# FAST (AND FUN!) COSINE TRANSFORMS

MATT BEVERIDGE*

**Abstract.** In this paper we explore improvements in computing the discrete cosine transform (DCT). We explain the relationship between the DCT and discrete Fourier transforms (DFTs), and how advancements in fast Fourier transforms (FFTs) can be applied to implement fast cosine transforms (FCTs). We explore several implementations of DCT and FCT algorithms, and evaluate their performance. Lastly, we look at a case study of Lee's algorithm and conjecture its usefulness.

**Key words.** Fourier Transforms, Numerical Methods, Compression, Signal Processing

**1. Introduction.** Since its inception [1, 10], the discrete cosine transform (DCT) has continued to be one of the most widely used transforms in signal processing[9]. In particular the DCT is extensively used within digital compression algorithms for images, videos, audio and so on [13]. It is derived from a similar transform called the fast Fourier transform (FFT) [15]; the only difference being the FFT outputs complex values and the DCT outputs the real part of the same values [18]. More specifically the DCT is a derivation of the discrete Fourier transform (DFT) which maps the "time domain" to the "frequency domain".

Formally, the DCT for an $N$ element array in one-dimension is computed as [8]:

$$y_n = c_n \sum_{k=0}^{N-1} \cos\left[\frac{n\pi}{2N}(2k+1)\right] \tag{1.1}$$

Where $c_0 = 1/\sqrt{N}$ and $c_n = \sqrt{2/N}$ for $n = 1, 2, \ldots, N-1$. For a two-dimensional matrix we simply apply (1.1) first to the rows and then to the columns, or vice versa [2]:

$$y_{n_1,n_2} = c_{n_1} \sum_{k_1=0}^{N_1-1} \left[ c_{n_2} \sum_{k_2=0}^{N_2-1} \cos\left(\frac{n_2\pi}{2N_2}(2k_2+1)\right)\right] \cos\left(\frac{n_1\pi}{2N_1}(2k_1+1)\right) \tag{1.2}$$

$$= c_{n_1} c_{n_2} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} \cos\left(\frac{n_2\pi}{2N_2}(2k_2+1)\right) \cos\left(\frac{n_1\pi}{2N_1}(2k_1+1)\right) \tag{1.3}$$

It follows that the DCT can be computed in any dimension by applying the one-dimensional definition repeatedly. Naïvely following the above definitions, the discrete cosine transform can be computed for the entire input of size $N$ in $O(N^2)$ operations regardless of the dimensionality of the input. As $N$ gets large this slow run time poses a significant computational burden to the point where operation on rather mundane matrices could take many minutes, or longer. However, using special properties of the DCT we are able to improve this to $O(N \log N)$.

In this paper we examine fast cosine transforms (FCTs) and how computation time can be improved from $O(N^2)$ using the naïve DCT to $O(N \log N)$ using a FCT. In section 2 we discuss the evolution of the discrete cosine transform, forms of the (fast) cosine transform are in section 3, performance comparison is in section 4, modern advancements are in section 5, and the conclusions follow in section 6.

*Department of EECS & Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA. (mattbev@mit.com).

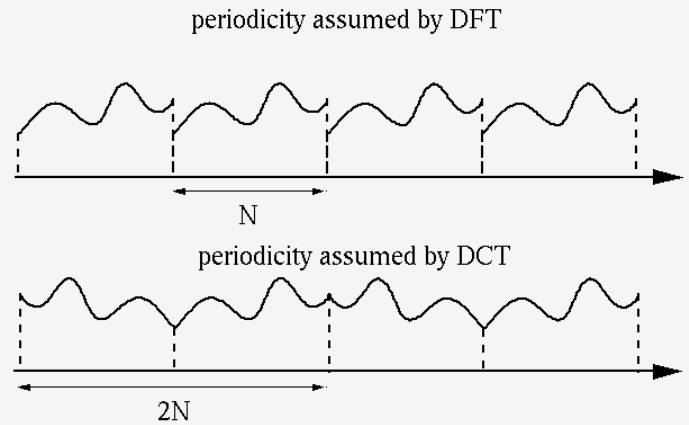


Fig. 1: Periodicity assumptions in the DCT and DFT. Note the discontinuities in the DFT [7].

