Student Name: Daniel Alconchel Vázquez
Semester: 1
Academic Year: 2022/2023

Faculty of Applied Physics and Mathematics
Institute of Physics and Applied Computer Science

Polynomials and the FFT

---

1. **Theoretical Introduction**

   In the previous assignment, we saw that the most common use for Fourier transforms, and hence the FFT, is in signal processing. A signal is given in the **time domain**: as a function mapping time to amplitude. Fourier analysis allows us to express the signal as a weighted sum of phase-shifted sinusoids of varying frequencies. The weights and phases associated with the frequencies characterize the signal in the **frequency domain**. Among the many everyday applications of FFT's are compression techniques used to encode digital video and audio information, including MP3 files.

   On the other hand, a **polynomial** in the variable x over an algebraic field F represents a function $A(x)$ as a formal sum:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

   We call the values $a_0, a_1, ..., a_{n-1}$ the **coefficients** of the polynomial. The coefficients are drawn from a field F, typically the set $\mathbb{C}$ of complex numbers.

   (a) **Polynomial Operations**

   A polynomial $A(x)$ has **degree** k if its highest nonzero coefficient is a_k; so we write then $degree(A) = k$. Any integer strictly greater than the degree of a polynomial is a **degree-bound** of that polynomial.

   We can define the **polynomial addition**, if $A(x)$ and $B(x)$ are polynomials of dregee-bound n, their **sum** is a polynomial $C(x)$, also of degree-bound n, such that if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j,\ B(x) = \sum_{j=0}^{n-1} b_j x^j$$

   then $C(x) = A(x) + B(x)$ where

$$C(x) = \sum_{j=0}^{n-1} c_j x^j,\ with\ c_j = a_j + b_j\ for\ j = 0, 1, ..., n-1$$

   For **polynomial multiplication**, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, their **product** $C(x)$ is a polynomial of degree-bound $2n-1$ such that $C(x) = A(x)B(x)\ \forall x$ in the underlying field. A mathematical way of express it is:

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j\ where\ c_j = \sum_{k=0}^{j} a_k b_{j-k} \tag{1}$$

Note that $degree(C) = degree(A) + degree(B)$. That implies that if A is a polynomial of degree-bound n_a and B is a polynomial of degree-bound n_b, then C is a polynomial of degree-bound $n_a + n_b - 1$. Since a polynomial of degree-bound k is also a polynomial of degree-bound $k+1$, we will normally say that the product polynomial C is a polynomial of degree-bound $n_a + n_b$.

(b) **Coefficient Representation**

A **coefficient representation** of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of a degree-bound n is a vector of coefficients $a = (a_0, a_1, ..., a_{n-1})$.

The coefficient representations is convenient for certain operations on polynomials, such as **evaluating** the polynomial $A(x)$ at a given point x_0; which consists of computing the value of $A(x_0)$. We can evaluate a polynomial in $\Theta(n)$ using the **Horner's rule:**

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + ... + x_0(a_{n-2} + x_0(a_{n-1}))...))$$

Similarly, adding two polynomials represented by the coefficient vector $a = (a_0, a_1, ..., a_{n-1})$ and $b = (b_0, b_1, ..., b_{n-1})$ takes $\Theta(n)$; because we just produce the coefficient vector $c = (c_0, c_1, ..., c_{n-1})$ where $c_j = a_j + b_j$ for $j = 0, 1, ..., n-1$.

Now, consider multiplying two degree-bound n polynomials $A(x)$ and $B(x)$ represented in coefficient form. If we use the method described in 1, multiplying polynomials takes time $\Theta(n^2)$, since we must multiply each coefficient in the vector a by each coefficient in the vector b. This operation of seems to be considerably more difficult than that evaluating a polynomial or adding two polynomials. The resulting coefficient vector c, given by the equation 1, is also called the **convolution** of the inputs vectors a and b, denoted $c = a \otimes b$.

(c) **Point-Value Representation**

A **point-value representation** of a polynomial $A(x)$ of degree-bound n is a set of n **point-value pairs**

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$$

such that all of the x_k are distinct and

$$y_k = A(x_k)$$

for $k = 0, 1, ..., n-1$. A polynomial has many different point-value representations, since we can use any set of n distinct points $x_0, x_1, ..., x_{n-1}$ as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straightforward, since all we have to do is select in n distinct points $x_0, x_1, ..., x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, ..., n-1$.

With Horner's method, evaluating a polynomial at n points takes time $\Theta(n^2)$, but we will see later that if we choose the points x_k cleverly, we can accelerate this computation to run in time $\Theta(n \log n)$.

The inverse of evaluation (*determining the coefficient form of a polynomial from a point-value representation*) is **interpolation**.

Theorem (*Uniqueness of an interpolating polynomial*). For any set $\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$ of n point-value pairs such that all the x_k values are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, ..., n-1$.

A fast algorithm for n-point interpolation is based on **Lagrange's formula:**

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)}$$

Thus, n-point evaluation and interpolation are well-defined inverse operations that transform between the coefficient representation of a polynomial and a point-value representation.

The point-value representation is quite convenient for many operations on polynomials. For addition, if $C(x) = A(x) + B(x)$, then $C(x_k) = A(x_k) + B(x_k)$ for any point x_k. More precisely, if we have a point-value representation for A,

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$$

and for B,

$$\{(x_0, y'_0), (x_1, y'_1), ..., (x_{n-1}, y'_{n-1})\}$$

then a point-value representation for C is

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), ..., (x_{n-1}, y_{n-1} + y'_{n-1})\}$$

Note that A and B are evaluated at the same n points.

Thus, the time to add two polynomials of degree-bound n in a point-value form is $\Theta(n)$.

