The Discrete Fourier Transform

Elijan Jakob Mastnak
Student ID: 28181157

November 2020

Contents

Assignment

1. Compute the discrete Fourier transform (DFT) and inverse DFT (IDFT) of a Gauss bell curve and other samples, e.g. linear combinations of sinusoidal functions. Investigate the phenomena of spectral leakage and aliasing. Investigate the effect of zero padding a signal before performing the DFT. Investigate the computation time of various DFT implementations.
2. Analyze the provided samples of a violin solo from a Bach partita, which are recorded at different sample rates. Qualitatively determine the effect of decreasing sample rate by listening, then try to quantify your observations with Fourier analysis.
3. *Optional*: Analyze signals of your choice (e.g. sound recordings from the `Acoustic Resonator` lab experiment, or your own recordings) using the discrete Fourier transform.

---

1 Theory

To jump right to the solution, see Section 2.

1.1 The Continuous and Discrete Fourier Transform

Consider a signal $h(t)$, giving amplitude as a function of time t. The signal's Fourier transform $H(f)$, which encodes the signal's amplitude as a function of frequency, is

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} \, \mathrm{d}t$$

We recover the original signal $h(t)$ from $H(f)$ with the inverse Fourier transform

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi i f t} \, \mathrm{d}t$$

In practice, the signal is a discrete-valued; i.e. defined at N samples $\{h_k\}_0^{N-1}$. Assuming the samples are evenly spaced by the time interval Δt, we define

$$h_k = h(t_k) \qquad \text{where} \qquad t_k = k\Delta t, \qquad k = 0, 1, 2, \ldots N-1$$

The discrete Fourier transform is then defined as

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi k n / N}, \qquad n = -\frac{N}{2}, \ldots, \frac{N}{2} \tag{1}$$

Note the convention of the index n running from $-\frac{N}{2}, \ldots, \frac{N}{2}$ and not from 0 to $N-1$. The continuous Fourier transform and discrete Fourier transform are related by

$$H_n \Delta t \approx H(f)\big|_{f=\frac{n}{N\Delta t}}$$

The corresponding inverse DFT is

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{2\pi i k n / N} \tag{2}$$

1.2 Nyquist Frequency and Other Considerations

The Nyquist (or critical) frequency f_c is defined as half of the sample rate:

$$f_c = \frac{f_s}{2} = \frac{1}{2\Delta t}$$

If the signal's spectrum is confined to $[-f_c, f_c]$ i.e. the signal does not have any frequencies outside of $[-f_c, f_c]$, then f's Fourier transform preserve's all of the original signal's information. If the signal has frequencies outside of the interval $[-f_c, f_c]$, information is lost in the Fourier transform—the portions of the spectrum outside the Nyquist range map into $[-f_c, f_c]$ and mix with the existing frequencies. Recovering the original signal from the muddles spectrum is impossible. This phenomenon is called aliasing.

A Few Miscellaneous Considerations:

- The DFT assumes its input signal is one complete period of a periodic signal. The DFT then outputs the discrete frequencies of this periodic signal. *Spectral leakage* occurs when the inputted signal is not sampled over a full period.
- The spacing Δf between DFT frequency samples depends only on the time over which the signal is sampled; the relationship is simply

  $$\Delta f = \frac{1}{T}$$

  A longer sampling time improves resolution between closely-spaced frequencies.
- For a complex-valued signal, the frequencies measured by the DFT run from zero to f_s. For a real valued signal, the largest measured frequency is the Nyquist frequency $f_c = \frac{f_s}{2}$. For a real signal, the DFT output is symmetric about the "direct-current" frequency $f = 0$, and only $\frac{N}{2}$ points are available to encode the signal's frequency.

2 Algorithm Implementations

2.1 Discrete Fourier Transforms

I experimented with two implementations of the DFT. The first version, shown in the `dft_loop` function below, is a straightforward implementation of (1). The second, shown in `dft_matrix`, uses matrix-vector multiplication to perform the summation. Section 5 discusses the computation time performance of the two methods.

```python
def dft_loop(signal, shift=True):
    """Calculates DFT of signal using a double loop"""
    N = signal.shape[0]  # number of samples
    dft = np.zeros(N, dtype=complex)  # preallocate
    for k in range(N):  # k indexes the DFT
        f_k = 0.0
        for n in range(N):  # n indexes the signal
            f_k += signal[n] * np.exp(-2j * np.pi * k * n / N)
        dft[k] = f_k
```

```
    if shift: return fftshift(dft)
    else: return dft
```

```
def dft_matrix(signal, shift=True):
    """Calculates DFT of signal using a matrix approach"""
    N = signal.shape[0]  # number of samples
    indices_row = np.arange(N)  # indices in row format
    indices_col = indices_row.reshape((N, 1))  # row to column transformation
    dft_mat = np.exp(-2j/*np.pi*indices_col*indices_row / N)  # nxn matrix
    if shift: return fftshift(np.dot(dft_mat, signal))
    else: return np.dot(dft_mat, signal)
```

2.2 Inverse DFT

I implemented two versions of the IDFT using a loop (`dft_loop`) and matrix (`dft_matrix`) approach analogous to the DFT case. The Python code is below

```
def idft_loop(dft, shift=True):
    """Calculates DFT of np array signal with a double loop"""
    N = dft.shape[0]  # number of samples
    idft = np.zeros(N, dtype=complex)  # preallocate
    for n in range(N):  # n indexes the IDFT
        idft_n = 0.0
        for k in range(N):  # k indexes the DFT
            idft_n += (dft[k]/N)*np.exp(2j*np.pi*k*n / N)
        idft[n] = idft_n
    if shift: return ifftshift(idft)
    else: return idft
```

```
def idft_matrix(dft, shift=True):
    """Calculates IDFT of dft using a matrix approach"""
    N = dft.shape[0]
    indices_row = np.arange(N)  # indeces as a row
    indices_col = indices_row.reshape((N, 1))  # row to column transformation
    idft_mat = (1/N)*np.exp(2j*np.pi*indices_col*indices_row/N)  # nxn matrix
    if shift: return ifftshift(np.dot(idft_mat, dft))
    else: return np.dot(idft_mat, dft)
```

3 DFT of a Gauss Bell Curve

In this section, I analyze the DFT of a Gauss curve as an exercise in understanding DFT indexing, the shift theorem, and the practical use of methods like Numpy's `fftshift`.

