2.6 The fast Fourier transform

We have so far seen how divide-and-conquer gives fast algorithms for multiplying integers and matrices; our next target is *polynomials*. The product of two degree-d polynomials is a polynomial of degree $2d$, for example:

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4.$$

More generally, if $A(x) = a_0 + a_1x + \cdots + a_dx^d$ and $B(x) = b_0 + b_1x + \cdots + b_dx^d$, their product $C(x) = A(x) \cdot B(x) = c_0 + c_1x + \cdots + c_{2d}x^{2d}$ has coefficients

$$c_k \;=\; a_0b_k + a_1b_{k-1} + \cdots + a_kb_0 \;=\; \sum_{i=0}^{k} a_ib_{k-i}$$

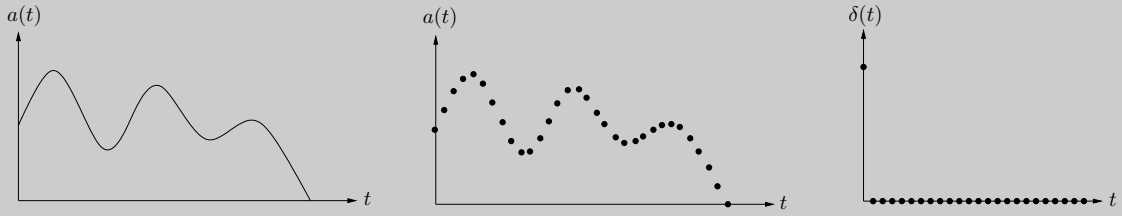
(for $i > d$, take a_i and b_i to be zero). Computing c_k from this formula takes $O(k)$ steps, and finding all $2d + 1$ coefficients would therefore seem to require $\Theta(d^2)$ time. *Can we possibly multiply polynomials faster than this?*

The solution we will develop, the fast Fourier transform, has revolutionized—indeed, defined—the field of signal processing (see the following box). Because of its huge importance, and its wealth of insights from different fields of study, we will approach it a little more leisurely than usual. The reader who wants just the core algorithm can skip directly to Section 2.6.4.

Why multiply polynomials?

For one thing, it turns out that the fastest algorithms we have for multiplying integers rely heavily on polynomial multiplication; after all, polynomials and binary integers are quite similar—just replace the variable x by the base 2, and watch out for carries. But perhaps more importantly, multiplying polynomials is crucial for *signal processing*.

A *signal* is any quantity that is a function of time (as in Figure (a)) or of position. It might, for instance, capture a human voice by measuring fluctuations in air pressure close to the speaker's mouth, or alternatively, the pattern of stars in the night sky, by measuring brightness as a function of angle.



(a) (b) (c)

In order to extract information from a signal, we need to first *digitize* it by sampling (Figure (b))—and, then, to put it through a *system* that will transform it in some way. The output is called the *response* of the system:

$$\text{signal} \longrightarrow \boxed{\text{SYSTEM}} \longrightarrow \text{response}$$

An important class of systems are those that are *linear*—the response to the sum of two signals is just the sum of their individual responses—and *time invariant*—shifting the input signal by time t produces the same output, also shifted by t. Any system with these properties is completely characterized by its response to the simplest possible input signal: the *unit impulse* $\delta(t)$, consisting solely of a "jerk" at $t = 0$ (Figure (c)). To see this, first consider the close relative $\delta(t - i)$, a shifted impulse in which the jerk occurs at time i. Any signal $a(t)$ can be expressed as a linear combination of these, letting $\delta(t - i)$ pick out its behavior at time i,

$$a(t) = \sum_{i=0}^{T-1} a(i)\delta(t - i)$$

(if the signal consists of T samples). By linearity, the system response to input $a(t)$ is determined by the responses to the various $\delta(t-i)$. And by time invariance, these are in turn just shifted copies of the *impulse response* $b(t)$, the response to $\delta(t)$.

In other words, the output of the system at time k is

$$c(k) \;=\; \sum_{i=0}^{k} a(i)b(k - i),$$

exactly the formula for polynomial multiplication!

2.6.1 An alternative representation of polynomials

To arrive at a fast algorithm for polynomial multiplication we take inspiration from an important property of polynomials.