Similarly, the point-value representation is convenient for multiplying polynomials. If $C(x) = A(x)B(x)$, then $C(x_k) = A(x_k)B(x_k)$ for any points x_k. We must face the problem, however, that degree $degree(C) = degree(A) + degree(B)$, if A and B are of degree-bound n, then C consists of n point-value pairs for each polynomial. When we multiply these together, we get n point-value pairs, but we need $2n$ pairs to interpolate a unique polynomial C of degree-bound $2n$. We must therefore begin with "extend" point-value representations for A and for B consisting of $2n$ point-value pairs each. Given an extended point-value representation for A.

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{2n-1}, y_{2n-1})\}$$

and for B

$$\{(x_0, y'_0), (x_1, y'_1), ..., (x_{2n-1}, y'_{2n-1})\}$$

then a point-value representation for C is

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), ..., (x_{2n-1}, y_{2n-1} y'_{2n-1})\}$$

Given two input polynomials in extended point-value form, we see that time to multiply them to obtain the point-value form of the results is $\Theta(n)$, much less than the time required to multiply polynomials in coefficient form.

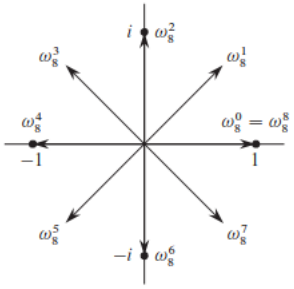
(d) **Complex roots of unity**

A **complex nth root of unity** is a complex number ω such that $\omega^n = 1$.

There are exactly n complex nth roots of unity: $e^{2\pi ik/n}$ for $k = 0, 1, ..., n-1$. If we use the definition of exponential of a complex number

$$e^{iu} = cos(u) + isin(u)$$

we can obtain the next picture,

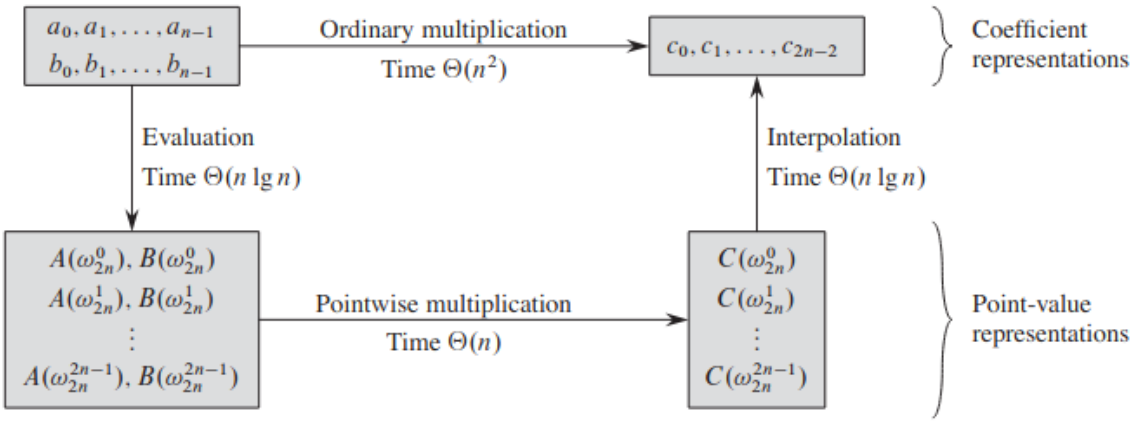


that shows that the n complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane, so the value $\omega_n = e^{2\pi i/n}$ is the **principal nth root of unity**; all other complex nth roots of unity are powers of ω_n. Then n complex nth roots of unity $\omega_n^0, \omega_n^1, ..., w_n^{n-1}$ form a group under multiplication.

(e) **Fast multiplication of polynomials in coefficient form**

Now, the question is if we can use the linear-time multiplication method for polynomials in point-value form (1c) to expedite polynomial multiplication in coefficient form. Well, the answer hinges on whether we can convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice versa (interpolate).

We can choose any points we want as evaluation points, but by choosing the evaluation points carefully, we can convert between representations in only $\Theta(n \log n)$ time, as we can see in the next diagram:



If we choose "complex roots of unity" as the evaluation points (1d), we can produce a point-value representation by taking the Discrete Fourier Transform (DFT) of a coefficient vector. We

can perform the inverse operation (interpolation) by taking the "inverse DFT" of point-value pairs, yielding a coefficient vector.

Another minor detail concerns degree-bounds. The product of two polynomials of degree-bound n is a polynomial of degree-bound $2n$. Before evaluating the input polynomials A and B, therefore, we first double their degree-bounds to $2n$ by adding n high-order coefficients of 0. Because the vectors have $2n$ elements, we use "complex $(2n)$th roots of unity", denoted by w_{2n}.

Given the FFT, we have the following $\Theta(n \log n)$ time procedure for multiplying two polynomials $A(x)$ and $B(x)$ of degree-bound n, where the input and output representations are in coefficient form. We assume that n is power of 2; we can always need this requirement by adding n high-order zero coefficients to each.

i. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-orden zero coefficients to each.

ii. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ by applying the FFT of order $2n$ on each polynomial. These representations contain the values of the two polynomials at the $(2n)$th roots of unity.

iii. *Point-wise multiply:* Compute a point-value representation for the polynomial $C(x) = A(x)B(x)$ by multiplying these values together point-wise. This representation contains the value of $C(x)$ at each $(2n)$th root of unity.

iv. *Interpolate:* Create the coefficient representation of the polynomial $C(x)$ by applying the FFT on $2n$ point-value pairs to compute the inverse DFT.

(f) **DFT and FFT**

We have seen many things about DFT and FFT in the previous assignment.

Recall that we wish to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound n at $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$. We assume that A is given in coefficient form $a = (a_0, a_1, ..., a_{n-1}$; so let us define the results y_k for $k = 0, 1, ..., n-1$, by

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj} \tag{2}$$

The vector $y = (y_0, y_1, ..., y_{n-1})$ is the DFT of the coefficient vector $a = (a_0, a_1, ..., a_{n-1}$. We also write $y = DFT_n(a)$.

