Discrete Fourier Transform in Image Compression and De-Noising

Ansh Anurag
2K20/B2/71

Ananya Ranjan
2K20/B2/57

Overview

A central concern of mathematical physics and engineering mathematics involves the transformation of equations into a coordinate system where expressions simplify, decouple, and are amenable to computation and analysis. Perhaps the most foundational and ubiquitous coordinate transformation was introduced by J.-B. Joseph Fourier in the early 1800s to investigate the theory of heat. Fourier introduced the concept that sine and cosine functions of increasing frequency provide an orthogonal basis for the space of solution functions. Indeed, the Fourier transform basis of sines and cosines serve as eigenfunctions of the heat equation, with the specific frequencies serving as the eigenvalues, determined by the geometry, and amplitudes determined by the boundary conditions. Fourier's seminal work provided the mathematical foundation for Hilbert spaces, operator theory, approximation theory, and the subsequent revolution in analytical and computational mathematics. Fast forward two hundred years, and the fast Fourier transform has become the cornerstone of computational mathematics, enabling real-time image and audio compression, global communication networks, modern devices and hardware, numerical physics and engineering at scale, and advanced data analysis. Simply put, the fast Fourier transform has had a more significant and profound role in shaping the modern world than any other algorithm to date.
In this report we will try to demonstrate the applications of fourier transform.

1. Fourier Series and the Fourier Transform

Before describing the computational implementation of Fourier transforms on vectors of data, here we introduce the analytic Fourier series and Fourier transform, defined for continuous functions. Naturally, the discrete and continuous formulations should match in the limit of data with infinitely fine resolution. The Fourier series and transform are intimately related to the geometry of infinite-dimensional function spaces, or Hilbert spaces, which generalize the notion of vector spaces to include functions with infinitely many degrees of freedom. Thus, we begin with an introduction to function spaces.

1.1 Inner products of functions and vectors

In particular, we will use the common Hermitian inner product for functions f(x) and g(x) defined for x on a domain x ∈ [a, b]:

$$\langle f(x), g(x)\rangle = \int_a^b f(x)\bar{g}(x)\,dx$$

Where $\bar{g}$ denotes the complex conjugate.
In particular, if we discretize the functions f(x) and g(x) into vectors of data, as in Fig. 2.1, we would like the vector inner product to converge to the function inner product as the sampling resolution is increased. The inner product of the data vectors f $=[f_1\ \ f_2\ f_3\ \ldots\ f_n]^T$ and g $=[g_1\ \ g_2\ g_3\ \ldots\ g_n]^T$ is defined by:

$$\langle \mathbf{f}, \mathbf{g}\rangle = \mathbf{g}^*\mathbf{f} = \sum_{k=1}^{n} f_k\bar{g}_k = \sum_{k=1}^{n} f(x_k)\bar{g}(x_k).$$

The magnitude of this inner product will grow as more data points are added; i.e., as n increases. Thus, we may normalize by $\Delta x = (b - a)/(n - 1)$:

$$\frac{b-a}{n-1}\langle \mathbf{f}, \mathbf{g}\rangle = \sum_{k=1}^{n} f(x_k)\bar{g}(x_k)\Delta x,$$

which is the Riemann approximation to the continuous function inner product. It is now clear that as we take the limit of $n \to \infty$ (i.e., infinite data resolution, with $\Delta x \to 0$), the vector inner product converges to the inner product of functions in (2.1).
This inner product also induces a norm on functions, given by,

$$\|f\|_2 = (\langle f, f\rangle)^{1/2} = \sqrt{\langle f, f\rangle} = \left(\int_a^b f(x)\bar{f}(x)\,dx\right)^{1/2}.$$

As in finite-dimensional vector spaces, the inner product may be used to project a function into a new coordinate system defined by a basis of orthogonal functions. A Fourier series representation of a function f is precisely a projection of this function onto the orthogonal set of sine and cosine functions with integer period on the domain [a, b]. This is the subject of the following sections.

1.2 Fourier Series

A fundamental result in Fourier analysis is that if f(x) is periodic and piecewise smooth, then it can be written in terms of a Fourier series, which is an infinite sum of cosines and sines of increasing frequency. In particular, if f(x) is 2π periodic, it may be written as:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos(kx) + b_k \sin(kx)\right).$$

The coefficients a_k and b_k are given by

$$a_k = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(kx) dx$$
$$b_k = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(kx) dx,$$

which may be viewed as the coordinates obtained by projecting the function onto the orthogonal cosine and sine basis {cos(kx), sin(kx)} as k goes from 0 to ∞.

