The Fast Fourier Transform

1 Motivation: digital signal processing

The *fast Fourier transform* (FFT) is the workhorse of digital signal processing. To understand how it is used, consider any *signal*: any quantity which is a function of time or of position. This signal might, for instance, capture a human voice by measuring air pressure over time (Figure 1.1(a)). In order to extract information from it, we need to first *digitize* it if it is analog – that is, to convert it to a discrete function $a(n), n \in \mathbf{Z}$, by sampling (Figure 1.1(b)) – and, then, to put it through a *system* which will transform it in some way. The output is called the *system response*.

In picking an appropriate transformation from the endlessly diverse space of possibilities, there are two especially desirable properties.

- *Linearity* – the response to the sum of two signals is just the sum of their individual responses. For instance, doubling a signal also doubles the system response.
- *Time invariance* – shifting the input signal by time t produces the same output, also shifted by t.

A system with these properties can be described concisely because it is completely characterized by its response to the simplest possible input signal: the *unit impulse* $\delta(n)$, consisting solely of a "jerk" at time zero (Figure 1.1(c)). To see this, first consider the close relative $\delta(n-i)$, a shifted impulse in which the jerk is moved to time i. Any signal $a(n)$ can be expressed as a linear combination of these, letting $\delta(n-i)$ pick out its behavior at time i,

$$a(n) = \sum_{i=-\infty}^{\infty} a(i)\delta(n-i).$$

By linearity, the system response to input $a(n)$ is determined by the responses to the various $\delta(n-i)$. And by time invariance, these are in turn just shifted copies of the *impulse response* $b(n)$, the response to $\delta(n)$. In other words, the output of the system on any possible input signal $a(n)$ can be read off easily from $b(n)$. It is

$$c(n) \;=\; \sum_{i=-\infty}^{\infty} a(i)b(n-i),$$

called the *convolution* of a and b. For example, a system with impulse response (Figure 1.1(d))

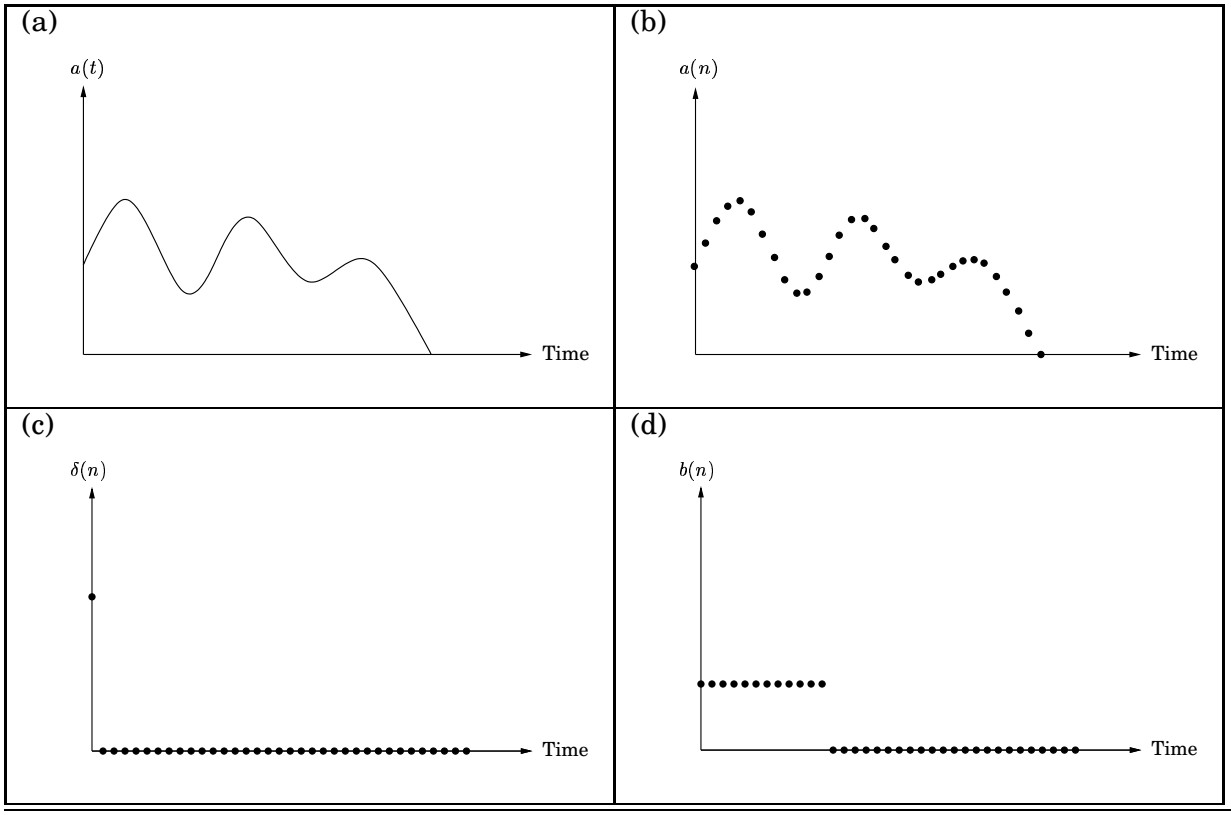
$$b(n) = \begin{cases} 1/T & \text{if } n = 0, 1, \ldots, T-1 \\ 0 & \text{otherwise} \end{cases}$$

