Reversible Integer FFT
PAT Project 2022

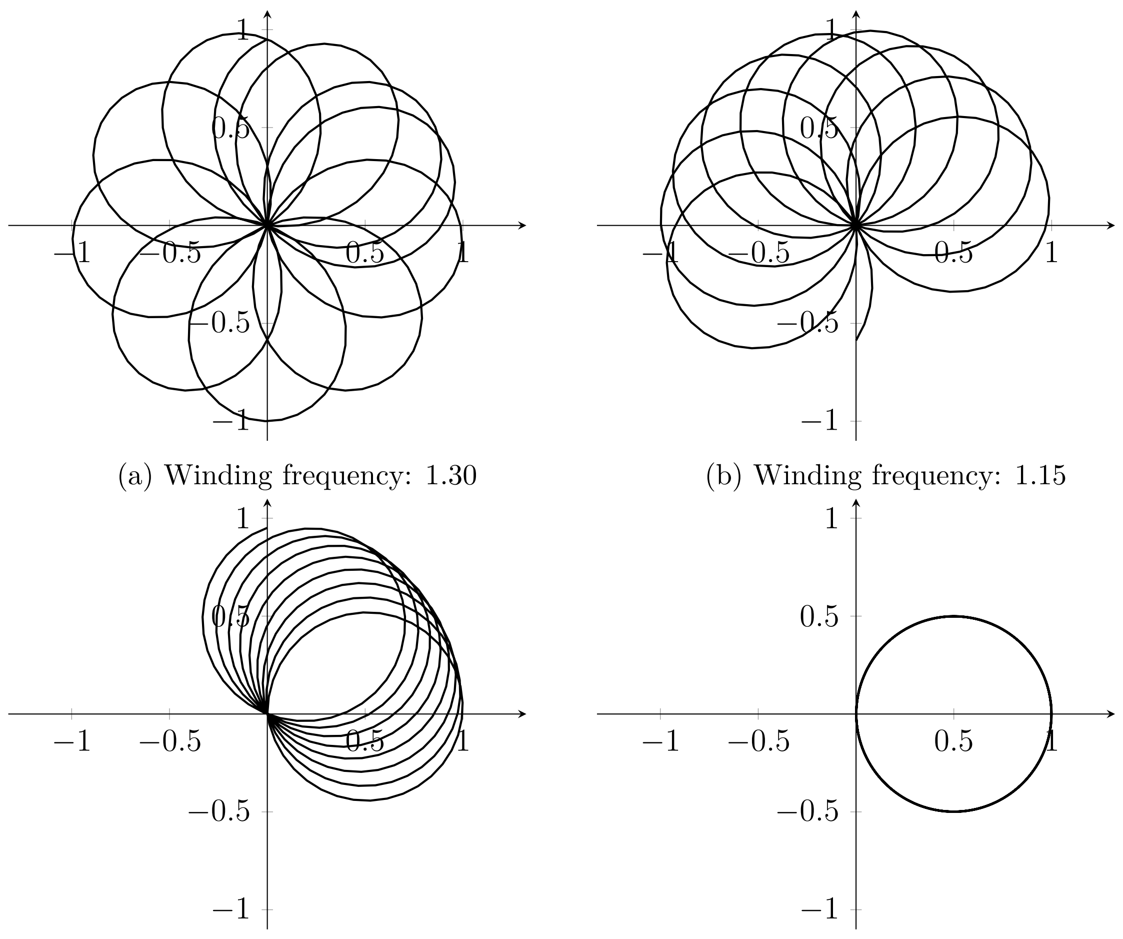
Troels Korreman Nielsen (xck773)

November 22, 2024

Abstract

The Fast Fourier Transform (FFT) is a widely-used efficient for computing the Discrete Fourier Transform (DFT) of signals. FFT and its derivatives have a variety of practical uses in mathematics, physics, signal processing, and many related fields. For example, the related Discrete Cosine Transform is employed in common media compression formats such as MP3, JPEG, and MP4. Especially in the cases of compression and de-noising, we are interested in utilizing both the transform and its inverse. FT is an injective function with a well-defined inverse transformation (IFT), but designing an approximation that retains this property presents some challenges. In this report, a reversible Fast Fourier Transform algorithm is presented that can be directly implemented in a reversible language. The ability to implement the algorithm in such a language can serve as proof of injectiveness. Furthermore, deriving the inverse transform from the forward transform can ensure that there are zero discrepancies between the two.

Contents


(a) Winding frequency: 1.30

(b) Winding frequency: 1.15

(c) Winding frequency: 1.05

(d) Winding frequency: 1.00

Figure 1: Plots of $sin(x)$ wound around zero in the complex plane with different winding "speeds". The integral of (1d) will deviate to the right in a much greater degree than the others.

1 The Fourier Transform

1.1 Definition

The Fourier Transform (FT) decomposes a function from the space or time domain into the frequency domain. If we view a function g as an infinite-dimensional vector, then FT transforms g from a representation in the standard basis to one in a basis of complex sinusoids. It can be defined as follows:

$$\hat{g}(f) = \int_{-\infty}^{\infty} g(x) \cdot e^{-i2\pi f x} dx \tag{1}$$

A continuous function $g : \mathbb{R} \to \mathbb{R}$ can be described as a sum of waves (in this case sinusoids). FT can tell us the presence of each frequency in the function.

It can be hard to develop an intuition for the transform. An excellent visual representation is provided by [6] in video form, which we can somewhat replicate with a sequence of images in figure 1. g is wound around zero in the complex plane with periodicity corresponding to f, and $\hat{g}(f)$ can be viewed as the "center of mass" for this winding, given by an integral. If a frequency is present in g, winding with the same periodicity will cause the integral to deviate from zero.

The Fourier inversion theorem states that, for certain functions, the inverse of the Fourier transform can be obtained by:

$$g(x) = \int_{-\infty}^{\infty} \hat{g}(f) \cdot e^{i2\pi fx} df \tag{2}$$

This can be used to build a function from a description of its sinusoid components.

