DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Report on

"Fast Polynomial Multiplication with DFT/FFT implementation, RSA Encryption, Image compression"

Master of Technology
in
Computer Science & Engineering

UE22CS644A – Topics In Advanced Algorithm

Submitted by:

MOOKAMBIKA S(PES1PG22CS027) **SHALINI T(PES1PG22CS045)**

Mookambika · S Shalini .T

2ND Semester M.tech

INTRODUCTION

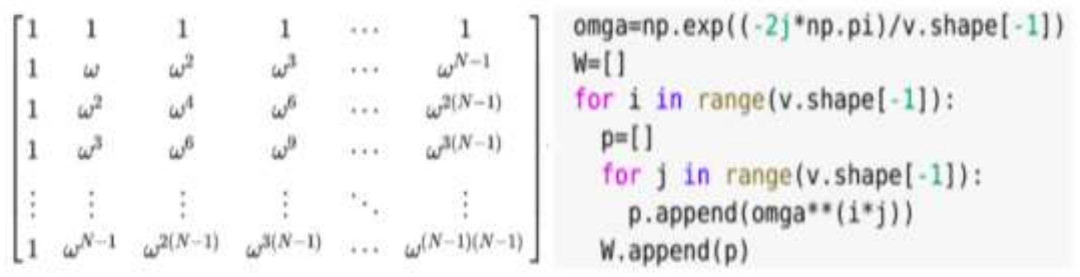
This project focuses on three primary objectives: accelerating polynomial multiplication, ensuring secure data transmission, and achieving efficient image compression. To prioritize essential elements, it leverages 1-D and 2-D Fourier Transforms to expedite polynomial multiplication and streamline image compression while disregarding less critical details. Additionally, the project employs multiple keys for data encryption and decryption, ensuring data security through RSA Encryption. It aims to assess the computational efficiency of these techniques, contributing to the advancement of fields like signal processing, cryptography, and image processing. This endeavor utilizes Python libraries.

1. Implement 1-D DFT, on coefficient vectors of two polynomials A(x), B(x) by multiplication of Vander-Monde matrix. (O(n2) - Complexity

We have successfully implemented the Discrete Fourier Transform (DFT) using a specialized matrix known as the DFT matrix. When this transformation matrix is applied to a signal through matrix multiplication, it yields the Fourier transform of that signal. This process is accomplished through the following components:

- Template for DFT Matrix: We define a template for the DFT matrix.
- Code for DFT Matrix: We provide the code to generate the DFT matrix and apply it to an array of coefficients to compute the Fourier transform.

The DFT matrix is employed to perform the transformation on the coefficient array, ultimately producing the Fourier transform of the signal.



$$\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

```
omga=np.exp((-2j*np.pi)/v.shape[-1])
W=[]
for i in range(v.shape[-1]):
  p=[]
  for j in range(v.shape[-1]):
    p.append(omga**(i*j))
  W.append(p)
```

Fig: Template of Vandermonde matrix

The resulting dft_coeff_A and dft_coeff_B arrays represent the DFT coefficients of A(x) and B(x). This process allows for the transformation of polynomial coefficients into frequency domain representations, facilitating various signal processing and mathematical operations.

2. Implement 1-D FFT on the same vectors, of A(x) and B(x). Ensure above two steps produce same results. (O(n logn) – Complexity)

Implementing a 2D Fast Fourier Transform (FFT) involves applying the 1D FFT to both rows and columns of a 2D array. This is commonly used in image processing and signal analysis to transform data from the spatial domain to the frequency domain.

The objective is to implement the Fast Fourier Transform (FFT) algorithm for 1-D signals, aiming for improved computational efficiency with O(n log n) complexity compared to the previous Discrete Fourier Transform (DFT) method.

The code works as follows:

It defines a versatile 1-D FFT function capable of handling both 1-D and 2-D arrays.

- The Cooley-Tukey Radix-2 Decimation in Time (DIT) algorithm is employed, which operates in O(N log N) time, making it substantially faster than the straightforward DFT, especially for larger input sizes.

- The "array of arrays" decorator is used to make the FFT function flexible. If provided with a 2-D array, it applies the FFT to each row and returns a list of results. For 1-D arrays, it directly computes the FFT.

- The FFT algorithm takes the coefficients of a polynomial A(x) and divides them into two sets: even-indexed coefficients and odd-indexed coefficients. This effectively creates two new polynomials, each having half the degree of the original.

- It multiplies all coefficients by x^2, effectively increasing their degrees by a factor of 2.

- The entire polynomial is multiplied by x, effectively increasing its degree by 1. After these operations, two modified polynomials are obtained.

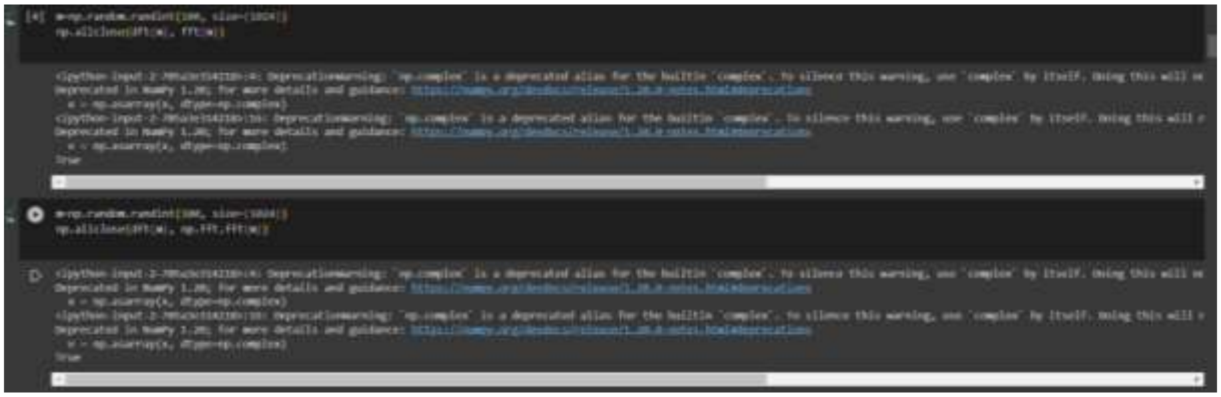
Finally, the results from these two modified polynomials are combined to obtain the DFT of the original polynomial A(x).

In essence, the FFT algorithm optimizes the computation of the Fourier Transform by breaking down the problem into smaller parts, exploiting symmetry and reducing redundant calculations. This significantly enhances efficiency compared to the traditional DFT approach.

Checking the correctness of our Implementation

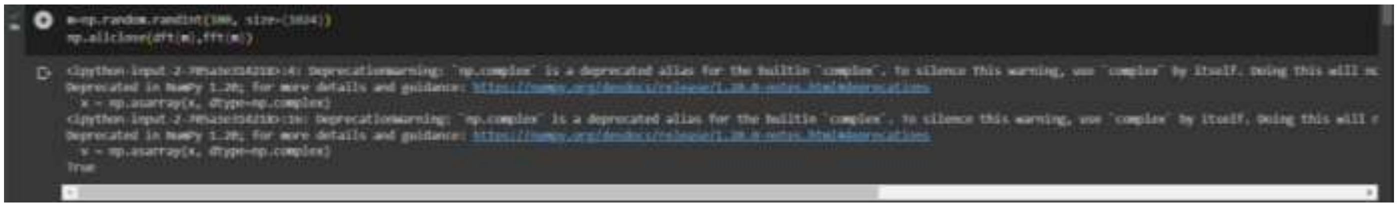
The correctness of our implementation is check against the FFT function from the NumPy package. We are verifying whether our custom FFT implementation produces the same results as the well-established FFT implementation in NumPy. If the results match, it suggests that your FFT implementation is correct and produces accurate results.