performs a simple averaging operation, $c(n) = \frac{1}{T}(a(n) + a(n-1) + a(n-2) + \cdots + a(n-T+1))$.

Most often, $a(\cdot)$ and $b(\cdot)$ are nonzero only at a small finite set of times, say 0 to $T-1$. In such cases, their convolution $c(\cdot)$ is nonzero only from 0 to $2T-2$:

$$c(n) = \begin{cases} \sum_{i=0}^{n} a(i)b(n-i) & \text{if } 0 \le n \le T-1 \\ \sum_{i=n-T+1}^{T-1} a(i)b(n-i) & \text{if } T \le n \le 2T-2 \end{cases}$$

Figure 1.1 (a) An analog signal. (b) A digitized version, obtained by sampling at regular intervals. (c) The unit impulse. (d) An averaging filter.



This operation is such a basic primitive that it is very important to compute efficiently. Each $b(i)$ takes $O(T)$ steps, so it looks like the overall computation time must be $O(T^2)$.[1] Remarkably, the fast Fourier transform is able to do it in just $O(T \log T)$ steps. This speedup from quadratic to almost linear time has revolutionized the practicality of digital signal processing.

The first step towards a more efficient algorithm is to reinterpret the problem as one involving polynomials. If we think of the $a(i)$ as coefficients of a polynomial $\sum_i a_i x^i$, and likewise the $b(i)$, then the output signal $c(\cdot)$ is given by the coefficients of their product,

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

In other words, *convolution can be reformulated as polynomial multiplication*. We will henceforth tackle the problem in this particular guise, letting the rich structure of polynomials guide us through various twists and turns towards a solution. In fact, it will soon become apparent that this chapter is all about *switching representations*.

[1] For simplicity we are assuming here, and for the rest of this chapter, that all coefficients are real numbers, and that basic arithmetic operations on reals take unit time. In general, if the numbers involved are large, the various time bounds we obtain will need to be multiplied by $O(\log^2 B)$, where B is the number of bits of precision.

2 Polynomial multiplication

2.1 An alternative representation of polynomials

The product of two degree-d polynomials $A(x) = a_0 + a_1x + \cdots + a_dx^d$ and $B(x) = b_0 + b_1x + \cdots + b_dx^d$ has degree $2d$:

$$C(x) \;=\; A(x)\,B(x) \;=\; c_0 + c_1x + \cdots + c_{2d}x^{2d}.$$

Its k^{th} coefficient is $c_k = \sum_{j=0}^{k} a_j b_{k-j}$, and takes $O(k)$ steps to compute. Therefore the baseline scheme for polynomial multiplication is essentially identical to the one for convolution, and has $O(d^2)$ time complexity.

A radically different approach is suggested by a special property of polynomials.

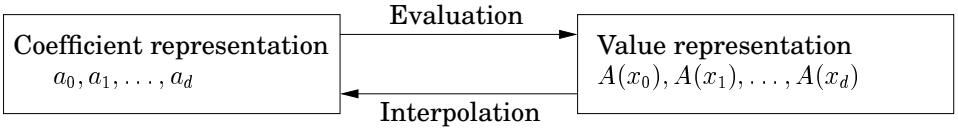
> **Fact.** A degree-d polynomial is uniquely characterized by its values at *any* $d+1$ distinct points.

We will later see why this is true, but for the time being it gives us an *alternative representation* of polynomials. Fix any distinct points $x_0, \ldots, x_d$. We can specify a degree-d polynomial $A(x)$ either by

(1) its coefficients $a_0, a_1, \ldots, a_d$; or

(2) the values $A(x_0), A(x_1), \ldots, A(x_d)$.

Going from the first representation to the second is merely a matter of *evaluating* the polynomial at the chosen points. Going in the reverse direction is called *interpolation*.



This alternative representation gives us another route to $C(x)$: since it has degree $\leq 2d$, we only need its value at any $2d+1$ points, and its value at any given point is easy enough to compute, simply $A(x)$ times $B(x)$. The resulting algorithm is shown in Figure 2.1.