First, some background: a DFT implementation like Equation 1 returns the DFT's N discrete values as follows: the first $N/2$ samples, with indices from 0 to $\frac{N}{2} - 1$, contain the DFT values corresponding to the frequencies from zero to the Nyquist frequency f_c, and the next $N/2$ samples, with indices from $\frac{N}{2}$ to $N - 1$, contain the DFT values corresponding to the frequencies from $-f_c$ to 0. Numpy's `fftshift`

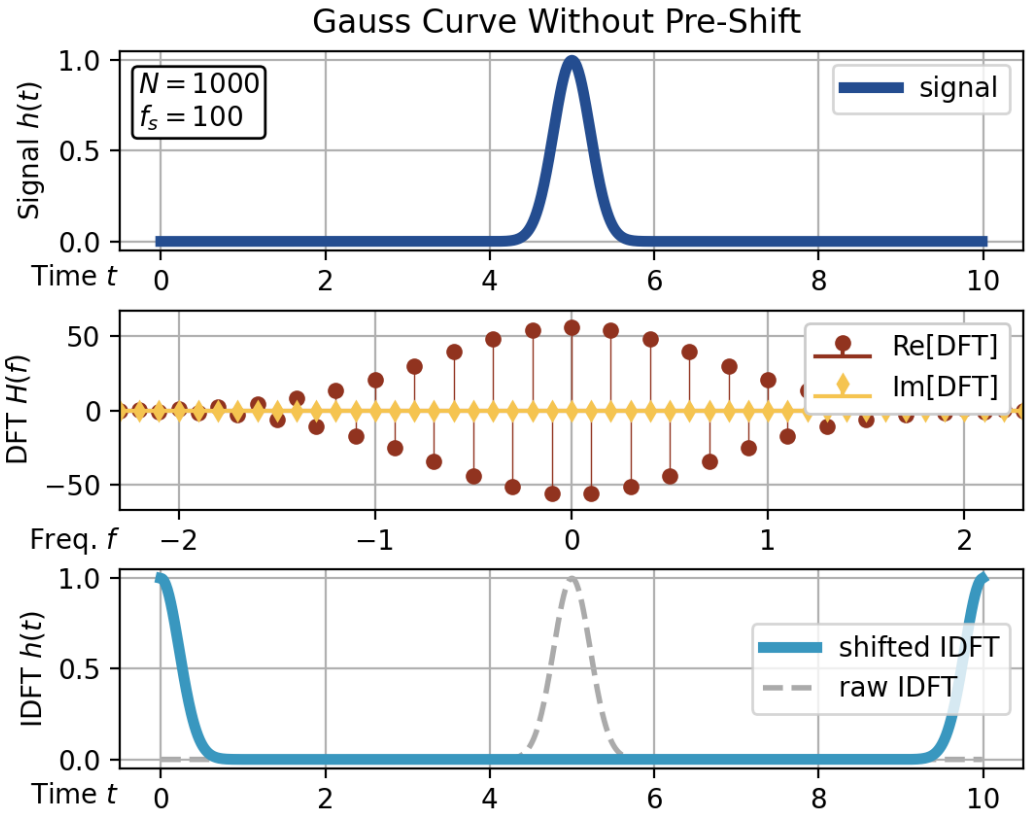

Figure 1: Directly computing the DFT of a Gauss curve. Since the input signal is shifted from the origin to $t = 5$, the DFT contains a corresponding modulation factor $e^{-2\pi ik}$, which produces sinusoidal oscillation about the expected Gauss-curve envelope. The last graph shows the IDFT without `ifftshift` (in grey) and with `ifftshift` (blue).

reorders the DFT's spectrum to run monotonically from $-f_c$ to f_c with the zero-frequency component at the middle index. The reciprocal `ifftshift` performs an analogous reordering of the time samples of the IDFT.

Figures 1 and 2 show how the use of `fftshift` (or lack of it) with a Gauss curve can give unexpected results. In particular, directly computing the DFT of a Gauss curve whose peak is shifted from the origin, in my case to $t = 5$, generates an additional modulation factor due to the shift theorem—see e.g. [2]. This modulation is remedied with a pre-application of `fftshift` to the inputted Gauss curve, as shown in Figure 2. Note that the DFT must also first be re-shifted to reconstruct a "proper" Gauss curve with the IDFT.

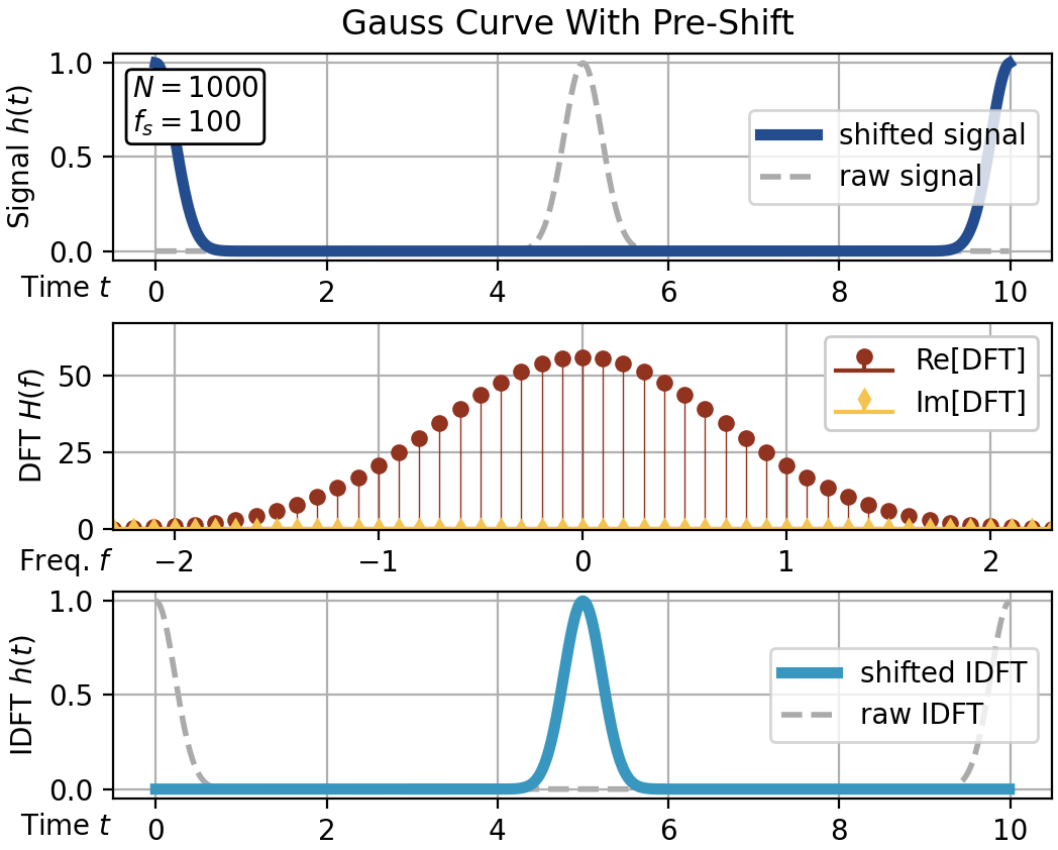

Figure 2: DFT of a Gauss curve, take two. The input signal is made "periodic" by a pre-application of `fftshift`, which results in an unmodulated spectrum. The raw IDFT recovers the "periodic" input signal, while the shifted IDFT produces a raw, un-shifted Gauss curve.