As the results obtained are matching between our custom FFT implementation and the NumPy



```
[4] m=np.random.randint(100, size=(1024))
    np.allclose(dft(m), fft(m))

<ipython-input-2-[illegible]>:4: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will n
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x = np.asarray(x, dtype=np.complex)
<ipython-input-2-[illegible]>:16: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will r
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x = np.asarray(x, dtype=np.complex)
True
```

```
m=np.random.randint(100, size=(1024))
np.allclose(dft(m), np.fft.fft(m))

<ipython-input-2-[illegible]>:4: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will n
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x = np.asarray(x, dtype=np.complex)
<ipython-input-2-[illegible]>:16: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will r
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x = np.asarray(x, dtype=np.complex)
True
```



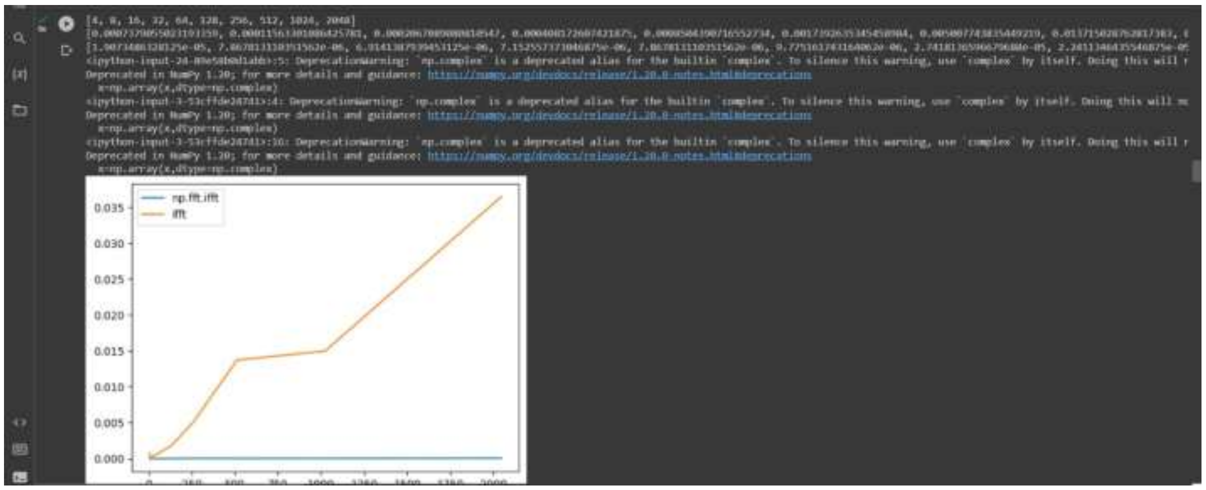
```
m=np.random.randint(100, size=(1024))
np.allclose(dft(m),fft(m))

<ipython-input-2-[illegible]>:4: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will no
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x = np.asarray(x, dtype=np.complex)
<ipython-input-2-[illegible]>:16: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will r
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x = np.asarray(x, dtype=np.complex)
True
```

FFT function, it provides us confidence in the correctness and reliability of our custom code. It

demonstrates that our code accurately performs Fourier transformations according to established mathematical principles.

Running time of the three 1D fourier transform functions

Now that we know our function works, let's check how long it takes to transform vectors of sizes 4, 8, 16, 32, 64, ... 1024 and 2048



```
[4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]
[0.0007379055023193359, 0.00011563301086425781, 0.0002067089080810547, 0.000408172607421875, 0.0008504390716552734, 0.0017392635345458984, 0.005007743835449219, 0.013715028762817383, [illegible]
[1.9073486328125e-05, 7.867813110351562e-06, 6.9141387939453125e-06, 7.152557373046875e-06, 7.867813110351562e-06, 9.775161743164062e-06, 2.7418136596679688e-05, 2.2411346435546875e-05
<ipython-input-24-09e58b0d1ab6>:5: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will r
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
<ipython-input-3-53cffde28741>:4: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will no
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
<ipython-input-3-53cffde28741>:10: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will r
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
```

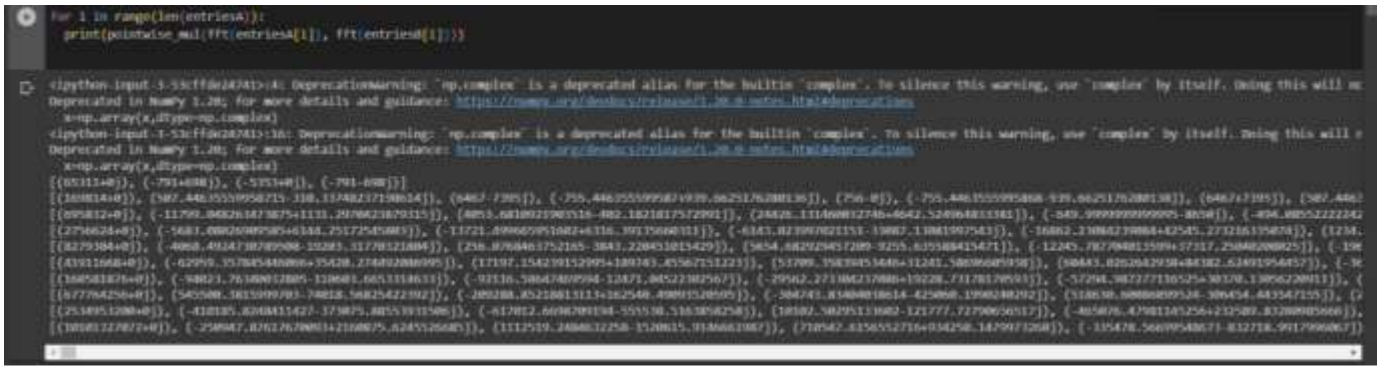
From the graph it is clear that fft is much faster than dft [n/log(n) times faster]. So, fft will now be used to transform various coefficient vectors for arbitrary polynomials A and B of length 4, 8, 16, 32, 64, … 1024 and 2048

3. Pointwise multiplying results of Step (ii) to produce C(x) in P-V form

To multiply two polynomials, A(x) and B(x), in the frequency or value domain, you follow these steps:
Perform pointwise multiplication: This means you multiply the coefficients at the same positions in the polynomials A(V) and B(V). This results in a new polynomial, which we'll call C(V), in either the frequency or value domain.
Apply the inverse FFT (IFFT): To convert C(V) back to the polynomial coefficient domain (P), you use the inverse Fast Fourier Transform (IFFT). This step gives you the polynomial C(x) in its coefficient form.

To achieve this using a programming function, you can define a function called pointwise_mul that multiplies