The Fourier series for an L-periodic function on [0, L) is similarly given by:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos\left(\frac{2\pi kx}{L}\right) + b_k \sin\left(\frac{2\pi kx}{L}\right)\right),$$

The coefficients a_k and b_k are given by

$$a_k = \frac{2}{L} \int_0^L f(x) \cos\left(\frac{2\pi kx}{L}\right) dx$$
$$b_k = \frac{2}{L} \int_0^L f(x) \sin\left(\frac{2\pi kx}{L}\right) dx.$$

Because we are expanding functions in terms of sine and cosine functions, it is also natural to use Euler's formula e^{ikx} = cos(kx)+isin(kx) to write a Fourier series in complex form with complex coefficients ck = αk + iβk:

$$f(x) = \sum_{k=-\infty}^{\infty} c_k e^{ikx} = \sum_{k=-\infty}^{\infty} (\alpha_k + i\beta_k)\left(\cos(kx) + i \sin(kx)\right)$$

$$= (\alpha_0 + i\beta_0) + \sum_{k=1}^{\infty} \left[(\alpha_{-k} + \alpha_k)\cos(kx) + (\beta_{-k} - \beta_k)\sin(kx)\right]$$

$$+ i \sum_{k=1}^{\infty} \left[(\beta_{-k} + \beta_k)\cos(kx) - (\alpha_{-k} - \alpha_k)\sin(kx)\right].$$

If f(x) is real-valued, then $\alpha_{-k} = \alpha_k$ and $\beta_{-k} = -\beta_k$.

Thus, the functions $\psi_k = e^{ikx}$ for k ∈ Z (i.e., for integer k) provide a basis for periodic, complex-valued functions on an interval [0, 2π). It is simple to see that these functions are orthogonal:

$$\langle \psi_j, \psi_k \rangle = \int_{-\pi}^{\pi} e^{ijx} e^{-ikx} dx$$

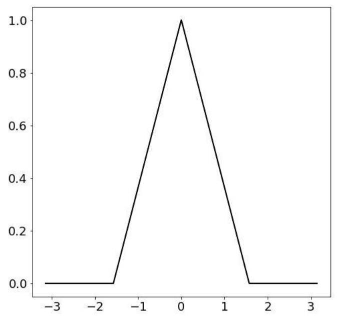
$$= \int_{-\pi}^{\pi} e^{i(j-k)x} dx = \left[\frac{e^{i(j-k)x}}{i(j-k)}\right]_{-\pi}^{\pi} = \begin{cases} 0 & \text{if } j \neq k \\ 2\pi & \text{if } j = k. \end{cases}$$

So $\langle\psi_j, \psi_k\rangle = 2\pi\delta_{jk}$, where δ is the Kronecker delta function. Similarly, the functions $e^{i2\pi kx/L}$ provide a basis for L^2 ([0, L)), the space of square integrable functions defined on x ∈ [0, L). In principle, a Fourier series is just a change of coordinates of a function f(x) into an infinite-dimensional orthogonal function space spanned by sines and cosines (i.e., $\psi_k = e^{ikx}$ = cos(kx) + isin(kx)):

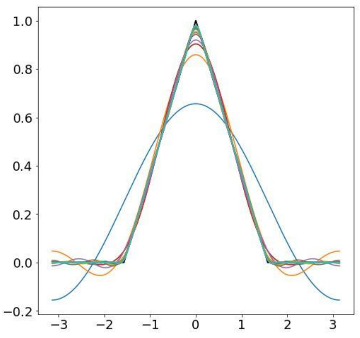
$$f(x) = \sum_{k=-\infty}^{\infty} c_k \psi_k(x) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} \langle f(x), \psi_k(x)\rangle \psi_k(x).$$

Example: Fourier series for a continuous hat function As a simple example, we demonstrate the use of Fourier series to approximate a continuous hat function, defined from −π to π:

$$f(x) = \begin{cases} 0 & \text{for } x \in [-\pi, \pi/2) \\ 1 + 2x/\pi & \text{for } x \in [-\pi/2, 0) \\ 1 - 2x/\pi & \text{for } x \in [0, \pi/2) \\ 0 & \text{for } x \in [\pi/2, \pi). \end{cases}$$