4 Sinusoids

I used the simple combination of sinusoidal signals

$$h(t) = \cos(2\pi f_1 t) + 0.5 \cdot \sin(2\pi f_2 t), \qquad f_1 = 2, f_2 = 5 \tag{3}$$

to investigate aliasing, spectral leakage, and zero padding. Figure 3 shows the sinusoidal signal and its DFT without "anything wrong," i.e. before I start tinkering when investigating aliasing, spectral leakage, etc... Note that the frequencies $f_1 = 2$ and $f_2 = 5$ are clearly visible in the spectrum, and that the frequency of the even harmonic—$f_1 = 2$ from the cosine term—appears in the real portion of the DFT, while the odd sine frequency $f_2 = 5$ appears in the imaginary portion of the DFT. Naturally, $\mathrm{Re}(H(f_k)$ is an even function, while $\mathrm{Im}(H(f_k)$ is odd.

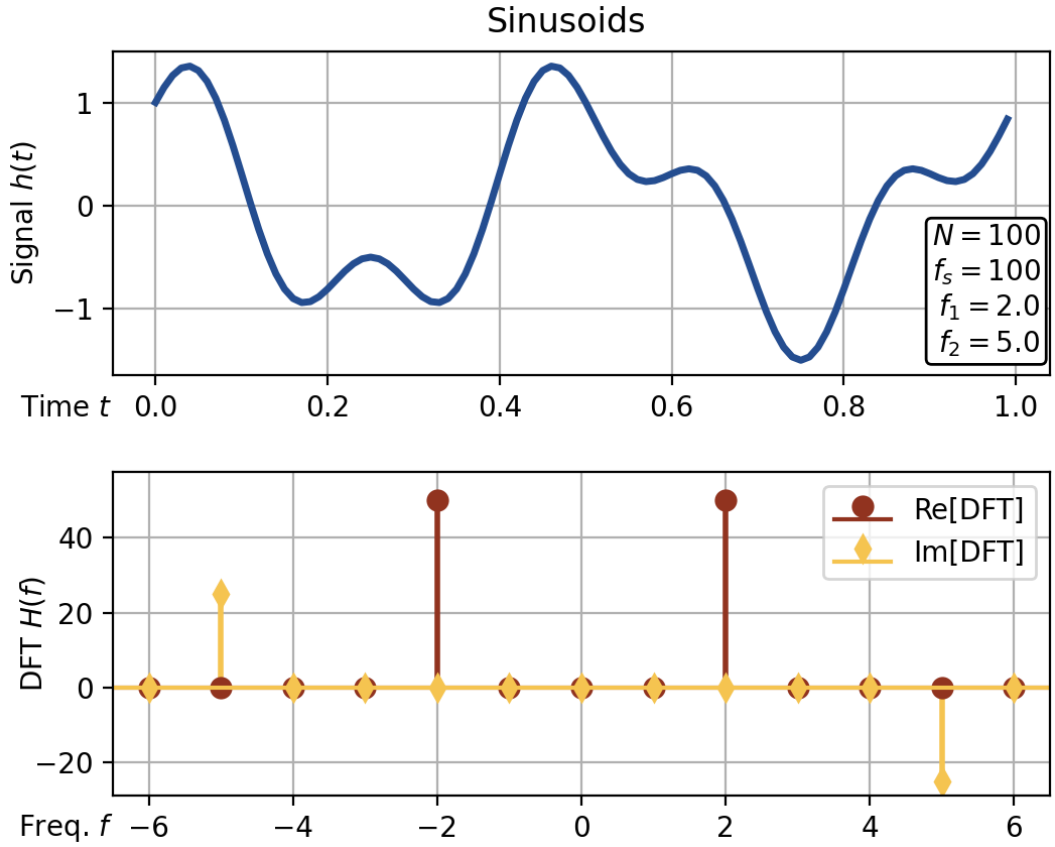

Figure 3: The unblemished waveform and spectrum of the sinusoidal signal (3) used for the remainder of this section. Note the clearly visible frequencies $f_1 = 2$ and $f_2 = 5$.

4.1 Aliasing

To investigate aliasing—the DFT's distortion for input frequencies above the Nyquist limit—I lowered the frequency at which I sampled the signal (3) until the Nyquist frequency f_c fell below the higher frequency $f_2 = 5$. As expected, the DFT fails to

properly detect the f_2 frequency, which falls just outside the Nyquist limit and is instead mapped back into the spectrum's available bandwidth at an incorrect position. Unsurprisingly, the IDFT fails to reconstruct the original signal. The problematic effects of aliasing would be more prominent in a signal containing many frequencies above the Nyquist limit, as seen in the analysis of the Bach violin recording in Section 6.

