FAST FOURIER TRANSFORM (COOLEY-TOOKEY)

Andrew Balch **Evan Conway** **Elliott Druga** **Zoe Hamilton**

April 28, 2025

1 Introduction / Problem Statement

The Fourier transform (FT), $\mathcal{F}$ is an integral transformation that takes as input a complex valued function f and produces another complex valued function typically denoted $\hat{f}$ defined as follows.

$$\hat{f}(\xi) = (\mathcal{F}f)(\xi) = \int_{-\infty}^{\infty} f(x)e^{-2\pi i\xi x}dx, \quad \forall \xi \in \mathbb{C} \tag{1}$$

In practice, we often use real-valued functions as inputs f. The Fourier transform represents the degree to which certain frequencies are present in the original function. If both f and its Fourier transform $\hat{f}$ are absolutely integrable, we can apply the inverse Fourier transform to recover the original function.

$$f(x) = (\mathcal{F}^{-1}\hat{f})(x) = \int_{-\infty}^{\infty} \hat{f}(\xi)e^{2\pi i\xi x}d\xi, \quad \forall x \in \mathbb{R} \tag{2}$$

In computing applications, we frequently represent continuous functions by sampling values over equally spaced intervals on some range. The discrete Fourier transform (DFT) transforms a sequence of N real or complex numbers $\{\mathbf{x_n}\} = x_0, x_1, \ldots, x_{N-1}$ for some $N \in \mathbb{N}$ into a sequence of complex numbers $\{\mathbf{X_k}\} = X_0, X_1, \ldots, X_{N-1}$ of the same length and is defined as follows.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2i\pi \frac{k}{N} n} \tag{3}$$

Here we can see the transformation is conducted on discrete time steps by simplifying the equation to a summation rather than an integral. Unlike the continuous case, the inverse transform always exists and is given by

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{2i\pi \frac{k}{N} n} \tag{4}$$

The DFT has wide a wide variety of applications in several domains, especially signal and image processing. Unfortunately, the naive implementation of the DFT has time complexity $\Theta(N^2)$. Since evaluation of the exponential term is computationally expensive, this makes the naive algorithm impractical for large amounts of data. Fast Fourier transforms (FFTs) are a class of algorithms for calculating the DFT which takes advantage of the properties of the complex exponential terms, which are called the roots of unity to reduce the time complexity to $\Theta(N \log N)$.

Given the variety of practical applications the Fourier transform has, it may be unsurprising to hear that the FFT algorithm itself has quite a varied history. Originally developed in the early 19th century by Carl Gauss, it was independently re-discovered a few more times in the subsequent centuries until James Cooley and John Tukey published the form we know today in 1965 (the Cooley-Tukey FFT algorithm). Cooley and Tukey themselves developed the algorithm with disparate practical applications in mind. Tukey's motivation was a directive from the Kennedy administration to utilize a network of seismic sensors for the detection of Soviet nuclear tests during the Cold War, which a DFT was not fast enough. When Cooley was recruited to the project, he was told that the algorithm was for calculating the spin orientation properties of a Helium-3 crystal.

2 Overall Intuitive Approach / Solution

The Cooley-Tookey FFT is a divide and conquer algorithm that splits the input sequence into 2 or more subsequences recursively computing the DFT of each subsequence and taking a linear combination of the solutions by applying a twiddle factor W_N^k. We will only examine the case where the input is a sequence whose length has a power of 2 so we can divide it into 2 subproblems on the even and odd elements at each recursive step. We must also determine how sub-results can be merged to create a valid DFT. The overall merge formulas will be as follows.

$$X_k = E_k + W_N^k \cdot O_k \qquad X_{k+N/2} = E_k - W_N^k \cdot O_k \tag{5}$$

The whole process is illustrated for a sequence of length 8 in the butterfly diagram below where W_N^k denotes a twiddle factor, a denotes the input data, and A denotes the transformed data.