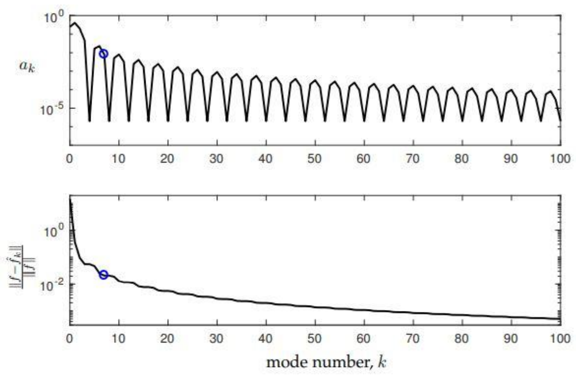
Here we will design and simulate the fourier series corresponding to f(x) and approximate the curve for n=20:



Code for the simulation in Python

```
import numpy as np
import matplotlib.pyplot as plt
import time
from matplotlib.cm import get_cmap

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

# Define domain
dx = 0.001
L = np.pi
x = L * np.arange(-1+dx,1+dx,dx)
n = len(x)
nquart = int(np.floor(n/4))

# Define hat function
f = np.zeros_like(x)
f[nquart:2*nquart] =
(4/n)*np.arange(1,nquart+1)
f[2*nquart:3*nquart] = np.ones(nquart) -
(4/n)*np.arange(0,nquart)

fig, ax = plt.subplots()
ax.plot(x,f,'-',color='k',LineWidth=2)

# Compute Fourier series
name = "Accent"
cmap = get_cmap('tab10')
colors = cmap.colors
ax.set_prop_cycle(color=colors)

A0 = np.sum(f * np.ones_like(x)) * dx
fFS = A0/2

A = np.zeros(20)
B = np.zeros(20)
for k in range(20):
    A[k] = np.sum(f *
np.cos(np.pi*(k+1)*x/L)) * dx # Inner
product
    B[k] = np.sum(f *
np.sin(np.pi*(k+1)*x/L)) * dx
    fFS = fFS +
A[k]*np.cos((k+1)*np.pi*x/L) +
B[k]*np.sin((k+1)*np.pi*x/L)
    ax.plot(x,fFS,'-')
```

Figure shows the coefficients a_k of the even cosine functions, along with the approximation error, for an increasing number of modes. The error decreases monotonically, as expected. The coefficients bk corresponding to the odd sine functions are not shown, as they are identically zero since the hat function is even.



1.2 Fourier Transform

The Fourier series is defined for periodic functions, so that outside the domain of definition, the function repeats itself forever. The Fourier transform integral is essentially the limit of a Fourier series as the length of the domain goes to infinity, which allows us to define a function defined on (−∞,∞) without repeating. We will consider the Fourier series on a domain x ∈ [−L, L), and then let L ⟶ ∞. On this domain, the Fourier series is:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty}\left[a_k \cos\left(\frac{k\pi x}{L}\right) + b_k \sin\left(\frac{k\pi x}{L}\right)\right]$$

$$= \sum_{k=-\infty}^{\infty} c_k e^{ik\pi x/L}$$

Restating the previous results, f(x) is now represented by a sum of sines and cosines with a discrete set of frequencies given by ω_k = kπ/L. Taking the limit as L ⟶ ∞, these discrete frequencies become a continuous range of frequencies. Define ω = kπ/L, Δω = π/L, and take the limit L ⟶ ∞, so that Δω ⟶ 0:

$$f(x) = \lim_{\Delta\omega\to 0} \sum_{k=-\infty}^{\infty} \frac{\Delta\omega}{2\pi} \underbrace{\int_{-\pi/\Delta\omega}^{\pi/\Delta\omega} f(\xi) e^{-ik\Delta\omega\xi}\, d\xi}_{\langle f(x),\psi_k(x)\rangle}\; e^{ik\Delta\omega x}.$$

When we take the limit, the expression <f(x), ψk(x)> will become the Fourier transform of f(x), denoted by Ƒ(f(x)). In addition, the summation with weight Δω becomes a Riemann integral, resulting in the following:

$$f(x) = \mathcal{F}^{-1}\left(\hat{f}(\omega)\right) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega x}\, d\omega$$

$$\hat{f}(\omega) = \mathcal{F}\left(f(x)\right) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x}\, dx.$$

2. Discrete Fourier Transform and the Fast Fourier Transform

Until now, we have considered the Fourier series and Fourier transform for continuous functions f(x). However, when computing or working with realdata, it is necessary to approximate the Fourier transform on discrete vectors of data. The resulting discrete Fourier transform (DFT) is essentially a discretized version of the Fourier series for vectors of data $f = [f_1\ f_2\ f_3 \ldots\ldots f_n]^T$.