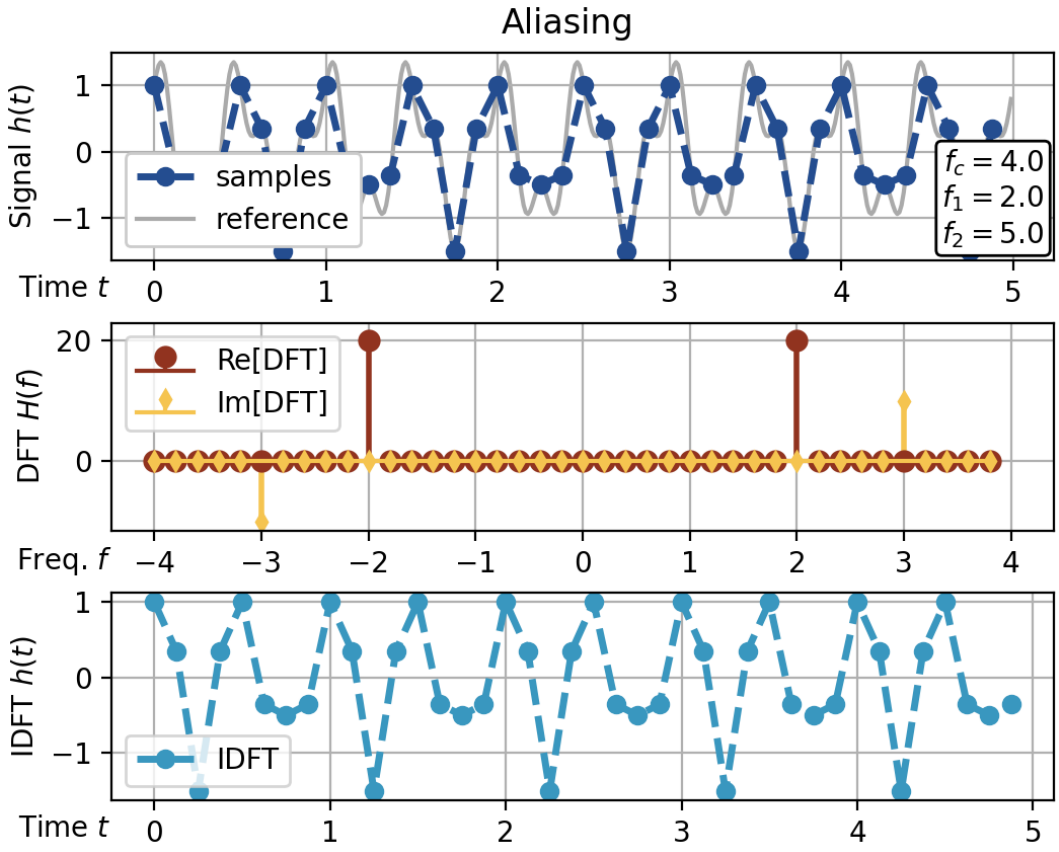

Figure 4: The effects of aliasing when the Nyquist frequency is lowered below $f_2 = 5$—the higher frequency f_2 is mapped into the spectrum at an incorrect position, and the DFT and IDFT are accordingly distorted.

4.2 Spectral Leakage

To investigate spectral leakage, I shifted the time over which I sampled the signal (3) to roughly one-and-a-quarter periods. Resultantly, the DFT's no long sees a periodic input, as visible in the top graph of Figure 5. The lack of periodicity wreaks havoc in the DFT, introducing a multitude of new frequencies and shifting existing ones from their correct positions. Unsurprisingly, the IDFT correspondingly fails to reproduce the original signal, which is shown for reference in the bottom graph.

4.3 Zero Padding

To investigate the use of zero padding, I decreased the time over which I sampled the signal (3) to slightly less than one period, as shown in the top graph of Figure

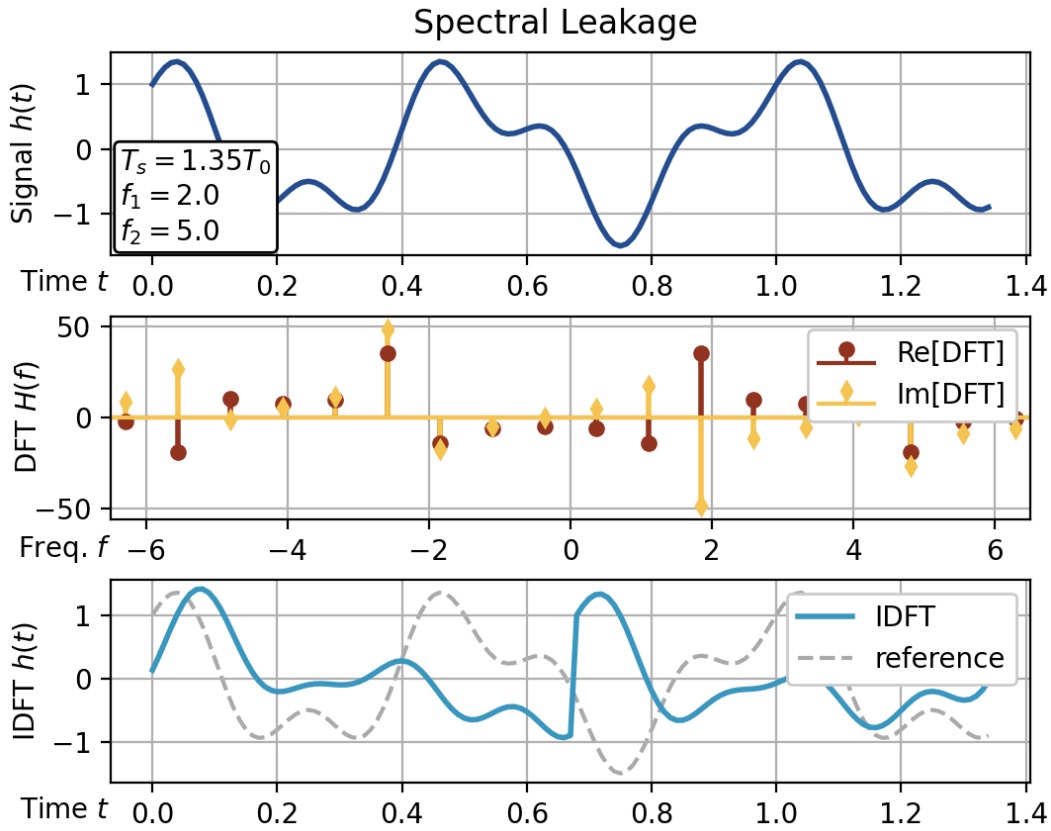

Figure 5: The effects of spectral leakage, which occurs when the input signal is not periodic over the sampling time. Spectral leakage introduces new frequencies, shifts the existing frequencies from their true values, and produces an incorrect inverse transform (the correct version is shown in dashed gray for reference).

6. Accordingly, the DFT suffers from poor spectral resolution and fails to accurately identify the two frequencies $f_1 = 2$ and $f_2 = 5$. This is remedied by zero padding—appending a large number of zeros to the input signal with the hope of artificially increasing resolution with increased sample duration. In my case, I appending zeros amounting to 10 times the number of original samples (not shown in the graph because the signal grows too compressed to be recognizable).

In practice, zero padding works as a sort of interpolation that connects the points of a sparsely populated spectrum. It doesn't add more information to an existing spectrum, but helps reveal, via the extrema positions of the zero padded DFT, information that is already present, just not easily visible. Figure 6 shows the results—indeed, the zero padded DFT helps identify the $f_1 = 2$ and $f_2 = 5$, which hidden in the raw spectrum.

I was pleasantly surprised to see that, although the inverse transform of the non-padded DFT fails to correctly reproduce the original signal, the inverse transform of the zero padded DFT does so almost flawlessly.

