Lecture Notes on the Fast Fourier Transform

1. Motivation: Digital Signal Processing

Consider a *signal* (any quantity that is a function of time, perhaps pressure in the air) affecting a *system* (anything that responds to a signal, like a microphone). The outcome of this interaction is called, naturally, *the response of the system to the signal* (see Figures (a) and (b)). The question is, how are we to compute such response digitally? In *many* applications (in which often the horizontal axis is not time, but also space, e.g. in image processing) it is of great interest to compute the system response, thereby simulating the system. The following is a rough description of how this is done.

First we *digitize* the signal, by *sampling it* often enough (Figure (c); how often is a whole subject in its own). Then we must know how the system behaves. It turns out that, if the system is *time invariant* (does not change its behavior over time) and *linear* (roughly, gives twice the response to twice the signal), then its behavior is completely captured by its *impulse response* —a description of what it does over time if it is given a sudden unit "jerk" at time zero (see Figures (d) and (e)). If we know that, since any signal is the sum of such jerks happening at various times, and since we know that the system is time-invariant and linear, then all we have to do is calculate the responses at various times, and add them to get the total system response.

To do the algebra, suppose that the signal is the sequence $(a_0, \ldots, a_{T-1})$ of real numbers, and the system impulse response is $(b_0, \ldots, b_{T-1})$ (assume that they both have the same time horizon, that is, they both die after T time units. Then at time 0 we have the system response $a_0 b_0$ —only the first pulse of the signal has arrived, and has only gotten the immediate response b_0. At time 1 we have the system response $a_0 b_1 + a_1 b_0$, because now a_1 gets the immediate response, while a_0 causes the delay-1 response of the system, and the two are added. After two steps, then the system response is $a_0 b_2 + a_1 b_1 + a_2 b_0$.

Question: What is the system response after t time units? If $t < T$, that is, if the signal keeps arriving at time t, then it is $c_t = \sum_{i=0}^{t} a_i b_{t-i}$. If $t \geq T$ —that is, if the signal has died out— then the system response keeps coming, since the system keeps responding to the signal in the past: The system response is then $c_t = \sum_{i=t-T+1}^{T-1} a_i b_{t-i}$. Finally, at $t = 2T - 1$ we have $c_{2T-2} = a_{T-1} b_{T-1}$, and thereafter $c_t = 0$: the system response has died out.

This sequence of formulas for $c_0, \ldots, c_{2T-2}$ should remind you of something from highschool algebra: *They are precisely the formulas for the coefficients of the product of two polynomials!*

$$(\sum_{i=0}^{T-1} a_i x^i) \cdot (\sum_{i=0}^{T-1} b_i x^i) = \sum_{i=0}^{2T-2} c_i x^i$$

This is not a coincidence: We can think of both the signal and the system response as two polynomials in x, *where x can be thought of as a unit delay,*, x^2 two delays, etc. Then the product of the two polynomials is, naturally enough, the system response.

Conclusion: It is of great interest to compute very fast the coefficients $c_t, t = 0, \ldots, 2T-2$, of the product of two given polynomials of degree $T - 1$.

2. The Fast Fourier Transform

Unfortunately, just by looking at the formulas, the number of operations required to compute the c_t's seems to be $\Omega(T^2)$: The number of terms increases from 1 to T and then down to 1 again, for a sum of about T^2. How can we calculate the c_t's much faster?

Here is a novel idea: Remember that the c_t's are the coefficient of the polynomial $C(x) = \sum_{i=0}^{2T-2} c_t x^t = A(x) \cdot B(x)$, where $A(x) = \sum_{i=0}^{T-1} a_t x^t$, and $B(x) = \sum_{i=0}^{T-1} b_t x^t$. And here is a scheme for calculating these coefficients:

1. Calculate the values of $A(x)$ and $B(x)$ at enough points $x_1, \ldots, x_n$ where $n \geq 2T - 1$.
2. Calculate the values of $C(x)$ at these points as $C(x_i) = A(x_i) \cdot B(x_i), i = 1, \ldots, n$
3. Now that we know at least $2T - 1$ values of the $2T - 2$-degree polynomial $C(x)$, we can interpolate *and recover the coefficients. (Recall that there is a unique d-degree polynomial that goes through $d + 1$ points.)*

But there are problems with this approach: Although step (2) is easy (it only requires n multiplications), step (1) seems to still require $\Omega(n^2)$ operations, and step (3) seems even harder. Have we accomplished nothing with this clever manoeuvre?

