EEE4120F Practical 01

Rory Schram[†] and Natasha Soldin[‡]
EEE4120F Class of 2023
University of Cape Town
South Africa
[†]SCHROR002 [‡]SLDNAT001

***Abstract*—This paper compares differently implemented testing methodologies that make use of the statistical operations builtin to Julia programming language and custom functions created to explore random generation of data functions and correlations that occur between data sets.**

I. Introduction

A. Aim

The aim of this practical is to familiarise students with the use of statistical operations in Julia programming language.

B. Theory

The statistical operations used throughout this practical are correlation and speedup.

Correlation is a statistical function that is used to compare the similarity between two datasets. It does this by means of a correlation coefficient r which ranges from 1 to -1. Where 1 refers to a perfect positive correlation (i.e. as one dataset changes, the other dataset changes in the same way), -1 refers to a perfect negative correlation (i.e. as one dataset changes, the other dataset changes in the opposite way) and 0 refers to zero correlation (i.e. there is no relationship between how the two datasets change). The coefficient is calculated using Pearson's correlation formula as follows:

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}} \tag{1}$$

Speed-up is a statistical function that compares the runtime of an un-optimised program (T_{p1}) to that of an optimised program (T_{p2}).

$$Speedup = \frac{T_{p1}}{T_{p2}} \tag{2}$$

II. Methodology

A. Hardware and Software

As the practical was mainly software based, the only hardware required was a PC running a Julia compatible IDE. The software was Julia programming language, which is commonly used for parallel computation, data science and machine learning.

B. Implementation and Experimental Procedure

The practical is broken into three sections: white noise generation, correlation and shifted signals. Time measurements will be taken using Julia's built in `TickTock` function where the `tick()` command is used to start the timer and the `tock()` command used to stop it as seen below:

Listing 1: TickTock Time Measurement Function

```
    tick()
    #   critical section of code needing to be timed
    tock()
```

To ensure maximum accuracy, each time measurement recorded is an average of 10 individual time measurements to ensure minimal discrepancies.

1) White Noise Generator: In this practical, white noise is generated using two methods both making use of Julia's built in `rand()` function. These two methods are tested on datasets of varying sizes and their time measurements are compared as well as used to calculate program speedup.

The first method is to use the `rand()` function alone to generate uniformly distributed random values within an interval as seen below:

Listing 2: Code for first method of White Noise Generation

```
numberOfSeconds = 1000

function createNoise(seconds)
    whiteNoise = (rand(seconds*48000)*2).-1
end
```

The second method involved writing a script called `createwhiten.m` which uses a forloop to generate one sample of white noise at a time over N seconds as seen below:

Listing 3: Code for second method of White Noise Generation

```
numberOfSeconds = 1
function createwhiten(n)
    whiteNoise = Array{Float64}(undef, 0)
    for i in 1:n
        for j in 1:48000
            append!(whiteNoise, (rand(1)*2).-1)
        end
    end
    println("Number␣of␣samples:␣", length(whiteNoise))
    WAV.wavwrite(whiteNoise, "whiteNoise2.wav", Fs=48000) #sample freq is 4800Hz
    return whiteNoise;
end
```

It is hypothesised that the first method will be faster and more efficient than the second as it calls the `rand()` function once off to generate the white noise array instead of building the white noise sample by sample compounded together by a for loop and calling the `rand()` function in each iteration.

2) Correlation: In this practical, correlation is calculated using two methods. The first method is using the `Statistics.cor` function which belongs to Julia's built in statistic package and the second method involving writing a script called `corr()` that implements Pearson's correlation formula 1 as seen below:

Listing 4: Code for second Correlation function

```
function corr(in1, in2)
    aveIn1 = 0.0
    for i in in1
        aveIn1 += i
    end
    aveIn1 = aveIn1/length(in1)
    #println(aveIn1)

    aveIn2 = 0.0
    for i in in2
        aveIn2 += i
    end
    aveIn2 = aveIn2/length(in2)
    #println(aveIn2)


    top = 0.0
    for i in 1:length(in1)
        top += (in1[i]-aveIn1)*(in2[i]-aveIn2)
    end

    bottom1 = 0.0
    for i in 1:length(in1)
        bottom1 += (in1[i]-aveIn1)^2
    end
    bottom1 = sqrt(bottom1)

    bottom2 = 0.0
    for i in 1:length(in1)
        bottom2 += (in2[i]-aveIn2)^2
    end
    bottom2 = sqrt(bottom2)

    bottom = bottom1*bottom2

    r = top/bottom
    println("The␣corrolation␣between␣the␣two␣datasets␣is:␣",r)
end
```