The DFT is tremendously useful for numerical approximation and computation, but it does not scale well to very large n>>1, as the simple formulation involves multiplication by a dense n×n matrix, requiring $O(n^2)$ operations.

2.1 Discrete Fourier Transform

Although we will always use the FFT for computations, it is illustrative to begin with the simplest formulation of the DFT. The discrete Fourier transform is given by:

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{-i2\pi jk/n},$$

and the inverse discrete Fourier transform (iDFT) is given by:

$$f_k = \frac{1}{n}\sum_{j=0}^{n-1} \hat{f}_j e^{i2\pi jk/n}.$$

For a given number of points n, the DFT represents the data using sine and cosine functions with integer multiples of a fundamental frequency, $\omega_n = e^{-2\pi i/n}$. The DFT may be computed by matrix multiplication:

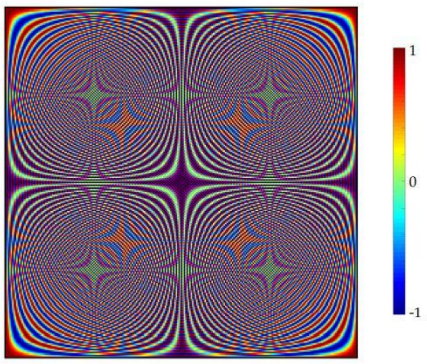
$$\begin{bmatrix}\hat{f}_1\\ \hat{f}_2\\ \hat{f}_3\\ \vdots\\ \hat{f}_n\end{bmatrix} = \begin{bmatrix}1 & 1 & 1 & \cdots & 1\\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1}\\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2}\end{bmatrix}\begin{bmatrix}f_1\\ f_2\\ f_3\\ \vdots\\ f_n\end{bmatrix}.$$

The output vector $\mathcal{F}(f(x))$ contains the Fourier coefficients for the input vector f, and the DFT matrix F is a unitary Vandermonde matrix. The matrix F is complex valued, so the output has both a magnitude and a phase, which will both have useful physical interpretations.

We tried coding the real part of the DFT matrix for n=256

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 8]
plt.rcParams.update({'font.size': 18})
```

```
n = 256
w = np.exp(-1j * 2 * np.pi / n)
DFT = np.zeros((n,n))

# Slow
for i in range(n):
    for k in range(n):
        DFT[i,k] = w**(i*k)

DFT = np.real(DFT)

plt.imshow(DFT)
J,K=
np.meshgrid(np.arange(n),np.arange(n))
DFT = np.power(w,J*K)
DFT = np.real(DFT)

plt.imshow(DFT)
plt.show()
```



2.2 Fast Fourier Transform

As mentioned earlier, multiplying by the DFT matrix F involves O(n 2) operations. The fast Fourier transform scales as O(n log(n)), enabling a tremendous range of applications, including audio and image compression in MP3 and JPG formats, streaming video, satellite communications, and the cellular network, to name only a few of the myriad applications. For example, audio is generally sampled at 44.1 kHz, or 44, 100 samples per second. For 10 seconds of audio, the vector f will have dimension n = 4.41 × 105 . Computing the DFT using matrix multiplication involves approximately 2 × 1011, or 200 billion, multiplications. In contrast, the FFT requires approximately 6 × 106 , which amounts to a speed-up factor of over 30, 000. Thus, the FFT has become synonymous with the DFT, and FFT libraries are built into nearly every device and operating system that performs digital signal processing.

The basic idea behind the FFT is that the DFT may be implemented much more efficiently if the number of data points n is a power of 2. For example, consider n = 1024 = 2^{10}. In this case, the DFT matrix F1024 may be written as:

$$\hat{\mathbf{f}} = \mathbf{F}_{1024}\mathbf{f} = \begin{bmatrix} \mathbf{I}_{512} & -\mathbf{D}_{512} \\ \mathbf{I}_{512} & -\mathbf{D}_{512} \end{bmatrix} \begin{bmatrix} \mathbf{F}_{512} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_{512} \end{bmatrix} \begin{bmatrix} \mathbf{f}_{\text{even}} \\ \mathbf{f}_{\text{odd}} \end{bmatrix},$$

where f_{even} are the even index elements of f, f_{odd} are the odd index elements of f, I_{512} is the 512 × 512 identity matrix, and D_{512} is given by

$$\mathbf{D}_{512} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & \omega & 0 & \cdots & 0 \\ 0 & 0 & \omega^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \omega^{511} \end{bmatrix}.$$

This expression can be derived from a careful accounting and reorganization of the terms in (2.26) and (2.29). If n = 2p , this process can be repeated, and F512 can be represented by F256, which can then be represented by F128 → F64 → F32 → · · · . If n 6≠2p , the vector can be padded with zeros until it is a power of 2. The FFT then involves an efficient interleaving of even and odd indices of subvectors of f, and the computation of several smaller 2 × 2 DFT computations.

3. Noise Filtering Using FFT

Understanding how FFT can be used to filter out undesired frequencies from a component is very important to go forward with image compression using the Fourier Transform.
To gain familiarity with how to use and interpret the FFT, we will begin with a simple example that uses the FFT to denoise a signal. We will consider a function of time f(t) and we will simulate the analysis in python side-by-side:

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize']=[16,12]
plt.rcParams.update({'font.size': 18})
```

