EEE4120F Practical 1 - Julia Performance Measurement

David Moore
MRXDAV015

I. Introduction

The objectives of this practical is to perform 3 experiments. The aim of the first 2 experiments is to measure the performance of the algorithms that generate white noise and compute Pearson's Correlation function with varying sample sizes. The aim of the last experiment is to investigate and compare the correlation of signals that are shifted in time.

The first experiment will measure the performance of the two algorithms that generate white noise, namely *simplewhiten()* and *createwhiten()*. The second experiment will measure the accuracy and performance of the implemented Pearson Correlation function *corr()*, and the *Statistics.cor()* function from the *Statistics* package.

The third experiment will investigate the correlation of signals shifted in time. Sinusoidal signals of varying frequency and sample sizes will be generated. These sinusoidal signals will then be correlated with its shifted counterpart as the samples are circulary shifted over the entire sample space.

II. Method and Implementation

A. White Noise Generation

The functions *simplewhiten()* and *createwhiten()* were used to generate an array of white noise samples that is needed to generate a sound wave. A *.wav* audio file can then generated by sampling the array of white noise at a frequency of 4,8kHz.

This was possible by making use of the *rand()* function which generates a random value within the interval of [0, 1). However, in order to create this sound wave, the random value will need to be in the interval of [-1.0, 1.0). Therefore, the generated random value will need to be multiplied by 2 then have 1 subtracted from it to satisfy this condition.

Where the functions differ is when the necessary mathematical operations are performed when generating a white noise samples. *simplewhiten()* first creates an array of random values using *rand()*, then iterates through this array twice to perform the necessary mathimatical operations. Whereas, *createwhiten()* uses a loop where only one random value is generated at a time then the mathimatical operations are performed on it. Then the white noise array is populated one element at a time.

```
function simplewhiten(n)
    white_noise = Array{Float64}(rand(4800*n)*2 .-1)
    println("Number of samples in simplewhiten: ", size(white_noise))
    return white_noise
end
```

Fig. 1. Code for *simplewhiten*

```
function createwhiten(n)
    samples = Array{Float64}(undef, 4800*n)

    for i in 1:size(samples, 1)
        samples[i] = rand()*2 - 1
    end

    println("Number of samples in createwhiten: ", size(samples))
    return samples
end
```

Fig. 2. Code for *createwhiten*

Therefore, *createwhiten()* should be better suited for generating white noise samples than *simplewhiten()*. The reason being is that *createwhiten()* should be computationally faster to compute as *simplewhiten()* needs to perform the necessary mathematical operations throughout the array twice.

In order to verify that *createwhiten()* is capable of generating an array of white noise samples, it needs to be able to create a uniform distribution of values within the interval of [-1.0, 1.0).

The first experiment conducted will measure and compare the performance of *simplewhiten()* and *createwhiten()* to generate white noise samples. The data that is recorded for this experiment will be the time it takes for both functions to generate *48 000, 480 000, 48 000 000* samples of white noise. The performance will then be determined by measuring the speed up of *createwhiten()* when compared to *simplewhiten()* as the benchmark.

B. Pearson's Correlation

The implemented *corr()* function is based on the formula for calculating Pearson's Correlation for a set of samples seen below:

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

```
function corr(x, y)
    # Mean for x
    Σx = 0
    for xᵢ in x
        Σx = Σx + xᵢ
    end
    x̄ = Σx / size(x, 1)

    # Mean for y
    Σy = 0
    for yᵢ in y
        Σy = Σy + yᵢ
    end
    ȳ = Σy / size(y, 1)

    σxy = sum((x .- x̄) .* (y .- ȳ))
    σx² = sum((x .- x̄).^2)
    σy² = sum((y .- ȳ).^2)

    r = σxy / (sqrt(σx²) * sqrt(σy²))
    return r
end
```

Fig. 3. Code for *corr*

The implemented *corr()* function will unlikely be better suited than the *Statistics.cor()* function to calculate the correlation of a set of samples. This is due to *corr()* using 2 loops to calculate the mean of both sample arrays in order to perform the correlation calculation. This will cause longer computational times as the size of these sample sets increase.

In order to verify that *corr()* is capable of calculating Pearson's Correlation it needs to demonstrate that it produces the same value as *Statistics.cor()* when given the same sample sets and that it produces a value of 1.0 when calculating the correlation of the sample set with itself.

The second experiment conducted will measure and compare the performance between the *Statistics.cor()* and *corr()* functions at calculating Pearson's Correlation with given sets of white noise samples. The data recorded will be the time it takes for *corr()* and *Statistics.cor()* to calculate the correlation of *48 000, 480 000, 48 000 000* samples of white noise generated by *simplewhiten()* and *createwhiten()*. The performance of these two correlation functions will then be determined by measuring the speed up of *corr()* when compared to *Statistics.cor()* as the benchmark.

C. Correlation of Shifted Signals

The third experiment is an investigation on the correlation of signals shifted in time. The signal used for this will be a sinusoidal waveform which will be repeatedly correlated with itself as it is circularly shifted by one sample at a time. The expected output for this will produce a new signal resembling a cosine waveform. This is due to the correlation oscillating between a postive and negative correlation as the samples are shifted.