**2. History.** One cannot mention the fast cosine transform without first talking about the function from which it is derived: the fast Fourier transform. The earliest version of the fast Fourier transform dates back to Gauss in 1805 [10]. Gauss was studying celestial bodies and was looking for a quick way to fit points on a line and doing so by hand so was tedious work. His main insight was that the data points could be divided into smaller periods and the fits of these subsections could be combined to form a fit for the entire data set. Thus, Gauss was ahead of his time and derived a divide and conquer algorithm over 150 years before they became commonplace in computer science [12]. Gauss's development comes even before Fourier for who the transform is named. While later proved revolutionary, Gauss never analyzed the complexity of his algorithm and merely noted:

> *"Truly, this method greatly reduces the tedium of mechanical calculation."*[12]

Unfortunately, Gauss's work on the FFT was only published after his death and even after it was published in encyclopedias it was still relatively unknown [10]. As a result several other mathematicians rediscovered the FFT over the years, and finally Cooley and Tukey famously popularized it in 1965 for use on a computer [6]. Cooley and Tukey showed that the FFT reduces complexity from $O(N^2)$ to $O(cN \log N)$ where $c$ is some constant factor, and laid a framework that could work with any method to compute the DFT. Today, work on improving the complexity of the FFT has mostly been limited to reducing the constant factor $c$, and most algorithms are a derivative of the Cooley-Tukey algorithm.

With the invention of the computer, digital compression blossomed as a field. While Fourier transforms are great for analysis, they compute unnecessary values if the intent is to be used in real applications. Concisely, there are two main advantages of the DCT compared to the DFT [7]:

1. The DCT is a real transform with better computational complexity than the DFT.
2. The DCT does not introduce discontinuity while imposing periodicity in the time domain, as shown in Figure 1.

As such the DCT became the method of choice for digital applications and a fast

version of it was needed to make its use practical. Thus the fast cosine transform (FCT) was born, and similarly to the FFT the FCT runs in $O(N \log N)$. Because the DFT and DCT are so closely related, early implementations of the fast cosine transform merely took the real part of the fast Fourier transform. However, this method is not optimal because the FFT works with complex numbers. The cosine transform only outputs real numbers, so the Fourier transform is doing unnecessary computation on imaginary numbers that will later be thrown away. It follows that there is post-processing involved in taking the real-valued output which also increases time complexity (although does not change asymptotic complexity). Thus, one common modern approach is to implement the FCT similar to the FFT instead of using the FFT itself; this means implementing a Cooley-Tukey style cosine transform. A Cooley-Tukey approach eliminates the post-processing and further speeds up the FCT because the algorithm can be implemented such that it only works with real numbers, not complex.

Each of the aforementioned methods for evaluating the discrete cosine transform are discussed in further detail in section 3.

**3. Fast Cosine Transforms.** There are several ways one can implement a DCT and in this paper we compare the following:

1. Naïve DCT
2. FCT from the FFT
3. Cooley-Tukey FCT

Each method has its own advantages and limitations which will be discussed in their respective section.

**3.1. Naïve DCT.** The naïve DCT is computed through directly following (1.1) for the one-dimensional DCT. As previously stated, this one-dimensional computation can be repeatedly applied over each dimension for a multi-dimensional input. A simple implementation could look like Algorithm 3.1 or Algorithm 3.2.

**Algorithm 3.1** (Scaled) One-Dimensional DCT

**function** 1D DCT($x$)
  $y$ = empty vector of the same dimensions as $x$
  **for** $k = 0, 1, \ldots, N-1$ **do**
    $\delta = \sqrt{\frac{1}{2}}$ if $k == 0$, else 1
    $\sigma = 0$
    **for** $n = 0, 1, \ldots, N-1$ **do**
      $\sigma = \sigma + x_n \cos\left(\frac{\pi k}{N}\left(n + \frac{1}{2}\right)\right)$
    **end for**
    $y_k = \sqrt{\frac{2}{N}}\delta\sigma$
  **end for**
  **return** $y$
**end function**

In both Algorithm 3.1 and Algorithm 3.2, it is clear to see the algorithm takes $O(N^2)$ time. While the runtime is slow compared to its competition, one advantage the naïve DCT has is that it can directly compute the transform for any size input without using any "tricks". As mentioned in subsection 3.2 and subsection 3.3, faster algorithms typically impose restrictions on the input while the naïve version does not.