In real-world applications, we usually aren't working with continuous functions. Typically we have a sequence of uniformly spaced samples approximating a recorded signal, and we wish to compute something FT-like for this. We cannot compute the FT of such a signal, but we can approximate it using the Discrete Fourier Transform (DFT):

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-i2\pi kn/N} \tag{3}$$

The DFT is also an invertible, linear transformation. The Inverse Discrete Fourier Transform (IDFT) is defined as:

$$x(n) = \sum_{k=0}^{N-1} X(k) \cdot e^{i2\pi kn/N} \tag{4}$$

We wish to write an efficient reversible algorithm that computes the DFT when run in the forward direction and computes the IDFT when run backwards.

2 Fast Fourier Transform

The DFT is not a hard sum to calculate, but naively computing it for $k \in [0; N)$ will take $O(N^2)$ time. By exploiting certain redundancies, we can recursively factorize the sum in such a way that it can be computed in $O(N \log N)$ time. An algorithm that takes advantage of such factorizations is called a Fast Fourier Transform (FFT).

A comprehensive explanation of FT, DFT, and FFT can be found in [2]. This book uses the Cooley-Tukey algorithm, but we will use a different factorization for simplicity and brevity.

2.1 Twiddle factor

Before we get into the factorization, let us abstract away some complexity. To simplify the representation and computation of the DFT, we define what is often referred to as the *twiddle factor*:

$$W_N = e^{-i2\pi/N} \tag{5}$$

Using this, we can then rephrase the DFT as:

$$X(k) = \sum_{n=0}^{N-1} W_N^{nk} x(n) \tag{6}$$

The twiddle factor can viewed as a rotation of the complex number $1 + 0i$ around $0 + 0i$ in the complex plane, scaled by $1/N$. Multiplying some complex number x by W_N^k will thus rotate x by an angle of $\theta = 2\pi \cdot k/N$ (see figure 2). As such, we can compute twiddle factors by

$$W_N^k = \cos\theta + i \sin\theta, \quad \theta = 2\pi k/N \tag{7}$$

As a result of being a rotated unit vector, the twiddle factor has certain properties that we can exploit. Namely, it wraps around after a full rotation, taking its negative is equivalent to

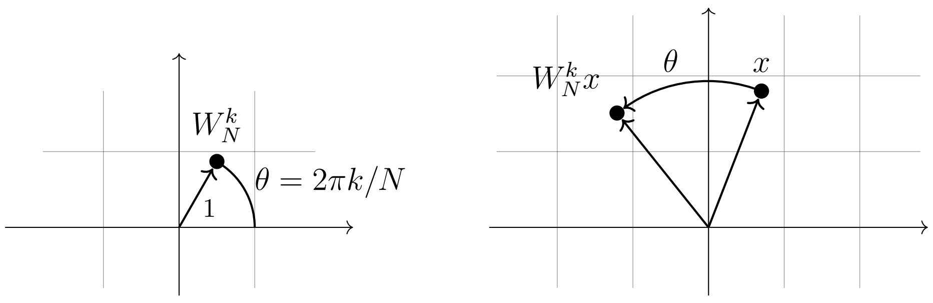


Figure 2: Representing the twiddle factor in the complex plane. The factor itself is a unit vector with angle θ. Multiplying some x by this number is equivalent to rotating x by θ.

making a half rotation, and upscaling k by some constant c is the same as downscaling N by $1/c$. These properties can be formally expressed as follows:

$$\textbf{Periodicity:}\quad W_N^k = W_N^{k+N} \tag{8}$$
$$\textbf{Symmetry:}\quad W_N^{k+N/2} = -W_N^k \tag{9}$$
$$\textbf{Scaling:}\quad W_N^{ck} = W_{N/c}^k \tag{10}$$

Having observed these, we can now move on to the factorization.

2.2 Factorization

There are numerous different ways in which the DFT sum can be factorized. They all have the same $O(N \log N)$ time complexity, but differ in the concrete amount of operations that they require. In the interest of simplicity, we adapt a factorization presented in [1]. FFT factorizations are traditionally expressed in matrix form, but this factorization is simple enough that we can skip that step. Note that the factorization as presented only works for inputs of length $N = 2^\gamma,\quad \gamma \in \mathbb{Z}^+$.

The first step of the factorization is to split the input list x into even- and odd-indexed elements:

$$x_{even} = \langle x_{2n} \mid n \in [0; N/2) \rangle \tag{11}$$
$$x_{odd} = \langle x_{2n+1} \mid n \in [0; N/2) \rangle \tag{12}$$

Next, we take the DFTs of these new lists. Using the scaling property of the twiddle factor, we can express these DFTs using W_N^{2nk} rather than $W_{N/2}^{nk}$:

$$X_{even}(k) = \sum_{n=0}^{N/2-1} W_{N/2}^{nk} x_{even}(n) = \sum_{n=0}^{N/2-1} W_N^{2nk} x(2n) \tag{13}$$
$$X_{odd}(k) = \sum_{n=0}^{N/2-1} W_{N/2}^{nk} x_{odd}(n) = \sum_{n=0}^{N/2-1} W_N^{2nk} x(2n+1) \tag{14}$$

Now we can reformulate $X(k)$ using X_{even} and X_{odd}. We split the sum into sums of even- and odd-indexed elements, factor out a coefficient of the odd sum, and then substitute the even and odd DFTs:

$$\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} W_N^{nk} x(n) && (15)\\
&= \sum_{n=0}^{N/2-1} W_N^{2nk} x(2n) + \sum_{n=0}^{N/2-1} W_N^{(2n+1)k} x(2n+1) && (16)\\
&= \sum_{n=0}^{N/2-1} W_N^{2nk} x(2n) + W_N^k \sum_{n=0}^{N/2-1} W_N^{2nk} x(2n+1) && (17)\\
&= X_{even}(k) + W_N^k X_{odd}(k) && (18)
\end{aligned}$$

In order to achieve our $O(N \log N)$ goal, we only wish to compute $X_{even}(k)$ and $X_{odd}(k)$ for $k \in [0; N/2)$. To still compute $X(k)$ for $k \in [N/2; N)$, we exploit the periodicity of the twiddle factor. Since $W_{N/2}^{k+N/2} = W_{N/2}^k$, we also have $X_{even}(k + N/2) = X_{even}(k)$. Thus we can "wrap around" for the upper half of X:

$$X(k) = \begin{cases} X_{even}(k) + W_N^k X_{odd}(k) & \text{if } k \in [0; N/2) \\ X_{even}(k') + W_N^k X_{odd}(k') & \text{if } k \in [N/2; N) \text{ where } k' = k - N/2 \end{cases} \tag{19}$$

The last step

- halves the number of multiplications,
- allows us to easily compute the FFT in-place,
- and defines the FFT in such a way that it can be directly inverted.