This experiment will use a sinusoidal signal with the varying frequencies of 1Hz, 10Hz, and 30Hz and sample sizes of 100, 1 000, and 10 000 samples. The data recorded will be the correlation of the sinusoid and its time shifted counterpart for each of these instances.

III. Results and Discussion

A. White Noise Generation

Fig. 4. verifies that *createwhiten()* does indeed have a uniform distribution of white noise, the same as *simplewhiten()*:

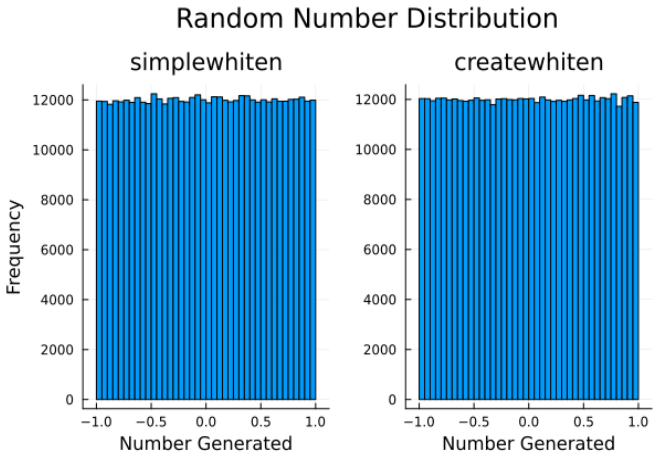

Fig. 4. Histogram plot of the distribution of samples produced by *simplewhiten()* and *createwhiten()*

TABLE I
Experimental Results for White Noise Sample Generation

Seconds(s)	Samples	simplewhiten(s)	createwhiten(s)	Speed Up
100	480 000	0.0221795	0.0147268	1.5060638
1000	4 800 000	0.0923815	0.0589503	1.5671082
10000	48 000 000	0.7962621	0.2247312	3.5431756

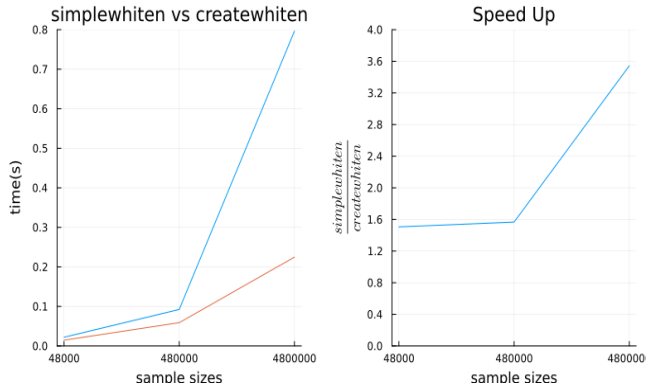

Fig. 5. White noise generation results

As seen in the results it is clear that *createwhiten()* is a better suited algorithm to generate an array of white

noise samples. This is due to the algorithm performing the necessary mathematical operations on every sample as it is being generated. Unlike *simplewhiten()* having to interate over the array twice to perform these operations.

B. Pearson's correlation

Fig. 6. demonstrates that *corr()* does indeed calculate a value of 1.0 when correlating a set of samples with itself. It is also able to calculate the same value as *Statistics.cor()* when correlating the white noise samples generated by *simplewhiten()* and *createwhiten()*. Thereby verifying that it does function as intended.

```
Number of samples in simplewhiten: (480000,)
Number of samples in createwhiten: (480000,)
Calculating the correlation of createwhiten with itself:
corr result: 1.0
Statistics.cor result: 1.0
Calculating the correlation of createwhiten and simplewhiten:
corr result:  -0.0016180980937638583
Statistics.cor result:: -0.0016180980937638585
```

Fig. 6. Verifitcation for the implemented *corr()* function

TABLE II
EXPERIMENTAL RESULTS FOR THE CORRELATION FUNCTIONS

Samples	simplewhiten(s)	createwhiten(s)	Speed Up
480 000	0.0221795	0.0147268	1.5060638
4 800 000	0.0923815	0.0589503	1.5671082
48 000 000	0.7962621	0.2247312	3.5431756

Fig. 7. Correlation function comparison results

The results indicate that *corr()* does not perform better than *Statistics.cor()*. This is due to the two loops in the algorithm that increases the computational time sharply as the size of the sample sets increase. The average speed being less than 1.0 also indicates that *corr()* is not a better suited function than *Statistics.cor()*.

C. Shifted sines and correlation

As seen in the results, the correlation between the sinusoidal signal and its time-shifting counterpart does produce a cosine signal as the correlation oscillates between a postive and negative correlation. There is an interesting observation that can be seen in Fig. 10. when there is not enough samples to accurately represent the sinusoid. The output correlation produces a signal that does not exhibit a cosine waveform.

