# Take-Home Interview Report - Luminosity Labs

Sarwan Shah

31st October, 2023

## Question 1.

### Prompt

Approximating the sin, cosine, and exponential function using only addition, multiplication, subtraction, and division operations.

### Thought Process & Observations

The sin, cosine, and exponential function were approximated using the Taylor Series expansion of these functions in C code. The following were some considerations and observations made during this process:

- The higher the order of the approximation - i.e. incorporating higher order terms of the series when computing the result for the respective operation - the better the approximation.
- It was important to remain mindful that if the approximation order was too high it would lead to higher order terms overflowing beyond the range of the variables that can store them.
- The aforementioned applied most strongly to the case of the exponential series approximation, which diverged quicker compared to the sin & cosine.
- It was also realized that Taylor Series approximation for sin & cosine exists about origin in the range of $[-\pi, \pi]$. Thus, a normalization function had to be written to ensure that values passed to function were normalized to this range in radians.
- The error between the actual exponential function and the approximation inevitably grew for larger values by virtue of the strictly increasing nature of the exponential function.
- The series expansion/approximation process in each case was optimized by keeping track of preceding terms and re-using them for the calculation of higher order terms.

### Results

```
Approx: Sin(-7.853982): -1.000000, Actual: Sin(-7.853982): -1.000000
Approx: Sin(4.712389): -1.000000, Actual: Sin(4.712389): -1.000000
Approx: Cos(-7.853982): -0.000000, Actual: Cos(-7.853982): -0.000000
Approx: Cos(6.283185): 1.000000, Actual: Cos(6.283185): 1.000000
Approx: Exp(5.000000): 148.379578, Actual: Exp(5.000000): 148.413159
Approx: Exp(10.000000): 20188.169922, Actual: Exp(10.000000): 22026.465795
```

Figure 1: The figure above shows results for the where the approximation functions have been compared with their actual implementations for various input values.