**Algorithm 3.2** (Scaled) Two-Dimensional DCT

**function** 2D DCT($x$)
  $y$ = empty vector of the same dimensions as $x$
  **for** $k_1 = 0, 1, \ldots, N_1 - 1$ **do**
    **for** $k_2 = 0 : N_2 - 1$ **do**
      $\delta_1 = \sqrt{\frac{1}{2}}$ if $k1 == 0$, else 1
      $\delta_2 = \sqrt{\frac{1}{2}}$ if $k2 == 0$, else 1
      $\sigma = 0$
      **for** $n_1 = 0, 1, \ldots, N_1 - 1$ **do**
        **for** $n_2 = 0, 1, \ldots, N_2 - 1$ **do**
          $\sigma = \sigma + x_{n_1,n_2} \cos\left(\frac{\pi k_1}{N_1}\left(n_1 + \frac{1}{2}\right)\right) \cos\left(\frac{\pi k_2}{N_2}\left(n_2 + \frac{1}{2}\right)\right)$
        **end for**
      **end for**
      $y_{k_1,k_2} = \sqrt{\frac{2}{N_1}}\sqrt{\frac{2}{N_2}}\delta_1\delta_2\sigma$
    **end for**
  **end for**
  **return** $y$
**end function**

**3.2. FCT from the FFT.** The FFT has been a widely studied transform in mathematics, and thus there are many readily available fast algorithms to compute it. Recall that the FCT was derived from the FFT, and thus we are able to quickly determine the FCT from the output of some FFT algorithm. The DFT is defined as [5]:

$$y_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi j n k}{N}} \tag{3.1}$$

$$= \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi n k}{N}\right) - j \sin\left(\frac{2\pi n k}{N}\right)\right] \tag{3.2}$$

The fast Fourier transform is thus a method of calculating (3.1) in $O(N \log N)$ time instead of $O(N^2)$ if you were to compute it directly. Usually, the FFT is computed as a Cooley-Tukey algorithm with radix $r = 2$ [6]. With this in mind, we can incorporate it to the FCT by taking the real part of the FFT's output as in Algorithm 3.3.

Algorithm 3.3 is a simple approach and performs much better than the algorithms in subsection 3.1. Normally the FFT takes $O(cN \log N)$ time where $c$ is some constant usually less than 2, but the modifications to get the cosine transform from it add some extra computational work:

**FFT:** $O(2N \log N)$. The FFT typically works on a mirrored version of the input $x$ which is thus length $2N$ [5].

**Taking the real part:** $O(N)$. We must iterate through the entirety of the output to take the real part.

Overall the time complexity is $O(2N \log N) + O(N) = O(N \log N)$. So getting the FCT from the FFT has the same asymptotic runtime as the FFT, but there are more operations that aren't necessary for the cosine transform. Explicitly, the cosine transform does not require the imaginary part, so we ideally don't need to operate

**Algorithm 3.3** Fast Cosine Transform using the Fast Fourier Transform

**function** FCT VIA FFT($x$)
  $N = \text{LENGTH}(x)$
  $y =$ empty vector of the same dimensions as $x$
  $x' = x$ concatenated with $\text{REVERSE}(x)$ ▷ if $x = [1, 2]$ then $x' = [1, 2, 2, 1]$
  $\omega =$ first N elements of $\text{FFT}(x)$
  **for** $k = 0, 1, \ldots, N-1$ **do**
    $y_k = Re\left[e^{\frac{-jk\pi}{2N}}\omega_k\right]$
  **end for**
  **return** $y$
**end function**

on a mirrored length $2N$ input. Also, an ideal FCT algorithm would not need to reiterate over the output to take the real part and instead only work with the real part. Both of these optimizations would reduce complexity and are discussed further in subsection 3.3.

**Note on inputs:** many FFT algorithms restrict the input size to be prime (e.g. Rader's FFT [16]) or to have dimensions that are equal to $2^m$ for some integer $m$. When the input does not meet these specifications, some algorithms such as Bluestein's FFT (also known as the *chirp z-transform*) zero-pad the input to meet them [3]. While this improves generality, it affects complexity.

**3.3. Cooley-Tukey FCT.** A better approach is modifying the Cooley-Tukey algorithm to approximate the DCT. This way, we avoid the issues talked about in subsection 3.2 and get a $O(N \log N)$ algorithm. For more information on Cooley-Tukey algorithms, see [12].

One such algorithm is Lee's algorithm [4]. Lee's algorithm recursively decomposes the DCT into its even and odd parts and operates as described in subsection 3.3.1 [13].