corresponding elements of two input lists. Then, apply this function to lists obtained by applying the Fast Fourier Transform (FFT) to the entries from entriesA and entriesB. This process effectively performs pointwise multiplication in the frequency domain, which is a crucial step in polynomial multiplication using the Fast Fourier Transform (FFT) algorithm. The result will be a list of products for each pair of entries from entriesA and entriesB.



```
for i in range(len(entriesA)):
  print(pointwise_mul(fft(entriesA[i]), fft(entriesB[i])))
```

```
<ipython-input-3-53cffde24741>:4: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will mo
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
<ipython-input-3-53cffde24741>:16: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will n
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
[(65311+0j), (-791+698j), (-5353+0j), (-791-698j)]
[(169814+0j), (507.4463555995871 5-310.33748237159614j), (6467-7395j), (-755.4463555995587+939.0625176288136j), (756-0j), (-755.4463555995868-939.0625176288138j), (6467+7395j), (507.4463
[(695832+0j), (-11799.048261873875+1131.2970623879315j), (4053.681092190351 6-482.1821817572991j), (24828.13146003274 6+4642.524964831381j), (-849.99999999999995-8850j), (-496.08552222242
[(2756624+0j), (-5683.000269095 85+6144.25172545 083j), (-13721.499665951082+6316.39135600313j), (-6143.823997021151-13087.13981997543j), (-16862.230842399 884+42545.2732161357 874j), (1234.
[(8279384+0j), (-4068.4924738789 5908-19283.317781521884j), (256.0768463752165-3843.228451013429j), (5654.682929457289-9255.635588415471j), (-12245.78270481 3599+57317.258482 08825j), (-198
[(43911668+0j), (-62959.357845448 066+35420.274892 080995j), (17197.154239152995+189743.85567151223j), (53709.358394534 66+31241.5869668 5958j), (90443.02620429 38+84382.62491954 57j), (-36
[(168581876+0j), (-98023.763080 12805-110601.66533 18633j), (-93316.580474 09594-12471.00522382567j), (-29562.27130623 7088+19228.73178170593j), (-57298.3872771 36525+30370.130562 2001 1j), (
[(677764256+0j), (545500.38159997 03-74018.568254223 92j), (-209288.0521 881313+162540.49 8915 20595j), (-304743.834 0408 10614-425060.199 8240 0292j), (518630.60000 89 95 28-306454.44335 47155j), (2
[(2534953200+0j), (-418185.8248 415427-373075.885 5393 3506j), (-617812.66 9670 9198-555538.5163 8582 58j), (10582.58 2951 33602-121777.7279 0656517j), (-465076.47 981 145256+232509.8320 0985 666j),
[(10181327872+0j), (-250947.8261767 0093+2160075.6245 5256 85j), (1112519.24 8463 2258-1520615.9146 6619 87j), (710542.6 3565552716+934258.1 4799 73268j), (-335478.5669 954 8671-832718.99179 90 6067j)]
```

Fig: C(x) i.e list of all polynommials obtained after multiplication

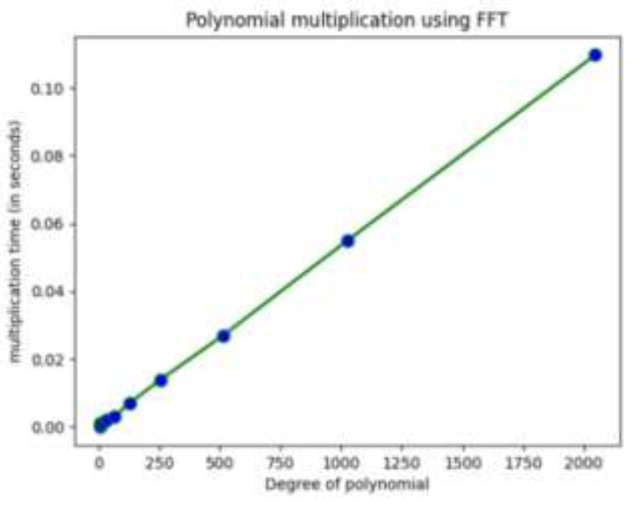


Fig: Graphical representation of pointwise multiplication

4. RSA encrypt (128-bit, 256-bit and 512-bit), with a public key, the C(x) in PV form, for transmission security and decrypt it with a private key and verify.

RSA is a public-key cryptosystem that is widely used for secure data transmission. It is also the oldest. Implementing RSA first required the implementation of a few helper functions.

To achieve this step, we use different methodologies:

→ Extended Euclidean GCD

The Extended Euclidean GCD algorithm is a method used to find the greatest common divisor (GCD) of two numbers, 'a' and 'b,' while also determining the coefficients 'x' and 'y' in the equation ax + by = gcd. This equation is known as Bézout's identity.

In contrast to the standard Euclidean algorithm, which focuses on obtaining remainders in a series of divisions, the extended version considers not only the remainders but also the successive quotients. This algorithm is designed to find the GCD of 'a' and 'b' in such a way that it identifies the smallest positive value that can be expressed as ax + by, where 'a' and 'b' are integers

→ Modular Inverse

A modular multiplicative inverse of an integer 'a' is another integer 'x' that, when multiplied with 'a,' results in a product 'ax' that is congruent to 1 when considered within the modulus 'm.' In simpler terms, it's a number 'x' that, when multiplied by 'a' and taken modulo 'm,' equals 1.

$$ax = 1 \pmod{m}$$

This concept is essentially a part of solving modular linear equations, particularly when 'b' is set to 1. The idea behind it relies on a theorem that states 'x' is a valid solution to the equation if and only if 'b' is divisible by the greatest common divisor (gcd) of 'a' and 'n,' where 'n' represents the modulus.

$$x_0 = x'(b/d) \bmod n \ .$$

→ RSA Encryption

RSA encryption is carried out using the following procedure:

1. Select at random two large prime numbers p and q such that $p \neq q$. The primes pand q might be, say, 1024 bits each.
2. Compute n = pq.

3. Select a small odd integer e , that is relatively prime to Φ(n) = (p-1)(q-1) .

   {Since by Euler's Theorem: Φ(n) = n (1 – 1/p) (1- 1/q) = (p-1)(q-1) }

4. Compute d, as the multiplicative inverse of e mod Φ(n) . [d exists and isuniquely defined. => ed ≡ 1 mod Φ(n)]
5. Publish the pair , P = (e , n) as the participant’s RSA public key.
6. Keep secret the pair S = (d , n) as the participant’s RSA secret key.

- To transform a message M associated with a public key P = (e , n) , compute

$$P(M) = M^e \bmod n$$

- To transform a ciphertext C associated with a secret key S = (d , n), compute

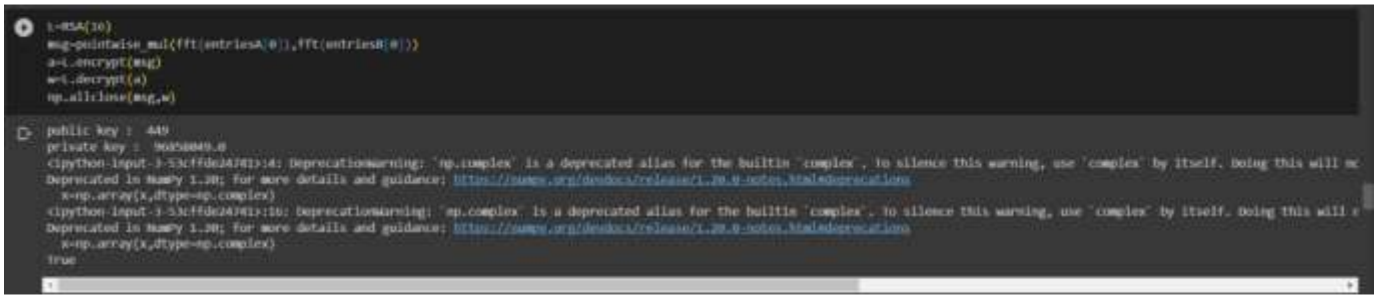
$$S(C) = C^d \bmod n .$$

- The working is as follows:

$$\begin{aligned} M &= S_A(P_A(M)) , \\ M &= P_A(S_A(M)) \end{aligned}$$

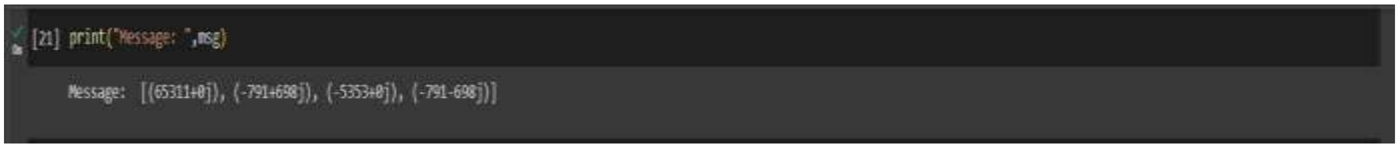
- We implemented RSA as a class. It creates an object that has the methods rsa, encrypt and decrypt.
- The rsa method generates the public and private key, the encrypt method encrypts the message and the decrypt method decodes it.
- The object is built for an array of values from the point value form of any given polynomial. The points are not encrypted because the points taken for the input polynomial are always the nth roots of unity.