From the symmetry and periodicity properties of the twiddle factor, we get $W_N^k = -W_N^{k-N/2} = -W_N^{k'}$. Because of this equivalence, we can multiply the odd DFT by the same coefficient when computing the lower and upper DFT of X. This gives us the final factorization:

$$X(k) = \begin{cases} X_{even}(k) + W_N^k X_{odd}(k) & \text{if } k \in [0; N/2) \\ X_{even}(k') - W_N^{k'} X_{odd}(k') & \text{if } k \in [N/2; N) \text{ where } k' = k - N/2 \end{cases} \tag{20}$$

This factorization can be recursively applied until we have divided our input into pieces of size 1. Conveniently, the DFT of $\langle v \rangle$ is simply $\langle v \rangle$, so this makes for a simple base case.

2.3 Algorithm

With the factorization in place, we can construct an algorithm to compute the FFT:

Algorithm 1: FFT(x, N)

```
Input: Discrete uniformly spaced signal $x$ with size $N = 2^\gamma,\quad \gamma \in \mathbb{Z}$.
Output: Discrete Fourier Transform of $x$.
1  if $N = 1$ then
2      return $x$
3  Let $x_{even} = \langle x(2n) \mid n \in [0; N/2) \rangle$
4  Let $x_{odd} = \langle x(2n+1) \mid n \in [0; N/2) \rangle$
5  Let $X_{even} = \mathrm{FFT}(x_{even},\ N/2)$
6  Let $X_{odd} = \mathrm{FFT}(x_{odd},\ N/2)$
7  for $k \leftarrow 0$ to $N/2 - 1$ do
8      Let $o = W_N^k X_{odd}(k)$
9      $X_{even}(k) \leftarrow X_{even}(k) + o$
10     $X_{odd}(k) \leftarrow X_{even}(k) - o$
11 return $\langle X_{even}, X_{odd} \rangle$
```

The algorithm performs two recursive calls with sizes $N/2$. Then it performs $O(N)$ work to combine the results. We can thus show the time bound with a recursive formula:

$$T(N) = O(N) + 2T(N/2) = O(N \log N) \tag{21}$$

Now that we have a general overview of the algorithm, we can move on to the subject of reversibility.

3 Reversible FFT

As mentioned in section 1, both FT and DFT are invertible sums. The definition of IDFT also permits a factorization that is essentially equivalent to that of the DFT. However, we wish to write a reversible DFT that can be directly inverted to obtain an IDFT. There are some challenges that must be overcome in order to do this. Most of the challenges are detailed in [5], which also presents solutions to these.

3.1 In-place computation

The FFT algorithm described in the previous section creates two new arrays of even- and odd-indexed elements. Rather than create new arrays, we can rearrange the evens and odds into the upper and lower half of the array, then recurse on these halfs. This allows the transform to be performed in-place.

Taking it further, rather than rearranging the array at every recursion step, we can perform a single equivalent rearrangement of the array at the start of the algorithm. This is usually referred to as a scrambling of the array. In order to get an arrangement equivalent to recursively separating evens and odds down to the base case, we swap the element at each index with the index obtained from reversing its bit string.

After scrambling, we have $N/2$ pairs of even and odd base cases placed next to each other. These can be merged, giving us $N/4$ pairs of size 2 FFT solutions. We then merge these solutions into $N/8$ size 4 solutions, and repeat until we have obtained an FFT for the full array. Figure 3 shows this strategy in action for an array of size 8.