**3.3.1. Lee's Algorithm.** Lee's algorithm is a recursive fast cosine transform in the structure of a Cooley-Tukey algorithm. Recall the formula for the DCT in one dimension (1.1). We are able to decompose this into even and odd parts and recursively evaluate it [13]. First, define:

$$g_k = x_k + x_{(N-1-k)} \tag{3.3}$$

$$h_k = \frac{x_k - x_{(N-1-k)}}{2\cos\left(\frac{n\pi}{2N}(2k+1)\right)} \tag{3.4}$$

(3.5)

These equations for $g_k$ and $h_k$ are the input to our recursive function calls during computation. They are used in the following cosine transform computations, $G_n$ and $H_n$.

$$G_n = \sum_{k=0}^{\frac{N}{2}-1} g_k \cos\left(\frac{n\pi}{2N}(2k+1)\right) \tag{3.6}$$

$$H_n = \sum_{k=0}^{\frac{N}{2}-1} h_k \cos\left(\frac{n\pi}{2N}(2k+1)\right) \tag{3.7}$$

For $n = 0, 1, \ldots, \frac{N}{2} - 1$. It follows that $G_n$ and $H_n$ are even and odd decompositions of the cosine transform, leading to the relationships below:

$$y_{2n} = G_n \tag{3.8}$$

$$y_{(2n+1)} = H_n + H_{(n+1)} \tag{3.9}$$

$$H_{\frac{N}{2}} = 0 \tag{3.10}$$

Lee's algorithm, like any Cooley-Tukey algorithm for the DCT, reduces time complexity to $O(N \log N)$. One possible implementation is expressed in Algorithm 3.4 [14].

**Algorithm 3.4** (Unscaled) One-Dimensional Lee's Algorithm

```
function LEE(x)
    n = the length of x
    if n == 1 then
        return COPY(x)
    else if n == 0 OR n mod 2 ≠ 0 then
        throw ValueError                          ▷ input must be of size 2^l
    else
        m = n mod 2
        γ = 1st to m^th elements of x
        ρ = n^th to (m + 1)^th elements of x      ▷ descending indices on x
        α = LEE(γ)                                ▷ recursive call
        β = LEE(ρ)                                ▷ recursive call
        y = empty vector of the same dimensions as x
        y_1, y_3, y_5, ..., y_{n-1} = α
        y_2, y_4, y_6, ..., y_n = β
        if LENGTH(β)> 1 then                      ▷ vectorized addition
            y_1, y_3, y_5, ..., y_{n-1} = y_1, y_3, y_5, ..., y_{n-1} + β_2, β_3, β_4, ...   ▷ exclude β_1
        end if
        return y
    end if
end function
```

As evident in Algorithm 3.4 the algorithm is a divide and conquer strategy recursively reducing the problem size half. Performance is discussed in section 4.

**Note on inputs:** inputs must be of size $2^m$ for some integer $m$ due to the currently implemented recursive structure of the algorithm.

**4. Performance.** Compared to the directly computed DCT, any FCT algorithm is an improvement. However, when asymptotic time complexity is equal, what separates good fast cosine transform algorithms from bad is the number of floating point operations and the amount of storage each of them take. In Figure 2, we see a comparison of the number of floating point operations (flops) between the algorithms.

From analysis, we know the time complexity of Lee's algorithm to be $O(N \log N)$[4]. However Figure 2 indicates that, despite performing asymptotically better than the naïve DCT, Lee's algorithm is a constant factor number of flops worse compared to optimal algorithms. Additionally, we can compare memory performance in Figure 3 and Figure 4.

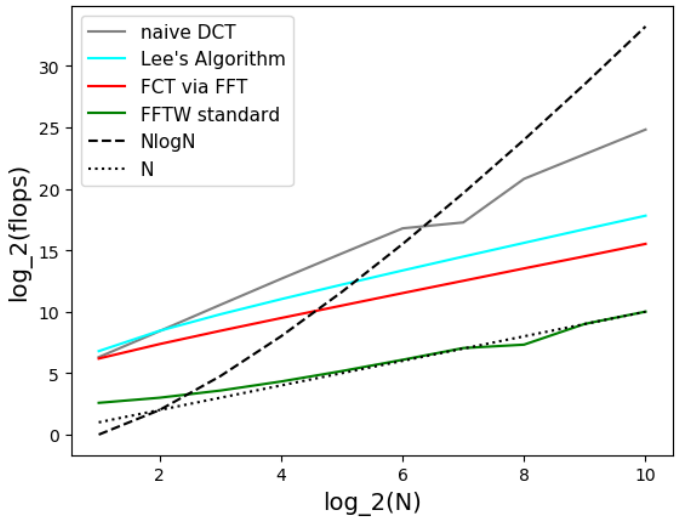


Fig. 2: Log-log scale of **flop counts** for different FCT algorithms over different sized inputs. FFTW is a widely used transform library.

