# Fourier Transform Final Project

Alex, Jeewon, Lucas, Merrick

April 17, 2023

## 1 Theory

### 1.1 Fourier Series

We will begin our discussion by first defining the Fourier series. To do so, we will invoke the following definition of the $L^2$ inner product:

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

For sake of example, say we have $f(x)$ defined from $-\pi$ to $\pi$.

Then, we can represent $f(x)$ as

$$\frac{A_0}{2} + \sum_{k=1}^{\infty}(A_k \cos(kx) + B_k \sin(kx))$$

where A and B are called *Fourier coefficients* and $\cos(kx)$ and $\sin(kx)$ are the cosines and sines of period 1, 2, $\ldots, \infty$. Note that we have chosen $\cos(kx)$ and $\sin(kx)$ here as they are $2\pi$ periodic, and $f$ is defined over a length of $2\pi$. We might think of the coefficients $A_k$ and $B_k$ as telling us how much of each cosine and sine of period 1, period 2, etc. we need to yield $f$.

We find $A$ and $B$ as follows:

- $A_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(kx)dx$
  - You might notice that this looks a lot like the inner product of $f(x)$ and $\cos(kx)$, which is exactly what it is!
- Similarly, $B_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(kx)dx$

In other words, the $k$th coefficient is the inner product of $f$ and the $k$th cosine or sine wave. That is,

$$A_k = \frac{1}{||\cos(kx)||^2}\langle f(x), \cos(kx)\rangle$$

and similarly for $B_k$. This is equivalent to the projection of the function onto the cosine or sine wave.

If $A_k$ and $B_k$ are projections of $f$ onto the orthogonal basis $\{\cos(kx), \sin(kx)\}$, we can think about the Fourier series as an expression of $f$ in the "coordinates" of $\{\cos(kx), \sin(kx)\}$. Recall that for a vector $\vec{f}$ in the $xy$-space, we can express $\vec{f}$ as

$$\vec{f} = \left\langle \vec{f}, \vec{x} \right\rangle \frac{\vec{x}}{||x||^2} + \left\langle \vec{f}, \vec{y} \right\rangle \frac{\vec{y}}{||y||^2}$$

or a sum of the components of $\vec{f}$ that are in the $x$-direction and $y$-direction. For a Fourier series, we replace $x$ and $y$ with $\cos(kx)$ and $\sin(kx)$.

In the previous example, we had a function defined from $-\pi$ to $\pi$, or a domain length of $2\pi$, that was expanded as an infinite sum. In practice, we often approximate the Fourier series by summing to domain length $L$ by using a *truncated Fourier series.* We then arrive at the following familiar definition of the Fourier series:

$$\frac{A_0}{2} + \sum_{k=1}^{\infty} \left( A_k \cos\left(\frac{2\pi k}{L} x\right) + B_k \sin\left(\frac{2\pi k}{L} x\right)\right)$$

where

$$A_k = \frac{2}{L} \int_0^L f(x) \cos\left(\frac{2\pi k}{L} x\right) dx$$

and

$$B_k = \frac{2}{L} \int_0^L f(x) \sin\left(\frac{2\pi k}{L} x\right) dx$$

### 1.1.1 Complex Fourier Series

While we will be focusing primarily on real-valued functions in the paper, we can certainly write complex-valued functions as Fourier series. Say we want to express a complex-valued function, once again defined from $-\pi$ to $\pi$, as a Fourier series. Then, we can write the function in the exponential form of the Fourier series:

$$f(x) = \sum_{k=-\infty}^{\infty} c_k e^{ikx}$$

where $c_k$ is some complex constant $(\alpha_k + i\beta_k)$, which is further defined in Section 1.3. Using Euler's identity, we can equivalently write this as

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

In the general case in which a function is defined $-L \leq x \leq L$, we write

$$f(x) = \sum_{k=-\infty}^{\infty} c_k e^{\frac{ikx\pi}{L}}$$

The intuition here is the same: we are once again projecting on to the orthogonal basis

$$\left\{ e^{\frac{ikx\pi}{L}} \right\}_{k=-\infty}^{\infty}$$

## 1.2 The Fourier Transform

The Fourier transform is the generalized Fourier series for a function defined on an infinite domain rather than $L$. It is simply a Fourier series as we take $L \to \infty$. We will use the following definitions to simplify our notation:

$$\begin{aligned}
f(x) &= \sum_{k=-\infty}^{\infty} c_k e^{\frac{ik\pi x}{L}} \\
c_k &= \frac{1}{2\pi} \langle f(x), \psi_k \rangle = \frac{1}{2L} \int_{-L}^{L} f(x) e^{\frac{-ik\pi x}{L}} dx \\
\omega_k &= \frac{k\pi}{L} = k\Delta\omega \\
\Delta\omega &= \frac{\pi}{L}
\end{aligned}$$

where $\omega_k$ is the frequency. We can equivalently write the limit as $L \to \infty$ as the limit as $\Delta\omega \to 0$. So, we see that for some dummy variable $\xi$,