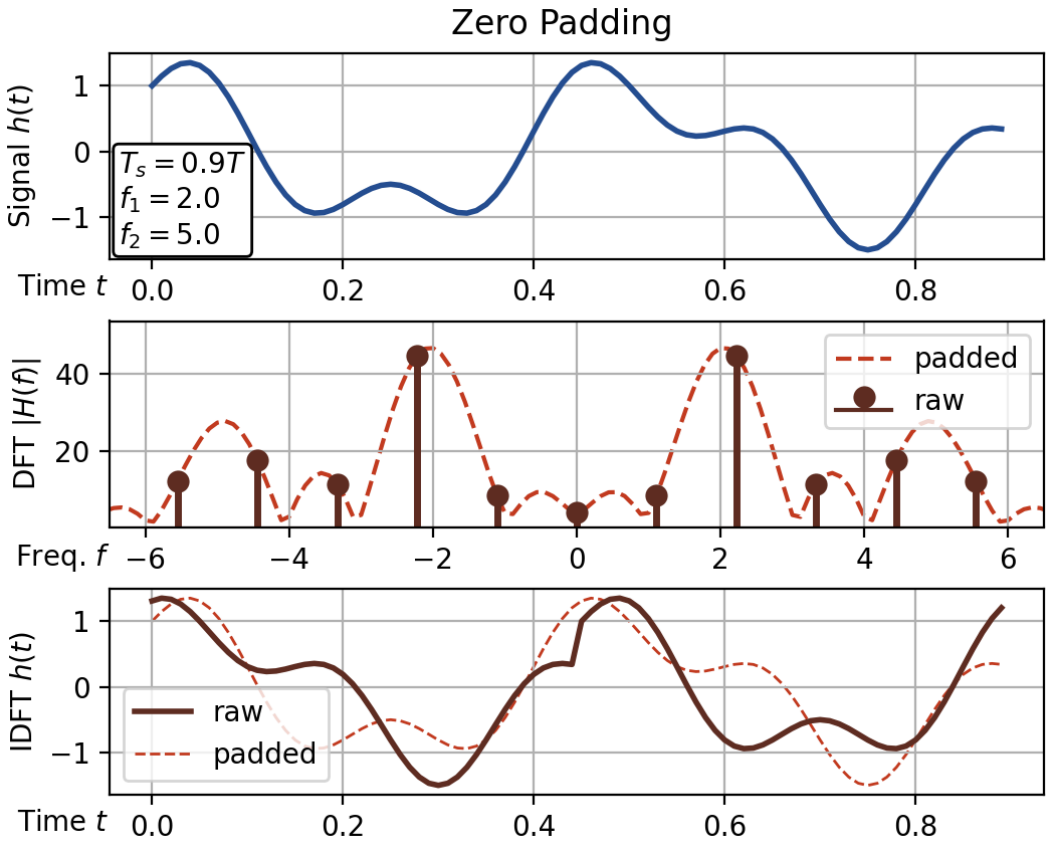

Figure 6: The use of zero padding to identify spectrum frequencies. The raw signal is increased in length by a factor of ten by appending zeros to the end (not shown), and the zero padded DFT's extrema positions reveal the frequencies the $f_1 = 2$ and $f_2 = 5$. Note that zero padding also improves the accuracy of the inverse DFT.

5 Computation Times

I compared the computation times of four DFT implementations: the loop and matrix algorithms in Section 2.1, Numpy's `fft`, which uses the fast Fourier transform, and Numpy's `fftr`, which is a fast Fourier transform optimized for real input. I measured the time each method took to compute the discrete Fourier transform of the sinusoidal signal $h(t) = \sin(2\pi \cdot 10 \cdot t)$ sampled at N points on the interval $[0, 10]$. Figure 7 shows the results.

Unsurprisingly, both FFT methods superlatively outperform my "handmade" implementations. Although the optimized `fftr` slightly outperforms `fft`, the difference is minimal for the small N I tested and is visible only on a logarithmic scale. Of my two basic DFT implementations, the matrix approach performs considerably better, likely by avoiding the explicit use of an expensive nested loop. In any case, both handmade methods show prohibitive asymptotic growth for consideration in any serious practical application. As a side note, since the matrix approach loads an entire $N \times N$ matrix into memory at once, it likely becomes prohibitively memory intensive for very large N. That said, I didn't have any problems on my 8 GB RAM computer for the moderate N I tested.

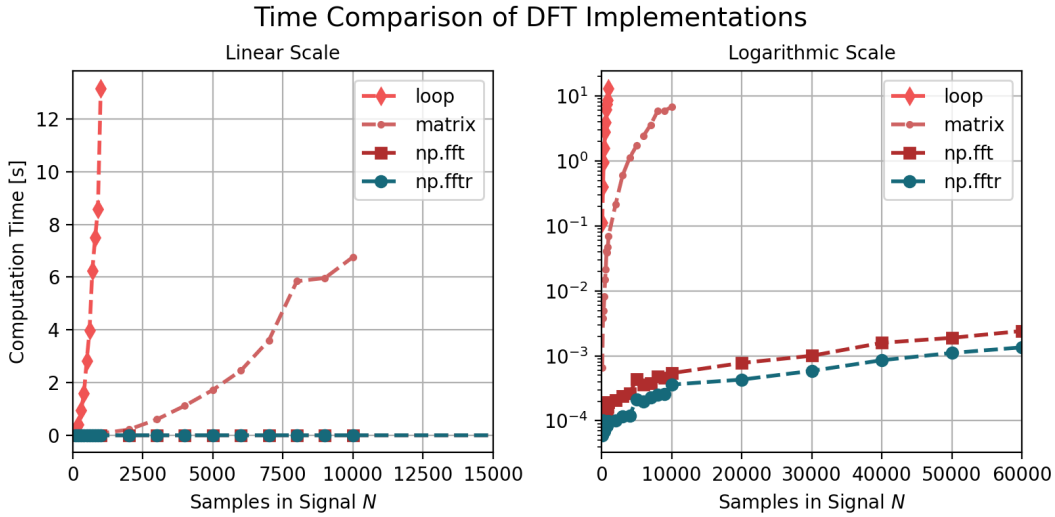

Figure 7: Time required for each of the four DFT implementations to compute the discrete Fourier transform of an N-sample signal. Note the vast superiority of the FFT implementations.

6 Analyzing a Bach Partita

First, some background: we are given a short excerpt from a Bach violin partita, recorded at the sample rates 44100, 11025, 5512, 2756, 1378, and 882 Hz. As the sample rate decreases, I notice two main trends:

1. The recording sounds increasingly distorted
2. Higher frequencies disappear from the recording, and accordingly, the violin's distinctive timbre vanishes with the higher harmonics

My first step was to try to find the musical notes played in the recording, which would tell me the corresponding frequencies and thus provide a quantitative reference for my analysis. Since I don't have perfect pitch, I turned to YouTube for improvised research. As far as I can tell, the sample recording comes from the opening seconds of the fourth movement, *Double (Presto)*, of Bach's Partita No. 1 in B minor, BWV 1002. Figure 8 shows the corresponding sheet music.