→ **Original message**



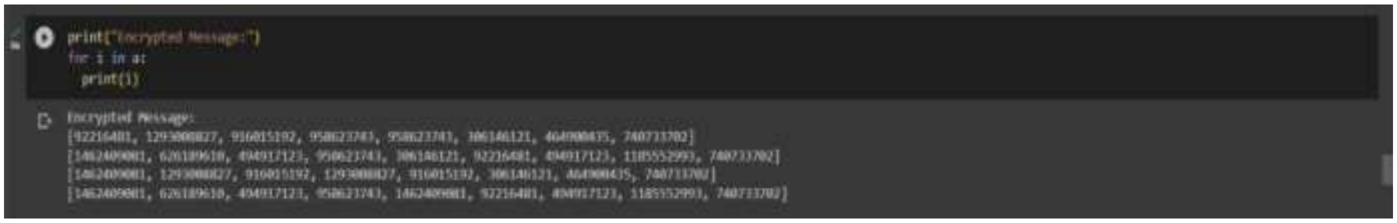
```
t=RSA(16)
msg=pointwise_mul(fft(entriesA[0]),fft(entriesB[0]))
a=t.encrypt(msg)
w=t.decrypt(a)
np.allclose(msg,w)
```

```
public key :  449
private key :  90458049.0
<ipython-input-3-53cffde24781>:4: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will nc
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
<ipython-input-3-53cffde24781>:16: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will r
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
True
```



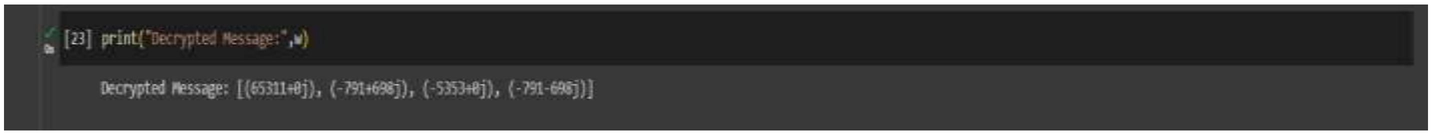
```
[21] print("Message: ",msg)

     Message:  [(65311+0j), (-791+698j), (-5353+0j), (-791-698j)]
```

→ Encrypted message



```
print("Encrypted Message:")
for i in a:
  print(i)

Encrypted Message:
[92216481, 1293000827, 916015192, 958623743, 958623743, 306146121, 464900435, 740733702]
[1462409081, 626189610, 494917123, 958623743, 306146121, 92216481, 494917123, 1185552993, 740733702]
[1462409081, 1293000827, 916015192, 1293000827, 916015192, 306146121, 464900435, 740733702]
[1462409081, 626189610, 494917123, 958623743, 1462409081, 92216481, 494917123, 1185552993, 740733702]
```

→ Decrypted message



```
[23] print("Decrypted Message:",w)

     Decrypted Message: [(65311+0j), (-791+698j), (-5353+0j), (-791-698j)]
```

Running time of RSA

- Generating keys can be time-consuming, especially for longer key lengths. This process can vary from taking seconds to minutes.
- In RSA encryption, the time complexity is directly related to the key size, but it's usually faster than key generation.
- Encrypting data is relatively quick, typically taking milliseconds to seconds, depending on the key's length and the hardware used.
- RSA decryption, in contrast, involves the private key, which can be larger than the public key.
- Decryption time also increases linearly with the key size and might take a bit more time than encryption, but it's still relatively fast compared to key generation.

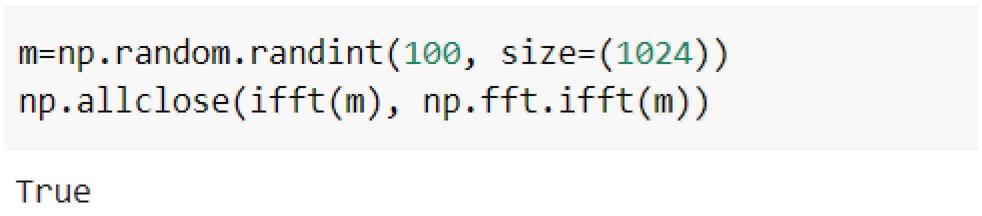
5. Implement 1-D Inverse FFT (I-FFT) on C(x), in PV form (Interpolation) to get C(x) in Coefficient form (CR) Polynomial.

The Inverse Fast Fourier Transform (IFFT) is a fast algorithm that performs reverse or backward or inverse Fourier Transform which undoes the process of FFT

The formula used here is as follows

$$x_i = \frac{1}{N}\sum_{n=0}^{N-1} F_n e^{\frac{2\pi j}{N} ni}$$

Checking the correctness of our Implementation

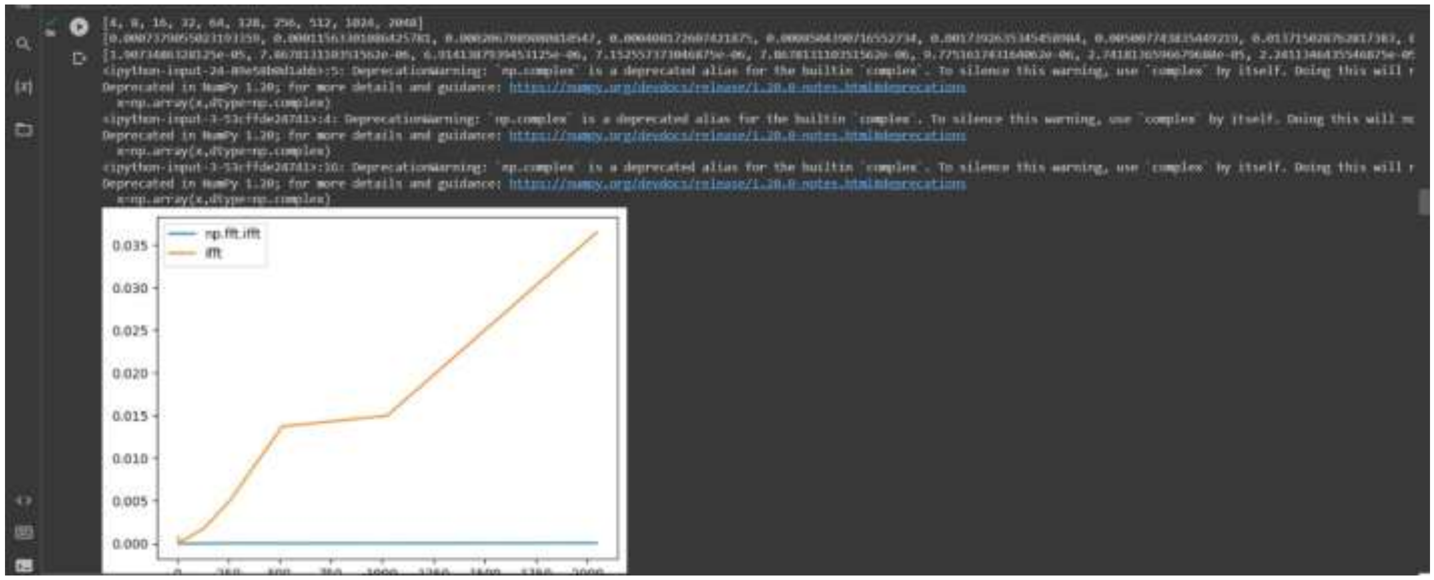


```
m=np.random.randint(100, size=(1024))
np.allclose(ifft(m), np.fft.ifft(m))
```

```
True
```

Comparing Multiplication using FFT and IFFT vs regular multiplication.