$$\begin{aligned}
f(x) &= \lim_{L\to\infty} \sum_{k=-\infty}^{\infty} \frac{1}{2L} \int_{-L}^{L} f(x) e^{\frac{-ik\pi x}{L}} dx e^{\frac{ik\pi x}{L}} \\
&= \lim_{\Delta\omega\to 0} \sum_{k=-\infty}^{\infty} \frac{\Delta\omega}{2\pi} \int_{\frac{\pi}{\Delta\omega}}^{\frac{\pi}{\Delta\omega}} f(\xi) e^{-ik\Delta\omega\xi} d\xi e^{ik\Delta\omega x} \\
&= \int_{-\infty}^{\infty} \frac{1}{2\pi} \int_{-\infty}^{\infty} f(\xi) e^{-i\omega\xi} d\xi e^{i\omega x} d\omega
\end{aligned}$$

From here, let

$$\begin{aligned}
\hat{f}(\omega) &= \int_{-\infty}^{\infty} f(\xi) e^{-i\omega x} d\xi \\
&= \int_{-\infty}^{\infty} f(x) e^{-i\omega x} dx \\
&= \mathcal{F}(f(x))
\end{aligned}$$

which is the inner product of $f$ with the basis functions previously discussed. Therefore, these are analogous to our $c_k$ coefficients, with the difference being that $\hat{f}$ is continuous. $\hat{f}(\omega)$ is known as the *Fourier transform.* To get our function $f(x)$, we can simply take the inverse of $\mathcal{F}$. So, returning to $f(x)$,

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

Here, our $\omega$ is continuously varying and not just a multiple of $\frac{\pi}{L}$. $f(x)$ and $\mathcal{F}$ are known as the *Fourier transform pair.*

Note that this Fourier Transform only works for $f(x)$ if it decays as $x \to \infty$ and $x \to -\infty$

### 1.2.1 Fourier Transform and Derivatives

An interesting thing to note is the usefulness of the Fourier transform for finding the derivatives of a function $f(x)$. We will show this by taking the Fourier transform of the derivative of a function:

$$\begin{aligned}
\mathcal{F}\left(\frac{d}{dx} f(x)\right) &= \int_{-\infty}^{\infty} \frac{df}{dx} e^{-i\omega x} dx \\
&= \left[f(x) e^{-i\omega x}\right]_{-\infty}^{\infty} - \int_{-\infty}^{\infty} f(x) \left(-i\omega e^{-i\omega x}\right) dx \\
&= i\omega \int_{-\infty}^{\infty} f(x) e^{-i\omega x} dx \\
&= i\omega \mathcal{F}\left(f(x)\right)
\end{aligned}$$

This means that we can compute the derivative of a function $f(x)$ in the Fourier domain, which simply becomes a product of $i\omega$ and $\mathcal{F}(f(x))$. We can take the inverse transform of $\mathcal{F}$ to find the derivative of $f(x)$.

## 1.3 Discrete Fourier Transform (DFT)

The DFT is used to compute the Fourier Transform in a computer. It is a mathematical transformation that can be written in terms of matrix multiplication. We use the DFT when we have measurement data defined at discrete locations and we believe that there is a continuous function underlying this data. So, we have a vector of data

$$f = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}$$

at points $x_0, x_1, \ldots, x_n$ which we will want to compute the discrete Fourier series of. We will want to get a vector of Fourier coefficients:

$$\hat{f} = \begin{bmatrix} \hat{f}_0 \\ \hat{f}_1 \\ \vdots \\ \hat{f}_n \end{bmatrix}$$

which is a Fourier vector of frequency components. Each $\hat{f}_i$ tells us how much of each frequency we need to add up to reconstruct this data in $f$. We can find this vector by solving

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

Just like with the Fourier transform pair, you can go from the data to the Fourier transform and vice versa using the following equation:

$$f_k = \frac{1}{n}\left(\sum_{j=0}^{n-1} \hat{f}_j e^{\frac{i2\pi jk}{n}}\right)$$

We notice that each equation contains

$$\omega_n = e^{\frac{-2\pi i}{n}}$$

which defines a *fundamental frequency* which determines the cosine and sine functions that can be approximated with the $n$-vector of data points.

It would be tedious to compute every single value of $\hat{f}_k$, so we wish to find some matrix which we can multiply by our vector of data that will give us the Fourier transform vector. That is, we wish to find $W$ such that

$$\begin{bmatrix} \hat{f}_0 \\ \hat{f}_1 \\ \vdots \\ \hat{f}_n \end{bmatrix} = W \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}$$

We will see that we can write $W$ in terms of the fundamental frequency $\omega_n$. We will show what the values of $W$ would look like by iterating through different values of $k$.

Say $k = 0$, then we see the the exponential term in the sum is raised to a power of 0 and thus the coefficient for every single data point $f_j$ is 1.