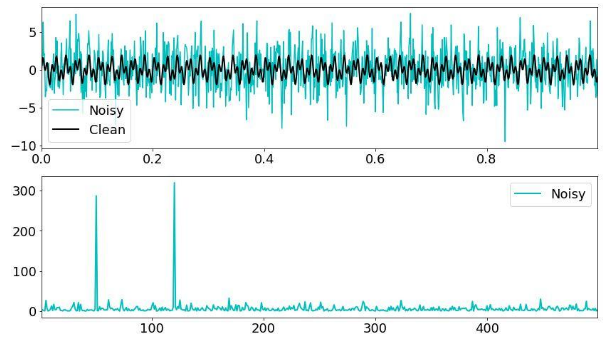
We have the function f(t) as:

$$f(t) = \sin(2\pi f_1 t) + \sin(2\pi f_2 t)$$

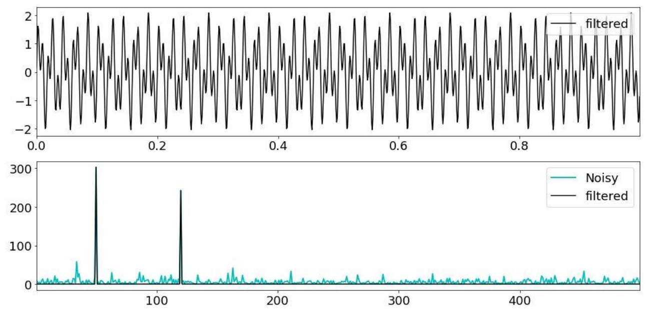
with frequencies f_1 = 50 and f_2 = 120.We then add a large amount of Gaussian white noise to this signal,

```
dt=0.001
t=np.arange(0,1,dt)
f=np.sin(2*np.pi*50*t)+np.sin(2*np.pi*120
*t)
f_clean=f
f=f+2.5*np.random.randn(len(t))
```

Now we find the power spectral density of the wave and then we will filter out those frequencies which have power spectral density less than 100.



```
indices=PSD>100
PSDclean=PSD*indices
fhat=indices*fhat
ffilt=np.fft.ifft(fhat)
fig,axs=plt.subplots(3,1)

plt.sca(axs[0])
plt.plot(t,f,color='c',LineWidth=1.5,labe
l='Noisy')
plt.plot(t,f_clean,color='k',LineWidth=2,
label='Clean')
plt.xlim(t[0],t[-1])
plt.legend()
plt.sca(axs[1])
plt.plot(t,ffilt,color='k',LineWidth=1.5,
label='filtered')
plt.xlim(t[0],t[-1])
plt.legend()
plt.sca(axs[2])
plt.plot(freq[L],PSD[L],color='c',LineWid
th=2,label='Noisy')
plt.plot(freq[L],PSDclean[L],color='k',Li
neWidth=1.5,label='filtered')
plt.xlim(freq[L[0]],freq[L[-1]])
plt.legend()
plt.show()
```

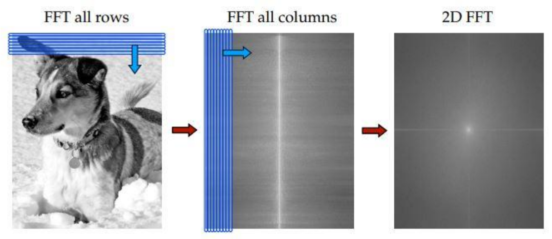


4. Image Processing using Fourier Transform

Although we analyzed both the Fourier transform and the wavelet transform on one-dimensional signals, both methods readily generalize to higher spatial dimensions, such as two-dimensional and three-dimensional signals. Both the Fourier and wavelet transforms have had tremendous impact on image processing and compression, which provides a compelling example to investigate higher-dimensional transforms.