This is likely due to the adjacent samples from the sinusoid having a high variance between them. However, as the number of samples increases the correlation signal produces a stable cosine waveform. This behavior resembles aliasing as the signals became indistinguishable due to undersampling.

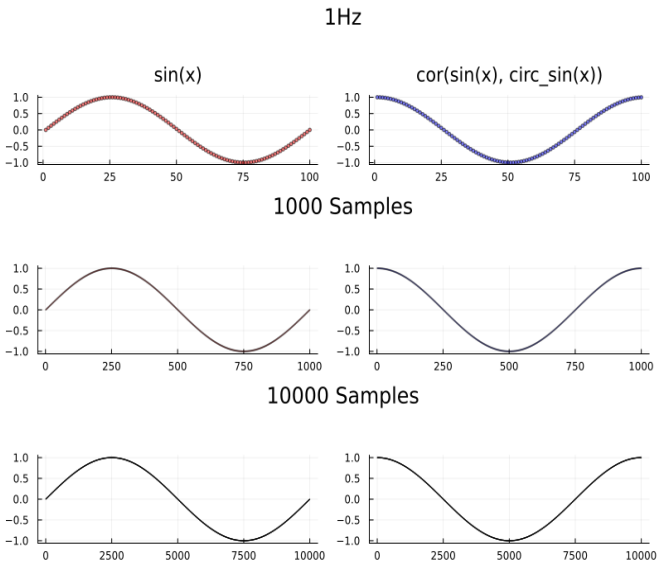

Fig. 8. Correlation results for 1Hz

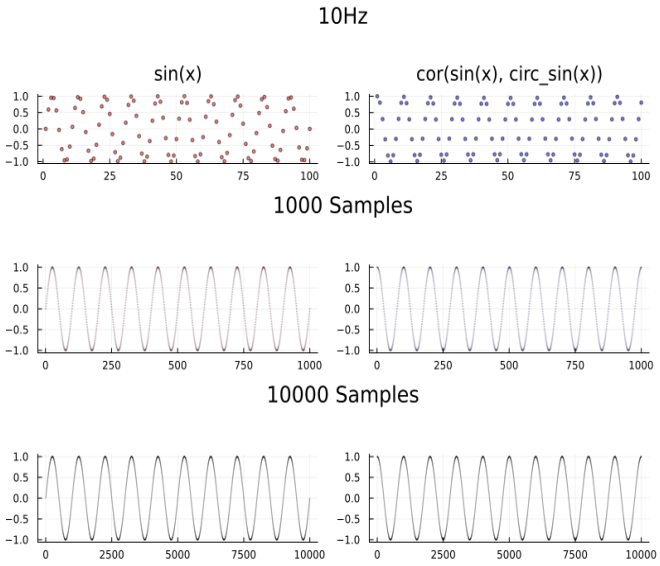

Fig. 9. Correlation results for 10Hz

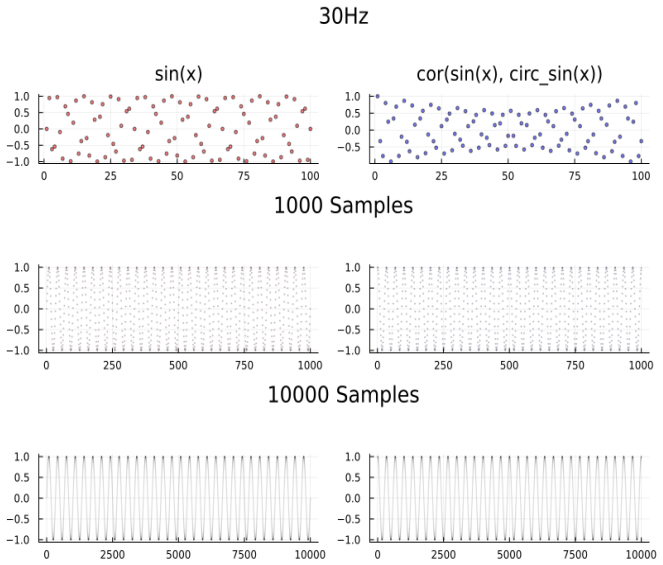

Fig. 10. Correlation results for 30Hz

IV. Conclusion

The results in the first experiment have proved that *createwhiten()* does perform better than *simplewhiten()* at generating white noise samples. This is due to the effciency of the mathematical operations being used to generate one sample at a time instead of having to iterate over the entire array.

The second experiment has shown that *corr()* does not perform better than *Statistics.cor()* when calculating the correlation of 2 white noise sample sets. The lacklustre computational performance of *corr()* stems from having to use 2 loops within its operation to calculate the mean of the given sample sets. This causes the computational time to drastically increase as the size of the sample sets increase.

The investigation in the third experiment has shown that the correlation between the sinusoidal signal and its time-shifting counterpart does produce a cosine signal. This is due to the correlation oscillating between a postive and negative correlation as the samples are shifted. However, as the frequency of the signal increases the number of samples also need to increase to prevent aliasing due to undersampling.

The *Julia* code for this practical can be viewed as a Jupyter Notebook file within this Github repository. Click here.