Now, say $k = 1$. In this case, when $j = 0$, we again have the exponent raised to a power of 0. For $j = 1$, the exponent is raised to $\frac{-i2\pi}{n}$, which gives $e^{\frac{-i2\pi}{n}}$, which is our $\omega_n$. By the same argument, when $j = 2$ we have $e^{\frac{-i2\pi}{n}}$ or $\omega_n^2$.

By following the same logic for all $k$, we can get $W$ as:

$$W = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

This matrix $W$ is known as the *Vandermonde matrix.* Thus, we can calculate the Fourier transform vector with the following matrix product:

$$\begin{bmatrix} \hat{f}_0 \\ \hat{f}_1 \\ \hat{f}_2 \\ \vdots \\ \hat{f}_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

In most cases, it is fairly expensive to compute the matrix and multiply the data to compute the Fourier Transform. This is where the Fast Fourier Transform comes in–it allows us to efficiently compute this transform.

## 1.4 Fast Fourier Transform (FFT)

As perhaps one of the most important algorithms ever developed, the Fast Fourier Transform is the enabling technology in most digital communication, audio compression, image compression, and much more. The FFT has essentially become synonymous with the DFT, in the sense that you'll never compute the DFT without using the FFT algorithm.

The DFT is an order $n^2$ calculation, so as $n$ gets increasingly larger, the calculation becomes significantly more expensive. On the other hand, the FFT is an order $n\log(n)$ calculation, which is significantly faster.

We start by considering the case where n is a power of 2. We observe that when n is a power of 2, then the Fourier Transform $\hat{f}$ can be simplified by reordering the entries of $f$ in the calculation $\hat{f} = Wf$. Say $n = 2^{10} = 1024$. In this case we have:

$$\hat{f} = W_{1024}f = \begin{bmatrix} I_{512} & D_{512} \\ I_{512} & -D_{512} \end{bmatrix} \begin{bmatrix} F_{512} & 0 \\ 0 & F_{512} \end{bmatrix} \begin{bmatrix} f_{even} \\ f_{odd} \end{bmatrix}$$

Where I is the identity matrix, D is a diagonal matrix, and $f_{even}$ consists of the even entries of $f$, i.e. $f_0, f_2, \ldots, f_n$, and similarly $f_{odd}$ consists of $f_1, f_3, \ldots, f_{n-1}$ stacked on top of each other. Both of the first two matrices in the product are diagonal, which makes the computation significantly easier and more efficient. The diagonal matrix $D$ is defined as:

$$D = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & \omega & 0 & \cdots & 0 \\ 0 & 0 & \omega^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & w^{511} \end{bmatrix}$$

We could go even further and split $F_{512}$ into smaller matrices and take the even and odd indices of $f_{even}$ (and similarly for $f_{odd}$) as we did for $F_{1024}$. We are essentially taking advantage of redundancies in our matrix $W$ and cutting it in half many times.

## 1.5 The Gabor Transform or the Short-Time Fourier Transform

Looking at a signal as a function of time has great precision about where in time events occur, but we don't know what frequencies are happening at those moments. On the other hand, when we look at the Fourier data of the signal, we have excellent data about which frequencies are present, but we do not know when those frequencies occurred. The *Gabor Transform* or the *short-time Fourier transform* (STFT) allows us to compute the *spectrogram*, which is a time-frequency plot of which frequencies are present at which time for a signal.

For signals that are perfectly periodic, the Fourier transform is great as the time information is not particularly relevant to the analysis of the signal. However, for real signals (for example, audio files) this is not as useful, since they are not perfectly periodic.

Suppose we have a signal as a function of time $f(t)$.

To get the Gabor transform of $f$, we take a short window over a portion of the signal and convolve it with the Fourier transform. If we do this repeatedly over the entire signal, then we are able to extract a time-frequency plot.

Let $g_{\tau,\omega}(\tau) = e^{i\omega\tau}g(\tau - t)$. This is the function defining our window. The kernel of this is $g(t)$, and is often chosen to be Gaussian, ie $g(t) = e^{t-\tau^2/a^2}$ Here, $a$ determines how wide the window is for the Fourier transform, and $\tau$ determines the center of the window. The Gabor transform, given by $G(f)$ is:

$$G(f) = \hat{f}_g(t,\omega) = \int_{-\infty}^{\infty} f(\tau)e^{-i\omega\tau}\bar{g}(t-\tau)d\tau =< f, g_{t,\omega} >$$

Since we are taking window, there is a tradeoff between resolution in the time domain and resolution in the frequency domain. That is, a narrower time window will result in better resolution in time, but a lower resolution in frequency. On the other hand, a wider time window will result in more precise frequency data, but will have lower time resolution.