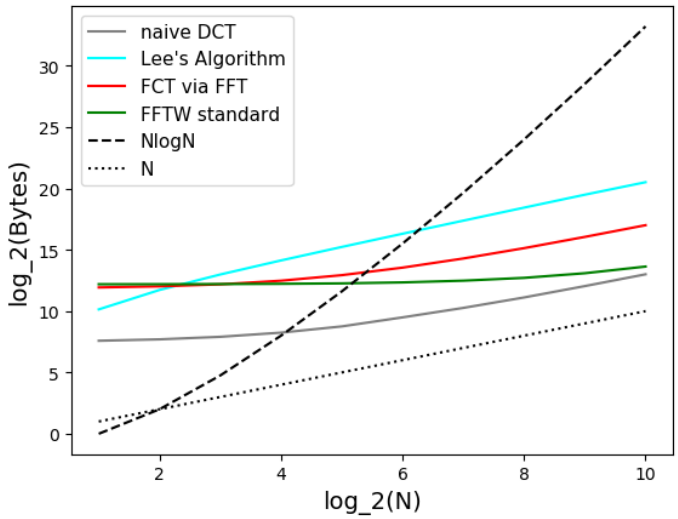


Fig. 3: Log-log scale of **memory** required for different FCT algorithms over different sized inputs. FFTW is a widely used transform library.

Figure 3 indicates that, except for small $N$ where results are difficult to interpret, Lee's algorithm looks to also be within a constant factor in terms of asymptotic memory performance. However, in terms of allocations, Figure 4 shows that Lee's algorithm is much worse than competing algorithms.

Lastly, Lee's algorithm has some deviation from the ground truth DCT output. This is because there is instability in the algorithm resulting in floating point errors

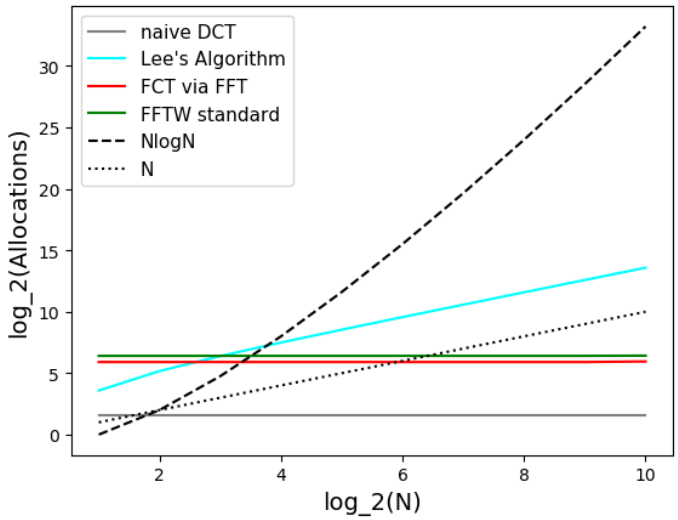


Fig. 4: Log-log scale of memory **allocations** for different FCT algorithms over different sized inputs. FFTW is a widely used transform library.

that propagate throughout [11].

**5. Further Advancements in Fast Cosine Transforms.** In typical compression applications, the FCT is applied to rather small input sizes: for example in JPEG compression $8 \times 8$ blocks of the input image are inputted into the FCT instead of the whole image at once. This is because compression applications use the cosine transform in the form of a mask, as shown in Figure 5. There has been extensive study

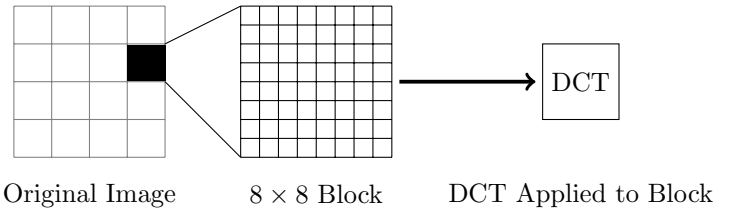


Fig. 5: Flow diagram of applying the DCT to an image. The discrete cosine transform is applied to blocks of size 8 in the form of a convolutional mask over the image. After this is done across the entire image, the image can be compressed using some encoding scheme.