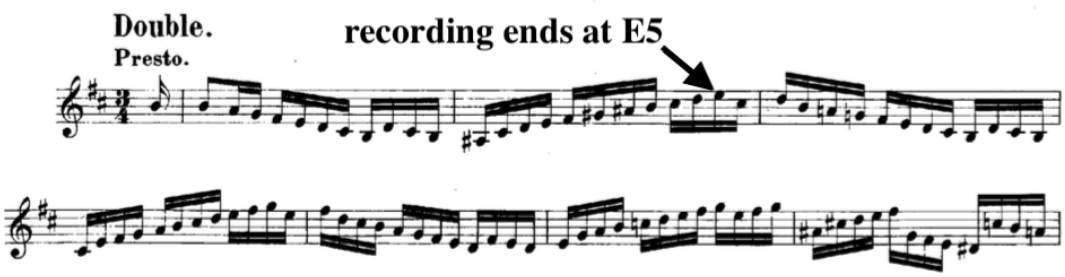

Figure 8: The partita's first few measures. Our sample recording ends at the `E5` note in the third measure.

In order, the notes in the excerpt are

```
B4 B4 A4 G4 F#4 E4 D4 C#4 B3 D4 C#4 B3
A#3 C#4 D4 E4 D4 G#4 A#4 B4 C#5 D5 E5
```

The notes span the frequency range from a low 233.1 Hz at `A#3` to a high 659.3 Hz at `E5`. Figure 9 shows the spectrum of the opening note, a `B4` with frequency 493.9 Hz, for each sample rate. Clearly, decreasing the sample rate, and thus the Nyquist frequency, removes higher harmonics and leads to aliasing, progressively degrading the recording's fidelity. At 5512 Hz and below, the opening note is no longer recognizable as a `B4`, discussed more below. Figure 10 shows the analogous spectra of the entire recording and displays similar problems with aliasing and higher harmonics.
An analysis of the change in sound quality with sample rate follows, with attempts at an explanation using the lessons of Fourier analysis.

- At 44 100 Hz the recording sounds excellent, at least to my non-expert ears. The instrument is clearly a violin. In the spectra of both the first note and the entire recording, higher harmonics are clearly visible.

- At 11 025 Hz, the recording sounds a bit "hollow"—it appears that the high harmonics (which are quite weak) are cut off and aliased into the available spectral bandwidth below the Nyquist frequency $f_c = 5512$ Hz. Nonetheless, the effect is minimal and, if I am honest, I am not sure I would detect something is wrong with the recording without *a priori* knowledge. This stance is validated by comparing the spectra of the 44 100 Hz and 11 025 Hz samples—they are nearly identical, at least up to the 11 025 Hz recording's Nyquist frequency.

- By 5512 Hz, the recording is audibly distorted, and the distinctive violin sound is fading. A look at the spectra suggest the lack of harmonics beyond $f_c \approx 2750$ Hz and a general shift to lower frequencies is to blame. In particular, the spectrum of the first note at 5512 Hz shows periodic frequencies, but these are no longer integer multiples of the fundamental frequency, and fail to match the actual `B4` note or its higher harmonics from the original sheet music.

- At 2756 Hz, I can hardly recognize the instrument is a violin, since the instrument's timbre—a fancy word for the instrument's distinctive sound or musical "fingerprint"—is progressively erased along with the higher harmonics, as confirmed in the spectra as the Nyquist frequency decreases.

- By 1378 Hz, I can just barely make out the melody. This agrees with theory—although the sample rate is very low, even the highest fundamental frequency in the melody, the `E5`'s 569 Hz, still falls withing the 689 Hz Nyquist range, so it makes sense that the melody is still (barely) distinguishable.

- At 882 Hz sampling, I can no longer detect the melody, since even the melody's fundamental frequencies begin to exceed the 441 Hz Nyquist limit. The recording is devoid of high frequencies and is strongly distorted, corresponding to a spectrum muddled by aliasing and stripped of frequencies above 440 Hz.