### 1.5.1 Discrete Implementation

In practice, similar to the FFT, the Gabor transform is used on discrete signals. Thus, the function is performed in a discretized manner. We instead set $\omega = j\Delta\nu$ and $\tau = k\Delta t$

The kernel function then becomes:

$$g_{j,k} = e^{i2\pi j \Delta\nu t} g(t - k\delta t)$$

and the Gabor transform becomes:

$$G_{\Delta} = \hat{f_{j,k}} =< f, g_{j,k} >= \int_{-\infty}^{\infty} f(\tau) \bar{g_{j,k}}(\tau) d\tau$$

Which is then approximated by taking a finite Reimman sum.

## 2 Algorithms

For the algorithms section of the project, I did a manual implementation of the DFT and FFT algorithms.

I compared my implementations with the built-in implemnntations from Scipy and confirmed that they produced the same results.

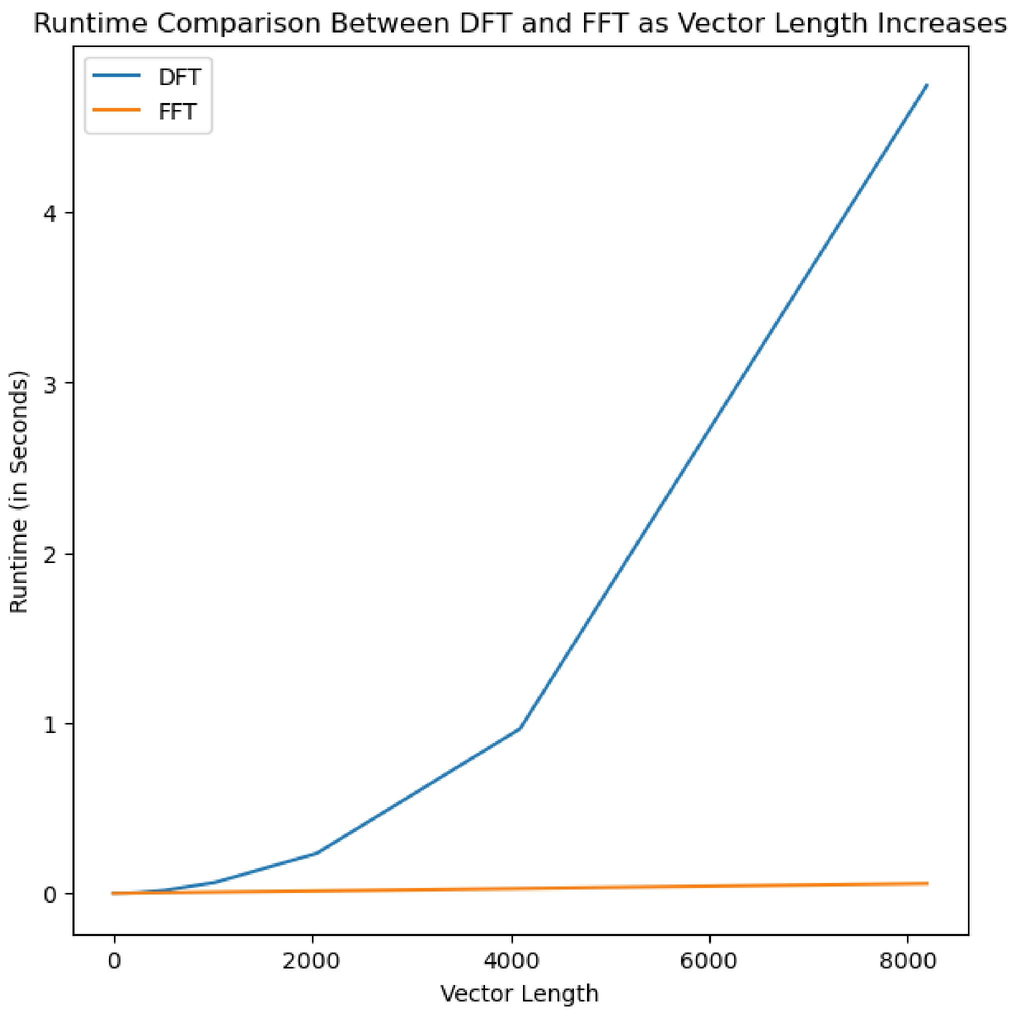
[6]:
```python
import matplotlib.pyplot as plt
import numpy as np
import scipy

#1st implementation of DFT
#does not use meshgrid
def DFT_simple(x):
    N = np.size(x)
    #creates the final vector that will store the fourier transform of x
    x_fourier_transform = np.zeros((N,),dtype= np.complex128)
    #loop through every row of the DFT matrix
    for m in range(0,N):
        #multiply each of the n values in the mth row of the DFT matrix by x[n]␣
↪to get the fourier transformed value x_fourier_transform[m] and add it to␣
↪the final vector
        for n in range(0,N):
            x_fourier_transform[m] += x[n]*np.exp(-np.pi*2j*m*n/N)
    return x_fourier_transform

#2nd implementation of DFT
#uses meshgrid
def DFT_meshgrid(x):
    N = np.size(x)
    #creates the DFT matrix
    J,K = np.meshgrid(np.arange(N), np.arange(N))
    w = np.exp(-np.pi*2j/N)
    DFT_matrix = np.power(w, J*K)
    #matrix multiply it by x
    return np.matmul(DFT_matrix, x)

#recursive implementation of FFT
def FFT(x):
    """
    note: input should have a length
    of a power of 2
    """

    N = len(x)
    if N == 1:
        return x
    #edge case handling for if not a power of 2
    if N % 2 > 0:
        raise ValueError("size of x must be a power of 2")
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = np.exp(-2j*np.pi*np.arange(N)/ N)
        X = np.concatenate([X_even+factor[:int(N/2)]*X_odd, X_even+factor[int(N/
↪2):]*X_odd])
        return X




#create a vector of 1024 random samples from a uniform distribution over [0, 1)
x = np.random.rand(1024,)

# compute DFT using our functions
X_dft_simple = DFT_simple(x)
X_dft_meshgrid = DFT_meshgrid(x)

# compute DFT using built-in scipy functions
X_dft_scipy = np.matmul(scipy.linalg.dft(np.size(x)), x)

# compute FFT using our function
X_fft_manual = FFT(x)

# compute FFT using built-in scipy functions
X_fft_scipy = scipy.fft.fft(x)

# now compare our estimates with the scipy estimates
print('Is our first manual estimate of DFT close to the scipy estimate of DFT?
↪',np.allclose(X_dft_simple - X_dft_scipy,1e-12))
print('Is our second manual estimate of DFT close to the scipy estimate of DFT?
↪',np.allclose(X_dft_meshgrid - X_dft_scipy,1e-12))
print('Is our  manual estimate of FFT close to the scipy estimate of DFT?',np.
↪allclose(X_fft_manual - X_dft_scipy,1e-12))

print('Is our first manual estimate of DFT close to the scipy estimate of FFT?
↪',np.allclose(X_dft_simple - X_fft_scipy,1e-12))
print('Is our second manual estimate of DFT close to the scipy estimate of FFT?
↪',np.allclose(X_dft_meshgrid - X_fft_scipy,1e-12))
```

```
print('Is our manual estimate of FFT close to the scipy estimate of FFT?',np.
↪allclose(X_fft_manual - X_fft_scipy,1e-12))
```

```
Is our first manual estimate of DFT close to the scipy estimate of DFT? True
Is our second manual estimate of DFT close to the scipy estimate of DFT? True
Is our  manual estimate of FFT close to the scipy estimate of DFT? True
Is our first manual estimate of DFT close to the scipy estimate of FFT? True
Is our second manual estimate of DFT close to the scipy estimate of FFT? True
Is our manual estimate of FFT close to the scipy estimate of FFT? True
```