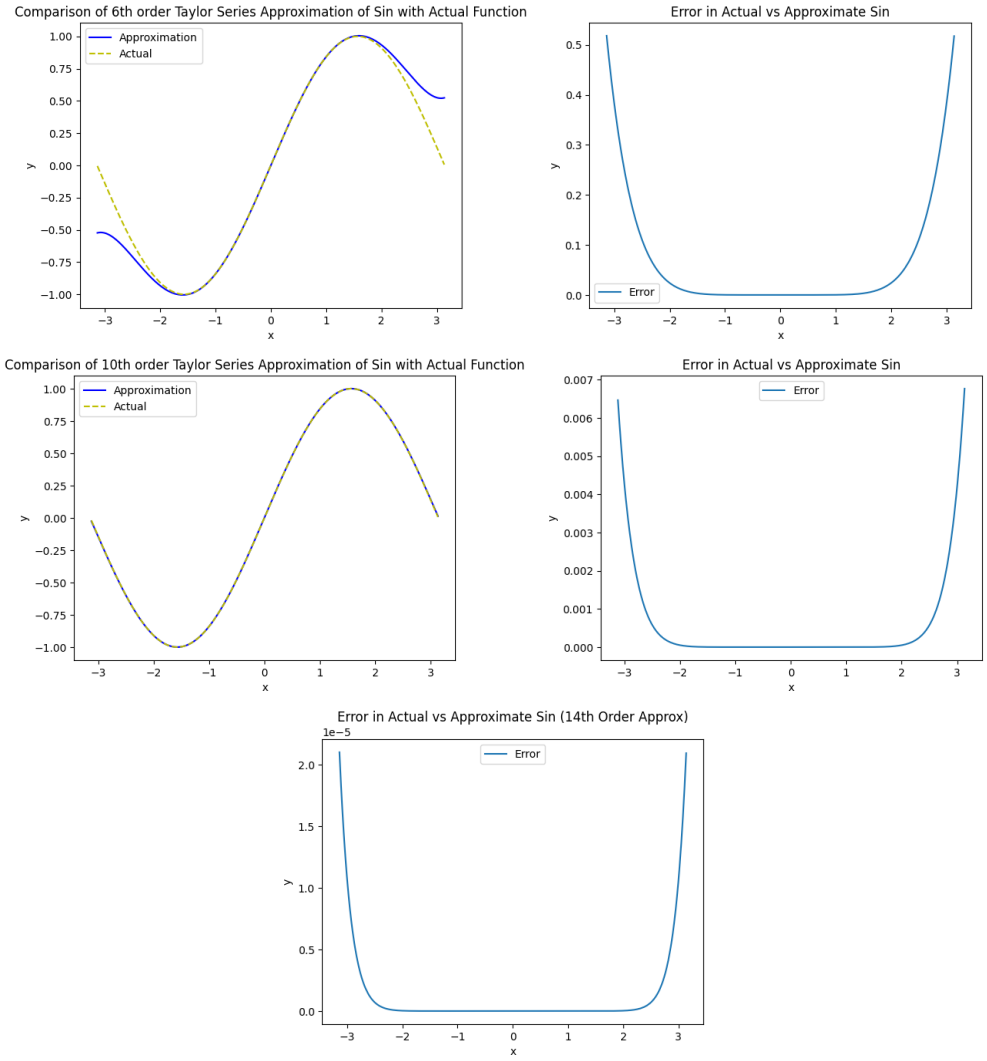


Figure 2: The figure shows us the comparison between how the 6th order approximation performs compared to the actual function. It is worth noting how it doesn't approximate well at the edges of 'pi' bounds, which is reflected in the error curve too! However, we see that this error at the bounds is significantly reduced for the 10th order approximation. However, it only drops below 0.0001 for beyond the 14th order approximation as show above.

# Question 2.

## Prompt

Implement a Fast Fourier Transform (FFT) in C using addition, subtraction, division, and multiplication operations. Ensure that the FFT size can be adjusted.

## Thought Process & Observations

- What is a Fourier Transform (FT)? How does multiplication with a complex exponential magically take us to the frequency domain?
- Answer to the above: Assume we have square signal and we compare it with a sin signal of similar frequency/periodicity (let's call this an nth frequency) and peak amplitude i.e. think in terms that they have some overlap in terms of their shape/trend.

  Moving forward, what if we check the correlation between these two signals by multiplying corresponding points? Naturally, if there is strong overlap in the shape of the signals, the sum of the multiplications will be

large value (let's call this an amplitude), and if not, then it would be a small value.

Now assume we have some arbitrary signal. Imagine we compare this signal against all possible nth frequency sinosoids. We'll realize some of the sinosoids will have a strong correlation with the signal and an associated strong amplitude, while others may not.

Then if we plot these results of all possible nth frequencies against the correlation amplitudes, MAGIC!, we will have a representation where the nth frequency in the signal will appear prominently in the plot by virtue of their amplitudes. This is essentially how the FT takes us into the frequency domain. The complex exponential in the FT represents all the sines and cosines through Euler's identity.

- It is worth noting that in the discrete version of the Fourier Transform i.e. the DFT, we can't actually consider all possible n frequencies (which would theoretically be infinite, but we do know the highest possible frequency based on our sampling rate, and we make use of that to sufficiently capture information) and hence are operating in finite resolution.
- Operating at this finite resolution is coupled with the aperiodic and sharp nature of real-life signals leads to the phenomena spectral leakage. This is because due to finite resolution we might allow a signal bit/frequency to be considered/correlated with multiple neighboring frequencies, instead of an exact one.
- What's the magic of FFT compared to the normal DFT? It takes us from $n^2$ (quadratic) time-complexity to $n * log(n)$ (almost linear) for computing the DFT of a signal. This is a huge difference in terms of computation requirement (1000 vs 1 million), and this improvement is super significant because almost all digital applications that exist today require computing the DFT for signal processing and digital communication.
- How does the FFT do it? By exploiting symmetries in the DFT calculation. It does this by first dividing the calculation into odd and even indexed calculations. Then for each odd and even indexed set of calculations.

