Image Compression using Fast Fourier Transform

Image Adhikari
4th Year/ 1st sem
Kathmandu University
imageadhikari@gmail.com

Yogesh Pant
4th Year/ 1st sem
Kathmandu University
yogeshpant293@gmail.com

Sambeg Shrestha
4th Year/ 1st sem
Kathmandu University
sthasuraj2@gmail.com

Sajag Silwal
4th Year/ 1st sem
Kathmandu University
sajag.silwal123@gmail.com

Abstract

In today's digital age, images are ubiquitous. With advancing technologies, the images that are getting better in quality, richer in metadata and information, and thus larger in size. Hence, image compression techniques are needed to improve image which optimizes storage space, network bandwidth usage as well as secure transmission. Among many ways to solve the problem with their own compression ratios and complexities, In this paper, however, we focus particularly on compression of images through Discrete fourier transform, particularly using the fast fourier transform.

1. Introduction

In today's digital world of internet and multimedia, the usage of images is ubiquitous and cosmopolitan. Additionally, with the advancement of technologies, the images that we use everyday have become better in quality, getting larger by the day, they now store more information than ever and hence their size is also getting higher. Image Formats like RAW take this even higher by including more meta data in the already hundreds of megapixels of images. So, image compression techniques are a necessity for improving image and video performance which will help in terms of storage space, network bandwidth usage as well as secure transmission.

Image compression is the process of reducing a graphics file's size in bytes without impairing the image's quality to an unacceptable degree. Image compression reduces the size of bytes in a graphics file without degrading the image's quality. Likewise, it also reduces the time it takes to transmit photos over the ground. There are a number of ways to compress images which leads to different outcomes in compression ratios and complexities. In this paper, however, we focus particularly on compression of images through Discrete fourier transform, particularly using the fast fourier transform.

2. Background

2.1 Digital Image Processing

Analyzing and processing analog and digital signals, as well as storing, filtering, and other operations on signals, are the focus of the mathematical and electrical engineering field of signal processing. These signals range from transmission signals to speech, sound, and image signals, among others. Out of all these signals, image processing is the field that works with the kind of signals where the input is an image and the output is also an image. These can be analog and digital signal processing. **Analog image processing** is done on analog signals. It includes processing on two dimensional analog signals. Meanwhile, **digital image processing** deals with developing a digital system that performs operations on a digital image.

Digital Image

An image is nothing more than a two dimensional signal. It is defined by the mathematical function f(x,y) where x and y are the two coordinates horizontally and vertically. The value of f(x,y) at any point gives the pixel value at that point of an image. Each number represents the value of the function f(x,y) at any point. An image is nothing but a two dimensional array of numbers ranging between 0 and 255. The dimensions of the picture is actually the dimensions of this two dimensional array.

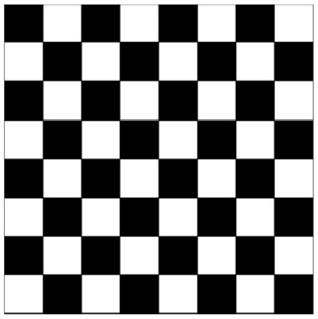


Fig: A simple 8x8 image

0	256	0	256	0	256	0	256
256	0	256	0	256	0	256	0
0	256	0	256	0	256	0	256
256	0	256	0	256	0	256	0
0	256	0	256	0	256	0	256
256	0	256	0	256	0	256	0
0	256	0	256	0	256	0	256
256	0	256	0	256	0	256	0

Fig: 8x8 Matrix representation of the image

2.2 Fourier transform in Image Processing

The Fourier transform, often known as the second language for image description, is a classical technique for converting images from the space domain to the frequency domain. It is found in applications such as direct harmonic analysis of musical signals and vibrations, but also reduces the rate coding of speech and music, compression, and digital transmissions.The Fourier transform (FFT), also known as frequency analysis or spectral involved in the implementation of many digital techniques for processing signals and images.

Discrete Fourier Transform (DFT)

Fourier Transform is a mathematical technique that helps to transform Time Domain function x(t) to Frequency Domain function X(ω). **Discrete Fourier transform (DFT)** converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency. The DFT is one of the most powerful tools in digital signal processing which enables us to find the spectrum of a finite-duration signal.