Next, I graphed the runtime of my DFT and FFT algorithms for vectors of increasingly length to verify that DFT has O(Nˆ2) runtime complexity and FFT has O(n(log(n))) runtime complexity.

[5]:
```
import time

# Simulation Parameters

#We do multiple iterations to help reduce random variance in runtime and get a␣
↪more realistic estimate
num_iterations   = 6

#We'll test our algorithms on increasingly long vectors to track how the␣
↪runtime increases
#Note: FFT requires vector length to be a power of 2
sample_size = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]


# We'll store all the runtimes for DFT and FFT in these matrices to be graphed␣
↪later
DFT_time = np.zeros((num_iterations, len(sample_size)))
FFT_time = np.zeros((num_iterations, len(sample_size)))

#generates
for j in range(len(sample_size)):
    #as j increases, numSamples iterates through the powers of 2
    num_samples = sample_size[j]

    #create a vector of random samples from a uniform distribution over [0, 1)␣
↪of length num_samples
    x = np.random.rand(num_samples,)

    #run DFT and FFT on the vector several times and record how long each␣
↪algorithm took each time
    for i in range(num_iterations):
        time_before_DFT = time.time()
        DFT_meshgrid(x)
        DFT_time[i, j] = time.time() - time_before_DFT

        time_before_FFT = time.time()
        FFT(x)
        FFT_time[i, j] = time.time() - time_before_FFT


#we take the median of our runtime samples to get our final runtime estimates␣
 ↪for DFT and FFT
DFT_time_median = np.median(DFT_time, 0)
FFT_time_median = np.median(FFT_time, 0)

#plot the results
import matplotlib.pyplot as plt
plt.figure(figsize=(7, 7))
plt.plot(sample_size, DFT_time_median, label = "DFT")
plt.plot(sample_size, FFT_time_median, label = "FFT")
leg = plt.legend(loc='upper left')
plt.xlabel('Vector Length')
plt.ylabel('Runtime (in Seconds)')
plt.title('Runtime Comparison Between DFT and FFT as Vector Length Increases')
plt.show()
```



As we can see, the runtime of DFT increases at an exponential pace as vector length increases, while the runtime of FFT increases at a much slower, non-exponential pace.