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-\frac{j2\pi kn}{N}} \tag{1}$$

$$X(k) = \sum_{m=0}^{N/2-1} x(n) \cdot e^{-\frac{j2m\pi k}{N}} + \sum_{m=0}^{N/2-1} x(n) \cdot e^{-\frac{j(2m+1)\pi k}{N}} \tag{2}$$

$$X(k) = \sum_{m=0}^{N/2-1} x(n) \cdot e^{-\frac{j2m\pi k}{N}} + \sum_{m=0}^{N/2-1} x(n) \cdot e^{-\frac{j(2m+1)\pi k}{N}} \tag{3}$$

$$X(k) = \sum_{m=0}^{N/2-1} x(n) \cdot e^{-\frac{jm\pi k}{N/2}} + \sum_{m=0}^{N/2-1} x(n) \cdot e^{-\frac{j(m+1/2)\pi k}{N/2}} \tag{4}$$

$$X(k) = \sum_{m=0}^{N/2-1} x(n) \cdot e^{-\frac{jm\pi k}{N/2}} + e^{\frac{j\pi k}{N/4}} \sum_{m=0}^{N/2-1} x(n) \cdot e^{-\frac{jm\pi k}{N/2}} \tag{5}$$

However, up till this point we've only gone from $N^2$ to $N^2/2 + N^2/2$ computations, which are the same. (Note: the additional exponent constant we've pulled out from the odd component is called the twiddle factor).

Furthermore, we realize now that for each of the these even and odd calculation sets the first half of the calculation/summation i.e. from $m = 0$ to $N/4 - 1$, is the symmetric to the second half i.e. from $m = N/4 - 1$ to $N/2 - 1$. This is because when m ¿ $N/4 - 1$, let's say $m = N/4 + r$, the term we have is:

$$= x(n) \cdot e^{-\frac{j(N/4+r)\pi k}{N/2}} \tag{6}$$

$$= e^{\frac{j(N/4)\pi k}{N/2}}(x(n) \cdot e^{-\frac{j(N/4+r)\pi k}{N/2}}) \tag{7}$$

$$= e^{j2\pi k}(x(n) \cdot e^{-\frac{j(r)\pi k}{N/2}}) \tag{8}$$

$$= x(n) \cdot e^{-\frac{j(r)\pi k}{N/2}} \tag{9}$$

Note that $e^{j2\pi k}$ is simply equal to 1 for an integer k, and $e^{-\frac{j(r)\pi k}{N/2}}$ is simply an equivalent term between $m = 0$ to $N/4 - 1$ and thus we don't need to re-compute it. This symmetry reduces our computation problem from $N^2/2 + N^2/2$ to $N^2/4 + N^2/4$ complexity. Thus, $N^2/2 < N^2$ .

But we realize that we can further divide each even and odd indexed set of calculations into their own even and odd indexed set of calculation. Thus, we go from $N^2/4 + N^2/4$ to $N^2/8 + N^2/8$ in the next step, and in this recursive fashion of computing the DFT, we are able to drop down the almost linear $n * log(n)$ time-complexity.

- Note, the above recursive approach works best only if we assume our signal length to be of a power of 2. This is because our recursive approach follows a division by 2 of the sample size at each step, and a power of 2 would be necessary to hit the base case. This is an assumption the implemented algorithm makes too!
- It was noted that the Taylor Series approximation of the exponential function failed very quickly due overflow in the imaginary part of the calculation. Thus, the Euler identity was used to effectively and efficiently calculate the value for the complex exponential.

## Results

```
Input: [2.000000, 1.000000, -1.000000, 5.000000, 0.000000, 3.000000, 0.
000000, -4.000000, ]

Output: [6.00 + 0.00i, -5.78 + 3.95i, 3.00 + 3.00i, 9.78 + 5.95i, -4.00
 + 0.00i, 9.78 + -5.95i, 3.00 + -3.00i, -5.78 + -3.95i, ]
```

```
>> fft([2, 1, -1, 5, 0, 3, 0, -4])

ans =

  Columns 1 through 5

   6.0000 + 0.0000i  -5.7782 - 3.9497i   3.0000 - 3.0000i   9.7782 - 5.9497i  -4.0000 + 0.0000i

  Columns 6 through 8

   9.7782 + 5.9497i   3.0000 + 3.0000i  -5.7782 + 3.9497i
```

Figure 3: The above figures shows the FFT calculation for a given signal of size 8. The first performed by the implemented algo and the other performed by MATLAB

```
CPU time used by DFT: 0.000073 seconds

CPU time used FFT: 0.000006 seconds
```

Figure 4: This figure shows the difference in computation time to compute the Fourier Transform using the DFT vs FFT for a signal size of 32. We can clearly observe the stark difference in the compution time, and thus, the powerfulness of the FFT algorithm.