The correctness of this high-level approach is a direct consequence of the equivalence of the two polynomial representations. What is not so immediate is its efficiency. Selection and multiplication are no trouble at all, just linear time. But how about evaluation? We know that evaluating a polynomial of degree $d \leq n$ at one point takes $O(n)$ steps, and so we would expect n points to take $O(n^2)$ steps. The FFT does it in $O(n \log n)$ time, for a particularly clever choice of $x_0, \ldots, x_{n-1}$ in which the computations required by the individual points overlap with one another and can be shared.

2.2 Evaluation by divide-and-conquer

Here's an idea for how to pick the points $x_0, x_1, \ldots, x_{n-1}$ at which to evaluate a polynomial $A(x)$ of degree $\leq n-1$. If we choose them to be positive-negative pairs, that is,

$$x_0, x_1, \ldots, x_{n-1} \;\equiv\; \pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

Figure 2.1 Polynomial multiplication

function PolyMult(A, B)
Input: Coefficients of two polynomials $A(x), B(x)$, of degree d
Output: Their product $C = A \cdot B$

Selection
Pick any points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d+1$
Evaluation
Compute $A(x_0), A(x_1), \ldots, A(x_{n-1})$
and $B(x_0), B(x_1), \ldots, B(x_{n-1})$
Multiplication
Compute $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n-1$
Interpolation
Interpolate to recover $C(x) = c_0 + c_1 x + \cdots + c_{2d}x^{2d}$

then the computations required for $A(x_i)$ and $A(-x_i)$ have a lot in common, because the even powers of x_i coincide with those of $-x_i$.

To investigate this, we need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 \;=\; (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4).$$

Notice that the terms in parentheses are polynomials in x^2. More generally,

$$A(x) \;=\; A_e(x^2) + xA_o(x^2),$$

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$ (assume for convenience that n is even). Given *paired* points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled towards computing $A(-x_i)$:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2). \end{aligned}$$

In other words, we can reduce the evaluation of polynomial $A(x)$, which has degree $\leq n$, at n *paired* points $\pm x_0, \ldots, \pm x_{n/2-1}$ to the evaluation of A_e and A_o, which have degree $\leq n/2 - 1$, at the $n/2$ points $x_0^2, \ldots, x_{n/2-1}^2$. In this way, the original problem of size n can be recast as two subproblems of size $n/2$, followed by some linear time arithmetic. If we could recurse, we would get a divide-and-conquer procedure with running time

$$T(n) \;=\; 2T(n/2) + O(n),$$

which is $O(n \log n)$, exactly what we want.

There are $\log n$ levels to this recursion, and the points at any given level are the *squares* of the points at the previous level. The divide-and-conquer works correctly under one condition.

Condition. The points are *paired* at *every* level of the recursion, that is,

1. the initial points $x_0, \ldots, x_{n-1}$ are paired,
2. their squares (of which there are $n/2$) are paired,
3. their fourth powers (of which there are $n/4$) are paired, and so on.

The top level, with n points, is easy enough to manage. But at the second level we already seem to be in trouble – if the numbers are squares, how can they be positive-*negative* pairs?

The answer is to use *complex* numbers, as becomes apparent when we reverse-engineer a solution. At the very bottom of the recursion, we are left with a single point. This point might as well be 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$. The next level up then has $\pm\sqrt{\pm 1} = \pm 1, \pm i$, where i is the imaginary unit. Continuing in this way, we can show that our condition is exactly satisfied when the initial numbers are taken to be the n^{th} *complex roots of unity*, that is, the n complex solutions to the equation $z^n = 1$. We now turn to these in more detail.

2.3 The complex roots of unity

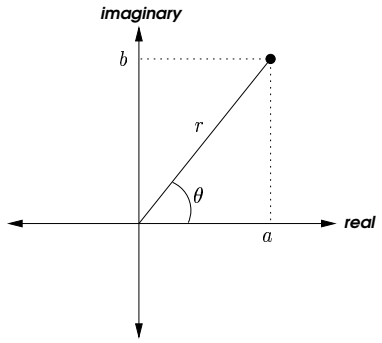
Figure 2.2 is a quick pictorial review of complex numbers. Assume n is even; the following observations from the diagram will be important to us.

1. The complex roots of unity are the n numbers whose angles are multiples of $2\pi/n$.
2. The squares of the n^{th} roots are the $(n/2)^{nd}$ roots.
3. These roots are paired: the numbers with angles $2\pi k/n$ and $2\pi(k + n/2)/n = 2\pi k/n + \pi$ are negatives of each other.

Observation (2) says that if we start our divide-and-conquer procedure with the n^{th} complex roots of unity, and n is a power of two, then at the k^{th} level of recursion the points being evaluated will be $(n/2^k)^{th}$ roots of unity. Observation (3) then confirms the recursive pairing condition we need. The resulting algorithm is the fast Fourier transform, shown in Figure 2.3.