# 3 Data Exploration

## 3.1 Example: Guitar Note

Let's look at an example of how we can use the Fourier transform to analyze an audio file. We'll use an audio recording of a guitarist plucking a note. We begin by importing the audio file as a .wav, then extract the audio data from the wav file. We can then perform the Fast Fourier Transform on the audio data to convert into into the frequency domain. Note that here, we use the real FFT in order to compute only the real components of the FFT.

[6]:
```
# importing packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile as wav    # to read wav files
from scipy.fftpack import fft, rfft, # to do fft
from scipy import stats
```

[38]:
```
# setting file name
file = 'guitar-audio.wav'

#getting the sound data from the file
rate, data = wav.read(file)

# performing the fft
fft_data = rfft(data.T[0])                      # performing real fft on the␣
 ↪left channel only (since file is stereo)
```

Once we've computed the FFT for the audio file, we can convert this data into decibels. Using the sample rate, we can set the corresponding frequencies for each of the magnitudes calculated from the FFT.

[39]:
```
# Compute the magnitude spectrum (in dB) of the FFT
mag_points = 20*np.log10(np.abs(fft_data))

# Generate the frequency values corresponding to the FFT coefficients
n = len(data)
freqs = np.arange(n) * (rate / n)
freqs_points = freqs
```

Here, we look for the maximum magnitude to attempt to identify the note being played by the guitar.

[40]:
```
# getting max of freq (identifies highest magnitude frequency)
magmodei = list(mag_points).index(max(mag_points))  #gets index of max
freqmode = freqs_points[magmodei]                   #gets corresponding␣
 ↪frequency
```

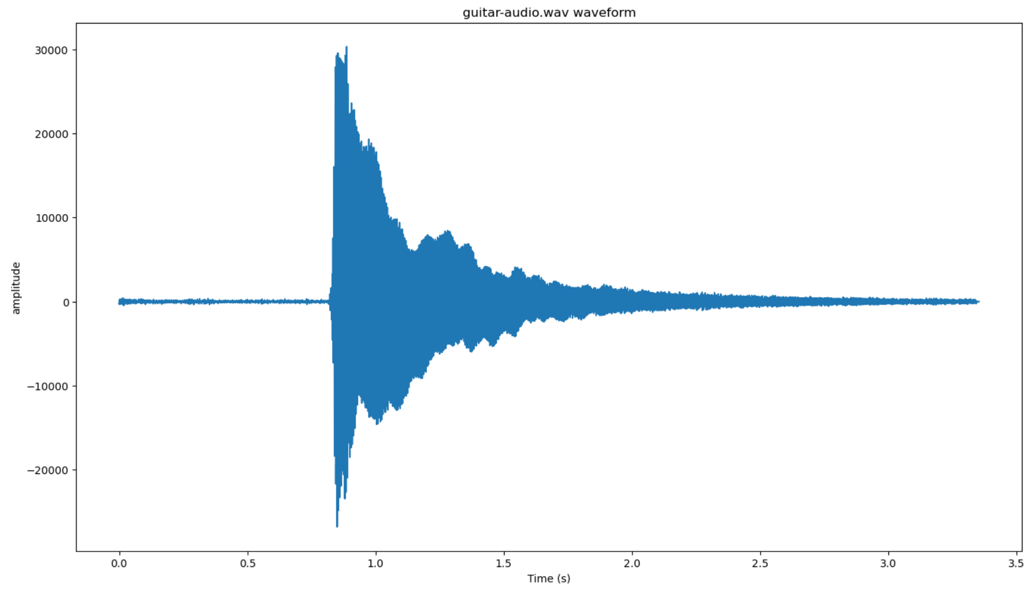
```
freqmode
```

[40]:
```
415.9420975768066
```

It looks like 415.94 is the most prominent frequency in the audio file. This is a G sharp!

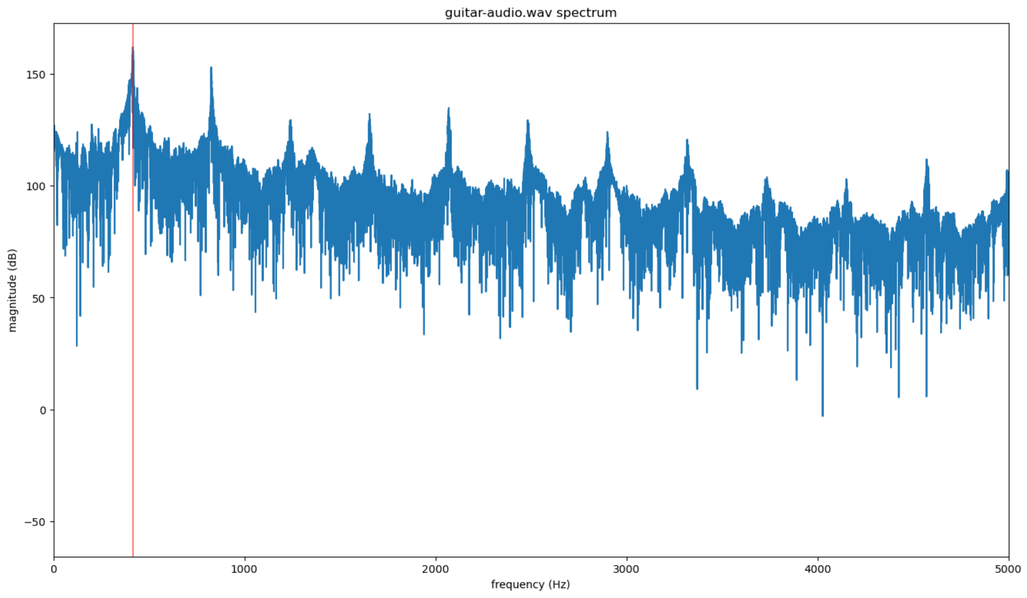
Lets look at the waveform for the audio file.