- Polynomial multiplication can be made more efficient by following this block diagram:
- First, use the Fourier transform to convert the two polynomials into their point value forms.
- Then, perform the multiplication using the function pointwise_mul(a, b).
- To reconstruct the resulting polynomial, apply interpolation using the implemented IFFT.
- This implementation's correctness and effectiveness can be easily confirmed by comparing its results with those obtained through a standard convolution for loop.
- Once its accuracy is verified, you can proceed to compare its speed with the traditional method of polynomial multiplication, which relies on the distributive property.



```
[4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]
[0.0007379055023193359, 0.0001156330108642578l, 0.0002067089080810547, 0.0004041726074218 75, 0.0005843907165527 34, 0.0017392635345458984, 0.005007743835449219, 0.01371502876281738 3, (
[1.9073486328125e-05, 7.867813110351562e-06, 6.9141387939453125e-06, 7.152557373046875e-06, 7.867813110351562e-06, 9.775161743164062e-06, 2.7418136596679688e-05, 2.2411346435546875e-05
<ipython-input-24-89e58b0d1ab6>:5: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will r
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
<ipython-input-3-53cffde287d1>:4: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will mc
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
<ipython-input-3-53cffde287d1>:10: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by itself. Doing this will r
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  x=np.array(x,dtype=np.complex)
```

6.Verify correctness of C(x) , by comparing with the coefficients generated by a Elementary "Convolution For Loop" on the Coefficients of A(x) and B(x)

We can compare the efficiency of these two polynomial multiplication methods through the following steps:

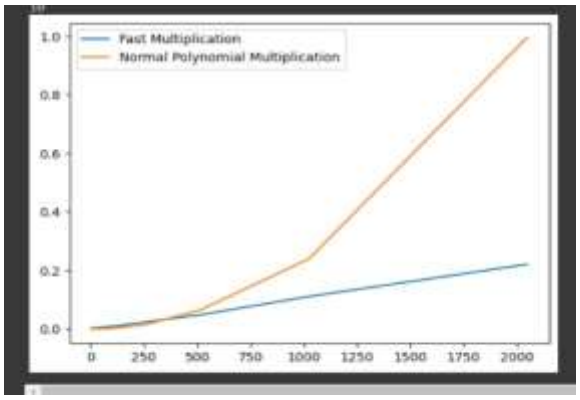
- Create a function called fast_multiplication(arr1, arr2) that takes coefficient arrays arr1 and arr2 representing polynomials A(x) and B(x) in coefficient representation (CR).

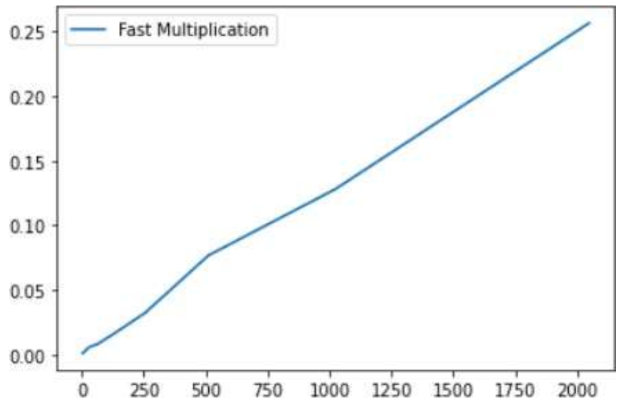
- Inside this function, compute the Fast Fourier Transform (FFT) of arr1 and arr2, perform pointwise multiplication of the FFT results, and then compute the Inverse FFT (IFFT) of the product. Round the result to obtain C(x) in coefficient representation.

- Implement another function called c_in_cr(arr1, arr2) that uses a nested for loop (convolution for loop) to multiply the two polynomials A(x) and B(x) represented by coefficient arrays arr1 and arr2. This function also returns the result in coefficient representation (CR) form.

- Use the np.allclose() function to compare the outputs of fast_multiplication and c_in_cr and check if they are close within a certain tolerance.

- To validate the correctness of these methods, specifically for the case of multiplying the polynomials [2, 1, 0, 0] and [2, 1, 0, 0], check if the np.allclose() function returns True. This indicates that both methods produce matching results for this particular case.





Based on the graph provided, it's evident that the multiplication process involving FFT, pointwise multiplication, and interpolation is significantly faster than the traditional method of polynomial multiplication.

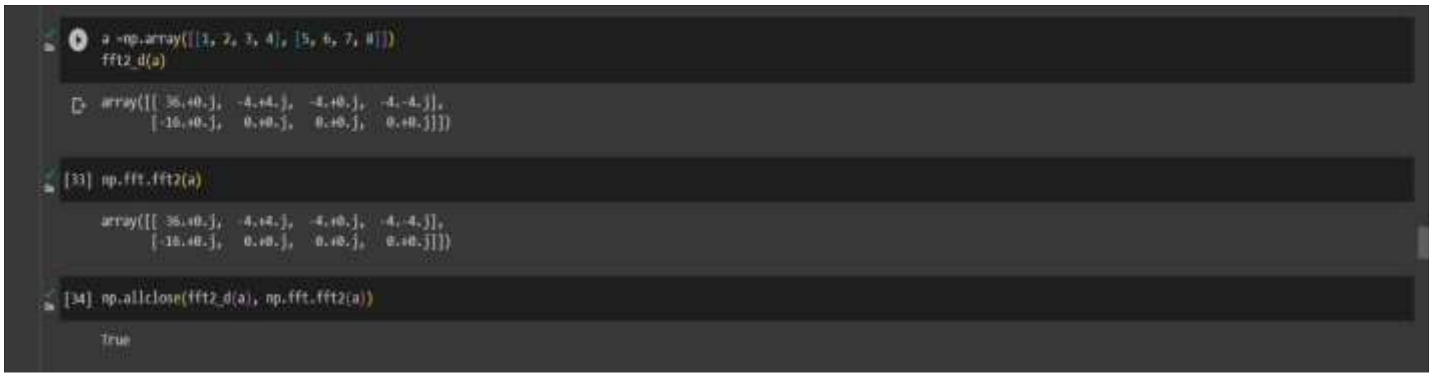
7.Implement a 2-D FFT and 2-D I-FFT module using your 1-D version (This just means, applying FFT on the Rows First and Columns Next on M x N matrix of numbers !!)

This procedure involves converting a matrix from its original spatial domain into the frequency domain using the Fast Fourier Transform (FFT), and then reversing this transformation using the Inverse Fast Fourier Transform (IFFT). This transformation enables efficient analysis and synthesis of two-dimensional data, particularly useful for tasks like image processing and signal analysis.

The 2-D FFT is a valuable tool, especially in applications such as image processing and signal analysis.