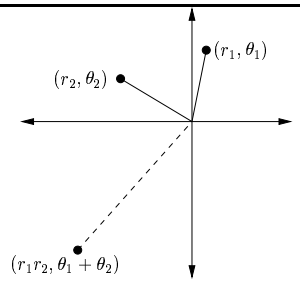
Why is n a power of two? The correctness of our polynomial multiplication scheme is assured for any $n \geq 2d+1$, but for the sake of efficiency n should not be too large. Fortunately, we can always find a power of two between $2d + 1$ and $4d$ (can you see why?). This modest increase in the input size is a small cost for the tremendous convenience it brings.

Figure 2.2 A review of complex numbers.



The complex plane

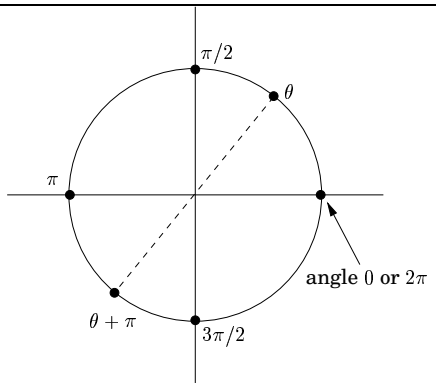
1. $z = a + bi$ is plotted at position (a, b).
2. Another format: $z = re^{i\theta}$, where
   - *length* $r = \sqrt{a^2 + b^2}$,
   - *angle* $\theta \in [0, 2\pi)$: $\cos\theta = a/r, \sin\theta = b/r$
   - $e^{i\theta}$ is the same as $\cos\theta + i\sin\theta$
   - θ can always be reduced modulo 2π.
3. Polar coordinates: (r, θ).



Multiplying complex numbers

The product of $r_1 e^{i\theta_1}$ and $r_2 e^{i\theta_2}$ is $r_1 r_2 e^{i(\theta_1+\theta_2)}$:

multiply the lengths and add the angles.

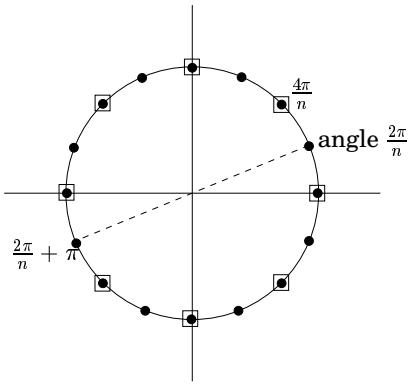


The unit circle

1. Complex numbers with length 1.
2. Specified completely by their angles, e.g.

Number	1	i	-1	$-i$
Angle	0	$\pi/2$	π	$3\pi/2$

3. Multiply numbers $\longleftrightarrow$ add their angles
4. Number z has angle θ $\longleftrightarrow$ $-z$ has angle $\theta + \pi$



The n^{th} complex roots of unity

(assume n is even)

1. Shown here for $n = 16$.
2. Their angles are multiples of $2\pi/n$.
   Raise to the n^{th} power $\Rightarrow$ multiply angle by $n \Rightarrow$ result has angle zero.
3. Square a number $\longleftrightarrow$ double its angle
4. Squares are the $(n/2)^{nd}$ roots of unity, shown here with boxes around them.

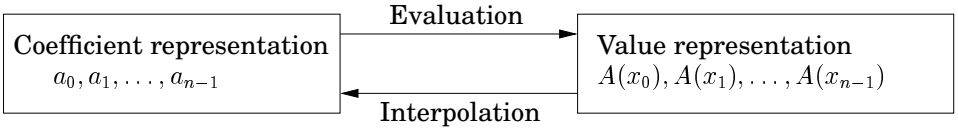
Figure 2.3 The fast Fourier transform (polynomial formulation)

```
function FFT(A, ω)
Input:  Coefficient representation of a polynomial A(x)
        of degree ≤ n − 1, where n is a power of two
        ω, an n^th root of unity
Output:  A(ω^0), ..., A(ω^{n−1})

if ω = 1 then return A(1)
rewrite A(x) = A_e(x^2) + x A_o(x^2)
FFT(A_e, ω^2) evaluates A_e at even powers of ω
FFT(A_o, ω^2), likewise for A_o
for j = 0 to n − 1:
   compute A(ω^j) = A_e(ω^{2j}) + ω^j A_o(ω^{2j})

return A(ω^0), ..., A(ω^{n−1})
```

2.4 A matrix reformulation

We are well on our way towards efficiently realizing the grand scheme of Figure 2.1. The last piece of the puzzle, the interpolation step, will be solved in the most simple and elegant way we could possibly have hoped for – using the same FFT algorithm, but called with ω^{-1} in place of ω! This might seem like a miraculous coincidence, but it will make a lot more sense when we recast our polynomial operations in the language of linear algebra.