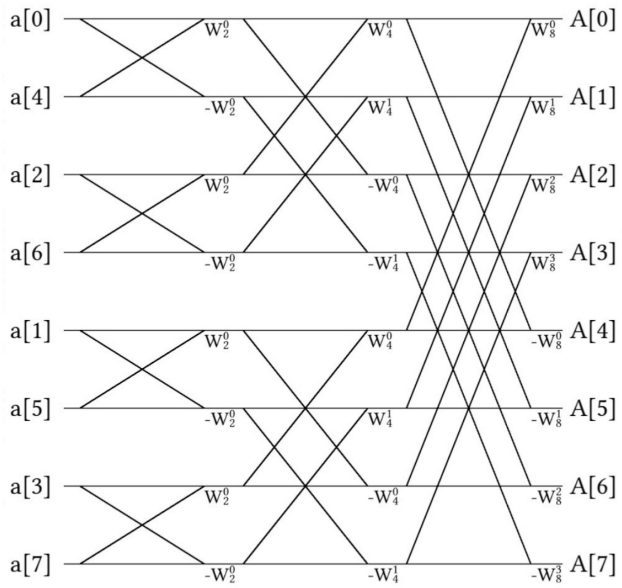

Figure 1: Butterfly diagram depicting how twiddle factors are (re)used to recursively calculate a DFT without summations over the whole sub-solution.

Now we must derive the twiddle factors. Splitting the summation by even and odd indices we get.

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{\frac{-2\pi i}{N}(2n)k} + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{\frac{-2\pi i}{N}(2n+1)k} \tag{6}$$

Additionally we can factor out an exponential term on the right summation and we get

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{\frac{-2\pi i}{N/2}nk} + e^{\frac{-2\pi i}{N}k} \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{\frac{-2\pi i}{N/2}nk} = E_k + e^{\frac{-2\pi i}{N}k} O_k \tag{7}$$

A crucial observation here is that both E_k and O_k are themselves Discrete Fourier Transforms on sequences of size $N/2$, however, solving them recursively will only give us solutions for $k = 0, 1, \ldots, \frac{N}{2} - 1$ We can derive a formula for the remaining $\frac{N}{2}$ terms $\frac{N}{2}, \frac{N}{2} + 1, \ldots, N - 1$ in terms of the recursive solutions as follows.

$$X_{k+\frac{N}{2}} = \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{\frac{-2\pi i}{N/2}n\left(k+\frac{N}{2}\right)} + e^{\frac{-2\pi i}{N}\left(k+\frac{N}{2}\right)} \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{\frac{-2\pi i}{N/2}n\left(k+\frac{N}{2}\right)} \tag{8}$$

$$= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{\frac{-2\pi i}{N/2}nk} e^{-2\pi in} + e^{\frac{-2\pi i}{N}k} e^{-\pi i} \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{\frac{-2\pi i}{N/2}nk} e^{-2\pi in} \tag{9}$$

$$= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{\frac{-2\pi i}{N/2}nk} - e^{\frac{-2\pi i}{N}k} \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{\frac{-2\pi i}{N/2}nk} \tag{10}$$

$$= E_k - e^{\frac{-2\pi i}{N}k} O_k \tag{11}$$

Now we have all the tools we need to provide an algorithm for computing the FFT.

3 Implementation

Note that while implementations exist to derive a DFT on sequences not of size 2^k, we do not discuss this case here.

Overall, the FFT recursively divides its input sequence into subsequences on the even and odd indices until our input sequence has length 1. Because the DFT of a single complex number is itself, we can simply return the original sequence. After computing the recursive solutions, the FFT algorithm merges the results together according to the formula (5) with twiddle factor $W_N^k = \exp(\frac{-2\pi i}{N}k)$ Analyzing the runtime of our algorithm, we see that we recursively compute

Algorithm 1 Fast Fourier Transfrom (Cooley-Tookey)

Require: $N = 2^k$ for some $k \in \mathbb{N}$
procedure FFT($(x_0, x_1, \ldots, x_{N-1})$)
  if N = 1 **then**
    return (x_0)
  end if
  $(E_0, E_1, \ldots E_{N/2-1}) \leftarrow$ FFT($(x_0, x_2, \ldots, x_{N-2})$)
  $(O_0, O_1, \ldots O_{N/2-1}) \leftarrow$ FFT($(x_1, x_3, \ldots, x_{N-1})$)
  for $k \leftarrow 0, N/2 - 1$ **do**
    $X_k \leftarrow E_k + e^{\frac{-2\pi i}{N}k} \cdot O_k$
    $X_{N/2+k} \leftarrow E_k - e^{\frac{-2\pi i}{N}k} \cdot O_k$
  end for
  return $(X_0, X_1, \ldots, X_{N-1})$
end procedure

two subproblems of size $N/2$ and do a linear amount work to merge the results giving the familiar

recurrence relation

$$T(N) = 2T\left(\frac{N}{2}\right) + O(N) \tag{12}$$

which gives an overall runtime of $\Theta(N \log N)$.