The Discrete Fourier Transform (DFT) can be expressed as:

$$\mathrm{F(k)} = \sum_{k=0}^{N-1} f(\mathrm{n})\, e^{-\frac{j2\pi nk}{N}} \qquad (\mathrm{n} = 0, 1, \ldots., \mathrm{N} - 1)$$

Similarly, the Inverse Discrete Fourier Transform (IDFT) can be expressed as:

$$f(\mathrm{n}) = \frac{1}{N}\sum_{k=0}^{N-1} F(\mathrm{k})\, e^{-\frac{j2\pi nk}{N}} \qquad (\mathrm{n} = 0, 1, \ldots., \mathrm{N} - 1)$$

Fast Fourier Transform

Fast Fourier Transform (FFT) is an algorithm that determines the Discrete Fourier Transform(DFT) of an input significantly faster than computing it directly. In computer science lingo, the FFT reduces the number of computations needed for a problem of size N from O(N^2) to O(NlogN). The basic FFT formulas are called radix-2 or radix-4 although other radix-r forms can be found for r = 2k, r > 4.

3. Methodology

3.1. Overview

The FFT compression works on the basics of the FFT algorithm. The algorithm converts an image matrix of 3 dimensions (RGB) into the frequency domain using FFT. The user-provided threshold is utilized in the algorithm to filter out low signals, and on the basis of that, we are filtering signals above the threshold, which is known as a high pass filter. This is done because, according to the theory behind it, low frequencies can be removed from an image while preserving the majority of its information because human vision can only detect frequencies beyond a particular threshold. The image is then converted back into its original domain using IFFT once the low frequencies have been removed.

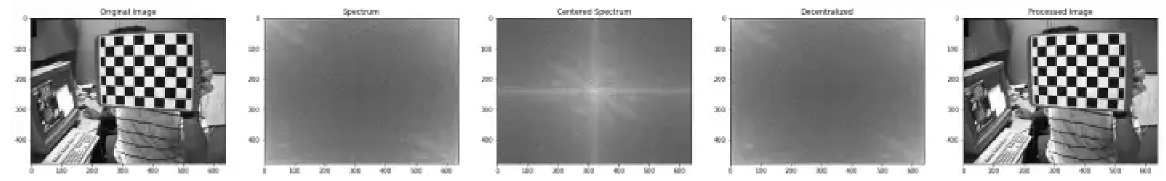


Fig: (From left to right) (1) Original image (2) FFT spectrum visual output (3) Centralized (4) Decentralized (5) Inverse FFT

3.2. Algorithms Implementation

The program makes use of an internally developed FFT function called Radix-2 Decimation In Time (DIT), which is derived from the Cooley-Tukey technique. DIT only works best with signals or sequences which can be expressed in terms of 2^n. According to the algorithm used by the program, the exponential matrix in the DFT can be written as the product of three matrices, of which two are fix-up matrices. A general equation to calculate the DFT of a sequence N is:

$$X = F_N . x$$

where X is the transform, *FN* is the exponential matrix for N terms and x is the sequence or the signal. According to the algorithm the *FN* can be expressed as:

$$F_N = \begin{bmatrix} I & D \\ I & -D \end{bmatrix} \cdot \begin{bmatrix} F_{N/2} & 0 \\ 0 & F_{N/2} \end{bmatrix} . P$$

where,

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & W & 0 & 0 & 0 \\ 0 & 0 & W^2 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & W(N/2-1) \end{bmatrix}$$

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.3. Permutation matrix of odd and even

The equations show that the N term can be split n times, i.e. N=1. The calculations are recursively reduced from N^2 to N $\log_2$N. The approach computes the exponential matrix only once for both the column and row dimensions of a full image matrix, which includes all three dimensions. This is advantageous because it cuts down on computation time. The exponential matrices that are produced can be utilized to convert the matrix to the frequency domain. We immediately pad the image with zeros to make it compatible with the DIT technique because the algorithm can only be applied to sequences that can be expressed in terms of 2n. The image is restored to its original size after being compressed. The overhead seen in this solution is that it expands the matrix size, which can further drive up the cost of operations and be a time-versus-cost trade-off.

3.4. Implementation

An implementation of the image compression using FFT was done in python to observe the application practically.

2D Fourier transform for images

The two-dimensional Fourier transform of a matrix of data X ∈ R n×m is achieved by first applying the one-dimensional Fourier transform to every row of the matrix, and then applying the one-dimensional Fourier transform to every column of the intermediate matrix. Switching the order of taking the Fourier transform of rows and columns does not change the result.

We start by importing libraries, then we load the image. Then, we convert the image into grayscale. We convert it to grayscale so the pixel value ranges from 0-255 instead of the rgb values. Then, the grayscale image is 1 dimensionally applied FFT to obtain the fourier transformed values, initially FFT is dome row-wise. This row-wise fourier transformed image is then later FFT transformed column-wise. Hence, the image is fourier transformed in 2 dimensions using FFT.

