1 Laboratory 3: The Fast Fourier transform

Exercise 1. *Implement the FFT and the IFFT.*

Given the input finite signal $x[0],\dots,x[N-1]$ its Fast Fourier Transform (FFT) is given by

$$X(\omega)\Big|_{\omega=2\pi j/N} = \sum_{n=0}^{N-1} x[n]e^{-inj2\pi/N} = y[j], \qquad 0 \le j < N.$$

If we denote the roots of unity by $W_N = e^{-2\pi i/N}$, then we can define a recursive implementation of the computation of the FFT by considering the interpretation of the Discrete Fourier Transform through two different representations of polynomials of degree $N-1$. Hence, the DFT is also interpreted as a way to convert a polynomial from its coefficients to its values at the roots of unity. This is, from the general polynomial representation we evaluate at the roots of unity in order to get the point-value representation of the given polynomial. In order to do so, a way to implement it is to consider the initial polynomial$A(x)$ in terms of the odd and even coefficients, i.e.

$$A(x) = \sum_{k=0}^{n-1} a_k x^k = A^{even}(x^2) + xA^{odd}(x^2)$$

Therefore, in order to find the corresponding values of the Transform at the n complex n-th roots of unity, $W_n^0, W_n^1, \dots, W_n^{n-1}$ we may just need to evaluate $A^{odd}(x), A^{even}(x)$ at the points $W_{n/2}^0, W_{n/2}^1, \dots,$ $W_{n/2}^{n/2-1}$. Therefore, the implemented algorithm in Matlab is given as follows:

```
function RFFT_ = RFFT__(a)
    n = length(a);
    if n == 1
        RFFT_ = a;
        return
    end
    w_n = exp(-2*pi*1i/n);
    w = 1;
    a_even = a(1:2:end);
    a_odd = a(2:2:end);
    y_even = RFFT__(a_even);
    y_odd = RFFT__(a_odd);
    y = zeros(1, n);
    for k = 0:(n/2 - 1)
        y(k+1) = y_even(k+1) + w*y_odd(k+1);
        y(k + n/2 + 1) = y_even(k+1) - w*y_odd(k+1);
        w = w*w_n;
    end
    RFFT_ = y;
end
```

Thus, we get the point-value representation, $y_k = A(x_k)$ for $x_k = W_N^k$, $k = 0,\dots,N-1$, where N is the initial degree of the polynomial.

By computing this code, note that for example, when considering row vectors input `a_1 = [1,2,3,4,0,0,0,0]`, `a_2 = [0,1,2,3,4,5,6,7]` we get as output

```
RFFT(a_1) = [10.000000-0.414214j, -2.000000+2.414214j, -2.000000+2.414214j,
    -2.000000-0.414214j, 0.000000-7.242641j, 2.000000-1.242641j, 0.000000+1.242641j,
    -2.000000+7.242641j],
RFFT(a_2) = [28.000000-4.000000j, -4.000000-4.000000j, -4.000000-4.000000j,
    -4.000000-4.000000j, 0.000000+9.656854j, 4.000000+1.656854j, 0.000000-1.656854j,
    -4.000000-9.656854j]
```

Note additionally that we can verify the validity of the computation of the FFT by considering the Matlab function `fft` that directly computes the Fast Fourier Transform.

Consider now the implementation of the Inverse Fast Fourier Transform (IFFT). By definition, the IFFT is given by the following expression:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} y[k]e^{ikn2\pi/N}, \quad 0 \leq n < N.$$

Thus, by considering again the same notation for the primitive roots of unity we have:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} W_N^{-kn} y[k]$$

Therefore, the IDFT is a conversion from the values of a polynomial to its coefficients, this is from the point-value representation to the original representation of the polynomial in terms of the coefficients. Notice that the recursive function implemented to compute the IFFT can be defined by modifying the previous function, by replacing the roles of a and y, replacing W_N by W_N^{-1} and by considering the normalization of the resulting values in terms of N. Thus, the corresponding implementation is conducted as follows:

```matlab
k_IFFT = RIFFT__(k_RFFT)/length(k_RFFT);
function IFFT_ = RIFFT__(a)
    n = length(a);
    if n == 1
        IFFT_ = a;
        return
    end
    w_n = exp(2*pi*1i/n); % Note the positive sign here
    w = 1;
    a_even = a(1:2:end);
    a_odd = a(2:2:end);
    y_even = RIFFT__(a_even);
    y_odd = RIFFT__(a_odd);
    y = zeros(1, n);
    for k = 1:(floor(n/2) )
        y(k) = y_even(k) + w*y_odd(k);
        y(k + floor(n/2)) = y_even(k) - w*y_odd(k);
        w = w*w_n;
    end
    IFFT_ = y; % Divide by n to incorporate the normalization factor
end
```

Again, we can check our results using the Matlab function `ifft()`. Moreover, notice that just by considering the mathematical definitions of the FFT and the IFFT we can see that the IFFT and FFT are complex conjugates, therefore an alternative approach to compute the IFFT, and moreover an additional way to check whether it is computed correctly is to implement the following function

```matlab
function x = IFFT__(X)
    X = conj(X);
    n = length(X);
    x = RFFT(X)/n;
    x = conj(x);
end
```

Observe that for this definition there is no need to normalize the output.