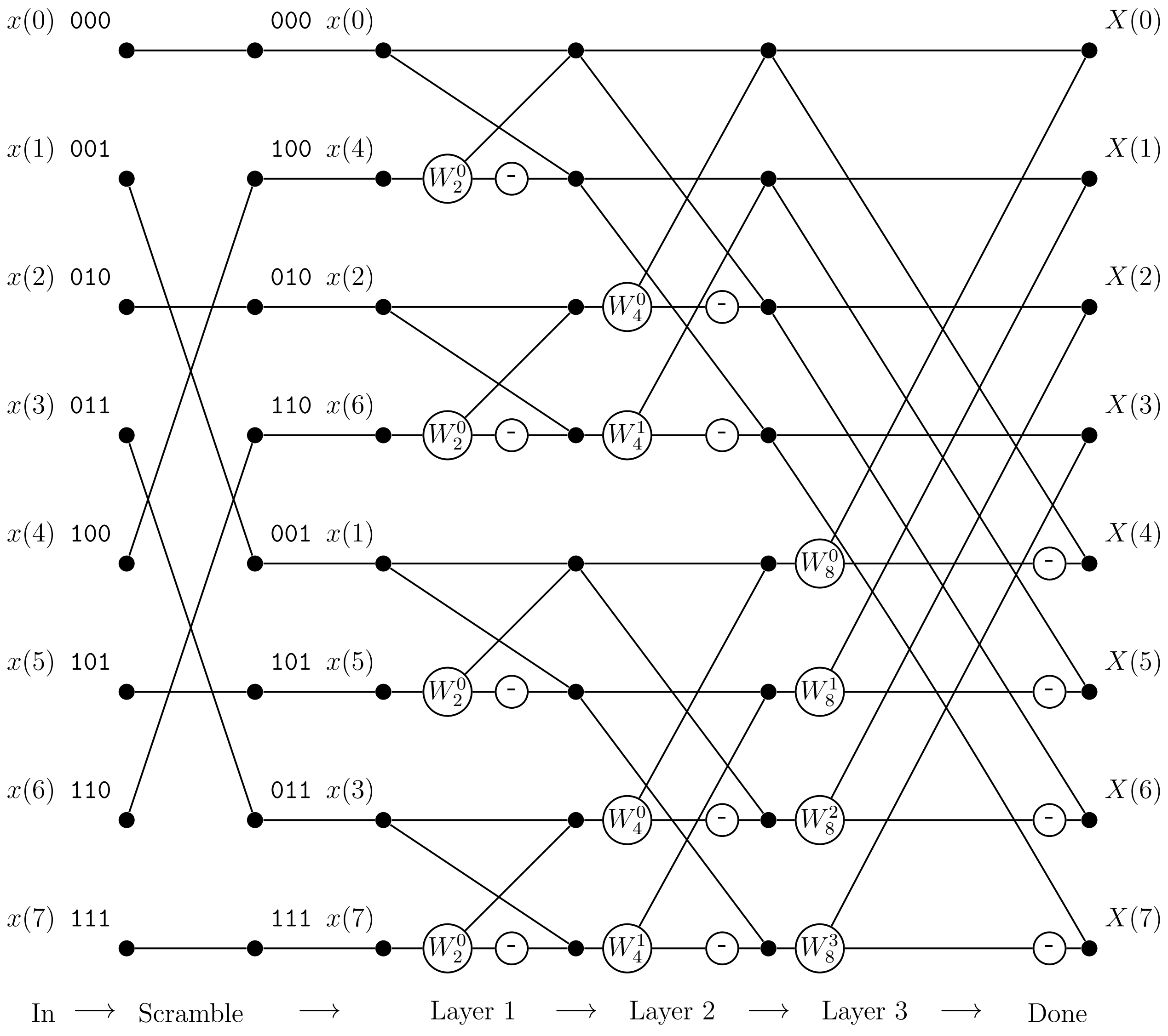


Figure 3: Example datapath representation of in-place FFT algorithm with $N = 8$.

3.2 Fixpoint arithmetic

The next problem is that we must find a way to represent complex numbers. Complex numbers can represented by two reals $a_r, a_i \in \mathbb{R}$ such that $a = a_r + ib_i$. Your first idea might be to use floating point numbers to represent a_r and b_i, but updates to floats are inherently destructive.

Instead, we use a fix-point representation. Each real number can be written as $r = x{\cdot}2^{-p}, \;\; x, p \in \mathbb{Z}$. The value of p is known as the fix-point for r, and can be visualized as a decimal point (binary point?) in the bit string. We will store r in an integer, and p will generally be implicit. Operations on these numbers can be performed with the integer equivalents on the x value.

Ignoring overflows, both addition and subtraction are invertible. Multiplication is only injective ie. left-invertible. This is quite clear when you consider what happens when dividing an odd number by 2. Furthermore, it should be noted that multiplication of two numbers with fix-points p_1 and p_2 results in a number with fix-point $p_1 + p_2$. We may wish to quantize (round) these integers to move the fixpoint back, but this is a destructive operation.

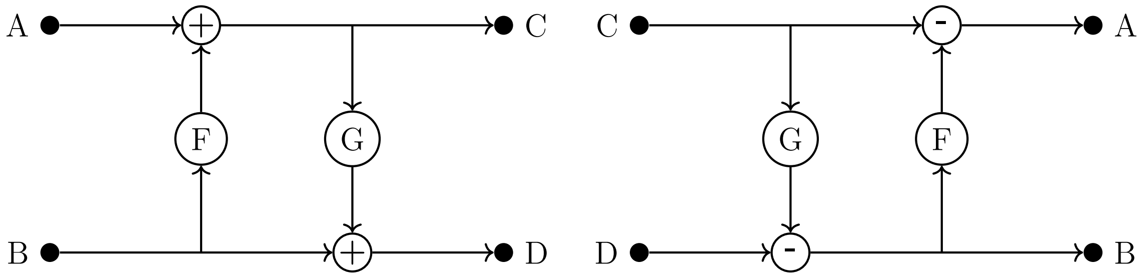
3.3 Lifting steps

Some central operations of our FFT algorithm must update two variables at once in a butterfly-like structure. That is to say, each value must be updated in a way that is dependent on both itself and the other, creating a datapath that looks like a "butterfly". Mutually dependent simultaneous updates are a problem for reversibility, as we cannot recover the original values. Luckily, our specific updates *can* be reversed, at least in a mathematical sense.

In order to make them practically reversible, we can convert them into a series of *lifting steps*. This method is described in detail in [3], although we use the method for somewhat simpler tasks.

Figure 4 shows the concept as datapaths. Say we need to perform a "butterfly" update on two variables A and B. Instead of performing simultaneous updates, we can run B through some function F and add it to A. Next, we can run A through another function G and add it back to B. Since the value added at each step can still be derived after the addition, we can reverse the operation by deriving and subtracting the same values in the opposite order.

If we can represent a butterfly update as a lifting scheme, we can reverse it. The method also has another benefit; the in-between functions like F and G do not have to be reversible. If the functions produce a lot of new information, we can quantize the result before addition. For example, fixpoint multiplication adds some amount p bits of resolution to the result, but this can be thrown out by rounding (eg. through left-shifting) the result before adding it to either A or B.



(a) Lifting steps. (b) Inverse path.

Figure 4: Dataflow path of a lifting step and its derived inverse.