Our two representations for a polynomial $A(x)$ of degree $\leq n-1$ are both vectors of n numbers,



and are related by the following equation.

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ & \vdots & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

Call the matrix in the middle M. Its highly specialized format – a *Vandermonde* matrix – gives it many interesting properties, of which the following is particularly relevant to us.

If $x_0, \ldots, x_{n-1}$ are distinct numbers then M is invertible.

The existence of M^{-1} allows us to invert the matrix equation above, so as to express coefficients in terms of values. In brief,

Evaluation is multiplication by M, while interpolation is multiplication by M^{-1}.

This reformulation of our polynomial operations reveals their essential nature more clearly. Among other things, it finally justifies an assumption we have been making throughout, that $A(x)$ is uniquely characterized by its values at any n points – in fact, we now have an explicit formula which will give us $A(x)$ in this situation. Vandermonde matrices also have the distinction of being quicker to invert than more general matrices, in $O(n^2)$ time instead of $O(n^3)$. However, we need to do interpolation a lot faster than this, so once again we turn to our special choice of points – the complex roots of unity.

2.5 Interpolation resolved

In linear algebra terms, the FFT multiplies an arbitrary n-dimensional vector (which we were calling the coordinate representation) by the $n \times n$ matrix

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & \vdots & & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & \vdots & & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{array}{ll} & \\ \leftarrow & \text{row for } \omega^0 = 1 \\ \leftarrow & \omega \\ \leftarrow & \omega^2 \\ & \vdots \\ \leftarrow & \omega^j \\ & \vdots \\ \leftarrow & \omega^{n-1} \end{array}$$

where ω is any complex n^{th} root of unity, and n is a power of two.

For simplicity, pick ω to be $e^{2\pi i/n}$, the root with angle $2\pi/n$, and as long as we are dealing with just one value of n, drop the subscript from M_n. It turns out that in addition to being a Vandermonde matrix, $M(\omega)$ is also (up to a scaling factor) *unitary*, which we will define later but for the time being implies that its inverse is easy to write down explicitly.

Inversion formula. $M(\omega)^{-1} = \frac{1}{n} M(\omega^{-1})$.

Therefore interpolation is the same as multiplication by $\frac{1}{n}M(\omega^{-1})$, and, since ω^{-1} is also an n^{th} root of unity, can be handled by a single FFT! Our $O(n \log n)$ polynomial multiplication algorithm (Figure 2.1) is now fully specified.

In terms of justification, however, there remains one loose end: the inversion formula. We need to verify that $M(\omega) \cdot M(\omega^{-1})$ is n times the identity matrix. Let's do this entry by entry. Position (j, k) of $M(\omega) \cdot M(\omega^{-1})$ is the dot product of the j^{th} row of $M(\omega)$ with the k^{th} column of $M(\omega^{-1})$. These are, respectively,

$$\begin{bmatrix} 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \end{bmatrix} \text{ and } \begin{bmatrix} 1 & \omega^{-k} & \omega^{-2k} & \cdots & \omega^{-(n-1)k} \end{bmatrix}'$$

(if you number rows and columns starting from zero), and their dot product is

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \cdots + \omega^{(n-1)(j-k)}.$$

If $j = k$ then all these numbers are one, and they add up to n, exactly the diagonal elements we want. If $j \neq k$, the numbers are arranged symmetrically around the unit circle and thus cancel each other out to give a sum of zero. In short,

$$1 + \omega^l + \omega^{2l} + \cdots + \omega^{(n-1)l} = \begin{cases} n & \text{if } l \text{ is a multiple of } n \\ 0 & \text{otherwise.} \end{cases}$$

Figure 3.1 A fast convolution algorithm.

```
function Convolution(a, b)
Input:  Signals a = (a_0, ..., a_{T-1}) and b = (b_0, ..., b_{T-1})
Output:  Their convolution c

    Let n be the smallest power of two above 2T - 2
    Pad a, b with zeros to get them to length n
Selection
    Let ω be e^{2πi/n}
Evaluation
    ã  ←—  FFT(a, ω)
    b̃  ←—  FFT(b, ω)
Multiplication
    c̃_j  ←—  ã_j · b̃_j  for all  j = 0, ..., n - 1
Interpolation
    c  ←—  (1/n) · FFT(c̃, ω^{-1})
```

A formal proof of the second case of this equation goes like this: since l is not a multiple of n, the angle $2\pi l/n$ is not a multiple of 2π, so $\omega^l \neq 1$. By the usual formula for the sum of a geometric series,

$$1 + \omega^l + \cdots + \omega^{(n-1)l} \;=\; \frac{\omega^{nl} - 1}{\omega^l - 1} \;=\; 0 \qquad (\text{note } \omega^{nl} = (\omega^n)^l = 1).$$

Another way to prove this is to argue that for some even k, these $1, \omega^l, \ldots, \omega^{(n-1)l}$ are the k^{th} roots of unity, repeated n/k times. For instance, when $l = n/2$ they are alternately $+1$ and -1. Therefore, the numbers are paired and add to zero.