However, instead of individually doing FFt for 1 dimensions, there already exists an inbuilt method in numpy called *fft.fft2* used for 2 dimensional FFT. So, we can directly use that.

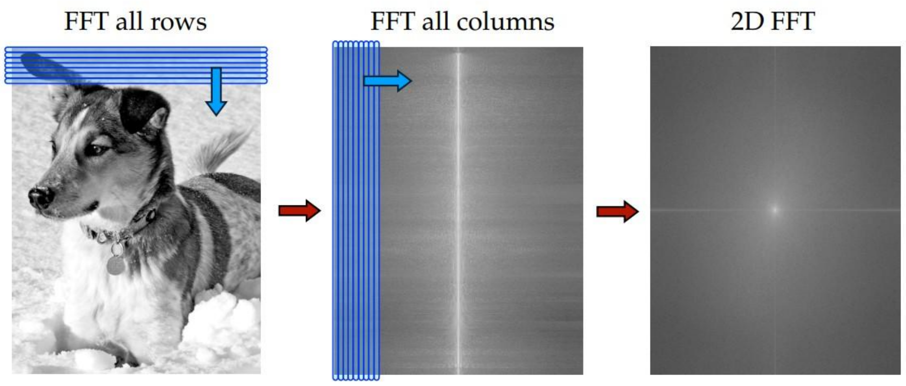


Fig: Applying FFT to the image in 2 dimensions, first row then column

Compression by Filtering larger Fourier coefficients (High pass filtering)

The two-dimensional FFT is effective for image compression, as many of the Fourier coefficients are small and may be neglected without loss in image quality. Thus, only a few large Fourier coefficients must be stored and transmitted.

We directly use the ***fft2*** function to apply FFT to the imported image to get fourier coefficients. From the obtained values of fourier coefficients, we sort them according to the threshold percentage (i.e. high pass filter cutt-off percentage), we find the **threshold fourier coefficient** below which all coefficients are discarded, in our implementation we keep the highest .

Upon finding the cutt-off value, we use a check if the fourier transformed values we previously obtained pass the threshold fourier threshold and store results. So, here all the values below the threshold are removed. We then perform Inverse discrete fourier transform using IFFT on the filtered coefficients to obtain the corresponding pixel intensity values from the fourier coefficient i.e. we get the compressed image.

So, the process performed are:

1. We find the threshold value for filtering.
2. Designing a High Pass filter.
3. Filtering the fourier coefficients according to threshold.
4. Performing Inverse Fourier Transform, IFFT, to get a compressed image.

3.5. Performance Indicators

Lossy compression techniques leave the reconstructed image with some distortions, as the reconstructed image is only the approximation to the original image. In order to measure and quantify the performance of compression technique, some performance indicators are used as follows:

1. **Mean Square Error (MSE):** MSE is the measure of error between the original image and the compressed image. Mean Square Error is the cumulative squared error between the compressed image and the original image. For the lesser distortion and high output quality, the MSE must be as low as possible. Mean Square Error may be calculated using following expression:

   $$\mathrm{MSE} = \frac{1}{\mathrm{m,n}} \sum\sum \left(\mathrm{X_{i,j}} - \mathrm{Y_{i,j}}\right)^2$$

   Higher MSE means higher the difference between original and compressed. So, lower is better.

2. **Peak Signal to Noise Ratio (PSNR):** PSNR is the ratio of maximum power of the signal and the power of unnecessary distorting noise. Here the signal is the original image and the noise is the error in reconstruction. For a better compression the PSNR must be high. The increase in the peak signal to noise ratio results in the decrease in compression ratio.
   Therefore a balance must be obtained between the compression ratio and peak signal to noise ratio for the effective compression. The peak signal to noise ratio may be calculated as:

   $$\mathrm{PSNR} = 20 * \log_{10} \left(\frac{255^2}{\mathrm{MSE}}\right)$$

   Higer the PSNR, the better the compression.

3. **The Structural Similarity Index (SSIM)**: SSIM is a perceptual metric that quantifies the image quality degradation that is caused by processing such as data compression or by losses in data transmission. This metric is basically a full reference that requires 2 images from the same shot, this means 2 graphically identical images to the human eye. The second image generally is compressed or has a different quality, which is the goal of this index. Its value ranges from 0 to 1, 1 being the most similar and 0 denoting something completely different..

4. Results

So, upon performing the task of Image compression through FFT, filtering and IFFT, we get the following image results.