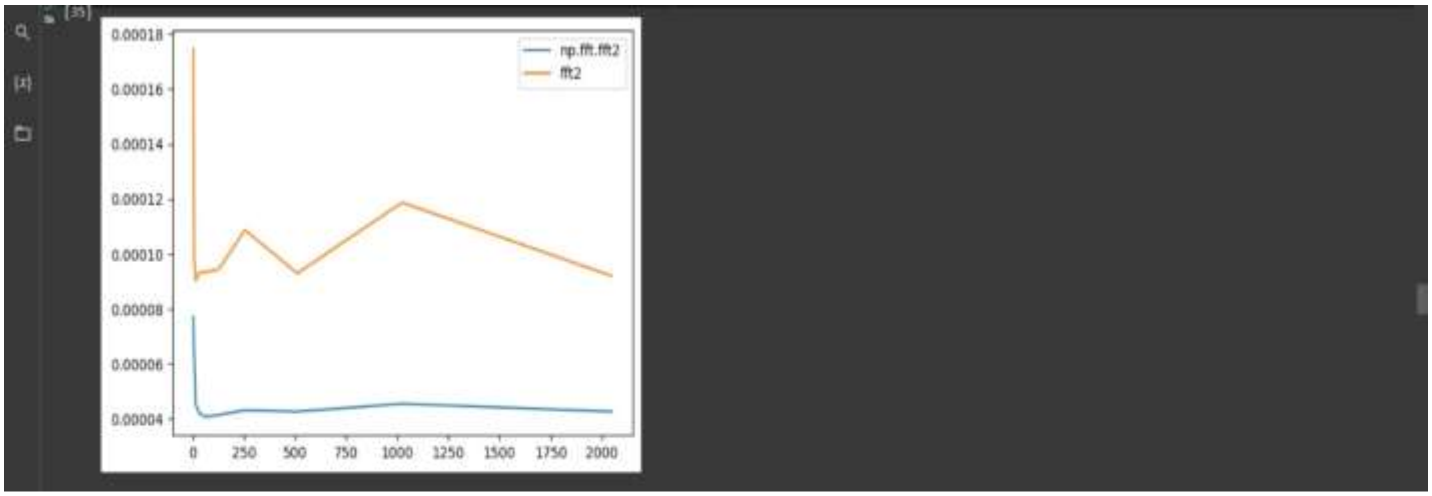
When your 2-D FFT and IFFT implementations produce the same results as established libraries or tools like NumPy across various test cases, it signifies that your implementations are likely accurate and correct. This confirms that your implementations effectively perform the necessary Fourier transforms and their inverses on 2-D matrices, which is a fundamental requirement for various tasks in signal processing and image analysis.

Checking the correctness of our Implementation



```
a =np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
fft2_d(a)

array([[ 36.+0.j,  -4.+4.j,  -4.+0.j,  -4.-4.j],
       [-16.+0.j,   0.+0.j,   0.+0.j,   0.+0.j]])

[33] np.fft.fft2(a)

array([[ 36.+0.j,  -4.+4.j,  -4.+0.j,  -4.-4.j],
       [-16.+0.j,   0.+0.j,   0.+0.j,   0.+0.j]])

[34] np.allclose(fft2_d(a), np.fft.fft2(a))

True
```

Running time of 2D FFT VS np.fft.fft2



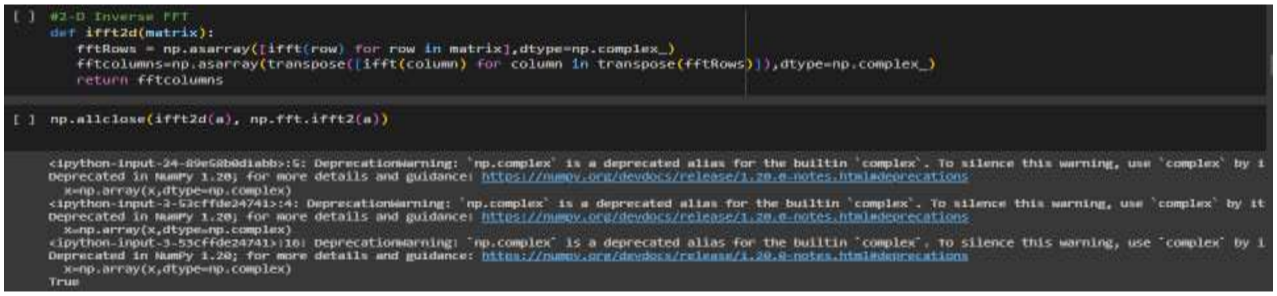
\

Implementing 2D IFFT

The implementation of the 2D Inverse Fast Fourier Transform (IFFT) is carried out by utilizing the 1D IFFT that has been previously generated. This process involves applying the 1D IFFT first to the rows of the 2D matrix and then to the columns.

By applying the 1D IFFT in both the row and column directions, you effectively reverse the transformation, bringing the entire 2D matrix back from the frequency domain to the spatial domain. This capability is particularly valuable for tasks like image reconstruction, especially after applying frequency-based operations such as filtering.A crucial aspect to note here is that this 2D IFFT operation is separable, which means you can achieve it by separately applying the 1D IFFT along the rows and then along the columns. This property simplifies the implementation and can significantly enhance the process's speed, especially when dealing with large matrices, thanks to the efficient 1D IFFT algorithms.

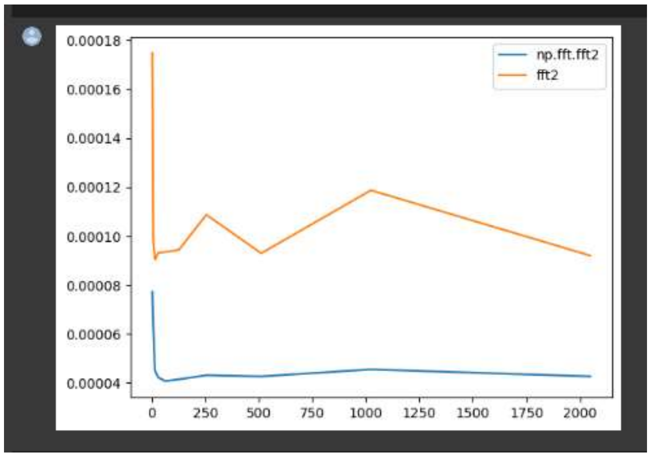
Checking the correctness of our Implementation



```
[ ] #2-D Inverse FFT
    def ifft2d(matrix):
       fftRows = np.asarray([ifft(row) for row in matrix],dtype=np.complex_)
       fftcolumns=np.asarray(transpose([ifft(column) for column in transpose(fftRows)]),dtype=np.complex_)
       return fftcolumns

[ ] np.allclose(ifft2d(a), np.fft.ifft2(a))

    <ipython-input-24-89e58b0d1abb>:5: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by i
    Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
      x=np.array(x,dtype=np.complex)
    <ipython-input-3-53cffde24741>:4: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by it
    Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
      x=np.array(x,dtype=np.complex)
    <ipython-input-3-53cffde24741>:16: DeprecationWarning: `np.complex` is a deprecated alias for the builtin `complex`. To silence this warning, use `complex` by i
    Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
      x=np.array(x,dtype=np.complex)
    True
```

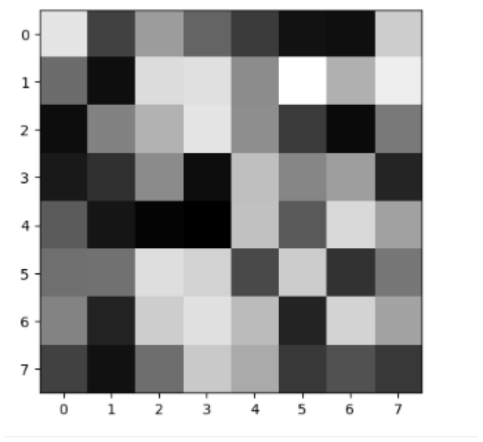
The np.allclose function is used to compare the entire output matrices for equality.

Running time of 2DIFFT VS np.fft.ifft2



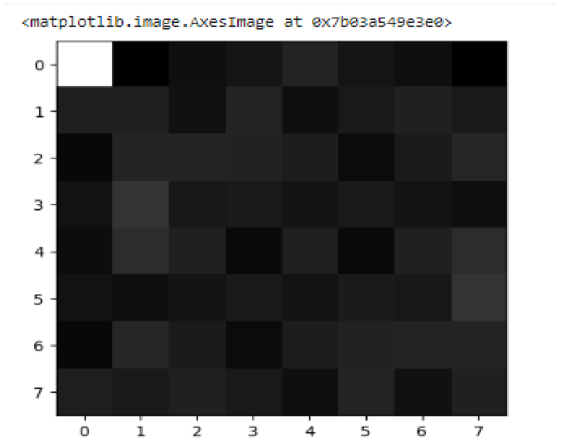
8.Verify your of Step (vii) correctness on a Grayscale matrix(which has random integer values in the range 0-255; 255 → White & 0 → Black))

- To confirm the accuracy of the grayscale matrix, follow these steps:
- Generate matrix A, which is a 4x4 matrix containing random values within the range of [0, 1).
- Calculate the 2D Fourier Transform of A using the fft2_d function, storing the result in y_A. This represents the Fourier coefficients of the image.