[41]:
```
#plotting the waveform
length = data.shape[0] / rate
time = np.linspace(0, length, data.shape[0])
plt.figure(figsize=(16, 9))
plt.title(file+' waveform')
plt.plot(time, data[:,0])
plt.xlabel('Time (s)')
plt.ylabel('amplitude')
plt.show()
```



Here's the spectrum of the audio file, with the loudest frequency indicated by the line in red. We notice also a regular pattern of peaks at higher frequencies moving down the line. These are the harmonics of the note being played.

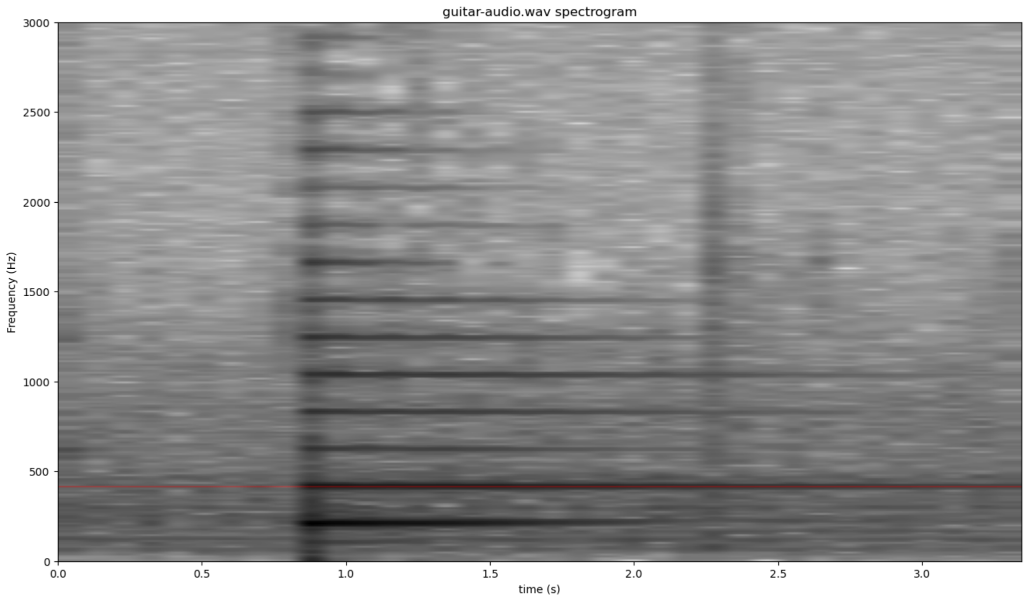
[43]:
```
# plotting the spectrum of the sound file
plt.figure(figsize=(16, 9))
plt.plot(freqs_points, mag_points )
plt.title(file+' spectrum')
plt.xlabel('frequency (Hz)')
plt.ylabel('magnitude (dB)')
plt.axvline(x = freqmode, color='r', alpha = 0.5)
plt.xlim(0,5000)
plt.show()
```



And here is the spectrogram of the audio file with the highest frequency highlighted again. We can see again that the harmonics are present in the spectrogram as in the frequency representation. We can also the how the sound of the guitar decays over time, which was also apparent in the waveform graphic generated directly from the audio file. This is ueful to inspect more aspects of the sound than just the time events or the the frequency spectrum.