In the last assignment we saw that FFT is a faster way to compute DFT, so it takes $\Theta(n \log n)$. In this case we will use a bit different recursive algorithm from the one we used in the first assignment:

```
RECURSIVE-FFT(a)
 1  n = a.length                    // n is a power of 2
 2  if n == 1
 3      return a
 4  ω_n = e^{2πi/n}
 5  ω = 1
 6  a^[0] = (a_0, a_2, ..., a_{n-2})
 7  a^[1] = (a_1, a_3, ..., a_{n-1})
 8  y^[0] = RECURSIVE-FFT(a^[0])
 9  y^[1] = RECURSIVE-FFT(a^[1])
10  for k = 0 to n/2 - 1
11      y_k = y_k^[0] + ω y_k^[1]
12      y_{k+(n/2)} = y_k^[0] - ω y_k^[1]
13      ω = ω ω_n
14  return y                        // y is assumed to be a column vector
```

To determinate the running time of the procedure, we note that exclusive of the recursive calls, each invocation takes time $\Theta(n)$, where n is the length of the input vector. The recurrence for the running time is therefore

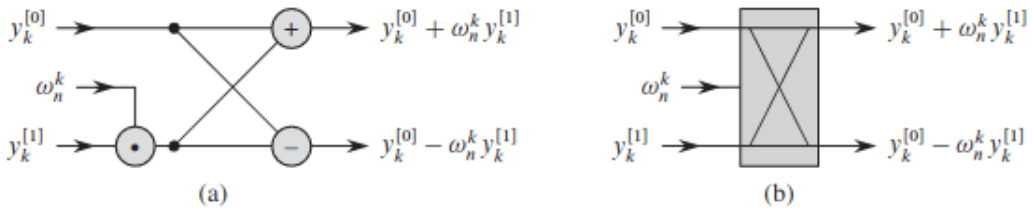
$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

Thus, we can evaluate a polynomial of degree-bound n at the complex nth roots of unity in time $\Theta(n \log n)$ using FFT.

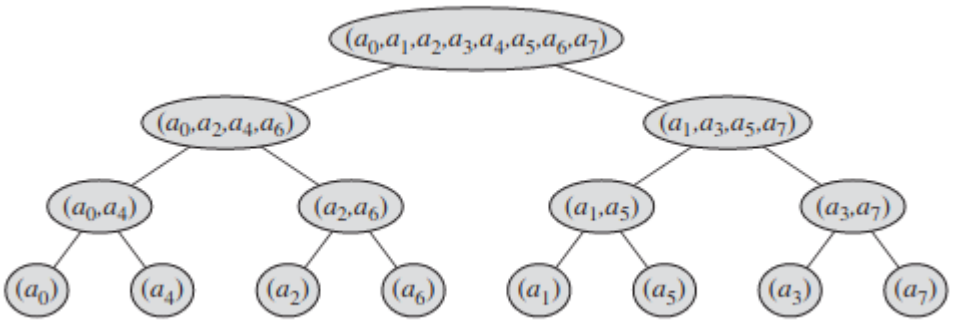
As the last time, we can do an **iterative implementation**. We first note that the *for loop* of lines 10-13 involves computing the value $\omega_n^k y_k^{[1]}$ twice, so we can change the loop to compute it only once, storing it in a temporary variable t:

```
for k = 0 to n/2 - 1
    t = ω y_k^[1]
    y_k = y_k^[0] + t
    y_{k+(n/2)} = y_k^[0] - t
    ω = ω ω_n
```

The operation in this loop, multiplying the twiddle factor $\omega = \omega_n^k$ by $y_k^{[1]}$, storing the product into t, and adding and subtracting t from $y_k^{[0]}$, is known as a **butterfly operation** and is shown schematically as:



Now, suppose a initial call of $n = 8$. The trees has one node for each call of the procedure, labeled by the corresponding input vector. Each invocation makes two recursive calls, unless it has received 1-element vector. The first call appears in the left child, and the second appears in the right child.



Looking at the tree, we observe that if we could arrange the elements of the initial vector a into the order in which they appear in the leaves, we could trace the execution of the recursive procedure, but bottom up instead. First, we take the elements in pairs, compute the DFT of each pair using one butterfly operation, and replace the pair with its DFT. The vector then holds $n/2$ 2-element DFTs. Taking this in pairs and compute the DFT of the four vector elements they come from by executing two butterfly operations, replacing two 2-element DFTs with one 4-element DFT. Then, the vector holds $n/4$ 4-element DFTs. We continue in this manner until the vector holds two $(n/2)-$element DFTs, which we combine using $n/2$ butterfly operations into final n-element DFT.

To turn this bottom-up approach into code, we use an array $A[0, ..., n-1]$ that initially holds the elements of the input vector a in the order in which they appear in the leaves of the tree. Because we have to combine DFTs on each level of the tree, we introduce a variable s to count the levels, ranging from 1 to $\log n$. Therefore, the algorithm has the following structure:

```
1  for s = 1 to lg n
2      for k = 0 to n − 1 by 2^s
3          combine the two 2^(s−1)-element DFTs in
               A[k..k + 2^(s−1) − 1] and A[k + 2^(s−1)..k + 2^s − 1]
               into one 2^s-element DFT in A[k..k + 2^s − 1]
```

We can express the body of the loop as more precise pseudo-code. We copy the *for loop* from the recursive algorithm, identifying $y^{[0]}$ with $A[k...k+2^{s-1}-1]$ and $y^{[1]}$ with $A[k+2^{s-1}...k+2^s-1]$. The twiddle factor used in each butterfly operation depends on the value of s; it is a power of ω_m, where $m = 2^s$. We introduce another temporary variable u that allows us to perform the butterfly operation in place. Also, we need an auxiliary procedure to copy vector a into array A in the initial order in which we need the values.

```
BIT-REVERSE-COPY(a, A)
1  n = a.length
2  for k = 0 to n − 1
3      A[rev(k)] = a_k
```