3.4 Complex multiplication

As a reminder, complex multiplication is performed with:

$$ax = (a_r + a_i \cdot i)(x_r + x_i \cdot i) = a_r x_r - a_i x_i + (a_r x_i + a_i x_r)i \tag{22}$$

If we treat complex numbers as vectors, we can represent this as a matrix multiplication:

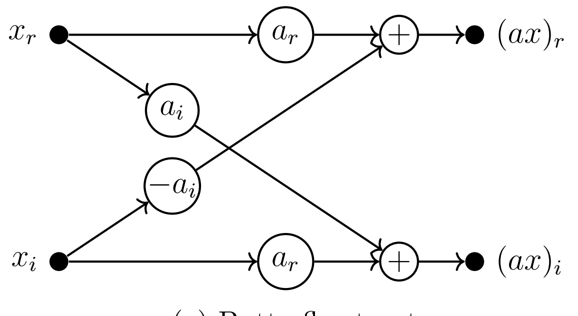
$$ax = \begin{bmatrix} a_r & -a_i \\ a_i & a_r \end{bmatrix} \begin{bmatrix} x_r \\ x_i \end{bmatrix} \tag{23}$$

It should be clear that an in-place multiplication of x with a requires a simultaneous update. Figure 5a shows the butterfly datapath for this update. In our case, we wish to multiply our values by a pre-known coefficient $a = W_N^k$. We also know this value to be of unit length, meaning that $\sqrt{a_r^2 + a_i^2} = 1$. Utilizing this equivalence, we can factorize the matrix as follows:

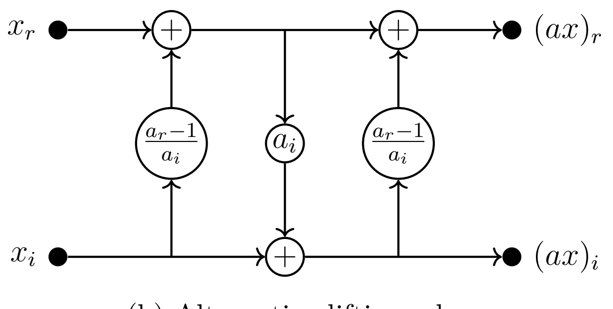
$$ax = \begin{bmatrix} 1 & \frac{c-1}{s} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{c-1}{s} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ x_i \end{bmatrix} \tag{24}$$

These multiplications can be performed by the lifting steps shown in figure 5b. Figure 5c shows the lifting scheme with added quantization steps. As for how to obtain these coefficients, we can pre-compute them and add them to the program as a lookup table. It is only necessary to compute the coefficients $W_N^k,\ k \in (0; N/4)$, as:

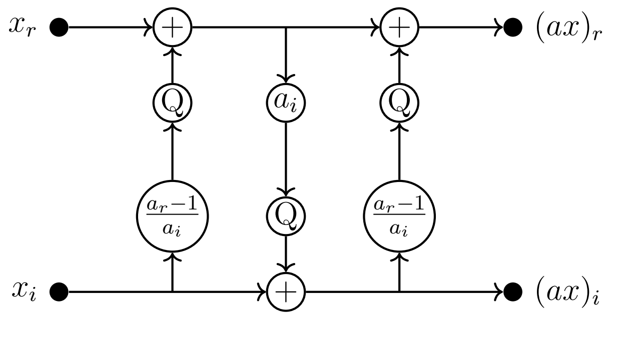
- $W_N^0 x$ and $W_N^{N/4} x$ can be trivially computed without coefficients.
- $W_N^k x$ for $k \in (N/4; N/2)$ can be computed by with $-W_N^{-k}$.
- W_N^k for $k \in [N/2; N)$ is never used.
- $W_{N/2^L}^k x$ can be obtained from $W_N^{2^L k} x$.



(a) Butterfly structure.



(b) Alternative lifting scheme.

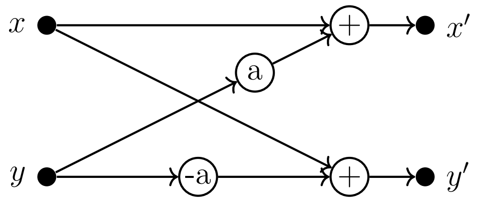


(c) Lifting scheme with quantization.

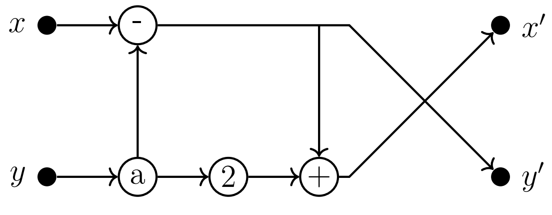
Figure 5: Datapaths for complex multiplication.

3.5 Reversible convolution

As can be seen quite clearly in the FFT datapath example (figure 3), merging the results of recursive steps requires a second butterfly update. This time, we need to compute $x_{\text{new}} = x + y$ and $y_{\text{new}} = x - y$ in-place (figure 6a). This is a reversible computation, as the original value of y can be retrieved with $y = (x_{\text{new}} - y_{\text{new}})/2$, and x can be retrieved with $x = x_{\text{new}} - y$. We can encode this reversibility by implementing the convolution as lifting steps, as seen in figure 6b.



(a) Butterfly structure.



(b) Alternative lifting scheme.

Figure 6: Datapaths for FFT convolution.

This introduces a significant challenge to the reversibility of our algorithm. This operation isn't *only* implemented through lifting steps, but requires an in-place doubling of y. Integer multiplications are only injective, as outputs must be a multiple of the factor for them to correspond to an input value. As a result, the inverse is only partially defined, for outputs that are a factor of 2 apart. This issue isn't covered by [5]. When reversing, you can divide by 2 and round off the result to get a decent result without changing the semantics for valid outputs. This strategy won't work in a fully reversible system however.

For our implementation, we write a multiplication function that panics when run in reverse on uneven output. This does have the disadvantage that our inverse FFT doesn't work on all outputs. There may be workarounds for this issue, but that wont be explored in the scope of this project.

3.6 Bit resolution

At this point, all parts of the algorithm have been molded into reversible operations. While the algorithm is reversible, the output *does* require more bits to represent than the input.

The offenders that produce these bits are the twiddle factor multiplications and convolutions. It should be fairly obvious to see that convolution increases bit-depth by one, as it effectively only performs either a subtraction or addition to the target variables. It isn't hard to see that convolution increases the bit-depth of the complex components by 1. For multiplication, the quantization in the lifting scheme (figure 5c) step removes the worst of the bit-depth increase. That leaves three additions, each increasing the depth by one bit. While [5] does prove that we can limit the increase for twiddle factor multiplication to one bit, we settle for a bound of 3 bits.