It turns out that we need another trick: *Pick the points $x_1, \ldots, x_n$ on which to evaluate $A(x)$ and $B(x)$ so that the n evaluations can be done* together *very fast.* It turns out that the most clever way to choose these points is to find n different points $x_1, \ldots, x_n$ such that the equation $x^n = 1$ holds for all of them.

Obviously, there are at most two real numbers that satisfy this equation -1, and perhaps -1, if n is even. But there are exactly n *complex* numbers that do: *The n complex roots of unity* (see the figure). They are n points lying, in the complex plane, on the unit circle. And since on the unit circle raising to the nth power means multiplying the angle by n, all n of these numbers are mapped to the real unity when raised to the nth power. Let us call then $x_1 = 1, x_2 = w, x_3 = w^2, x_4 = w^3, \ldots, x_n = w^{n-1}$. What we need to remember from now on about these numbers is that w is some number satisfying $w^n = 1$ —nothing else. Except one thing: If we take a root of unity like w^i, and add its powers $1 + w^i + w^{2i} + \ldots + w^{(n-1)i}$ then we get 0 —because the powers of w^i are just points around the origin, "pulling it in all different directions." With one exception: If $i = 0$, then of course the sum is $1 + 1 + \ldots + 1 = n$.

We conclude that we want to find a fast way to compute these n values:

$$A(w^j) = \sum_{i=1}^{n-1} a_i w^{ij}, j = 0, \ldots, n-1. \tag{1}$$

In this equation, j varies over $0, \ldots, n-1$ to define the n points $x_1, \ldots, x_n$ on which to evaluate $A(x)$, and the summation is the value $A(x_{j+1}) = A(w^j)$. We also need to remember all the time that $w^n = 1$ —this is the fact that is going to save us from the $\Omega(n^2)$ algorithm.

We are going to compute all n values in (1) together, *by divide-and-conquer.* Assume that n is a power of two —presumably the next power of two above $2T - 2$. Then, we *divide* the sequence of coefficients $a_0, \ldots, a_{n-1}$ into two subsequences: The *even subsequence* $a_0, a_2, a_4, \ldots, a_{n-2}$, and the *odd subsequence* $a_1, a_3, a_5, \ldots, a_{n-1}$. Then we can write (1) as

$$\begin{aligned} A(w^j) &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} w^{2ij} + \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w^{2ij+j} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} (w^2)^{ij} + w^j \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} (w^2)^{ij} \end{aligned}$$

Now this equation is just two problems of the same kind applied to sequences of length $\frac{n}{2}$ (compute the values of a degree $\frac{n}{2} - 1$-polynomial at the $\frac{n}{2}$-nd roots of 1 —notice that that is $1, w^2, w^4, \ldots, w^{2n-2}$ are precisely the $n/2$nd roots of unity). To obtain the $A(w^j)$ from the results of the conquered subproblems, we just multiply the $j \bmod \frac{n}{2}$th result of the second

evaluation by w^j and add it to the corresponding $j \bmod \frac{n}{2}$th result of the first. The points on which we need to evaluate the subproblems are just $\frac{n}{2}$, because this is the number of the possible values of $(w^2)^j$. This is quintessential divide-and-conquer, with recurrence

$$T(n) = 2T(\frac{n}{2}) + n,$$

where the $O(n)$ term is the work required to "put together" the results of the conquered parts (n multiplications and additions). The total complexity of the alorithm is, as we know, $O(n \log n)$.

This algorithm, which computes the values of an n-degree polynomial at the n n-th roots of unity in $O(n \log n)$ time by divide-and conquer is called *the Fast Fourier Transform (FFT).* It is perhaps one of the most important and widely used algorithms; it was discovered by Cooley and Tuc key in the 1950's.