When we replace line 3 of the overall structure by the loop body, we get the following pseudo-code:

```
ITERATIVE-FFT(a)
 1  BIT-REVERSE-COPY(a, A)
 2  n = a.length                    // n is a power of 2
 3  for s = 1 to lg n
 4      m = 2^s
 5      ω_m = e^{2πi/m}
 6      for k = 0 to n − 1 by m
 7          ω = 1
 8          for j = 0 to m/2 − 1
 9              t = ω A[k + j + m/2]
10              u = A[k + j]
11              A[k + j] = u + t
12              A[k + j + m/2] = u − t
13              ω = ω ω_m
14  return A
```

The call to **BIT-REVERSE-COPY** procedure runs in $O(n \log n)$ since we iterate n times and can reverse an integer between 0 and $n-1$, with $\log n$ bits, in $O(\log n)$ time. To complete the proof that **ITERATIVE-FFT** runs in time $\Theta(n \log n)$, we show that $L(n)$, the number of times the body of the innermost loop execute is $\Theta(n \log n)$. The *for loop* of lines 6-13 iterates $n/m = n/2^s$ times for each value of s, and the innermost loop of lines 8-13 iterates $m/2 = 2^{s-1}$ times. Thus,

$$L(n) = \sum_{s=1}^{\log n} \frac{n}{2^s} 2^{s-1} = \sum_{s=1}^{\log n} \frac{n}{2} = \Theta(n \log n)$$

(g) **Interpolation at the complex root of unity**

We now complete the polynomial multiplication scheme by showing how to interpolate the complex roots of unity by a polynomial, which enables us to convert from point-value form back to coefficient form. We interpolate by writing the DFT as a matrix equation and then looking at the form of the matrix inverse.

We can write the DFT as the matrix product $y = V_n a$ where V_n is a Vandermonde matrix containing the appropriate powers of ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

The (k, j) entry of V_n is ω_n^{kj}, for $j, k = 0, 1, ..., n-1$. The exponents of the entries of V_n form as multiplication table.

For the inverse operation, which we write as a $DFT_n^{-1}(y)$, we proceed by multiplying y by the matrix V_n^{-1}.

Given the inverse matrix V_n^{-1}, we have that $DFT_n^{-1}(y)$ is given by

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{3}$$

for $j = 0, 1, ..., n-1$. By comparing equation 2 and 3, we see that by modifying the FFT algorithm to switch the roles of a and y, replace ω_n by ω_n^{-1} and divide each element of the result by n, we compute the inverse DFT in $\Theta(n \log n)$ time as well.

Convolution Theorem. for any two vectors a and b of length n, with n is a power of 2,

$$a \otimes b = DFT_{2n}^{-1}(DFT_{2n}(a) \cdot DFT_{2n}(b)) \tag{4}$$

where the vectors a and b are padded with 0s to length $2n$.

2. Description of implementation

(a) Implementation of the Algorithm

We are going to start with the recursive implementation. Before it, we need to import:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.fft import fft, ifft
```

Now we follow the steps explained in 1f:

```
def RECURSIVE_FFT(A,m,w):
    if m==1:
        return A
    else:
        A_even = A[::2]
        A_odd = A[1::2]
        F_even = RECURSIVE_FFT(A_even,m/2,w**2)
        F_odd = RECURSIVE_FFT(A_odd,m/2,w**2)
        F = [0]*int(m)
        x = 1
        for j in range (0,int(m/2)):
            F[j] = F_even[j] + x*F_odd[j]
            F[int(j+m/2)] = F_even[j] - x*F_odd[j]
            x = x * w
        return F
```

I have used some extra parameters, like m, that is the length of the array, and w for indicating the roots (so we can use the same code for the FFT and the inverse FFT).

Now, we can create the final algorithm:

```
def RECURSIVE_CONVOLUTION(A,B):
    if (len(A)!=len(B)) or np.log2(len(A))%1 > 0:
        raise ValueError("Both coefficient representations must have same
size and be power of 2")

    # Double degree bound
    n = len(A)
    for i in range (0,n):
        A.append(0)
        B.append(0)
```

```
    n = len(A)
    wn = np.exp((-2j*np.pi)/n)

    # Evaluate using FFT
    FFTA = RECURSIVE_FFT(A,n,wn)
    FFTB = RECURSIVE_FFT(B,n,wn)

    # Point-Wise Multiplication
    FC = []
    for i in range (0,n):
        FC.append(FFTA[i]*FFTB[i])

    # Interpolate
    F = RECURSIVE_FFT(FC,n,wn**(-1))
    for i in range (0,n):
        F[i] = F[i]*(1/n)

    return F
```

Now, lets go with the iterative version. First, we need to implement the auxiliary method **BIT-REVERSE-COPY**, that uses a bit-reversal permutation, so:

```
def REV(n,bits):
    result = 0
    for i in range(int(np.log2(bits))):
        result <<= 1
        result |= n & 1
        n >>= 1
    return result
```

```
def BIT_REVERSE_COPY(a,A):
    n = len(a)
    k = 0
    while k<=n-1:
        A[REV(k,n)] = a[k]
        k+=1
```

Now, we can implement the **ITERATIVE-FFT** and its inverse (is only changing the roots from $\omega_n \rightarrow \omega_n^{-1}$:

```
def ITERATIVE_FFT(a):
    A = [0]*len(a)
    BIT_REVERSE_COPY(a,A)
    n = len(a)

    s = 1
    while s <= np.log2(n):
        m = pow(2,s)
        wm = np.exp((-2j*np.pi)/m)
        k = 0
        while k <= n-1:
            w = 1
            j = 0
            while j <= m/2-1:
                t = w*A[int(k+j+m/2)]
                u = A[int(k+j)]
                A[int(k+j)] = u+t
                A[int(k+j+m/2)] = u-t
                w = w*wm
                j+=1
            k+=m
        s+=1
    return A
```

```
def ITERATIVE_INVERSE_FFT(a):
    A = [0]*len(a)
    BIT_REVERSE_COPY(a,A)
    n = len(a)

    s = 1
    while s <= np.log2(n):
        m = pow(2,s)
        wm = np.exp((2j*np.pi)/m)
        k = 0
        while k <= n-1:
            w = 1
            j = 0
            while j <= m/2-1:
                t = w*A[int(k+j+m/2)]
                u = A[int(k+j)]
                A[int(k+j)] = u+t
                A[int(k+j+m/2)] = u-t
                w = w*wm
                j+=1
            k+=m
        s+=1
    return A
```

So, the final algorithm is:

```
def ITERATIVE_CONVOLUTION(A,B):
    if (len(A)!=len(B)) or np.log2(len(A))%1 > 0:
        raise ValueError("Both coefficient representations must have same
size and be power of 2")

    # Double degree bound
    n = len(A)
    for i in range (0,n):
        A.append(0)
        B.append(0)

    # Evaluate using FFT
    FFTA = ITERATIVE_FFT(A)
    FFTB = ITERATIVE_FFT(B)

    n = len(A)

    # Point-Wise Multiplication
    FC = []
    for i in range (0,n):
        FC.append(FFTA[i]*FFTB[i])

    # Interpolate
    F = ITERATIVE_INVERSE_FFT(FC)
    for i in range (0,n):
        F[i] = F[i]*(1/n)

    return F
```

(b) **Implementation of the unit test**

We are going to use a similar method to the one we used in the previous assignment. In **numpy** exists the function **numpy.polymul(A,B)** that return the convolution of A and B, so:

```
for i in range (10):
    A = np.random.random(1024)
```

```
        A = A.tolist()
        B = np.random.random(1024)
        B = B.tolist()
        S = np.polymul(A,B)
        SR = RECURSIVE_CONVOLUTION(A,B)
        SI = ITERATIVE_CONVOLUTION(A,B)
        print("Case", i+1, ":")
        print("\t","Recursive Solution:", np.allclose(SR[0:len(S)],S,rtol=1e-05,
    atol=1e-08,equal_nan=False))
        print("\t","Iterative Solution:", np.allclose(SI[0:len(S)],S,rtol=1e-05,
    atol=1e-08,equal_nan=False))
```

As the last time. we generate a random array, but this time, because we use the method *append* in the convolution, we need to convert it to a list.

3. **Description of unit tests and computational complexity**

   (a) **Unit test** We have just seen the idea in 2b

   (b) **Parameter N of the algorithm**

   In the last assignment and in 1f we have seen that we can compute FFT in $\Theta(N \log N)$ time. Also, the inverse FFT (*same algorithm but using* ω_n^{-1}) is computed in $\Theta(N \log N)$ time. Finally, the product in point-wise form takes $\Theta(n)$ time, so we have a $\Theta(N \log N)$ time procedure for multiplying two polynomials as we see in 1e.

   As the last time, we will see that the iterative implementation is faster then the recursive one, but it is still slower than the numpy implementation.

   (c) **Some examples**

   Lets prepare some inputs data. We will generate them randomly:

```
    A1 = np.random.random(2048) # 2^11
    B1 = np.random.random(2048)
    A1 = A1.tolist()
    B1 = B1.tolist()
    A2 = np.random.random(1024) # 2^10
    B2 = np.random.random(1024)
    A2 = A2.tolist()
    B2 = B2.tolist()
    A3 = np.random.random(8192) # 2^13
    B3 = np.random.random(8192)
    A3 = A3.tolist()
    B3 = B3.tolist()
```

We can prepare the following script:

```
    def unitest(A,B):
        S = np.polymul(A,B)
        SR = RECURSIVE_CONVOLUTION(A,B)
        SI = ITERATIVE_CONVOLUTION(A,B)
        print ("Input: \n", "A:", np.asarray(A), "\n", "B:", np.asarray(B))
        print("\t","Recursive Solution:", np.allclose(SR[0:len(S)],S,rtol=1e-05,
    atol=1e-08,equal_nan=False))
        print("\t","Iterative Solution:", np.allclose(SI[0:len(S)],S,rtol=1e-05,
    atol=1e-08,equal_nan=False))
        print("\n Output: \n")
        print("Numpy: \n", np.asarray(S))
        print("Recursive: \n", np.asarray(SR))
        print("Iterative: \n", np.asarray(SI))
```

We get the following results:

```
unitest(A1,B1)

Input:
 A: [0.74567901 0.42716859 0.83864014 ... 0.         0.         0.        ]
 B: [0.25796635 0.2640148  0.448177   ... 0.         0.         0.        ]
        Recursive Solution: True
        Iterative Solution: True

 Output:

Numpy:
 [0.19236009 0.30706542 0.66331594 ... 0.81945899 0.39836784 0.63093124]
Recursive:
 [1.92360090e-01+2.19485367e-12j 3.07065416e-01+2.37576061e-12j
 6.63315944e-01+2.54853844e-12j ... 3.98367836e-01+5.45494595e-12j
 6.30931243e-01+5.65826654e-12j 1.99833039e-10+4.61609918e-12j]
Iterative:
 [1.92360090e-01-4.24670975e-12j 3.07065416e-01-4.19110210e-12j
 6.63315944e-01-4.17457674e-12j ... 1.13043463e-11-1.30006762e-11j
 1.13763443e-11-1.30207429e-11j 1.14209843e-11-1.31035742e-11j]

unitest(A2,B2)