For every recursive step of the algorithm, each cell of our input array may be multiplied by one coefficient and will be convoluted with another cell. That makes for a 4 bit depth-increase per layer, and total a limit of $4 \log_2 N$ additional bits.

4 Implementation

Putting the theory into practice, let us implement our algorithm in a reversible system. Our language of choice is the extended version[4] of Janus[7]. The extra features provided make managing a larger program much easier than it would've been in base Janus.

Despite the lack of documentation and downloadable executables, it was possible to implement and run the FFT algorithm. The code is included in appendix A. The results from a test run is shown in figure 7, where we see the program successfully computing the FFT of an input signal. The test signal is a combination of two sine curves with different frequencies, and the FFT is shown to identify these.

Complex numbers are represented by arrays of 2 elements each. Two arrays are used for the real and imaginary components of the signal. The same is true for the twiddle factors. To simplify things, numerical operations for complex numbers are implemented as procedures.

The program tries to mimic the structure of the lattice diagram shown in figure 3. There are procedures for scrambling, multiplication of odds, and convolution of even-odd solution pairs. The `fft` and `step` invoke these procedures in layered steps with parameters unique to each step.

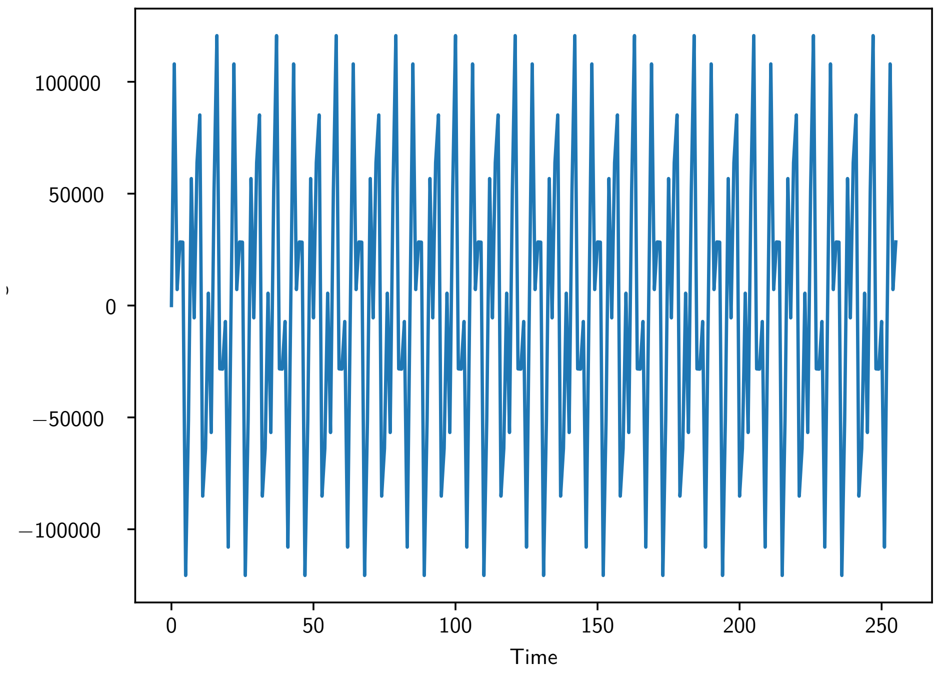
The bit depth of the twiddle factor was chosen somewhat arbitrarily to be 16. The depth of the input is up to the user, but 16 bits was also chosen for the test. This seemed fitting, as most sound is stored in a signed 16 bit format.

5 Conclusion

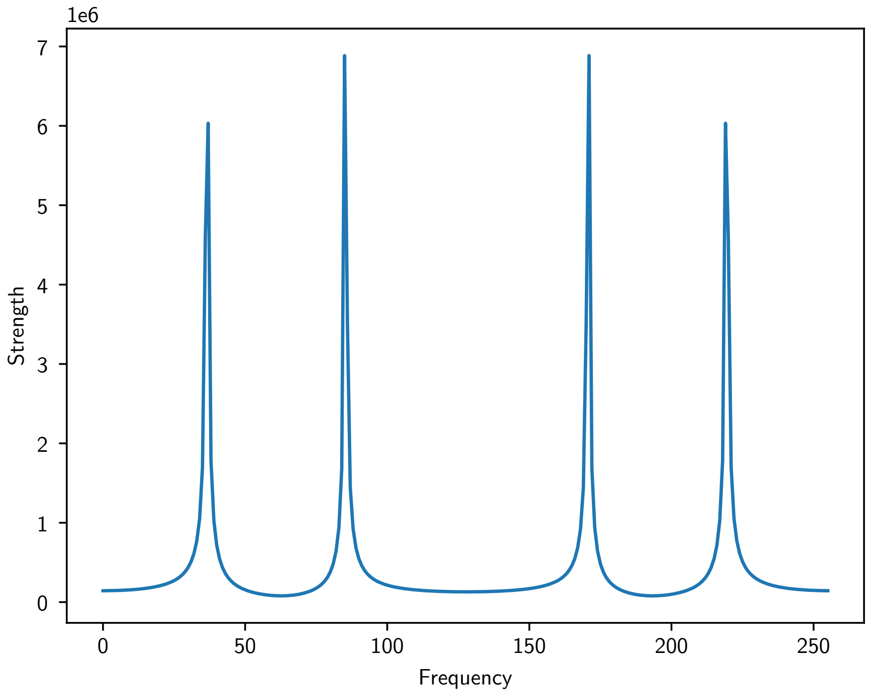
In this report, we have explored the Fourier Transform and its discrete fixed-time sibling, the Discrete Fourier Transform. We have seen how the DFT sum can be factorized into a recursive formulation, allowing us to develop an $O(N \log N)$-time algorithm for computing it, called the Fast Fourier Transform. We have shown how, using even-odd scrambling, fixpoint arithmetic, and lifting schemes, the FFT can be computed in-place and reversibly. Finally, we have implemented this FFT in the reversible programming language Janus, demonstrating that our algorithm is reversible in practice.

Further work might explore the challenge of making the transform bijective, with the goal of deciding whether or not this is possible. It would also be interesting to explore the possibility of implementing the Discrete Cosine Transform [1] in a reversible language.

