Abstract

This report corresponds to the evaluation of the obtained results in the third laboratory of Applied Harmonic Analysis. We shall divide the report in two parts, one for each exercise.

1 Exercise 1

In our first exercise, we focused on implementing both the FFT and the IFFT. To implement the FFT, we referred to the pseudocode provided in the statement and adapted it for Octave code. As for the IFFT, we followed the notation in the statement and observed that the inverse of the matrix $M = (W_N^{ij})_{i,j=0}^{N-1}$ is $M^{-1} = \frac{1}{N}\overline{M}$. Therefore, the algorithm for computing the IFFT is essentially identical to the one for computing the FFT, except for changing the sign of the exponential term and dividing the resulting vector by the length of the vector N.

We have implemented the FFT and IFFT algorithms in two files named *recursive_fft.m* and *recursive_ifft.m*. These functions can be called in the main file to compute the FFT and IFFT of a given input signal. To test the correctness of our implementations, we created a simple vector $a = (1, \dots, 32)$ and compared the results with the precompiled functions. The discrepancy between my FFT and the precompiled version was of the order of 10^{-13}, while the difference between my IFFT and the precompiled one was of the order of 10^{-14}. Based on these results, we can confidently conclude that our implementations are correct!

2 Exercise 2

During our second exercise, we focused on developing a script that calculates the number of triplets formed by the initial 100,000 prime numbers. To achieve this, we crafted a function named *Eratosthenes* that calculates the first 100,000 prime numbers. This function utilizes the sieve of Eratosthenes algorithm, an ancient method for identifying all prime numbers up to a specified limit with a time complexity of $\mathcal{O}(nloglog(n))$. To utilize this function in our main script, we created a separate file named *Eratosthenes.m*.

Let us return to the main script, we utilize the Eratosthenes function with a parameter of $n = 1,299,709$, which corresponds to the 100,000th prime number. By doing so, we can effortlessly generate the coefficients of our polynomial, denoted as p, where a_i is 1 if i is prime and 0 otherwise. As recommended, we proceed to calculate p^2. To achieve this, we rely on the FFT and IFFT polynomial multiplication algorithm, which operates as follows:

- **Padding**: Pad the two polynomials with zeros so that their degree becomes a power of 2.
- **FFT**: Next, we apply the FFT to each of the padded polynomials. This will convert them from the polynomial domain to the frequency domain. Let A and B be the FFTs of the polynomials.
- **Pointwise multiplication**: We then multiply the two FFTs A and B pointwise (element-wise). Let C be the resulting FFT after the pointwise multiplication.
- **IFFT**: Finally, we apply the IFFT to C to get the product polynomial

After obtaining the resulting polynomial, we can easily calculate the number of triplets present. It's worth remembering that (p_i, p_j, p_k) forms a triplet if and only if $p_i + p_k = 2p_j$. Therefore, by iterating through the coefficients given by two times the list of primes, and then subtracting one from each coefficient and dividing it by two[1], we can determine the number of triplets for each prime. Finally, we just need to sum all of them up.

By executing this algorithm, we discover that there are a total of 250,155,454 triplets formed by the first 100,000 prime numbers. It's worth noting that this process takes approximately 8 seconds to complete, which is significantly faster than the time it would have taken if we had attempted to solve the problem using a "brute force" approach.

A Delivery format

We provide four code files and this report. The code files are as follows:

- **Lab3_DavidRosado.m**. The main script.
- **Eratosthenes.m**. File that contains the function **Eratosthenes** to compute the first n primes.
- **recursive_fft.m**. File that contains the function to compute FFT.
- **recursive_ifft.m**. File that contains the function to compute IFFT.

[1]We subtract one to exclude the trivial combination, such as $11 + 11 = 22$, and divide by two to consider each triplet only once.