3 The definitive algorithm

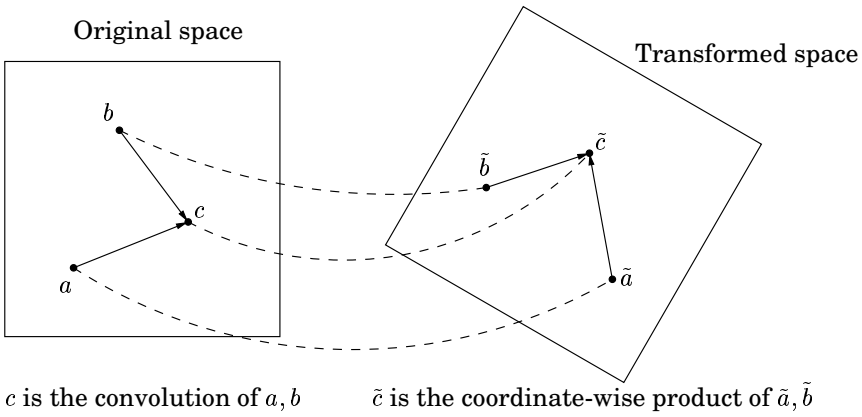
3.1 Putting polynomials aside

The scenic journey of the previous section culminated in an $O(T \log T)$ algorithm for computing the convolution of two length-T signals. Much of the inspiration came from casting the problem in terms of polynomials. But now their job is done, and we can put them aside and rewrite the algorithm without them. The result is shown in Figure 3.1.

Let's step back and solidify our understanding of this algorithm. The overall idea is that convolution is just a disguised form of a simpler, linear-time operation: coordinate-wise multiplication. The real work is done in switching between disguises, and this is the province of the FFT. Figure 3.2 shows the correspondences between the two spaces involved – the default space, which contains the original vectors (signals) a, b, and the transformed space, with their counterparts $\tilde{a}, \tilde{b}$.

The transformation is extremely simple. It is just multiplication by the matrix $M_n(\omega)$ which we defined earlier, and which we will occasionally call M for convenience. What features give M its special properties?

Figure 3.2 Switching between two vector spaces.



> **Feature 1.** M is a *Vandermonde* matrix – each row starts with 1 and has ascending powers of some number.

By simple algebra (which you should check), this implies exactly the claim of Figure 3.2, that

> M times the convolution of a, b is the coordinate-wise product of Ma with Mb.

Can you also show the converse, that any matrix with this property must be Vandermonde?

> **Feature 2.** M is *unitary* (up to a scaling factor) – in other words, it is invertible and its inverse is obtained by replacing each entry $e^{i\theta}$ of M' by $e^{-i\theta}$.

The most concrete consequence of this feature is that M^{-1} looks a lot like M itself, so much so that both transformations can be carried out using the same procedure. But it also tells us something about the geometry of the two spaces we are managing. A unitary transformation is a rigid rotation. Therefore, multiplication by M is (not just figuratively but also technically) a *change of basis*. We rotate the original vectors into the new *Fourier basis*, multiply them coordinate-wise, and then rotate the result back.

> **Feature 3.** The powers of ω are recursively paired.

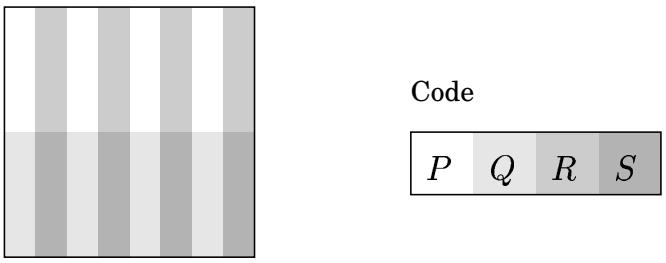
By making $M_n(\omega)$ amenable to divide-and-conquer, this serves as our source of efficiency. Let's take a closer look at how this works in matrix format.

3.2 Divide-and-conquer revisited

For reference, here again is $M_n(\omega)$.

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & \vdots & & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & \vdots & & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

Its n^2 entries are all n^{th} roots of unity and therefore consist of only n distinct values. Moreover, they are arranged in a highly regular pattern, which is easiest to appreciate when the matrix is divided into four $n/2 \times n/2$ regions.