[1] `https://en.wikipedia.org/wiki/Discrete_cosine_transform`



(a) Input signal: two combined sine functions with frequencies 1/3 and 1/7.



(b) Magnitude of FFT result.

Figure 7: Results from a test run of the Janus FFT implementation.

References

[1] Steve Brunton. The Fast Fourier Transform Algorithm, 2020. `https://www.youtube.com/watch?v=toj_IoCQE-4`.

[2] James W. Cooley, Peter A. W. Lewis, and Peter D. Welch. The fast fourier transform and its applications. *IEEE Transactions on Education*, 12(1):27–34, 1969.

[3] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analaysis and Applications*, 4:247–269, 1998.

[4] Claus Skou Nielsen, Michael Budde, and Michael Kirkedal Thomsen. Extended janus playground. `https://topps.di.ku.dk/pirc/?id=janusP`.

[5] S. Oraintara, Y.J. Chen, and T.Q. Nguyen. Integer fast fourier transform. *IEEE Transactions on Signal Processing*, 50(3):607–618, 2002.

[6] Grant Sanderson. But what is the Fourier Transform? A visual introduction., 2018. `https://www.youtube.com/watch?v=spUNpyF58BY`.

[7] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '07, page 144–153, New York, NY, USA, 2007. Association for Computing Machinery.