Outcome

As we can see for the given image, there is barely any difference when compressed using 5% and 1% thresholds, whereas there is noticeable lack of detail in the 0.2% threshold.

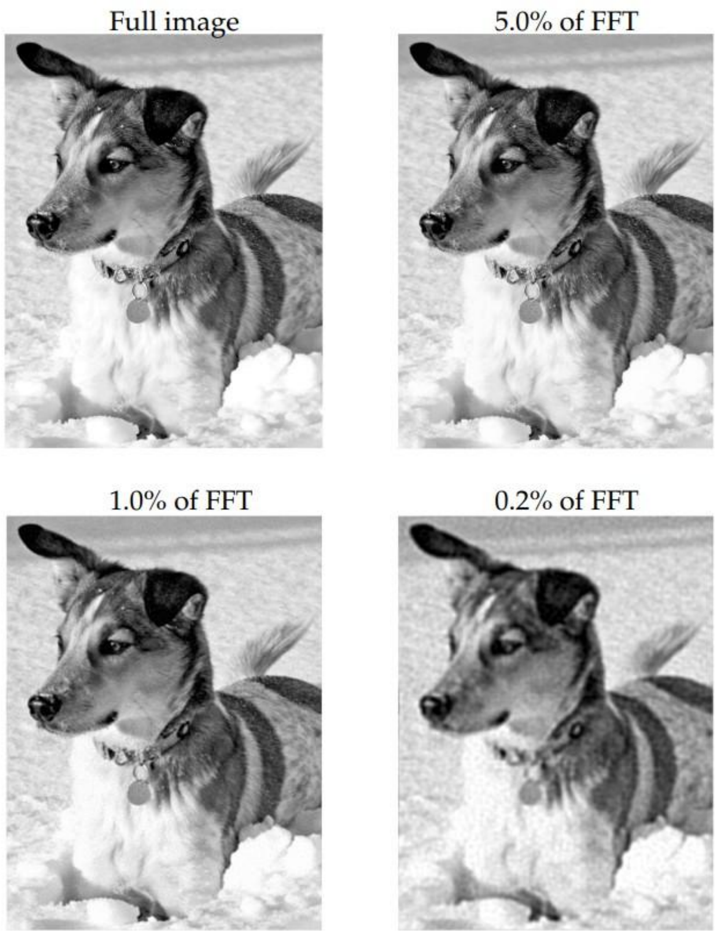
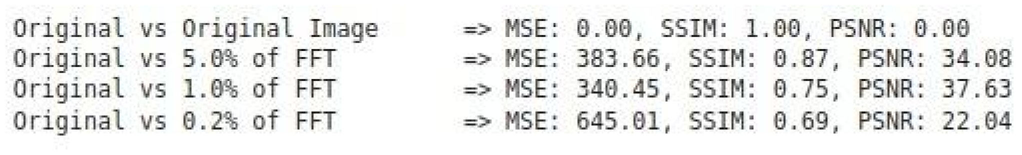


Fig: Compressed image using various thresholds to keep 5%, 1%, and 0.2% of the largest Fourier coefficients.

Performance Indicator

Meanwhile, in Image performance, we can see the following results:



```
Original vs Original Image      => MSE: 0.00, SSIM: 1.00, PSNR: 0.00
Original vs 5.0% of FFT         => MSE: 383.66, SSIM: 0.87, PSNR: 34.08
Original vs 1.0% of FFT         => MSE: 340.45, SSIM: 0.75, PSNR: 37.63
Original vs 0.2% of FFT         => MSE: 645.01, SSIM: 0.69, PSNR: 22.04
```

The original MSE is 0 for original, SSIM is 1 and PSNR is 0 (incalculable). But, as the threshold decreases MSE increases showing there is higher error, similarity decreases and PSNR also decreases.

Compression Sizes

In terms of file size of the image, we have the following results:

Image	**Size (KB)**
Original	322.1
5% of FFT	268.6
1% of FFT	192.8
0.2% of FFT	138.7

We can see, as the threshold increases i.e. compression increases, the file size of the images decreases. So, from the results obtained, we can see that compression is a trade-off between quality vs. size.

5. Conclusion

In this paper, we have presented the method for grayscale image compression based on the Fourier transform(FFT). The transformed compressed images were reconstructed afterwards. The reconstructed images' quality was assessed. After transformation, a compressed image that has been recreated has good picture quality and better compression ratio. In conclusion, we can say that image compression is a balance between quality and size. As we increase the FFT compression, the image starts to get blurrier and lose details, which indicates more compression.