Input:
 A: [0.46312433 0.16717358 0.38295353 ... 0.         0.         0.        ]
 B: [0.10389584 0.56885436 0.2258708  ... 0.         0.         0.        ]
        Recursive Solution: True
        Iterative Solution: True

 Output:

Numpy:
 [0.04811669 0.28081894 0.23949096 ... 1.34739651 1.12482685 0.79462368]
Recursive:
 [ 4.81166907e-02-7.74510664e-13j  2.80818935e-01-8.52423422e-13j
  2.39490961e-01-7.84187912e-13j ...  1.12482685e+00-2.24130722e-12j
  7.94623676e-01-2.14606307e-12j -5.65307801e-11-1.52979166e-12j]
Iterative:
 [4.81166907e-02-3.54249095e-12j 2.80818935e-01-3.65692016e-12j
 2.39490961e-01-3.53320400e-12j ... 2.00695016e-12-3.44816456e-12j
 1.92795779e-12-3.37623799e-12j 1.98772371e-12-3.49074296e-12j]

unitest(A3,B3)

Input:
 A: [0.05665022 0.4304864  0.73375027 ... 0.         0.         0.        ]
 B: [0.25626075 0.50503733 0.04738541 ... 0.         0.         0.        ]
        Recursive Solution: True
        Iterative Solution: True

 Output:

Numpy:
 [0.01451723 0.13892725 0.4081275  ... 1.05071733 0.37688471 0.40818471]
Recursive:
 [ 1.45172260e-02-5.13906626e-13j  1.38927244e-01+5.09954973e-13j
  4.08127492e-01+4.74030169e-12j ...  3.76884703e-01-7.69462218e-12j
  4.08184701e-01-9.16294990e-13j -5.23459676e-09+2.41890829e-12j]
Iterative:
 [ 1.45172284e-02+3.75023448e-10j  1.38927246e-01+3.71747752e-10j
  4.08127495e-01+3.77760109e-10j ... -4.10948636e-10+5.30681301e-10j
 -4.13139939e-10+5.37767841e-10j -4.09338953e-10+5.28422001e-10j]
```

If we want to use a higher value of N we should consider deleting the recursive implementation for the script, because it computational time is elevated, so lets see what happens now:

```
A4 = np.random.random(32768) # 2^15
B4 = np.random.random(32768)
A4 = A4.tolist()
B4 = B4.tolist()
A5 = np.random.random(131072) # 2^17
B5 = np.random.random(131072)
A5 = A5.tolist()
B5 = B5.tolist()

def unitest2(A,B):
    S = np.polymul(A,B)
    SI = ITERATIVE_CONVOLUTION(A,B)
    print ("Input: \n", "A:", np.asarray(A), "\n", "B:", np.asarray(B))
    print("\t","Iterative Solution:", np.allclose(SI[0:len(S)],S,rtol=1e-05,
atol=1e-08,equal_nan=False))
    print("\n Output: \n")
    print("Numpy: \n", np.asarray(S))
    print("Iterative: \n", np.asarray(SI))
```

We get:

```
unitest2(A4,B4)

Input:
 A: [0.67190028 0.07042499 0.86666143 ... 0.         0.         0.        ]
 B: [0.34024541 0.84426884 0.56223703 ... 0.         0.         0.        ]
        Iterative Solution: True

 Output:

Numpy:
 [0.22861099 0.59122625 0.73210242 ... 0.         0.         0.        ]
Iterative:
 [2.28610989e-01-4.23653132e-09j 5.91226255e-01-4.16360065e-09j
 7.32102423e-01-4.24956610e-09j ... 3.69597537e-11-8.93674448e-10j
 5.58507892e-11-9.62360232e-10j 3.51363355e-11-8.81404123e-10j]

unitest2(A5,B5)

Input:
 A: [0.61019576 0.89804163 0.8757521  ... 0.         0.         0.        ]
 B: [0.35731999 0.12891496 0.47110385 ... 0.         0.         0.        ]
        Iterative Solution: True

 Output:

Numpy:
 [0.21803514 0.39955158 0.7161603  ... 0.97705647 0.54336369 0.53004141]
Iterative:
 [2.18035160e-01+2.08977720e-10j 3.99551603e-01+2.97004785e-11j
 7.16160319e-01+2.13914363e-10j ... 5.43363786e-01-1.44650086e-11j
 5.30041500e-01+1.49538974e-10j 9.06002242e-08-6.99891215e-11j]
```

4. **Results**

From now on we are going to work only with the most optimal algorithm, which is the iterative implementation. For this part we are going to take 15 different values of N and we are going to run the algorithm 10 times for each one. Then, we are going to calculate the average time and the standard deviation of those data.

Lets prepare a script for this:

```
from timeit import default_timer as timer

N = 2
for i in range (15):
    A = np.random.random(N)
    B = np.random.random(N)
    A = A.tolist()
    B = B.tolist()

    times = []
    for j in range (10):
        start = timer()
        S = ITERATIVE_CONVOLUTION(A,B)
        end = timer()
        time = end-start
        times.append(time)

    average = np.mean(times)
    deviation = np.std(times)

    print("Size of N", N)
    print("Average Time:", average)
    print("Standard Deviation:", deviation)
    print("\n")
    N*=2
```

We have the following output (*Not all*):

```
Size of N 2
Average Time: 0.030400244900101826
Standard Deviation: 0.06195259833138895


Size of N 4
Average Time: 0.019209442299779767
Standard Deviation: 0.031180883324244475