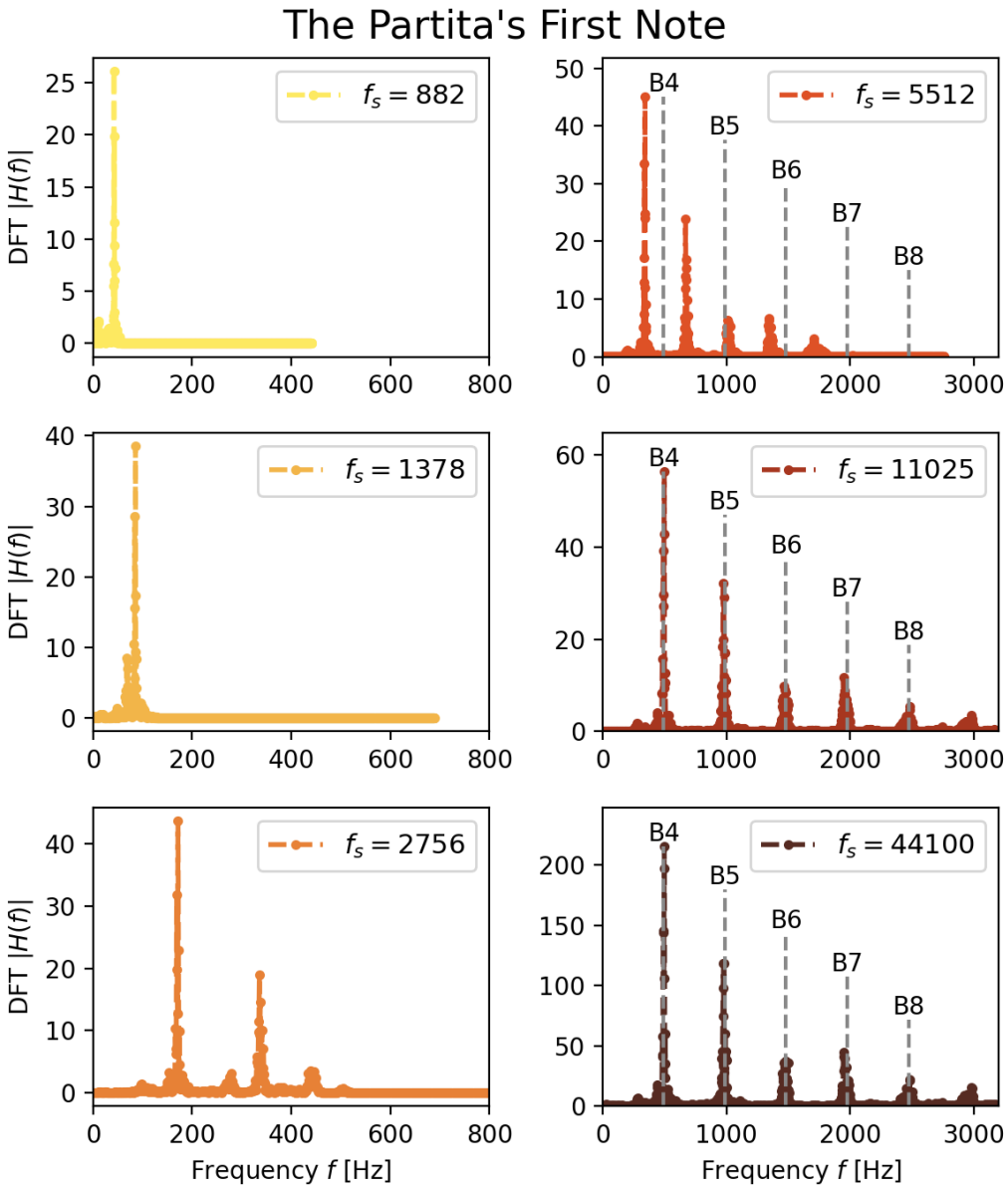

Figure 9: Spectrum of the partita's first note, a 493 Hz `B4`. For the highest three sample rates, the theoretically expected positions of the note and its higher harmonics are shown for reference. Note the agreement at 44100 and 11 025 Hz and disagreement at (and below; not shown) 5512 Hz, meaning the note is no longer recognizable as a `B`. In general, the spectrum grows weaker and more aliased with decreasing sample rate.

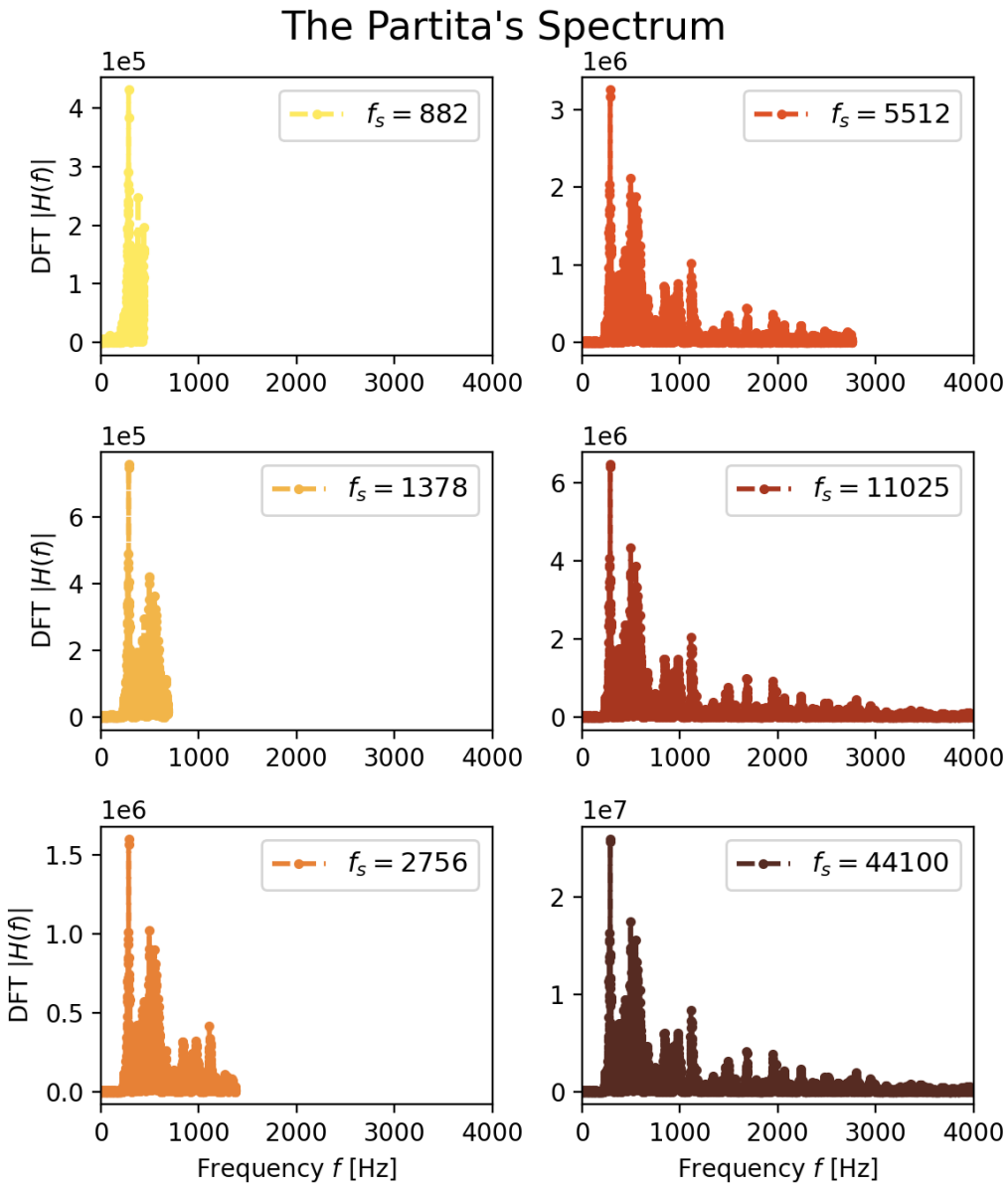

Figure 10: The entire recording's spectrum at various sample rates. Note that the 44100 and 11 025 Hz spectra are nearly identical (at least below the Nyquist limit, beyond the graph's axis limits). Besides the obvious truncation of harmonics and aliasing at lower sample rates, note that spectrum's amplitude grows weaker as well.

7 Extra: Decomposing Guitar Chords

7.1 Extracting Notes in the Chords

For this section, I recorded a few simple chords on a nylon-string acoustic guitar in `wav` format at 44 100 Hz, and then attempted to extract the chords' constituent notes using the DFT. Since I knew the notes in each chord *a priori* (e.g. an A major chord played across all six string contains the notes `A2`, `E3`, `A3`, `C#4`, `E4` and `A4`), I could assign each note a corresponding frequency using the 12-tone equal temperament tuning system.
To avoid bogging down this report with music theory, I'll refer interested readers to [4]. For our purposes, it is enough to know that we can bijectively map each note to a corresponding frequency (see e.g. [3] for tabulated values). I could then theoretically predict which frequencies "should" be present in a given chord. My goal was to test if the theoretically expected frequencies agreed with the experimental data from a DFT analysis of the `wav` files.
Happily, the experiment worked! Figure 11 shows the results. The dashed gray lines show the theoretically expected positions of the chords' notes and their first few harmonics, in good agreement with the spectral peaks.