References

Fourier Analysis. (2022, May 19). Investopedia.
https://www.investopedia.com/terms/f/fourieranalysis.asp

Maklin, C. (2021, December 13). *Fast Fourier Transform - Towards Data Science*. Medium.
https://towardsdatascience.com/fast-fourier-transform-937926e591cb

Pandey, S. S., Singh, M. P., & Pandey, V. (2015, April 10). *Image Transformation and Compression using Fourier Transformation* [Review of *Image Transformation and Compression using Fourier Transformation*].

Amaar Hassan, & G. AlAzawi. (2006, December 4). *Digital Image Compression Using Fourier Transform and Wavelet Technique*. https://www.iasj.net.

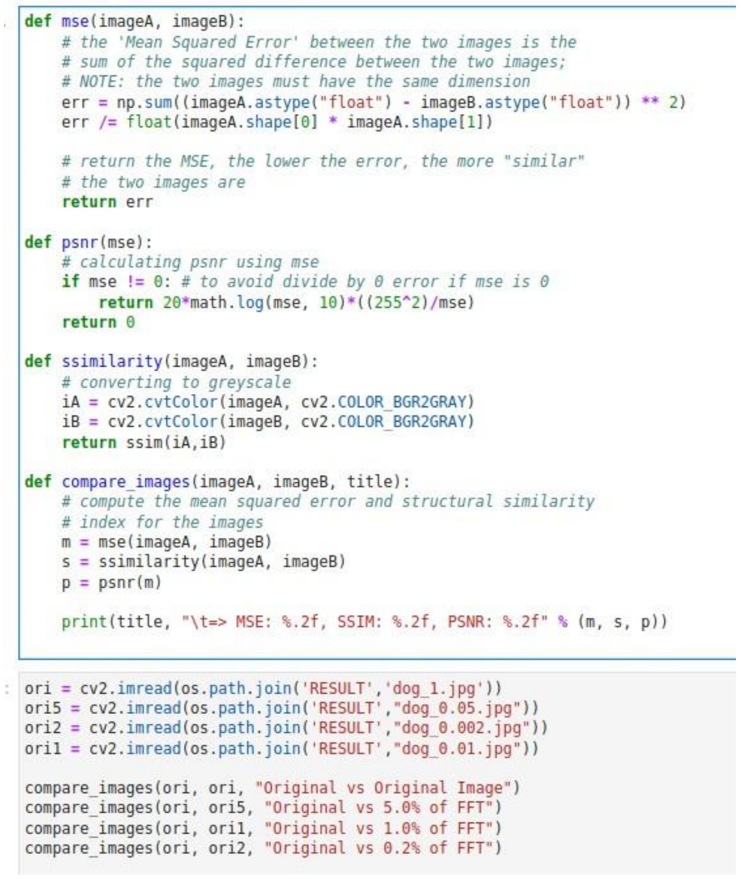
Naushad, R. (2022, June 13). *Fourier Transform for Image Processing in Python from scratch.* Medium. https://medium.datadriveninvestor.com/fourier-transform-for-image-processing-in-python-from-scratch-b96f68a6c30d

Chen, C. (2021, December 13). *Digital Image Processing using Fourier Transform in Python*. Medium. https://hicraigchen.medium.com/digital-image-processing-using-fourier-transform-in-python-bcb49424fd82

Brunton, S. L., & Kutz, J. N. (2019). Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control. Cambridge University Press. http://databookuw.com/databook.pdf