Notice that the FFT, as described so far, only takes care of Step 1 of our scheme: Evaluating $A(x)$ and $B(x)$ at n points. How are we to carry out Step (3) —recovering the coefficients of $C(x)$ from these n values? The amazing fact is that *this can be done with another FFT*—but this time w^{-1} playing the role of w. This is just algebra. Suppose that we apply this "inverse" FFT to the values of $C(w^j), j = 1, \ldots, n-1$.

$$\begin{aligned} \sum_{j=0}^{n-1} C(w^j) w^{-jk} &= \sum_{j=0}^{n-1} (\sum_{i=0}^{n-1} c_i w^{ij}) w^{-jk} \\ &= \sum_{i=0}^{n-1} (\sum_{j=0}^{n-1} c_i w^{j(i-k)}) \end{aligned}$$

Consider however the inner sum of the last line, for some fixed values of i and k. If these values are the same, then the parenthesis contributes n. If they are not, then the powers of $w^{i-k} \neq 1$ cancel each other! So, the above sum is just $\sum_{j=0}^{n-1} C(w^j) w^{-jk} = c_k$ —it recovers the coefficients of $C(x)$.

To summarize, the three steps of our signal processing–polynomial multiplication algorithm now are these:

Step 1: Compute the FFT of $a_0, \ldots, a_{n-1}$ top obtain the sequence $A_0, \ldots, A_{n-1}$. Repeat with $b_0, \ldots, b_{n-1}$ top obtain the sequence $B_0, \ldots, B_{n-1}$.
Step 2: Compute $C_i := A_i \cdot B_i, i = 0, \ldots, n-1$ (note that these are complex number multiplications.
Step 3: Compute the inverse FFT (FFT with w^{-1} in the place of w), and then divide the results by n, to obtain $c_0, \ldots, c_{n-1}$.

Two last remarks: We often need to solve the system response problem when the input signal is *periodic* —that is, it is repeated after n time units (see Figure 1(f)). The system response then is found by an FFT with *half* the points ($n = T$). Can you see why?

Also, since all we needed from w is the equations $w^n = 1$ and $\sum w^i = 0$, we could use any arithmetical domain where these equations hold —not necessarily the complex numbers with their slow multiplications. We shall see in good time that there are domains in which we are computing modulo a large integer, in which such equations hold.

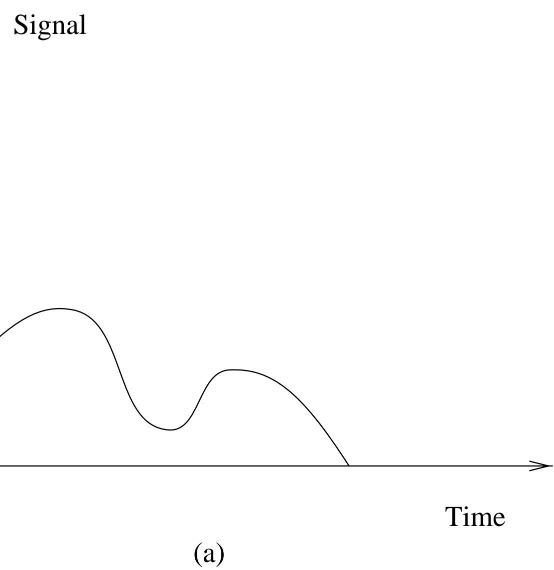
3. The FFT Circuit As with all divide-and-conquer algorithms, it is worthwhile to *unravel*

the recursion, to see what the algorithm *really* does. If we do this, the algorithe becomes the

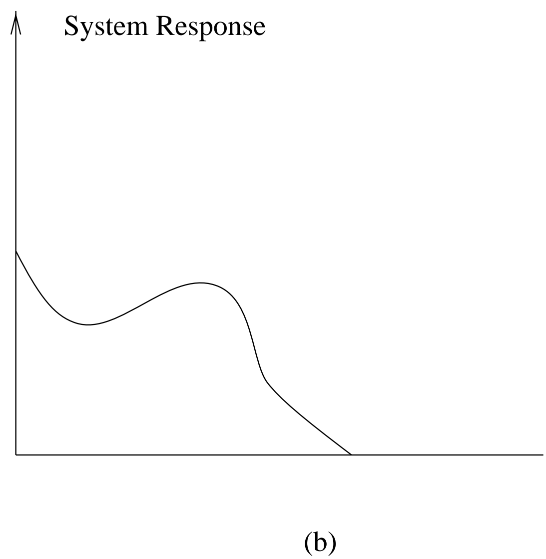
circuit shown in Figure 3. A word of explanation: The nodes are complex variables. The nodes on the left are the inputs (but in a funny order), and those on the right are the outputs. An arrow labeled with the integer j (unlabeled arrows are labeled "0") from x to y can be though of as carrying the value $x \cdot w^j$ to y. The two arrows coming into a node (other than the input nodes) are added together. Under this interpretation, Figure 3 shows the FFT of eight points.

