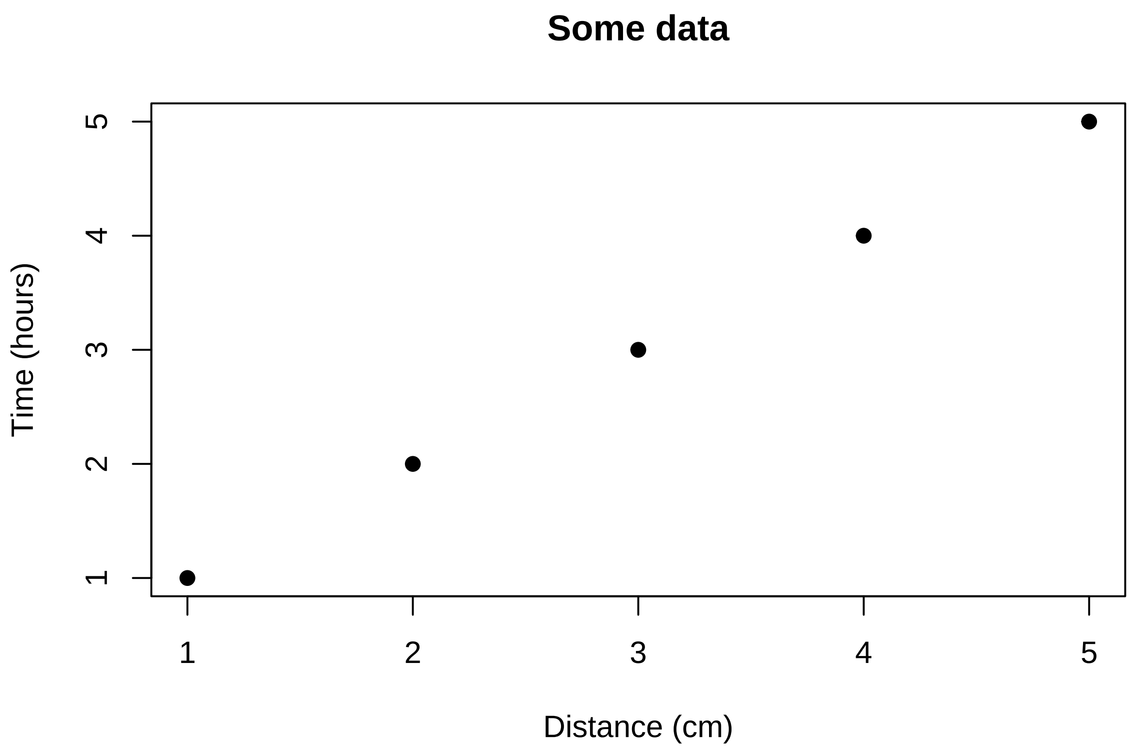


Figure 1: This is the second figure.