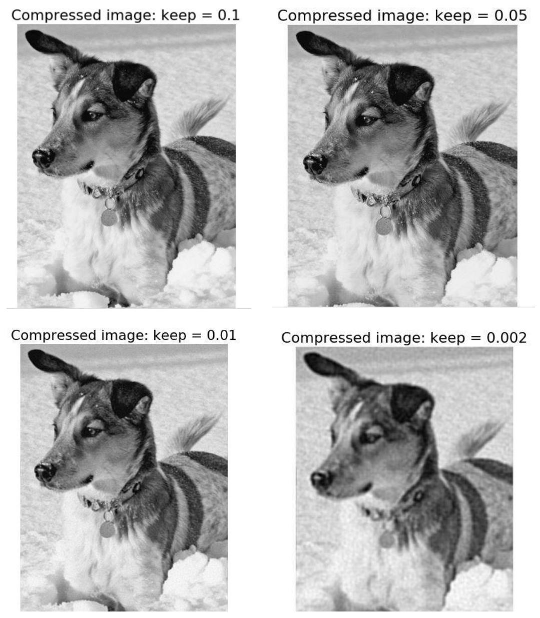
4.1 Fourier Transform for Image Compression

The two-dimensional Fourier transform of a matrix of data X ∈ R $^{n\times m}$ 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. This sequential row-wise and column-wise Fourier transform is shown in Fig. below. Switching the order of taking the Fourier transform of rows and columns does not change the result



```
from matplotlib.image import imread
import numpy as np
import matplotlib.pyplot as plt
import os
plt.rcParams['figure.figsize'] = [12, 8]
plt.rcParams.update({'font.size': 18})

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

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

# Zero out all small coefficients and
inverse transform
for keep in (0.1, 0.05, 0.01, 0.002):
    thresh =
Btsort[int(np.floor((1-keep)*len(Btsort))
)]
    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))
```

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.


Compressed image: keep = 0.1
Compressed image: keep = 0.05
Compressed image: keep = 0.01
Compressed image: keep = 0.002

Compressed image using various thresholds to keep 10%, 5%, 1%, and 0.2% of the largest Fourier coefficients.

4.1 Fourier Transform for Image De-Noising

Finally, the FFT is extensively used for denoising and filtering signals, as it is straightforward to isolate and manipulate particular frequency bands. We can demonstrate the use of a FFT threshold filter to denoise an image with Gaussian noise added. In this example, it is observed that the noise is especially pronounced in high frequency modes, and we therefore zero out any Fourier coefficient outside of a given radius containing low frequencies.

```
## Denoise
Bnoise = B +
200*np.random.randn(*B.shape).astype('uin
t8') # Add some noise
Bt = np.fft.fft2(Bnoise)
Btshift = np.fft.fftshift(Bt)
F = np.log(np.abs(Btshift)+1)
# Put FFT on log scale

fig,axs = plt.subplots(2,2)

axs[0,0].imshow(Bnoise,cmap='gray')
axs[0,0].axis('off')

axs[0,1].imshow(F,cmap='gray')
axs[0,1].axis('off')

nx,ny = B.shape
X,Y=np.meshgrid(np.arange(-ny/2+1,ny/2+1)
,np.arange(-nx/2+1,nx/2+1))
R2 = np.power(X,2) + np.power(Y,2)
ind = R2 < 150**2
Btshiftfilt = Btshift * ind
Ffilt = np.log(np.abs(Btshiftfilt)+1)
# Put FFT on log scale

axs[1,1].imshow(Ffilt,cmap='gray')
axs[1,1].axis('off')

Btfilt = np.fft.ifftshift(Btshiftfilt)
Bfilt = np.fft.ifft2(Btfilt).real
axs[1,0].imshow(Bfilt,cmap='gray')
axs[1,0].axis('off')

plt.show()
```