Notice these properties of this circuit:

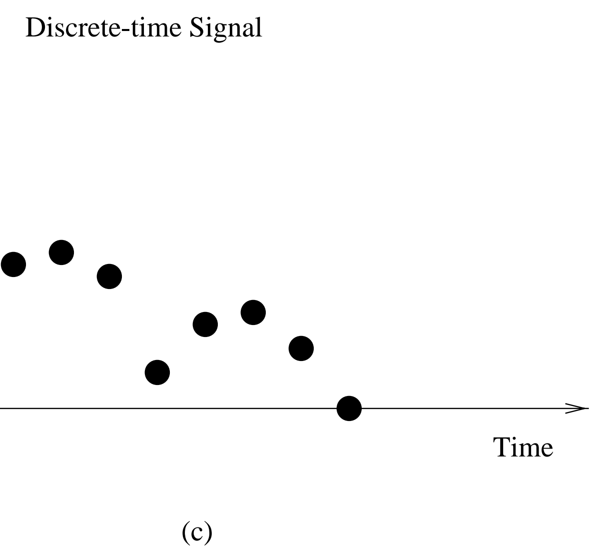
- There are $3 = \log n$ levels, with n variables each, and four complex operations per variable (actually, seven real operations), for a total of $7n \log n$ operations.
- There is a unique path between every input node and every output node.
- The path between a_i and $A(w^j)$ has label sum equal to ij modulo 8 (and it makes sense to take powers of w modulo 8, since we know that $w^8 = 1$).
- The previous two facts ensure that the circuit correctly computes the FFT.
- Why are the inputs mixed up this weird order? Can you find the pattern? (Hint: Compare the *binary representations* of the indices of an input and an output that are opposite one another.)
- Notice how neatly arranged this circuit is for *parallel evaluation.* Indeed, the FFT is a natural for parallelism, and can be carried out in $\log n$ short paralle stages. Often the FFT is computed by specialized embedded parallel hardware.
- Each stage of the FFT consists of $\frac{n}{2}$ "butterfly" operations —a typical butterfly is the subcircuit shown in bold in Figure 3.



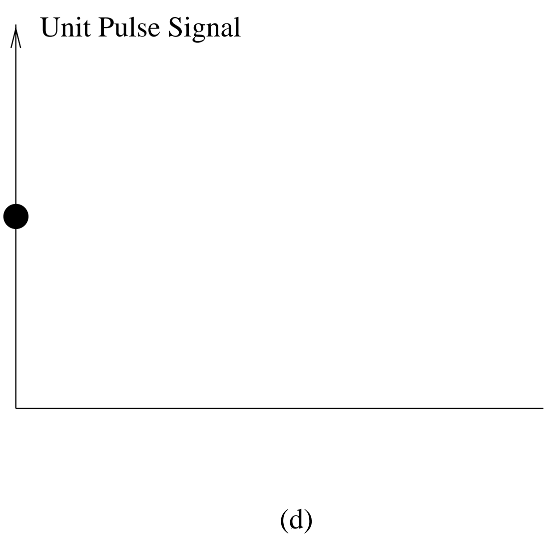
(a)



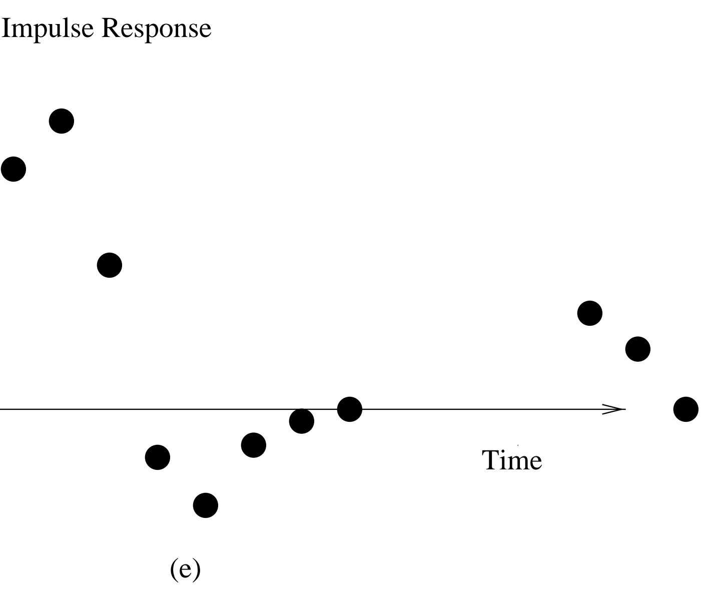
(b)