Region P consists of the top halves of all the even-numbered columns (starting count at zero), while Q has the bottom halves of these columns. Regions R and S are the corresponding halves of the odd-numbered columns.

The four submatrices are intimately related.

1. $P = Q = M_{n/2}(\omega^2)$.
2. The j^{th} row of R is ω^j times the j^{th} row of P.
3. The j^{th} row of S is $\omega^{j+n/2} = -\omega^j$ times that of P.

For the first of these, notice that the j^{th} row of matrix P contains all *even* powers of ω^j:

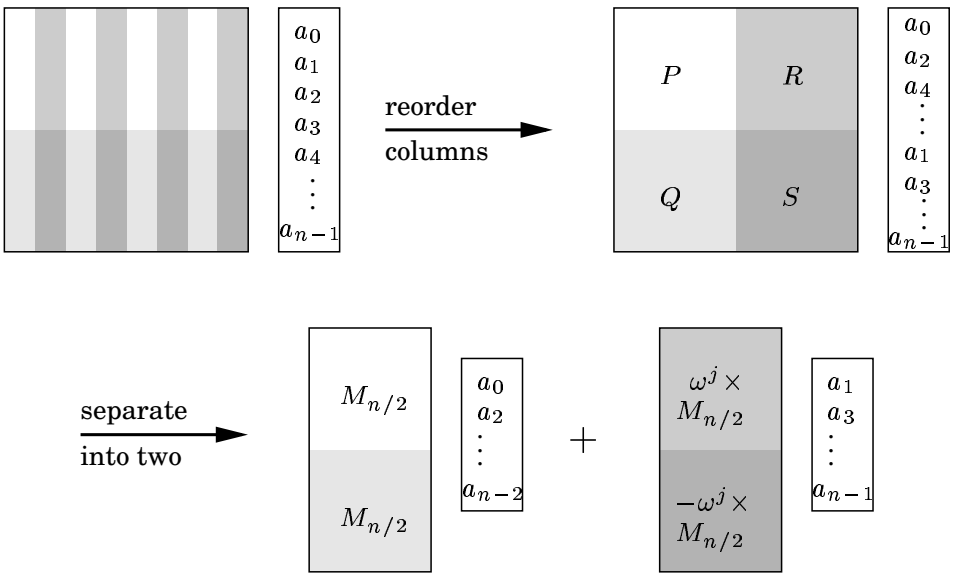
$$\begin{bmatrix} 1 & \omega^{2j} & \omega^{4j} & \cdots & \omega^{j(n-2)} \end{bmatrix}$$

while the j^{th} row of Q contains even powers of $\omega^{j+n/2}$:

$$\begin{bmatrix} 1 & \omega^{n+2j} & \omega^{2n+4j} & \cdots & \omega^{n(n-2)/2+j(n-2)} \end{bmatrix}.$$

Since $\omega^n = 1$, these two matrices are identical. The other claims can be checked similarly.

By rewriting $M_n(\omega)$ in terms of P, Q, R, S, and then in terms of $M_{n/2}(\omega^2)$, the multiplication of a vector $a = (a_0, \ldots, a_{n-1})'$ by $M_n(\omega)$ can be decomposed into subtasks:



This leads to the definitive FFT algorithm of Figure 3.3.

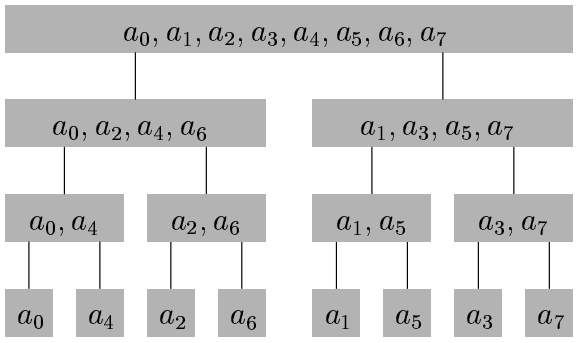
Figure 3.3 The fast Fourier transform

```
function FFT(a, ω)
Input:  A vector a = (a_0, a_1, ..., a_{n-1}), for n a power of two
        A primitive n^th root of unity, ω
Output:  M_n(ω) a

if ω = 1 then return a
(s_0, s_1, ..., s_{n/2-1}) ← FFT((a_0, a_2, ..., a_{n-2}), ω^2)
(s'_0, s'_1, ..., s'_{n/2-1}) ← FFT((a_1, a_3, ..., a_{n-1}), ω^2)
for j = 1 to n/2 - 1:
    r_j = s_j + ω^j s'_j
    r_{j+n/2} = s_j - ω^j s'_j
return (r_0, r_1, ..., r_{n-1})
```