4 Summary of Programming Challenge

The programming challenge involves taking a waveform, represented in as a series of discrete samples that the fast Fourier transform can handle, and determining which musical chord that waveform represents. In solving this problem, students are expected to learn how to implement the fast Fourier transform, as well as how to interpret its output in a real-world context. The transform outputs a sequence of complex numbers, so some work is needed to extract useful information from it, which is the main learning objective of this assignment. Determining the chord that is represented by the waveform does not require very complicated code, but it does require some basic knowledge of music theory, which provides students a fun opportunity to become familiar with what might be a new subject field, and how it intersects with computer science. This challenge also helps to motivate why the fast Fourier transform is useful - it has concrete applications, and larger test cases demonstrate the utility of the improved runtime compared to the standard DFT.

5 Key Ideas for Solving Programming Challenge

The programming challenge can roughly be divided into three parts: calculating the Fourier transform, extracting peak frequencies from the transform, and interpreting the frequencies as a chord. The first challenge is to actually implement the Fourier transform and apply it to the input data. This is a fairly straightforward application of the FFT algorithm - the twist comes from the later steps.

Once results from the algorithm are calculated, you must extract accurate frequency peaks. As the FFT algorithm returns a series of complex numbers, you must first extract the amplitude of each frequency by applying taking the absolute value of the results. Then, some form of peak-finding algorithm must be used to identify the constituent frequencies. Students are permitted to use existing library code for this step, but some experimentation is still required from students to properly tune their peak-finding approach to work effectively with a wide range of inputs. In particular, it is important to avoid detecting false peaks, as this will cause subsequent steps to not work properly. Approaches for avoiding this include smoothing the FFT output, imposing a minimum height for peaks, or imposing a minimum distance between peaks.

The final step is to take the extracted peak frequencies and interpret them as a chord. There are several necessary steps in doing this, and they all depend somewhat on knowledge of music theory. However, we believe that the required knowledge is fairly simple, and an undergraduate taking this course would probably be able to quickly grasp enough to tackle this problem. This twist adds just enough challenge and real world context to make the problem more interesting.

The first task for this step is to assign note names to frequencies. The key insight here is that the ratio of an octave is 2:1. This means that for any given frequency, you can double or half the frequency and the note name stays the same. Thus, only 12 unique frequency-note pairs need to be stored. For a given frequency, simply double of half until the value is in the range of stored pairs, and assign it the closest value.

Next, you must determine the root and quality of the chord (e.g. A minor). This is perhaps the most challenging step for those with limited musical knowledge. To do this, we want to take our frequencies and arrange them by intervals of a third - it is guaranteed that this can be done for any chord given. The key insight is that this can be done by repeatedly moving the lowest frequency note up by an octave (double its frequency). If this is done repeatedly, it is guaranteed that the notes will eventually all be separated by thirds. Then, the chord quality can be determined by looking at whether the specific intervals are minor thirds (1.189207:1) or major thirds (1.259921:1). More on this is in the assignment writeup.

Finally, once the chord has been rearranged into root position to determine the root and quality, it is simple to determine the inversion. The inversion is determined simply by which note of the chord in root position is in the bass of the original chord. If the root is in the bass, it is in root position; if the third is in the bass, it is in 1st inversion; and if the fifth is in the bass, it is in 2nd inversion. This can be checked in constant time using the notes in root position (returned from the last step) and the bass, simply the lowest frequency note.

The runtime of the model solution is $O(n \log(n))$, bounded by the fast Fourier transform, where n is the number of samples in the input. Extracting peaks from the transformed waveform can be done in $O(n)$, and all following operations are $O(1)$ relative to the size of the original input, given that they always operate on the three extracted frequencies. Thus, meeting runtime requirements is a matter of implementing FFT correctly.

6 Conclusion

The Fast Fourier Transform makes it feasible to transform discretely-sampled data from the time domain to the frequency domain, and vice-versa. To do this in $\Theta(n \log(n))$ time, the Cooley-Tukey FFT algorithm recursively divides the sequence of data until it contains only a single point, then merges the subsolutions in linear time by leveraging some algorithmic manipulation and the periodicity of the complex exponential. Our programming challenge asks potential students to implement the FFT for a real world problem: determining the musical chords present in a set of input data. The twist required by the challenge involves going beyond the FFT algorithm, which will only provide frequency data, to process the frequencies into chords without adding to the overall time complexity. This challenge focuses on teaching students to understand the practical applications of the FFT, and how it can be extended to fit these needs.