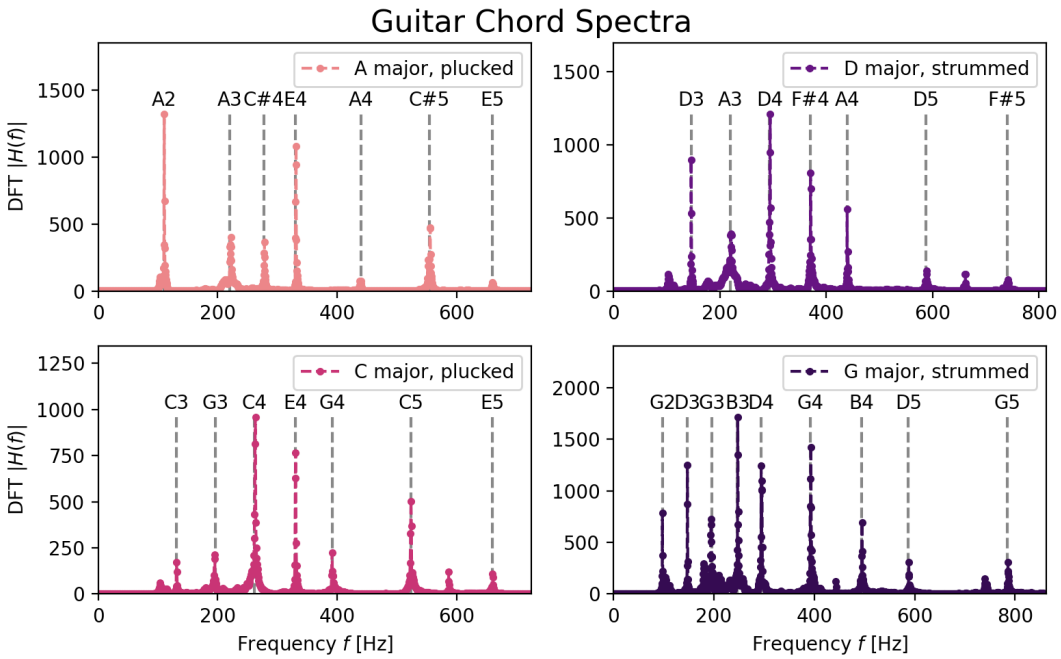

Figure 11: Comparing the theoretically expected notes (dashed grey lines and labels) in nylon-string acoustic guitar chords to the chords' spectra. Note the good agreement in all four cases. The frequency positions are found using the 12-tone equal temperament tuning system described in [4].

7.2 Detecting Slightly Flat Tuning

As a last experiment, I intentionally tuned my guitar 8 Hz flat relative to the standard `A4` frequency of 440 Hz, ending up with a reference tone of `A4` $\mapsto$ 432 Hz. I recorded

(again in `wav` format at 44 100 Hz) a few simple chords in 432 Hz tuning and compared them to a 440 Hz reference version. I was curious to see if the DFT would detect the slight difference in the tuning. Figure 12 shows the results for a strummed A major chord, with the 440 Hz and 432 Hz versions plotted on the same axis, along with the theoretically expected constituent note positions for each tuning system. The difference is subtle, but present—the 432 Hz chord is shifted to slightly lower frequencies.

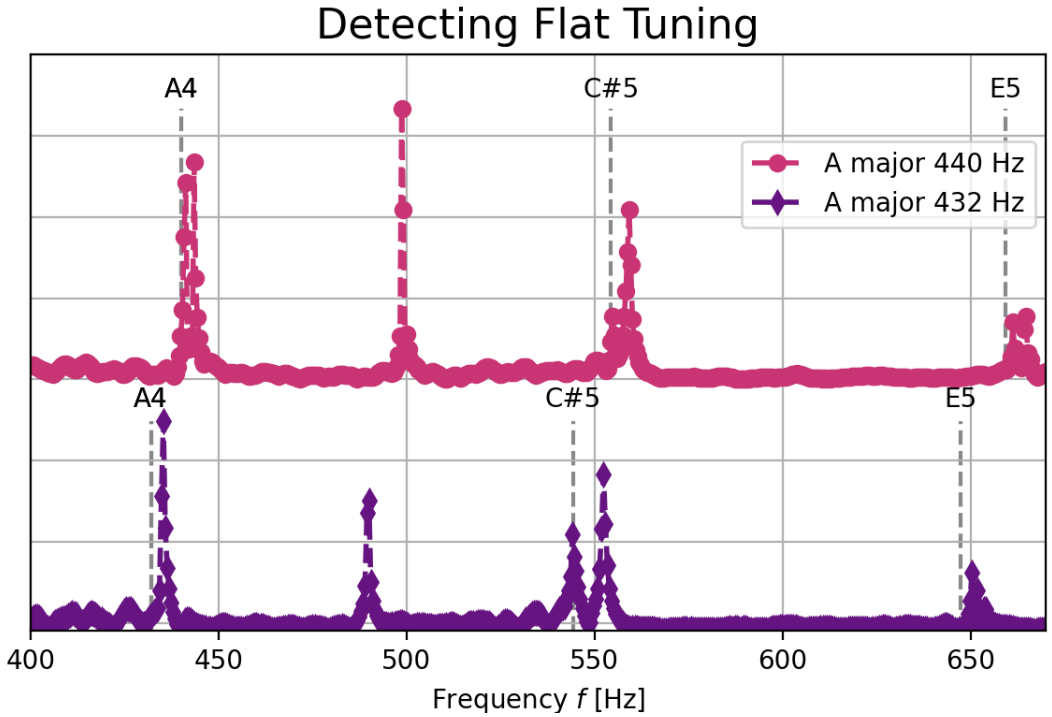

Figure 12: Comparing an A major chord in 440 Hz and 432 Hz tuning. As expected, the 432 Hz version's notes and spectral peaks are shifted to slightly lower frequencies.

References

[1] "Spectral Leakage and Zero Padding of the Discrete Fourier Transform". https://dspillustrations.com/pages/posts/misc/spectral-leakage-zeropadding-and-frequency-resolution.html

[2] Wikipedia contributors. "Discrete Fourier transform." *Wikipedia, The Free Encyclopedia.* 1 November 2020. https://en.wikipedia.org/wiki/Discrete_Fourier_transform#Shift_theorem

[3] B. H. Suits. "Frequencies for equal-tempered scale, A4 = 440 Hz" Physics Department, Michigan Technological University. https://pages.mtu.edu/~suits/notefreqs.html

[4] Wikipedia contributors. "12 equal temperament." *Wikipedia, The Free Encyclopedia.* 27 October 2020. https://en.wikipedia.org/wiki/12_equal_temperament#Mathematical_properties.