Appendix

Appendix 1: Performance Indicator Implementation

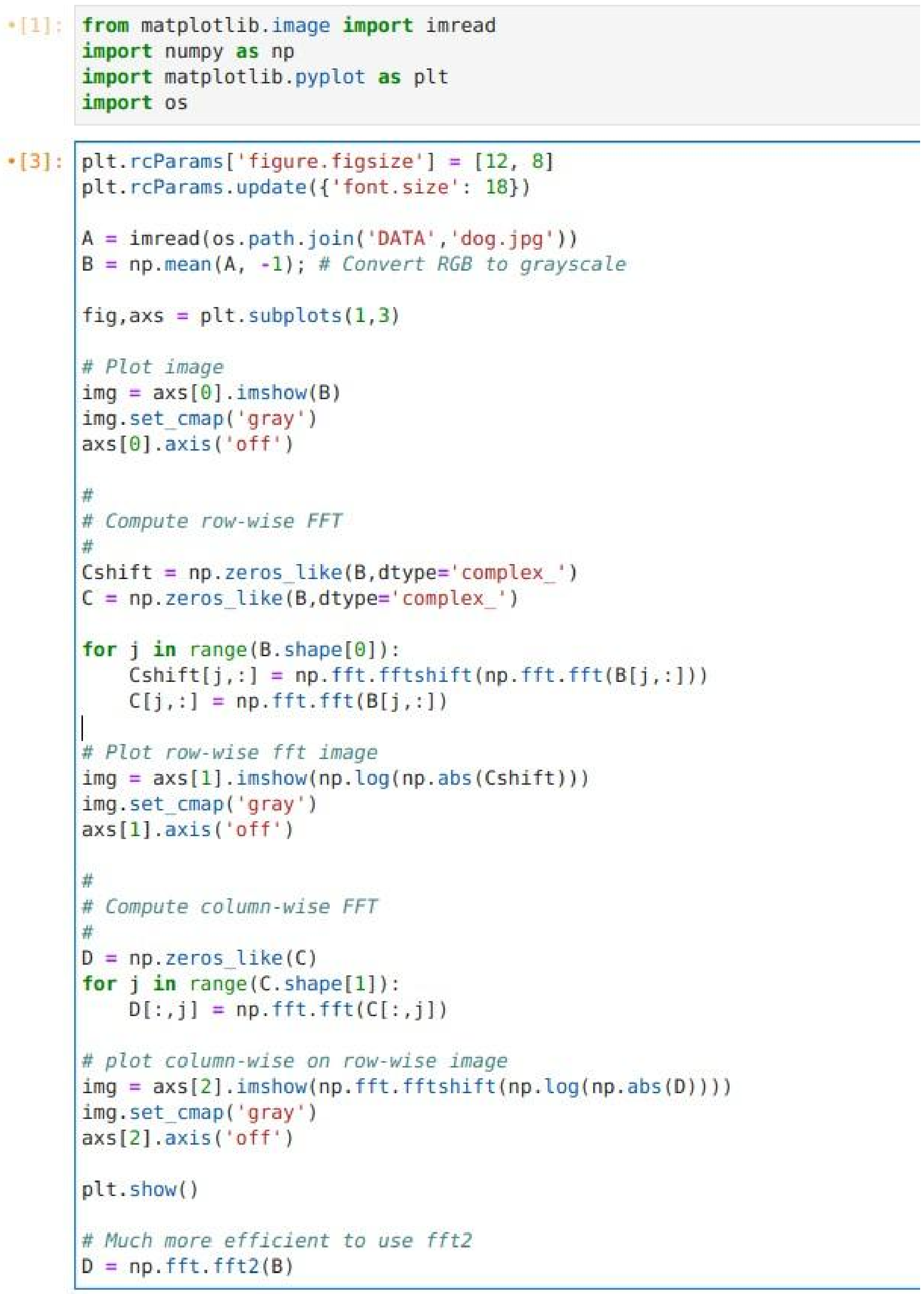


```
def mse(imageA, imageB):
    # the 'Mean Squared Error' between the two images is the
    # sum of the squared difference between the two images;
    # NOTE: the two images must have the same dimension
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar"
    # the two images are
    return err

def psnr(mse):
    # calculating psnr using mse
    if mse != 0: # to avoid divide by 0 error if mse is 0
        return 20*math.log(mse, 10)*((255^2)/mse)
    return 0

def ssimilarity(imageA, imageB):
    # converting to greyscale
    iA = cv2.cvtColor(imageA, cv2.COLOR_BGR2GRAY)
    iB = cv2.cvtColor(imageB, cv2.COLOR_BGR2GRAY)
    return ssim(iA,iB)

def compare_images(imageA, imageB, title):
    # compute the mean squared error and structural similarity
    # index for the images
    m = mse(imageA, imageB)
    s = ssimilarity(imageA, imageB)
    p = psnr(m)

    print(title, "\t=> MSE: %.2f, SSIM: %.2f, PSNR: %.2f" % (m, s, p))
```

```
ori = cv2.imread(os.path.join('RESULT','dog_1.jpg'))
ori5 = cv2.imread(os.path.join('RESULT',"dog_0.05.jpg"))
ori2 = cv2.imread(os.path.join('RESULT',"dog_0.002.jpg"))
ori1 = cv2.imread(os.path.join('RESULT',"dog_0.01.jpg"))

compare_images(ori, ori, "Original vs Original Image")
compare_images(ori, ori5, "Original vs 5.0% of FFT")
compare_images(ori, ori1, "Original vs 1.0% of FFT")
compare_images(ori, ori2, "Original vs 0.2% of FFT")
```