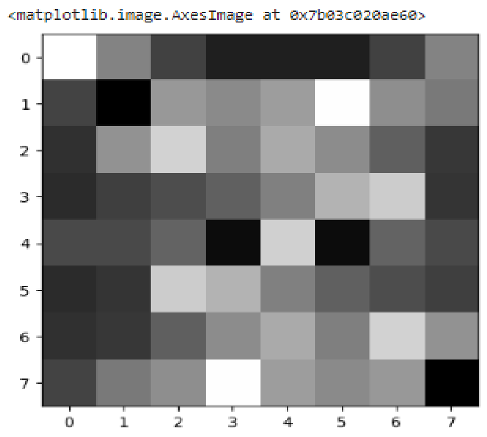
- Perform Inverse Fourier Transforms on both y_A and y_c to obtain reconstructed images. Save the result of the inverse Fourier Transform using the custom ifft2d function in A_c, and the result using np.fft.ifft2 in A_y

- To validate the correctness of the inverse Fourier Transform, visually examine the displayed images of A, A_c, and A_y. If the reconstruction is accurate, A_c and A_y should closely resemble the original A. However, due to potential differences in precision and numerical approximations, there might be slight variations between the images.



```
<matplotlib.image.AxesImage at 0x7b03a549e3e0>
```

Fig: A_y



```
<matplotlib.image.AxesImage at 0x7b03c020ae60>
```

Fig: A_c

9.Apply your 2D-FFT on TIFF/JPG (lossless) Grayscale image and drop Fourier coefficients below some specified magnitude and save the 2D- image to a new file.

The use of digital images has significantly expanded over the past four decades, primarily due to advancements in image storage, transmission, and modification. One powerful technique for image compression is the Fast Fourier Transform (FFT), which can reduce the size of an image to nearly one-tenth of its original size without compromising its clarity. The process involves the following straightforward steps:

- Begin by converting a grayscale image of your choice into a matrix.
- Apply a 2D Fast Fourier Transform (FFT) to this matrix.
- Extract the value from the matrix corresponding to the top 'c%' of coefficients, where 'c' represents the compression rate. This step identifies the signal components that carry 100-c% of the image's information.
- Eliminate all values in the matrix that are lower than this threshold by setting them to 0.
- Apply a 2D Inverse Fast Fourier Transform (IFFT) to the modified matrix.
- Finally, visualize the resulting image using the imshow() function from the matplotlib.pyplot library.

Fig: Original image

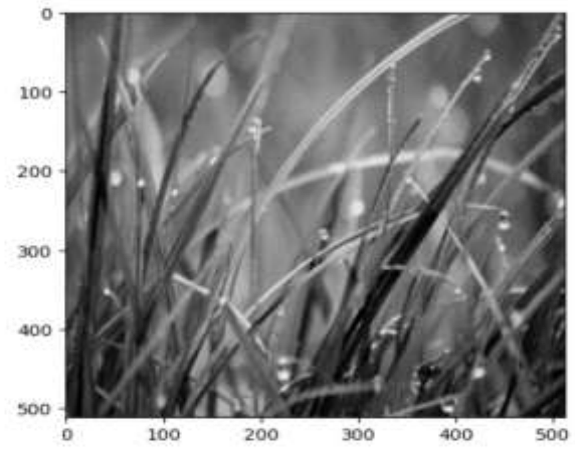


Fig : Grayscale Image

In essence, this process achieves compression by reducing the image's size while retaining important information, albeit with some permanent loss due to the removal of coefficients below a certain threshold value.

10.Apply 2D I-FFT, on the Quantized Grayscale image and render it to observe Image Quality.

In the compression process, we convert the Fourier transform of the image into a vector and then sort these values in descending order. We subsequently retain only specific percentages of the highest values, such as the top 10%, 5%, 1%, or 0.2% of the threshold.

In the context of lossy compression, intentional information loss occurs to achieve higher compression ratios. When you repeatedly compress an image using substantial factors like 0.1 or 0.01, you progressively sacrifice image details. The more you compress, the more apparent the decline in image quality becomes. This can lead to visual artifacts, blurriness, and a reduction in image sharpness.For instance, when compressing an image at a 10% compression rate, this means only 10% of the highest-value coefficients are retained, and the remaining 90% are discarded. This level of compression can result in a noticeable loss of image quality, as some of the image's fine details and nuances may be compromised