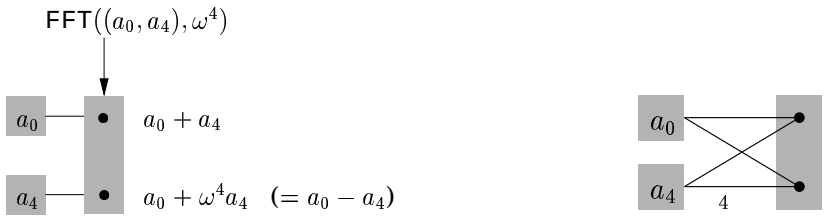
4 The fast Fourier transform unraveled

Through all our discussions so far, the fast Fourier transform has remained tightly cocooned within the strictures of a divide-and-conquer formalism. To fully expose its structure, we need to unbind it, to unravel the recursion.

The FFT decomposes an input vector into its even- and odd-numbered components. On an input of length eight, this results in the following pattern of recursion.

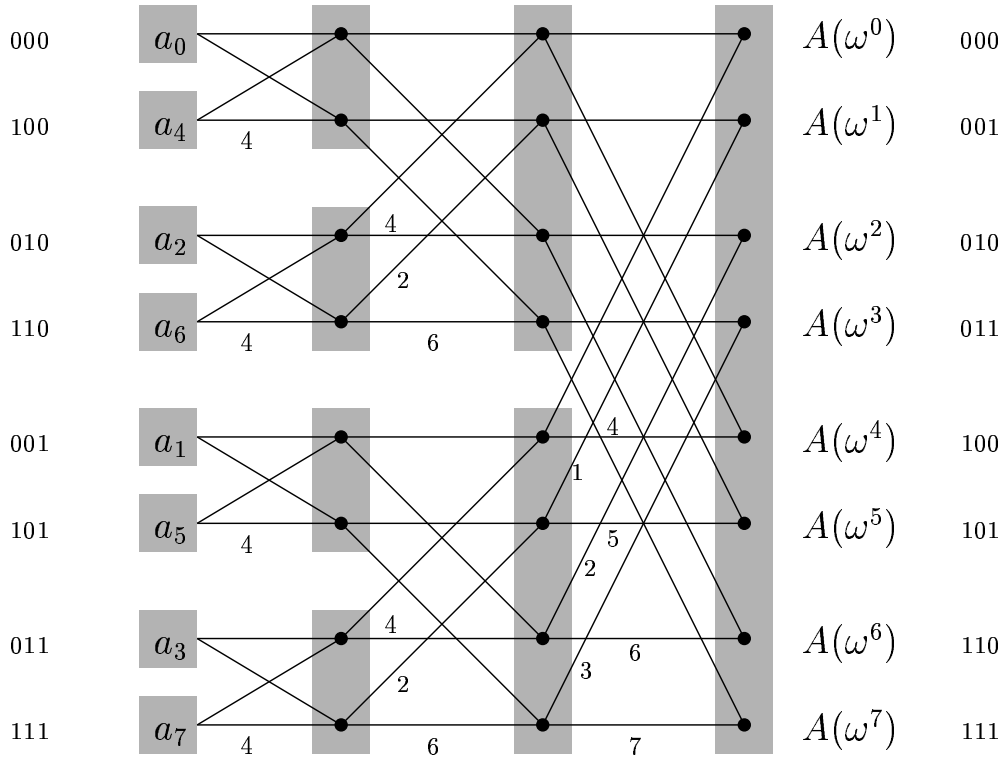


The intermediate computations are extremely simple. What happens with (a_0, a_4), for instance, is shown below on the left.



This kind of computation is called a *butterfly*. The FFT is made up entirely of these, so it will be convenient to instead use the shorthand depicted above on the right. The edges are wires carrying complex numbers from left to right. A weight of j means "multiply the number on this wire by ω^j". And the numbers coming into a node get added up.

Figure 4.1 The fast Fourier transform circuit.



Expressed in terms of these circuit elements, the FFT computation on a length-eight vector is shown in Figure 4.1. Notice the following.

1. For n inputs there are $\log_2 n$ levels, each with n nodes.

2. There is a unique path between each input a_j and each output $A(\omega^k)$.

This path is most easily described using the binary representations of j and k (shown in the figure for convenience). There are two edges out of each node, one going up (the 0-edge) and one going down (the 1-edge). To get to $A(\omega^k)$ from any input node, simply follow the edges specified in the bit representation of k, starting from the leftmost bit. (Can you similarly specify the path in the reverse direction?)

3. On the path between a_j and $A(\omega^k)$, the labels add up to $jk \bmod 8$.

Since $\omega^8 = 1$, this means that the contribution of input a_j to output $A(\omega^k)$ is $a_j\omega^{jk}$, and therefore the circuit correctly the values of polynomial $A(x)$.

4. And finally, the circuit is a natural for parallel computation.