(c)



(d)



(e)

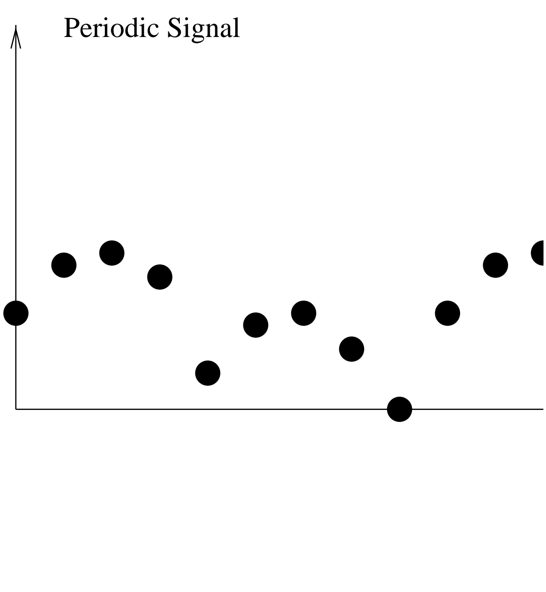


Figure 1: Signal and system response

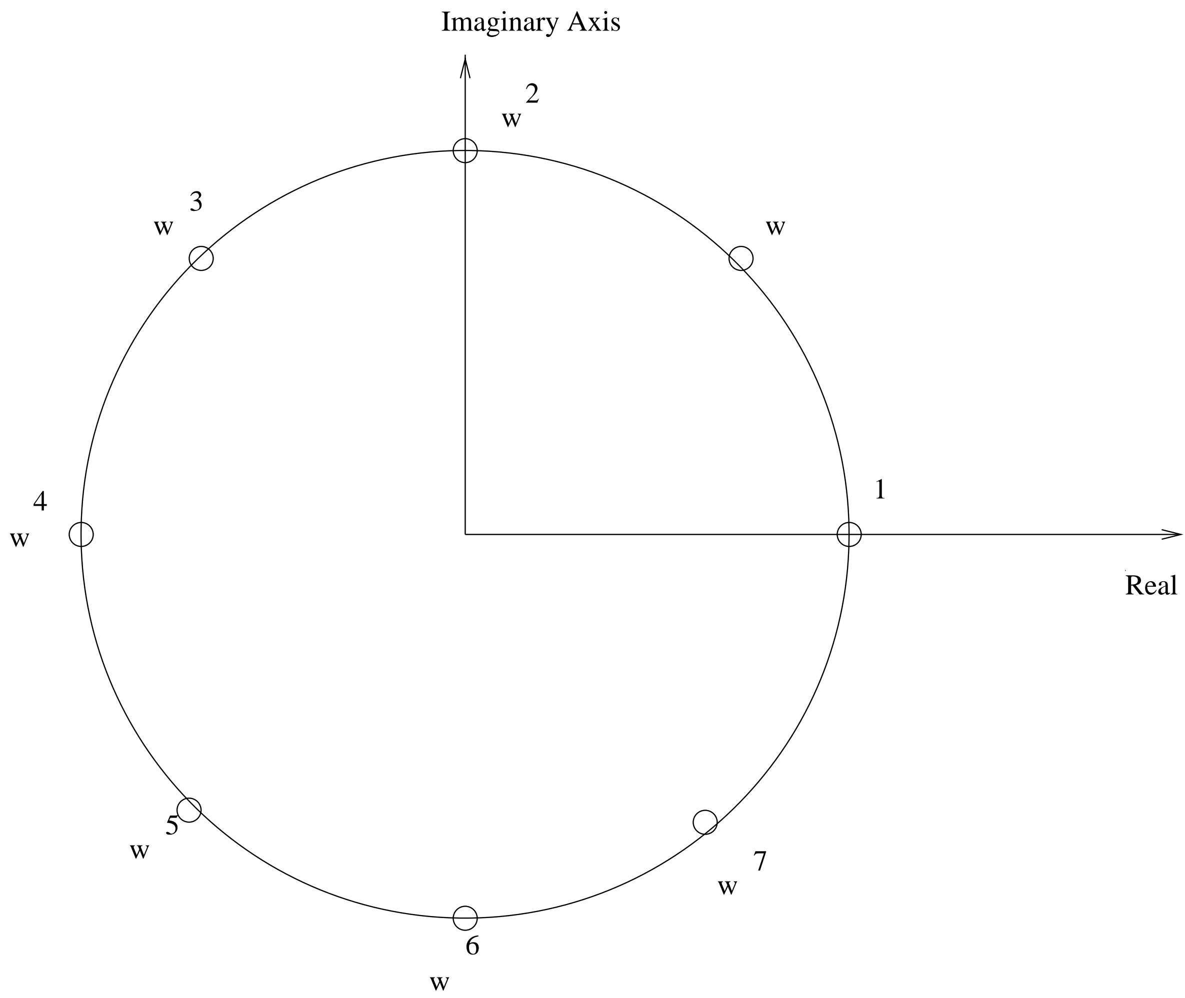