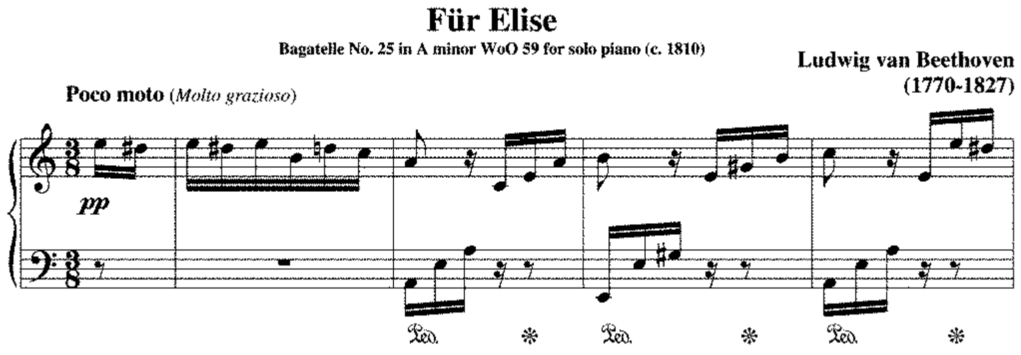
[45]:
```
# plotting the spectrogram of the data
plt.figure(figsize=(16, 9))
plt.specgram(data.T[0], NFFT=4096, Fs=rate, noverlap=0, cmap='gist_yarg')

#titles
plt.title(file+' spectrogram')
plt.xlabel('time (s)')
plt.ylabel('Frequency (Hz)')

#plotting loudest frequency
plt.axhline(y = freqmode, color = 'r', alpha = 0.3)
```

```
#limiting to interesting part
plt.ylim(0, 3000)


plt.show()
```



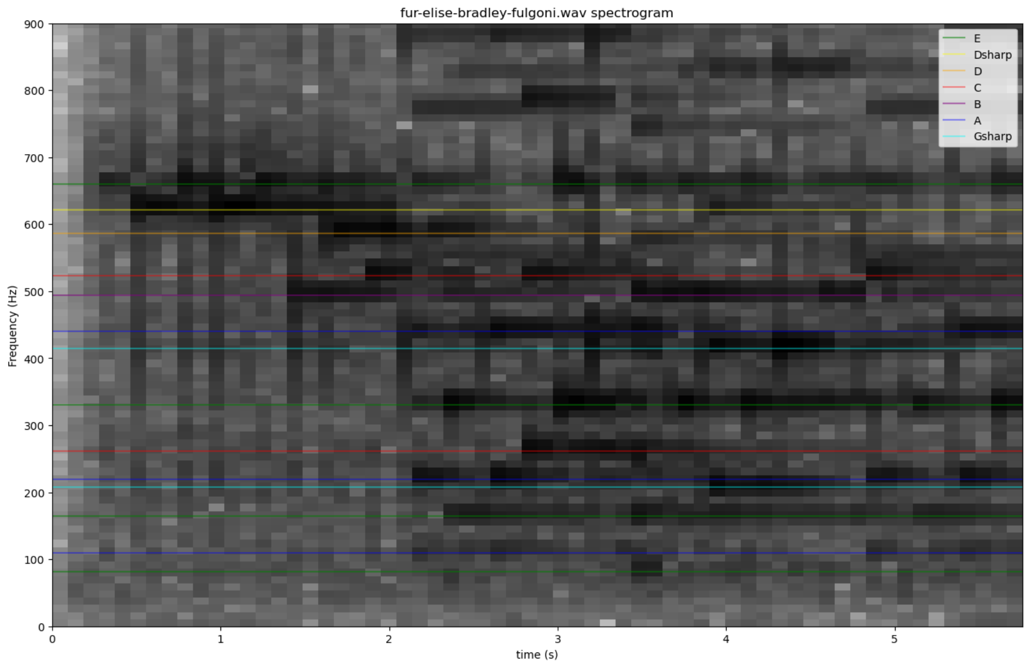
## 3.2 Example: The Gabor Transform on a Famous Piano Bagatelle

To explore the usefulness of the Gabor transform, this time we will examine an audio file of the well-known *Fur Elise* piano bagatelle by Ludvig van Beethoven (1770-1827), performed by Peter Bradley-Fulgoni.


Für Elise
Bagatelle No. 25 in A minor WoO 59 for solo piano (c. 1810)
Ludwig van Beethoven
(1770-1827)
Poco moto (Molto grazioso)
pp

```
[70]: # plotting the spectrogram of the data
plt.figure(figsize=(16, 10))
plt.specgram(b_data.T[0], NFFT=4096, Fs=rate, noverlap=0, cmap='gist_yarg')
```

Using the above code (plus some additional formatting), we are able to extract the following spectrogram of *Fur Elise*:



Those who are able to read music will recognize the note movements are reflected quite well in the spectrogram, providing a convenient time-frequency representation of this piece of music. The image is somewhat affected by the presence of overtones, but with the helpful addition of the note-frequency indication lines, our interpretation is helped.

## References

[BF] Bradley-Fulgoni, Peter: *Fur Elise* https://imslp.org/wiki/Special:ReverseLookup/415774

[BK] Brunton, Steve L. ; Kutz, J. N.: *Data Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. https://databookuw.com/page-2/page-21/

[mit] *Lecture 26: Complex matrices; fast fourier transform*. MIT OpenCourseWare https://ocw.mit.edu/courses/18-06-linear-algebra-spring-2010/resources/lecture-26-complex-matrices-fast-fourier-transform/

[sci] *Fourier transforms (scipy.fft)*. https://docs.scipy.org/doc/scipy/tutorial/fft.html