Each of the two methods are then used to calculate the correlation between the same instance of white noise generation from the same function (i.e. second method described in section II-B1) and different instance of white noise generation from different functions (i.e. both methods described in section II-B1).

It is hypothesised that the first experiment's data will be perfectly correlated (i.e. r = 1) and the second experiments data will have little to no correlation (i.e. r $\approx$ 0). The first method will be faster and more efficient than the second as it implements a pre-defined method built into Julia which typically executes faster than performing calculations from first principles as done in the second method.

3) Shifted Signals: Correlation is further explored in this section between sinusoidal waves with varying frequency and sampling sizes. Julia's pre-defined `Statistics.cor` function is used to calculate correlation and the `circshift()` function is used to generate the shifted sinusoidal waves as seen below:

It is hypothesised that the correlation between the shifted signals vary between 1 and -1 as the sinusoids are shifted in and out of phase.

Listing 5: Code for Shifted Signal Correlation

```
using Plots
using Statistics

x = range(0, 16*pi, length = 1000)
y = sin.(x)
corrolationAfterShift = zeros(size(x))

for i in 0:999
    global y_shifted = circshift(y, i)
    corrolationAfterShift[i+1] = Statistics.cor(y, y_shifted)
    #println(Statistics.cor(y, y_shifted))
end

l = @layout [a ; b]
p1 = plot([y,y_shifted])
title!("Shifted␣Sinusoids")
xlabel!("Samples")
ylabel!("y1␣and␣y2")

p2 = plot(corrolationAfterShift)
xlabel!("Samples")
ylabel!("Correlation")
title!("Corrolation␣between␣the␣two␣shifted␣sinusoids")
fig = plot(p1, p2, layout = l)

display(fig)
```

III. Results

A. White Noise Generator

The first method of white noise generation produced the following result as seen in figure 1 below.

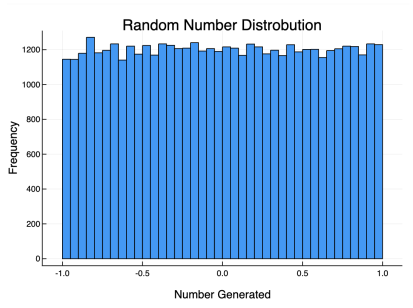

Fig. 1: White Noise Histogram Representation

The white noise generation results over varying sample sizes for the two methods can be summarised in table I below:

Number of Samples	**Execution time of Method 1 (ms)**	**Execution time of Method 2 (ms)**	**Speed-up**
1 * 48000	93,4	116,8	1,25
10 * 48000	77,6	272,2	3,51
100 * 48000	82,4	2349	28,5
1000 * 48000	77,8	20245	260,22

TABLE I: Results of White Noise Generation Experiment

The data in table I above indicates that the first method is significantly faster than the second method as originally hypothesised. The first method's execution time stays relatively constant for increasing sample sizes whereas the second method's execution time increases linearly. This implies that the second method's implementation is more computationally expensive as a result of the for loops creation of unnecessary overhead and delayed execution.

The speed-up can be calculated using equation 2 above where the second method is the un-optimised program and the first method is the optimised program. This speed-up can be graphically represented in figure 2 below:

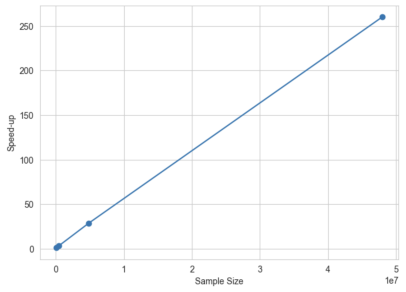

Fig. 2: White Noise Methods Speed-up

The speedup dramatically increase as the sample size increases. This is because the second method has a time complexity of order O(n^2) due to its implementation of two nested for loops. As the sample size varies, the second method's execution time increases significantly. The first method is very optimised (potentially utilizing parallel processing) which results in a speed-up that linearly increases with sample size.