Figure 2: The eight 8th roots of unity

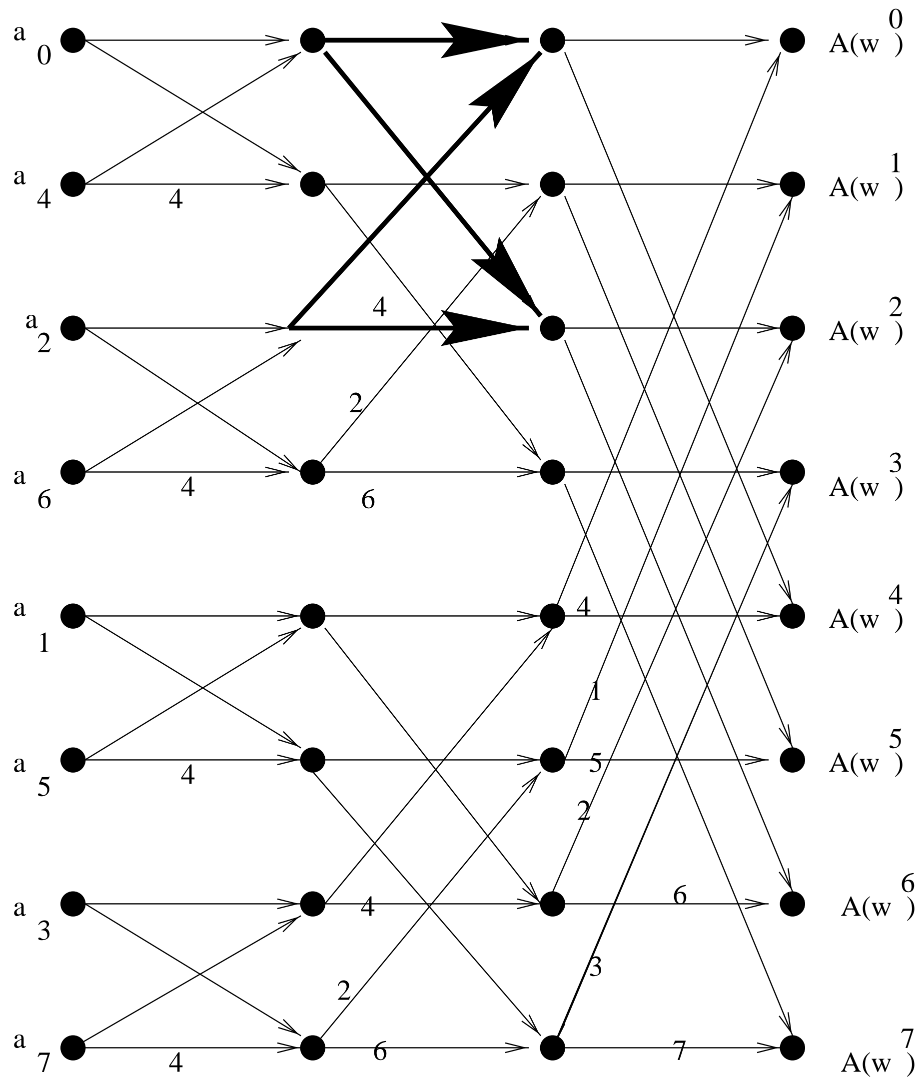


Figure 3: The FFT circuit