of these small scale cases [9, 8, 17]. Here, a prevailing algorithm is Feig's algorithm, which further reduces the constant factor in the $O(cN \log N)$ time complexity but is meant only for $8 \times 8$ inputs. These algorithms show that there are further matrix properties of the DCT that can be exploited beyond Cooley-Tukey algorithms, and currently they are best suited in the small scale.

**6. Conclusions.** It is clear from section 4 that Lee's algorithm is much quicker than directly computing the discrete cosine transform. It is also clear that Lee's

algorithm is asymptotically as quick as the best available methods, but is slower by some constant factor. However, looking at Algorithm 3.4 we see that implementation is very simple: it is around 20 lines of code. The optimal algorithms likely take thousands of times more lines of code compared to Lee's algorithm in order to fully utilize system hardware (e.g. optimal cache usage). Despite the error propagation described in subsection 3.3.1, the ease of implementation compared to slightly better algorithms and the vast time complexity superiority compared to the naïve DCT computation make Lee's algorithm a quick and effective algorithm to deploy when one needs to compute a cosine transform.

**Acknowledgments.** I am grateful to Professor Steven Johnson and Jacob Gold, without whom this work would not be possible.

## REFERENCES

[1] N. Ahmed, T. Natarajan, and K. R. Rao, *Discrete cosine transform*, IEEE Transactions on Computers, C-23 (1974), pp. 90–93.
[2] F. Bellifemine, A. Capellino, A. Chimienti, R. Picco, and R. Ponti, *Statistical analysis of the 2d-dct coefficients of the differential signal for images*, Signal Processing: Image Communication, 4 (1992), pp. 477–488.
[3] L. Bluestein, *A linear filtering approach to the computation of discrete fourier transform*, IEEE Transactions on Audio and Electroacoustics, 18 (1970), pp. 451–455.
[4] Byeong Lee, *Fct – a fact cosine transform*, in ICASSP '84. IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 9, 1984, pp. 477–480.
[5] W. Cochran, J. Cooley, D. Favin, H. Helms, R. Kaenel, W. Lang, G. Maling, D. Nelson, C. Rader, and P. Welch, *What is the fast fourier transform?*, Proceedings of the IEEE, 55 (1967), pp. 1664–1674.
[6] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Mathematics of computation, 19 (1965), pp. 297–301.
[7] N. Drakos and R. Moore, *Discrete cosine transform - e186 handout*, 2013.
[8] E. Feig, *Fast scaled-dct algorithm*, in Image Processing Algorithms and Techniques, vol. 1244, International Society for Optics and Photonics, 1990, pp. 2–13.
[9] E. Feig and S. Winograd, *Fast algorithms for the discrete cosine transform*, IEEE Transactions on Signal processing, 40 (1992), pp. 2174–2193.
[10] M. Heideman, D. Johnson, and C. Burrus, *Gauss and the history of the fast fourier transform*, IEEE ASSP Magazine, 1 (1984), pp. 14–21.
[11] Hsieh Hou, *A fast recursive algorithm for computing the discrete cosine transform*, IEEE Transactions on Acoustics, Speech, and Signal Processing, 35 (1987), pp. 1455–1461.
[12] S. Johnson, *Introduction to numerical methods - lecture 35*, 2020.
[13] K. Lagerström, *Design and implementation of an mpeg-1 layer iii audio decoder (master's thesis)*, 2001.
[14] P. Nayuki, *Fast discrete cosine transform algorithms*, 2018.
[15] H. J. Nussbaumer, *The fast fourier transform*, in Fast Fourier Transform and Convolution Algorithms, Springer, 1981, pp. 80–111.
[16] F. Rader's, *Rader's fft algorithm*.
[17] Shue-Lee Chang and T. Ogunfunmi, *A fast dct (feig's algorithm) implementation and application in mpeg1 video compression*, in 38th Midwest Symposium on Circuits and Systems. Proceedings, vol. 2, 1995, pp. 961–964 vol.2.
[18] G. Strang, *The discrete cosine transform*, SIAM review, 41 (1999), pp. 135–147.