B. Correlation

When applied to the same instance of white noise both the first and second method produced a correlation coefficient of 1 which was to be expected as the data is identical and is indicative of proper correlation method functionality.

The correlation functions applied to two instance of white noise generated from different functions produced a correlation co-efficient of 0,0005 which was expected as the data is randomly generated and thus should not be correlated. The execution times for this experiment were measured for varying sample sizes and speed-up was calculated as shown in table II below:

Sample Size	**corr speed (ms)**	**Statistics.cor speed (ms)**	**Speed-up**
1 * 48000	30,78	1,4	21,99
2 * 48000	36,81	1,5	24,54
3 * 48000	40,27	1,5	26,85
4 * 48000	45,49	1,6	28,43
5 * 48000	49,89	1,7	29,35
6 * 48000	55,61	1,8	30,89
7 * 48000	62,14	1,9	32,71
8 * 48000	65,4	1,7	38,47
9 * 48000	70,91	1,9	37,32
10 * 48000	75,6	2	37,80

TABLE II: Results of Correlation Experiment

The data in table II above indicates that the first correlation method is significantly faster than the second method as originally hypothesised. The execution time of the first method remains relatively constant over varying sample sizes whereas the second method's execution time increases linearly. This is due to the efficacy of a built in Julia function over a formula implementation by first principles.

Once again, the speedup as seen in figure 3 is quite dramatic for the correlation methods. This is again due to the second method's utilization of for loops which have a time complexity of O(n), and the first method using parallel processing to handle larger datasets more efficiently.

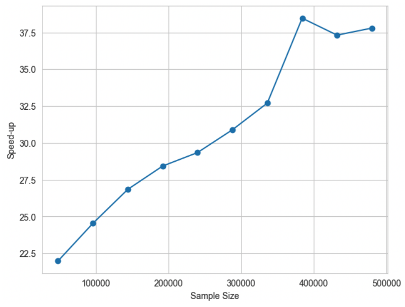

Fig. 3: Correlation Methods Speed-up

C. Shifted Signals

The results of the shifted signals can be shown graphically in plot 4 below:

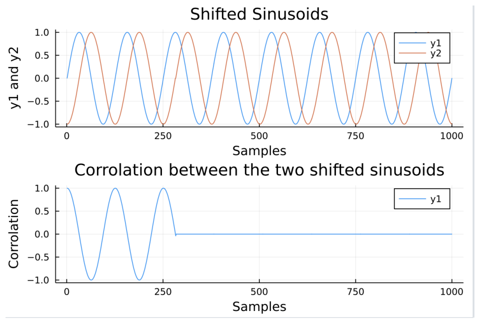

Fig. 4: Shifted Signal Experiment

The correlation between the two sinusoids oscillates periodically as the phase shift between the sinusoids ϕ is changed. When $\phi = 0$ or any multiple of 2π then the signals are in phase and the correlation co-efficient is 1, when $\phi = \pi$ or any multiple of π then the signals are out of phase and the correlation co-efficient is -1. This oscillation between 1 and -1 is visualized underneath the plot of the two sinusoids. This second graph shows how shifting one of the sinusoids by increasing values of ϕ, while testing the correlation between the two, gives values of correlation that are sinusoidal in nature themselves.

IV. Conclusion

White noise generation using a for loop to iteratively build each sample (method 2) produced a speed-up of on average 73,37 when compared to a once off white noise generation (method 1). Pearson's correlation formula (method 2) produced a speed-up of on average 30,83 when compared to Julia's built in statistics correlation function (method 1). For the two aforementioned experiments the execution time of the optimised methods (method 1) remained relatively constant as sample size increased whereas the un-optimised methods (method 2) showed a linear execution time increase, this is due to the parallel capability of the optimised programs. There is a relationship between the shift between two sinusoidal signals and the correlation between them, as the shift varies, the correlation co-efficient varies in a sinusoidal fashion.