A Janus FFT code

```
// swap x[0] with arr0[idx], x[1] with arr1[idx]
procedure swap_arr(int x[2], int arr0[], int arr1[], int idx)
    x[0] <=> arr0[idx]
    x[1] <=> arr1[idx]

// swap ‘x‘ and ‘y‘
procedure swap(int x[2], int y[2])
    x[0] <=> y[0]
    x[1] <=> y[1]

// add ‘y‘ to ‘x‘
procedure add(int x[2], int y[2])
    x[0] += y[0]
    x[1] += y[1]

// subtract ‘y‘ from ‘x‘
procedure sub(int x[2], int y[2])
    x[0] -= y[0]
    x[1] -= y[1]

// multiply ‘x‘ by ‘2‘
// fails when reverse-executing if LSB is ‘1‘
procedure mul2(int x[2])
    local int tmp[2]
    call add(tmp, x)
    call add(x, tmp)
    tmp[0] -= x[0] / 2
    tmp[1] -= x[1] / 2
    delocal int tmp[2]

// x’ = -x
procedure negate(int x[2])
    local int tmp[2]
    tmp[0] += x[0]
    tmp[1] += x[1]
    x[0] -= 2 * tmp[0]
    x[1] -= 2 * tmp[1]
    tmp[0] += x[0]
    tmp[1] += x[1]
    delocal int tmp[2]

// rotate x 90 degrees clockwise
procedure rot90(int x[2])
    x[0] <=> x[1]
    local int tmp
    tmp += x[0]
    x[0] -= 2 * tmp
    tmp += x[0]
    delocal int tmp

// decide the index ‘other‘ to swap element index ‘i‘ with, given the size
   of the input
procedure scramble_idx(int i, int other, int log2N)
    local int j
    from j = 0
    loop
        if (i & (1 << j)) != 0 then
            other += 1 << log2N - j - 1
        fi (i & (1 << j)) != 0
        j += 1
    until j = log2N
    delocal int j = log2N

// perform a full even-odd scramble of ‘arr‘, making it ready for the FFT
   algorithm
procedure scramble(int arr[], int log2N)
    local int N = 2 ** log2N
    local int i
    from i = 0
    loop
        local int other
        call scramble_idx(i, other, log2N, N)
        local int tmp
        if i < other then
            arr[i] <=> tmp
            arr[other] <=> tmp
            arr[i] <=> tmp
        fi i < other
        delocal int tmp
        uncall scramble_idx(i, other, log2N, N)
        delocal int other
        i += 1
    until i = N
    delocal int i = N
    delocal int N = 2 ** log2N

// multiplies ‘x‘ by ‘a‘
// ‘a‘ is not a direct representation of a complex number, instead:
// ‘a[0] = -(a_r - 1)/a_i‘
// ‘a[1] = a_i‘
procedure mul(int x[2], int a[2], int twiddle_fixpoint)
    x[0] -= (a[0] * x[1]) >> twiddle_fixpoint
    x[1] += (a[1] * x[0]) >> twiddle_fixpoint
    x[0] -= (a[0] * x[1]) >> twiddle_fixpoint

// multiply indices ‘[offset; offset + length)‘ by ‘W^(k * k_step)‘ for ‘k
   in [0; length)‘
procedure mul_coefficients(
    int reals[],
    int imags[],
    int twiddle_m[],
    int twiddle_i[],
    int twiddle_fixpoint,
    int offset,
    int length,
    int k_step
)
    if length > 1 then
        local int halflength = length / 2
        local int idx = 1 // skip 0, as it isn’t rotated
        from idx = 1
        loop
            local int a[2]
            a[0] += twiddle_m[idx * k_step]
            a[1] += twiddle_i[idx * k_step]

            local int y[2]
            call swap_arr(y, reals, imags, idx + offset)
            call mul(y, a, twiddle_fixpoint)
            call swap_arr(y, reals, imags, idx + offset)

            call swap_arr(y, reals, imags, length - idx + offset)
            uncall mul(y, a, twiddle_fixpoint)
            call negate(y)
            call swap_arr(y, reals, imags, length - idx + offset)
            delocal int y[2]

            a[1] -= twiddle_i[idx * k_step]
            a[0] -= twiddle_m[idx * k_step]
            delocal int a[2]
            idx += 1
        until idx = halflength

        local int y[2]
        call swap_arr(y, reals, imags, idx + offset)
        call rot90(y)
        call swap_arr(y, reals, imags, idx + offset)
        delocal int y[2]
        delocal int idx = halflength
        delocal int halflength = length / 2
    fi length > 1

// convolves x and y resulting in:
// x' = x + y
// y' = x - y
procedure convolve(int x[2], int y[2])
    call sub(x, y)
    call mul2(y)
    call add(y, x)
    call swap(x, y)

// convolve even and odd indexes
procedure convolve_pairs(int reals[], int imags[], int offset, int length)
    local int halflength = length / 2
    local int idx = 0
    from idx = 0
    loop
        local int x[2]
        local int y[2]

        call swap_arr(x, reals, imags, idx + offset)
        call swap_arr(y, reals, imags, idx + halflength + offset)
        call convolve(x, y)
        call swap_arr(y, reals, imags, idx + halflength + offset)
        call swap_arr(x, reals, imags, idx + offset)

        delocal int y[2]
        delocal int x[2]

        idx += 1
    until idx = halflength
    delocal int idx = halflength
    delocal int halflength = length / 2

procedure step(
    int layer,
    int log2N,
    int reals[],
    int imags[],
    int twiddle_m[],
    int twiddle_i[],
    int twiddle_fixpoint
)
    local int N = 2 ** layer
    local int num_sections = 2 ** (log2N - layer)

    local int section
    from section = 0
    loop
        local int section_offset = section * N
        call mul_coefficients(
            reals,
            imags,
            twiddle_m,
            twiddle_i,
            twiddle_fixpoint,
            section_offset + N / 2,
            N / 2,
            num_sections
        )
        call convolve_pairs(reals, imags, section_offset, N)
        delocal int section_offset = section * N
        section += 1
    until section = num_sections
    delocal int section = num_sections

    delocal int num_sections = 2 ** (log2N - layer)
    delocal int N = 2 ** layer

procedure fft (
    int log2N,
    int reals[],
    int imags[],
    int twiddle_m[],
    int twiddle_i[],
    int twiddle_fixpoint
)
    call scramble(reals, log2N)
    local int layer = 0
    from layer = 0
    do
        layer += 1
        call step(layer, log2N, reals, imags, twiddle_m, twiddle_i,
   twiddle_fixpoint)
    until layer = log2N
    delocal int layer = log2N

procedure main()
    int log2N = 8

    // twiddle factor values
    int twiddle_fixpoint = 16
    constant int twiddle_m[] = {0, 804, 1608, 2413, 3219, 4026, 4834, 5643,
   6454, 7267, 8083, 8900, 9721, 10544, 11371, 12201, 13035, 13874, 14716,
   15563, 16415, 17273, 18136, 19005, 19880, 20761, 21650, 22545, 23449,
   24360, 25280, 26208, 27145, 28092, 29050, 30017, 30996, 31986, 32988,
   34002, 35029, 36070, 37125, 38195, 39280, 40382, 41500, 42635, 43789,
   44962, 46155, 47369, 48604, 49862, 51144, 52451, 53784, 55143, 56531,
   57949, 59398, 60879, 62395, 63946}
```

```
 constant int twiddle_i[] = {0, 1608, 3215, 4821, 6423, 8022, 9616,
11204, 12785, 14359, 15923, 17479, 19024, 20557, 22078, 23586, 25079,
26557, 28020, 29465, 30893, 32302, 33692, 35061, 36409, 37736, 39039,
40319, 41575, 42806, 44011, 45189, 46340, 47464, 48558, 49624, 50660,
51665, 52639, 53581, 54491, 55368, 56212, 57022, 57797, 58538, 59243,
59913, 60547, 61144, 61705, 62228, 62714, 63162, 63571, 63943, 64276,
64571, 64826, 65043, 65220, 65358, 65457, 65516}

 int reals[] = {0, 107993, 7136, 28434, 28319, -120649, -51240, 56754,
-5518, 63891, 85190, -85192, -63894, 5516, -56757, 51237, 120647, -28321,
 -28437, -7138, -107995, -2, 107993, 7136, 28434, 28319, -120649, -51240,
 56754, -5518, 63891, 85190, -85192, -63894, 5516, -56757, 51237, 120647,
 -28321, -28437, -7138, -107995, -2, 107993, 7136, 28434, 28319, -120649,
 -51240, 56754, -5518, 63891, 85190, -85192, -63894, 5516, -56757, 51237,
 120647, -28321, -28437, -7138, -107995, -2, 107993, 7136, 28434, 28319,
-120649, -51240, 56754, -5518, 63891, 85190, -85192, -63894, 5516,
-56757, 51237, 120647, -28321, -28437, -7138, -107995, -2, 107993, 7136,
28434, 28319, -120649, -51240, 56755, -5518, 63891, 85190, -85192,
-63894, 5516, -56757, 51237, 120647, -28321, -28437, -7138, -107995, -2,
107993, 7136, 28434, 28319, -120649, -51240, 56754, -5518, 63891, 85190,
-85192, -63894, 5516, -56756, 51237, 120647, -28321, -28437, -7138,
-107995, -2, 107993, 7136, 28434, 28319, -120649, -51240, 56754, -5518,
63891, 85190, -85192, -63894, 5516, -56757, 51237, 120647, -28321,
-28437, -7138, -107995, -1, 107993, 7136, 28434, 28319, -120649, -51240,
56754, -5518, 63891, 85190, -85192, -63894, 5516, -56757, 51237, 120647,
-28321, -28437, -7138, -107995, -2, 107993, 7136, 28434, 28319, -120649,
-51240, 56755, -5518, 63891, 85190, -85192, -63894, 5516, -56756, 51237,
120647, -28321, -28437, -7138, -107995, -2, 107993, 7136, 28434, 28319,
-120649, -51240, 56754, -5518, 63891, 85190, -85192, -63894, 5516,
-56756, 51237, 120647, -28321, -28437, -7138, -107995, -2, 107993, 7136,
28434, 28319, -120649, -51240, 56754, -5518, 63891, 85190, -85192,
-63894, 5516, -56757, 51237, 120647, -28321, -28437, -7138, -107995, -2,
107993, 7136, 28434, 28319, -120649, -51240, 56755, -5518, 63891, 85190,
-85192, -63894, 5516, -56757, 51237, 120647, -28321, -28437, -7138,
-107995, -2, 107993, 7136, 28434}
 int imags[256]

 call fft(log2N, reals, imags, twiddle_m, twiddle_i, twiddle_fixpoint)
```