The corresponding `Matlab` implementations are presented on code `Lab3_ex1.m`.

Exercise 2. *A problem in additive combinatorics: Consider the set formed by the first N primes: $p_1, \ldots, p_N$. We want to know in how many ways we can choose three primes in this set such that they are consecutive terms of an arithmetic progression. Meaning that, how many triplets (i, j, k) are there such that $1 \leq i < j < k \leq N$ verifying that $p_j - p_i = p_k - p_j$.*
So the triplets (5, 11, 17), 7, 157, 307), (3, 5, 7) are valid as they are three consecutive terms of an arithmetic progression. But the triplets (2, 5, 7), (3, 7, 17) are not.
Write a program that computes the number of triplets formed by the first 100.000 primes.

In order to do so we may consider the polynomial $p(x) = \sum_{i=1}^{M} a_i x^i$ where $a_i = 1$ if i is prime and zero otherwise and compute its square $q = p^2$. Observe therefore that when computing the coefficients of the product polynomial we are considering

$$q(x) = \sum_{k=0}^{2M} c_k x^k,$$

where actually $c_k = \sum_{j=0}^{k} a_j \cdot a_{k-j}$. Hence, observe that this is directly the definition of the convolution, i.e. $c = a * a$.
On the other hand, notice that the coefficient of the polynomial, c_k indicates the number of possible combinations of the remaining monomials in order to get k, this is the possible combinations of indexes i, j such that $i + j = k$. Additionally, drawn from the definition of p we have that each of these indexes is prime. Therefore, we are considering the possible combinations of two prime numbers that add up to k. Therefore, if we find the coefficients that are exactly 3 we may have the three prime numbers that combined in pairs can add up to k, this is $k = i + j = j + l \iff j - l = j - i$ which are prime consecutive terms of an arithmetic progression. Consequently, we seek to count the number of coefficients of q equal to 3.

By directly computing the multiplication of polynomials and checking for the desired coefficients we get that among the first 100.000 primes we have exactly 16 possible combinations of 3 primes, this is 3 triplets of primes consecutive terms of an arithmetic progression.

Notice however, that we can have the case in which more than 2 combinations of prime numbers add up to the exponent $2k$ considered above. In these cases, we may have again the central combination, this is the term corresponding to x^k coefficient (since we look for a triplet of prime numbers and in any other case we may have both the combinations of $x^i \cdot x^j = x^k = x^j \cdot x^i$, and therefore a set of even length). Consequently, we may need to consider the coefficients that are strictly equal to 3, and also the coefficients that are odds greater than 3. For example, in the case of having as a term of the polynomial $q(x)$ the monomial $5x^{2k}$, then we may have two possible triplets, both containing k, and the corresponding equidistant prime numbers on both directions. This extends for any odd number greater than 3. Hence, we may consider the odd coefficients of q and sum the values of $(a_k - 1)/2$ in order to get the total number of triplets from the whole set of prime numbers considered. The number of triplets of prime consecutive terms of an arithmetic progression is: 250155454 triplets. It is computed using the code in `LAB3_EX2_v1.m`.

Observe however that computing this polynomial multiplication takes the cost of $N \times N$. However, we may consider the point-value representation of the polynomial $p(x)$ which is given as $\{(x_k, y_k\}_{k \in \{0, \ldots, N-1}$ for $y_k = p(x_k)$. Hence we can compute the evaluation of the polynomial on the N points given by the complex roots of unity by using the FFT. Then we can compute the corresponding multiplication by considering the point-value form of the given $q(x_k) = p(x_k)p(x_k)$, which may be computed by taking the pointwise multiplication from this representation. Thus we can interpolate back the point-value form into the polynomial form by considering the inverse FFT. With this approach, we may be able to compute the corresponding number of coefficients equals to 3 in $\mathcal{O}(nlog(n))$.

In order to do so, the algorithm may be:

1. We construct the polynomial of degree M with the first N primes, this is $p(x) = \sum_{i=1}^{M} a_i x^i$ where $a_i = 1$ if i is prime and zero otherwise.

```
suma = 0;
i = 2;
p = [];
while suma < N
    if isprime(i)
        p(i) = 1;
        fprintf("%.2f\n", i);
    else
        p(i) = 0;
    end
    suma = sum(p);
    i = i+1;
end
```

2. Now we have a polynomial, of degree M, with the first N primes. We seek to consider the point-value representation Therefore, we may consider the polynomial of degree $2M$ by adding the M zero coefficients to p.

```
M = length(p);
p_2 = zeros(1, M);
p_3 = cat(2, p, p_2);
```

3. Evaluate $p(x)$ at the $2M$ roots of unity using the FFT. Which could have actually been implemented by using the Matlab function `fft(p_3)`.

4. We compute the pointwise multiplications of the point-value forms: $q(x_k) = p(x_k) \cdot p(x_k)$.

5. Finally we interpolate $q(x)$ by using the inverse FFT. The implementation of the last 3 steps together would be:

```
p_fft = RFFT(p_3);
q_fft = p_fft .* p_fft;
q_ = RIFFT(q_fft);
```

6. Once we have back the coefficient, we can compute the number of triplets by again considering the coefficients of q that are odds greater or equal to 3, and considering the sum of the coefficients $(a_k - 1)/2$.

The corresponding code is presented on file `LAB3_EX3_v2.m`.