Size of N 8
Average Time: 0.04163716730017768
Standard Deviation: 0.06738677297284106
```

Lets make now a table with all the data:

No	N	Average Time	Standard Deviation
1	2	0.030400244900101826	0.06195259833138895
2	4	0.019209442299779767	0.031180883324244475
3	8	0.04163716730017768	0.06738677297284106
4	16	0.08927487399987513	0.14539602418841532
5	32	0.19235024299987344	0.31183207850056643
6	64	0.401094406899756	0.6389681834724148
7	128	0.8414097208002204	1.3486561014390892
8	256	1.8357663106000472	2.959723901628359
9	512	3.7138247564999802	5.920570433751573
10	1024	7.9705723978999234	12.781655664302042
11	2048	17.189995763000024	27.810203018716045
12	4096	36.173929070099895	56.71719479293871

It is taking so much time, but why? . That's because in the convolution code we have 2 extra-for, one for get a double degree-bound and the other for multiplying the interpolation by $1/n$. Lets make some adjust them. We are going to do the double degree-bound out of the main function and the for that multiply the ifft by $1/n$ (*We add this multiplication in the ITERATIVE-INVERSE-FFT at the end*),so:

```
    def ITERATIVE_CONVOLUTION2(A,B):
        # Evaluate using FFT
        FFTA = ITERATIVE_FFT(A)
        FFTB = ITERATIVE_FFT(B)

        n = len(A)

        # Point-Wise Multiplication
        FC = []
        for i in range (0,n):
            FC.append(FFTA[i]*FFTB[i])

        # Interpolate
        F = ITERATIVE_INVERSE_FFT(FC)

        return F

    N = 2
    for i in range (15):
        A = np.random.random(N)
        B = np.random.random(N)
        A = A.tolist()
        B = B.tolist()
```

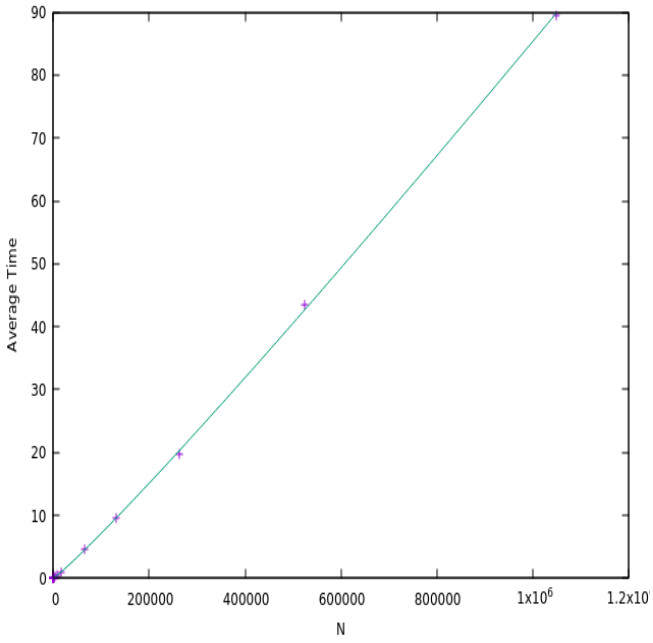
```
# Double degree bound
n = len(A)
for i in range (0,n):
    A.append(0)
    B.append(0)

times = []
for j in range (10):

    start = timer()
    S = ITERATIVE_CONVOLUTION2(A,B)
    end = timer()
    time = end-start
    times.append(time)

average = np.mean(times)
deviation = np.std(times)

print("Size of N", N)
print("Average Time:", average)
print("Standard Deviation:", deviation)
print("\n")
N*=2
```

Now we get more decent data:

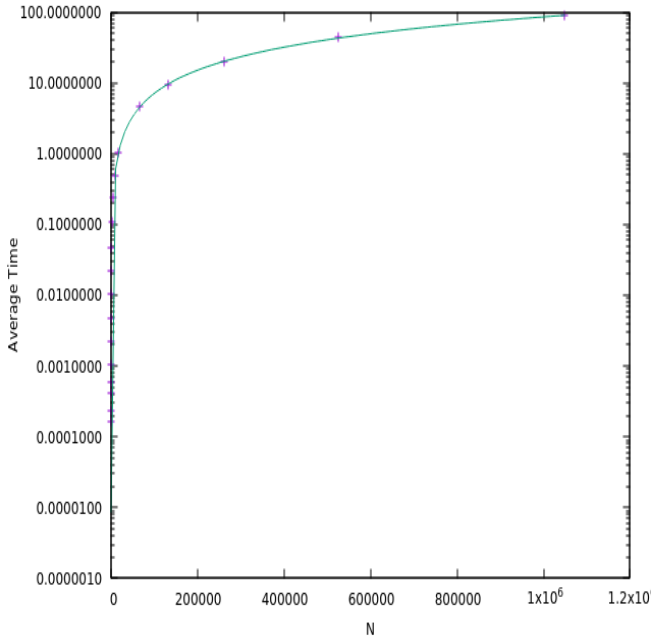
No	N	Average Time	Standard Deviation
1	2	0.00041064500001084523	0.0010467618477896581
2	4	0.0001625653000701277	5.869787404954675e-05
3	8	0.0002329482000277494	2.0856667959705398e-05
4	16	0.0005800605000331416	0.00015750598663067378
5	32	0.0010646675002135453	9.834163701536125e-05
6	64	0.0021999719998348154	0.00015993857597299007
7	128	0.004629416200168634	4.259312863119057e-05
8	256	0.010245481400033896	0.0001005938536301305
9	512	0.02230287180009327	0.00031327570333500424
10	1024	0.047172959499766874	0.0007432197061259509
11	2048	0.10659039850006594	0.0087655697552925
12	4096	0.2366943951000394	0.017638536812417473
13	8192	0.48730179309995947	0.03411094753141976
14	16384	1.0264710055999786	0.03322173797039719
15	65536	4.562282113999845	0.19401084234808288
16	131072	9.607307113000115	0.25822360335246625
17	262144	19.811323657600134	0.4137360866685898
18	524288	43.526238708100024	1.2957055359545409
19	1048576	89.50924947950007	3.611458089156712

For generating the graphics I will use **gnuplot**. I have a "data.dat" document with the column N, Average Time and Standard Deviation, so we proceed as follows:

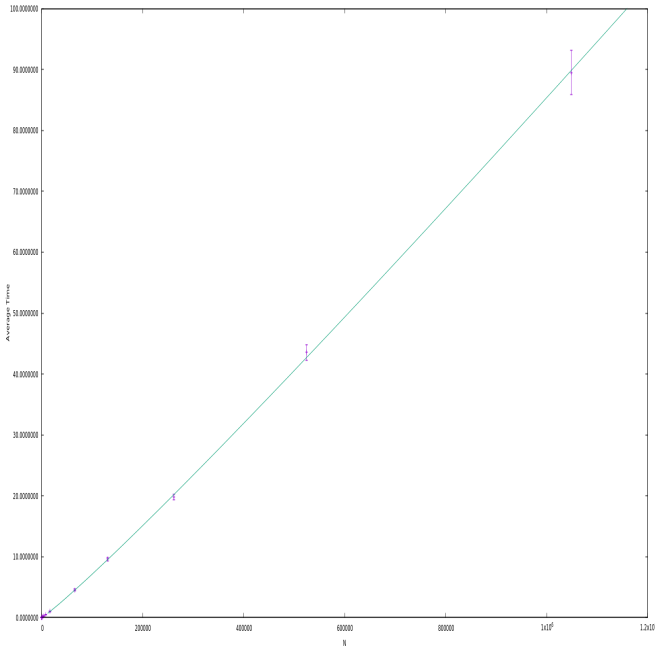
```
gnuplot> set xlabel "N"
gnuplot> set ylabel "Average Time"
gnuplot> unset key
gnuplot> f(x) = a*x*log(x)
gnuplot> fit f(x) "data.dat" u 1:2 via a
iter      chisq       delta/lim  lambda   a
 0 2.7262113585e+14   0.00e+00  3.79e+06    1.000000e+00
 1 6.8155283964e+11  -3.99e+07  3.79e+05    5.000587e-02
 2 1.8859823505e+05  -3.61e+11  3.79e+04    3.248185e-05
 3 1.0102884752e+00  -1.87e+10  3.79e+03    6.180201e-06
 4 1.0102832510e+00  -5.17e-01  3.79e+02    6.180062e-06
iter      chisq       delta/lim  lambda   a