Fig: Performance indicator Implementation

Appendix 2: ***Sequential row-wise and column-wise Fourier transform***

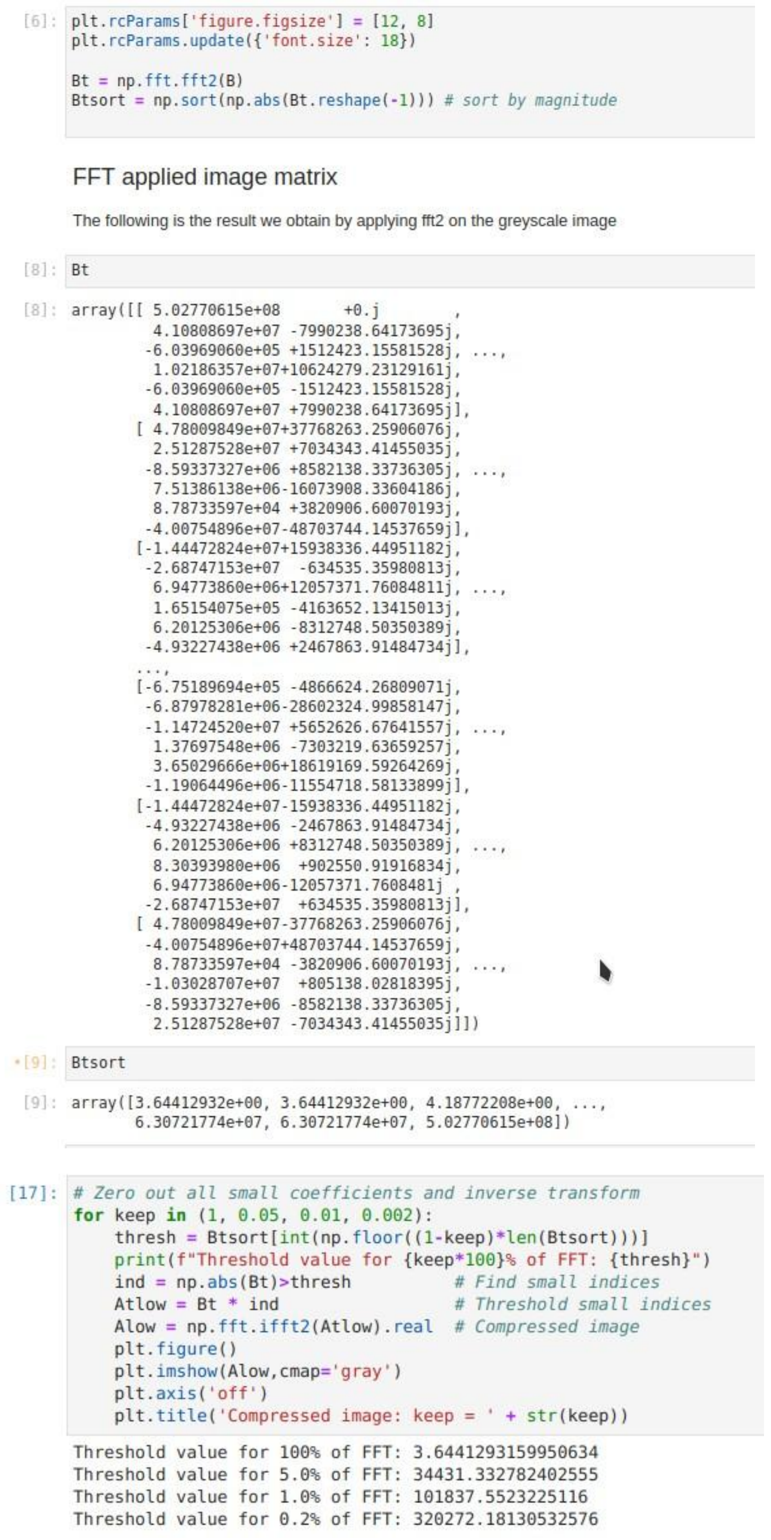


```
•[1]: from matplotlib.image import imread
      import numpy as np
      import matplotlib.pyplot as plt
      import os
```

```
•[3]: plt.rcParams['figure.figsize'] = [12, 8]
      plt.rcParams.update({'font.size': 18})

      A = imread(os.path.join('DATA','dog.jpg'))
      B = np.mean(A, -1); # Convert RGB to grayscale

      fig,axs = plt.subplots(1,3)

      # Plot image
      img = axs[0].imshow(B)
      img.set_cmap('gray')
      axs[0].axis('off')

      #
      # Compute row-wise FFT
      #
      Cshift = np.zeros_like(B,dtype='complex_')
      C = np.zeros_like(B,dtype='complex_')

      for j in range(B.shape[0]):
          Cshift[j,:] = np.fft.fftshift(np.fft.fft(B[j,:]))
          C[j,:] = np.fft.fft(B[j,:])

      # Plot row-wise fft image
      img = axs[1].imshow(np.log(np.abs(Cshift)))
      img.set_cmap('gray')
      axs[1].axis('off')

      #
      # Compute column-wise FFT
      #
      D = np.zeros_like(C)
      for j in range(C.shape[1]):
          D[:,j] = np.fft.fft(C[:,j])

      # plot column-wise on row-wise image
      img = axs[2].imshow(np.fft.fftshift(np.log(np.abs(D))))
      img.set_cmap('gray')
      axs[2].axis('off')

      plt.show()

      # Much more efficient to use fft2
      D = np.fft.fft2(B)
```