When image resolution is greater 1024*1024

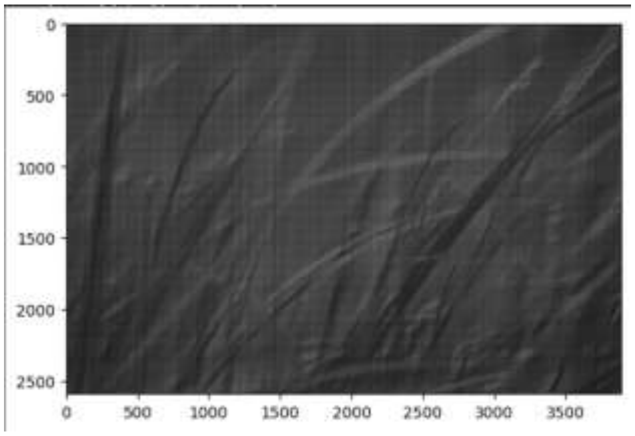


Image compression at 0.1 %

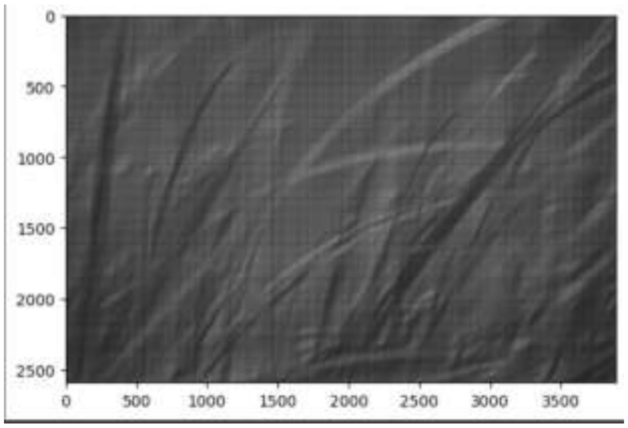


Image compression at 0.01 %

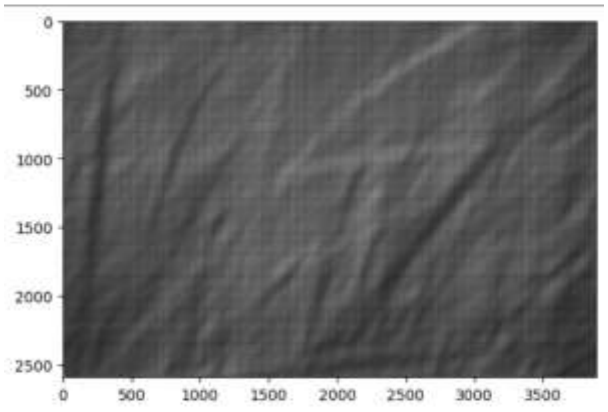


Image compression at 0.001 %

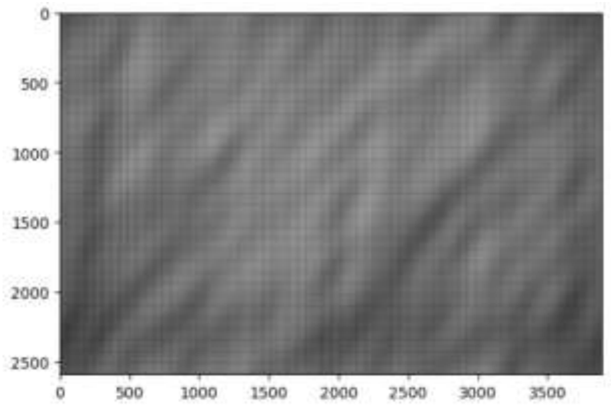


Image compression at 0.0001 %

When the image resolution is smaller that is 512 * 512 compression occurs faster

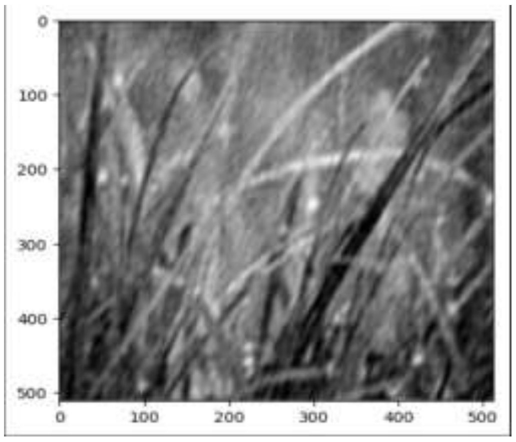


Image compression at 0.1 %

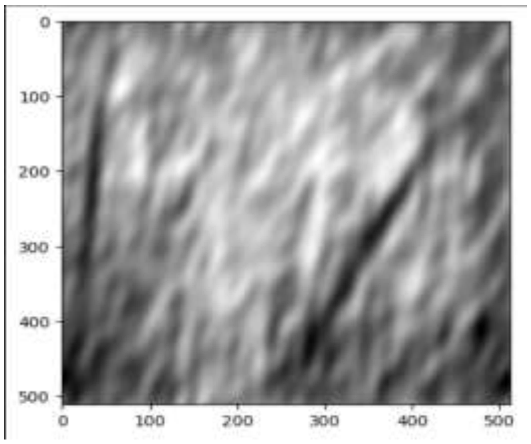


Image compression at 0.01 %

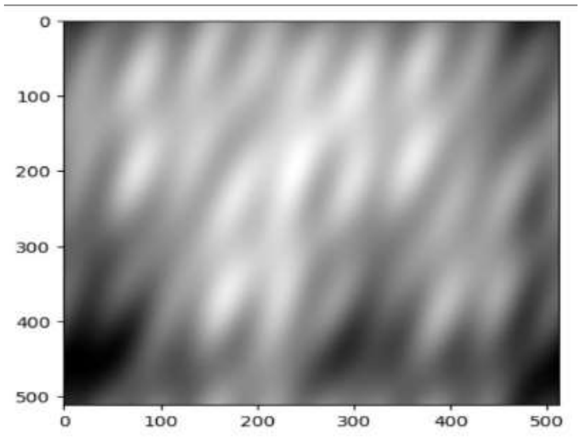


Image compression at 0.001 %

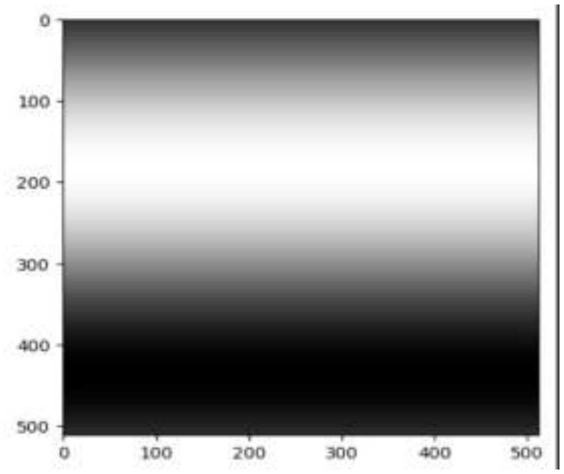


Image compression at 0.0001 %

- In summary, the impact of repeatedly compressing an image depends on the compression method and the compression ratio used.
- Lossless compression maintains image quality, while lossy compression sacrifices some quality for higher compression.
- Very high compression ratios (e.g., 0.1 or 0.01) using lossy compression will likely lead to visible degradation in image quality.

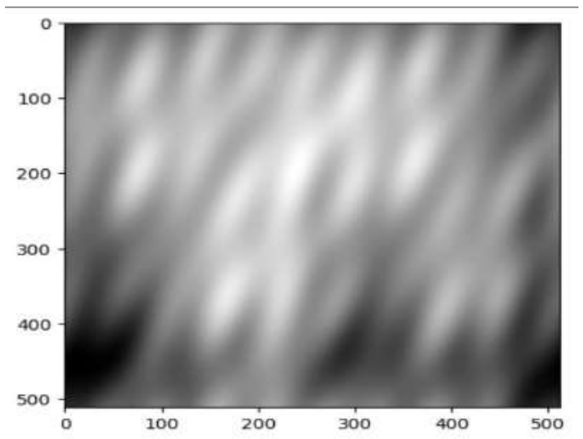


Image compression at 0.001 %

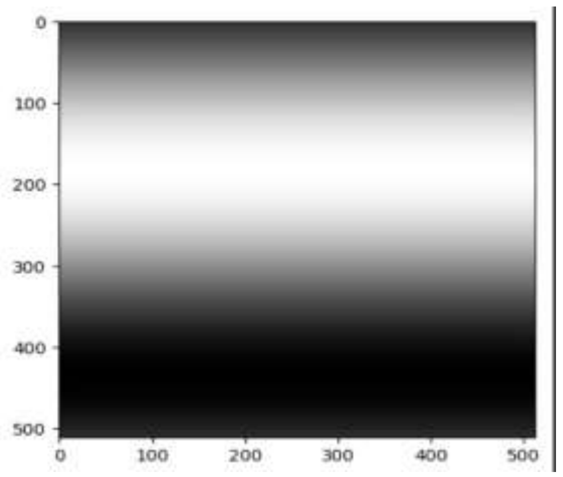


Image compression at 0.0001 %

- In summary, the impact of repeatedly compressing an image depends on the compression method and the compression ratio used.
- Lossless compression maintains image quality, while lossy compression sacrifices some quality for higher compression.
- Very high compression ratios (e.g., 0.1 or 0.01) using lossy compression will likely lead to visible degradation in image quality.

Observations and Learning Outcomes

1. The 1-D DFT has a time complexity of O(n^2) time as compared to the 1-D FFT which has a time complexity of O(n log n) time. DFTn, which is evaluating the coefficient vector A(x) at the n complex nth units of unity, has been reduced to evaluating two polynomials at the n/2 complex, n/2th roots of unity (i.e., two instances of DFTn/2).

2. 1-D FFT is then performed on the row vectors followed by 1-D FFT on the column vectors of the transformed matrix to get 2-D FFT.

3. RSA is a public key cryptographic algorithm in which two different keys are used to encrypt and decrypt the message. Each user has to generate two keys: a private key and a public key. The public key is circulated or published to all and hence others are aware of it whereas, the private key is secretly kept with the user only. A sender has to encrypt the message using the intended receiver's public key. Only the intended receiver can crack the message. In between the communication no one can harm the confidentiality of the message as the message can only be decrypted by the intended receiver's private key which is only known to that receiver. 4. Image compression - The quantized grayscale image is obtained by considering the largest 10%, 5% , 1% and 0.2% of fourier coefficients. As the percentage of largest fourier coefficients considered is reduced, image quality reduces