After 4 iterations the fit converged.
final sum of squares of residuals : 1.01028
rel. change during last iteration : -5.17107e-06

degrees of freedom    (FIT_NDF)                        : 18
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.236911
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0561268

Final set of parameters            Asymptotic Standard Error
=======================            =========================

a                 = 6.18006e-06    +/- 1.435e-08    (0.2322%)

gnuplot> plot "data.dat" u 1:2, f(x)
gnuplot> set log y
gnuplot> replot
gnuplot> plot "data.dat" with yerrorbars, f(x)
gnuplot> unset log y
gnuplot> replot
```



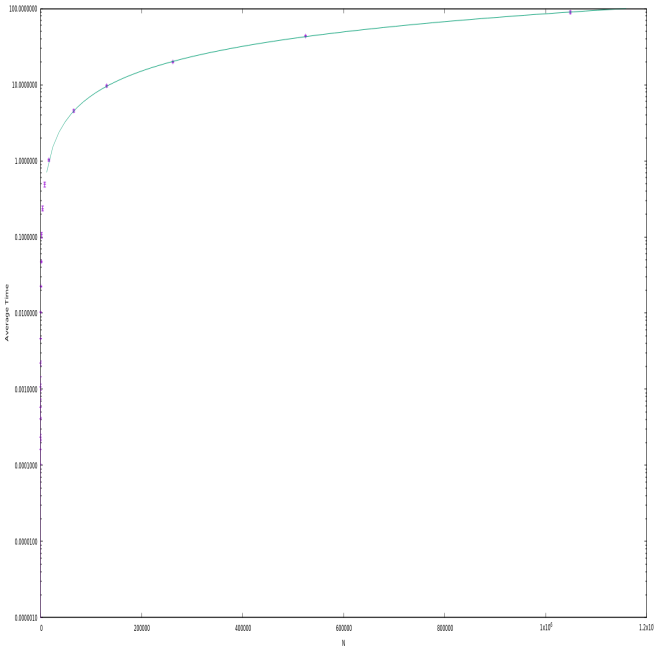
(a) Results Plot



(b) Log Scale Y



(a) Results Plot With Error



(b) Log Scale Y With Error

So, as we can see, the points fit fairly well to the complexity curve calculated above (*By using gnuplot toll the adjustment is quite fast and easy to calculate*). The only drawback of the previous adjustment is that we have very few data to perform the adjustment and these are far apart as N grows as a consequence of the fact that our algorithm requires that the size of the data be a power

5. **Conclusion**

We have seen throughout this work the importance of optimizing algorithms. We have implemented the convolution of 2 polynomials of degree-bound n, that has complexity $\Theta(n^2)$, but, by using FFT and some maths we can get an $\Theta(N \log N)$ complexity algorithm. Also, we have made 2 different implementation of this, one recursive and other iterative and, as usually, the iterative one is faster, but, the numpy implementation is still the fastest, because it use *FORTRAN*.

Finally, throughout the work we have seen different tools that have helped us in our goal, such as LaTex, Gnuplot or JupyterNotebooks, which are very useful and used in the scientific and research world.

References

[1] FFT for polynomials

[2] Polynomials and FFT

[3] How to do fast polynomial multiplication with FFT

[4] Advanced Algorithm Design and Analysis: Polynomials and the FFT

[5] Understanding Fast Fourier Transform from scratch — to solve Polynomial Multiplication.

[6] Numpy

[7] Bit Reversal

[8] Bit Reversal II