Fig: Sequential row-wise and column-wise Fourier transform

Appendix 3: FFT, Filtering and IFFT for image compression



```
[6]: plt.rcParams['figure.figsize'] = [12, 8]
     plt.rcParams.update({'font.size': 18})

     Bt = np.fft.fft2(B)
     Btsort = np.sort(np.abs(Bt.reshape(-1))) # sort by magnitude
```

FFT applied image matrix

The following is the result we obtain by applying fft2 on the greyscale image

```
[8]: Bt

[8]: array([[ 5.02770615e+08       +0.j        ,
              4.10808697e+07 -7990238.64173695j,
             -6.03969060e+05 +1512423.15581528j, ...,
              1.02186357e+07+10624279.23129161j,
             -6.03969060e+05 -1512423.15581528j,
              4.10808697e+07 +7990238.64173695j],
            [ 4.78009849e+07+37768263.25906076j,
              2.51287528e+07 +7034343.41455035j,
             -8.59337327e+06 +8582138.33736305j, ...,
              7.51386138e+06-16073908.33604186j,
              8.78733597e+04 +3820906.60070193j,
             -4.00754896e+07-48703744.14537659j],
            [-1.44472824e+07+15938336.44951182j,
             -2.68747153e+07  -634535.35980813j,
              6.94773860e+06+12057371.76084811j, ...,
              1.65154075e+05 -4163652.13415013j,
              6.20125306e+06 -8312748.50350389j,
             -4.93227438e+06 +2467863.91484734j],
            ...,
            [-6.75189694e+05 -4866624.26809071j,
             -6.87978281e+06-28602324.99858147j,
             -1.14724520e+07 +5652626.67641557j, ...,
              1.37697548e+06 -7303219.63659257j,
              3.65029666e+06+18619169.59264269j,
             -1.19064496e+06-11554718.58133899j],
            [-1.44472824e+07-15938336.44951182j,
             -4.93227438e+06 -2467863.91484734j,
              6.20125306e+06 +8312748.50350389j, ...,
              8.30393980e+06  +902550.91916834j,
              6.94773860e+06-12057371.7608481j ,
             -2.68747153e+07  +634535.35980813j],
            [ 4.78009849e+07-37768263.25906076j,
             -4.00754896e+07+48703744.14537659j,
              8.78733597e+04 -3820906.60070193j, ...,
             -1.03028707e+07  +805138.02818395j,
             -8.59337327e+06 -8582138.33736305j,
              2.51287528e+07 -7034343.41455035j]])
```

```
*[9]: Btsort

[9]: array([3.64412932e+00, 3.64412932e+00, 4.18772208e+00, ...,
            6.30721774e+07, 6.30721774e+07, 5.02770615e+08])
```

```
[17]: # Zero out all small coefficients and inverse transform
      for keep in (1, 0.05, 0.01, 0.002):
          thresh = Btsort[int(np.floor((1-keep)*len(Btsort)))]
          print(f"Threshold value for {keep*100}% of FFT: {thresh}")
          ind = np.abs(Bt)>thresh          # Find small indices
          Atlow = Bt * ind                 # Threshold small indices
          Alow = np.fft.ifft2(Atlow).real  # Compressed image
          plt.figure()
          plt.imshow(Alow,cmap='gray')
          plt.axis('off')
          plt.title('Compressed image: keep = ' + str(keep))

      Threshold value for 100% of FFT: 3.6441293159950634
      Threshold value for 5.0% of FFT: 34431.332782402555
      Threshold value for 1.0% of FFT: 101837.5523225116
      Threshold value for 0.2% of FFT: 320272.18130532576
```

Fig: FFT, Filtering and IFFT for image compression