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

A familiar instance of this is that "any two points determine a line." We will later see why the more general statement is true (page 72), 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 $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ by either one of the following:

1. Its coefficients $a_0, a_1, \ldots, a_d$
2. The values $A(x_0), A(x_1), \ldots, A(x_d)$

Of these two representations, the second is the more attractive for polynomial multiplication. Since the product $C(x)$ has degree $2d$, it is completely determined by its value at any $2d+1$ points. And its value at any given point z is easy enough to figure out, just $A(z)$ times $B(z)$. Thus *polynomial multiplication takes linear time in the value representation.*

The problem is that we expect the input polynomials, and also their product, to be specified by coefficients. So we need to first translate from coefficients to values—which is just a matter of *evaluating* the polynomial at the chosen points—then multiply in the value representation, and finally translate back to coefficients, a process called *interpolation*.

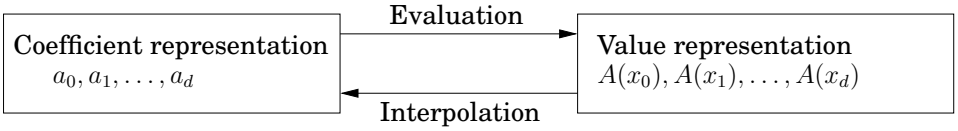


Figure 2.5 presents the resulting algorithm.

Figure 2.5 Polynomial multiplication

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

Selection
Pick some 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
Recover $C(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$

The equivalence of the two polynomial representations makes it clear that this high-level approach is correct, but how efficient is it? Certainly the selection step and the n multiplications are no trouble at all, just linear time.[3] But (leaving aside interpolation, about which we know even less) how about evaluation? Evaluating a polynomial of degree $d \leq n$ at a single point takes $O(n)$ steps (Exercise 2.29), and so the baseline for n points is $\Theta(n^2)$. We'll now see that the fast Fourier transform (FFT) does it in just $O(n \log n)$ 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.

2.6.2 Evaluation by divide-and-conquer

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

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

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, 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

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

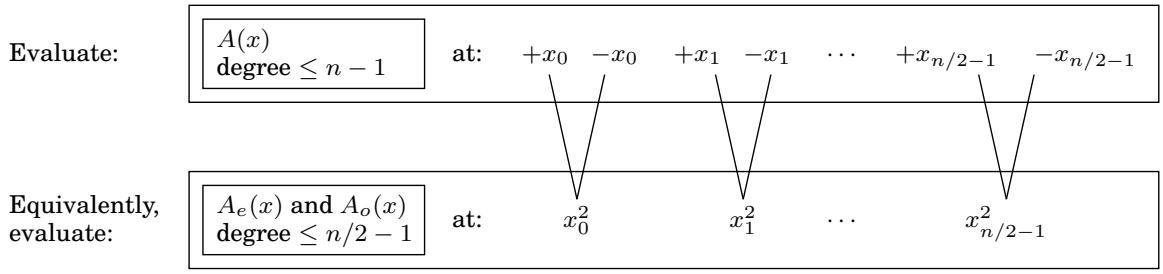
Notice that the terms in parentheses are polynomials in x^2. More generally,

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

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 toward 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, evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ (which each have half the degree of $A(x)$) at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.



[3] In a typical setting for polynomial multiplication, the coefficients of the polynomials are real numbers and, moreover, are small enough that basic arithmetic operations (adding and multiplying) take unit time. We will assume this to be the case without any great loss of generality; in particular, the time bounds we obtain are easily adjustable to situations with larger numbers.

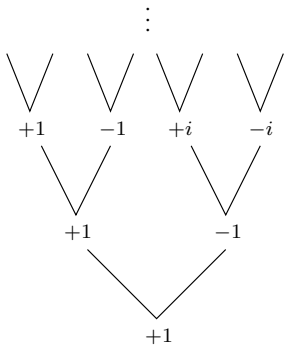
The original problem of size n is in this way recast as two subproblems of size $n/2$, followed by some linear-time arithmetic. If we could recurse, we would get a divide-and-conquer procedure with running time

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

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

But we have a problem: The plus-minus trick only works at the top level of the recursion. To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be *themselves* plus-minus pairs. But how can a square be negative? The task seems impossible! *Unless, of course, we use complex numbers*.

Fine, but which complex numbers? To figure this out, let us "reverse engineer" the process. At the very bottom of the recursion, we have 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 $\pm\sqrt{+1} = \pm 1$ as well as the *complex* numbers $\pm\sqrt{-1} = \pm i$, where i is the imaginary unit. By continuing in this manner, we eventually reach the initial set of n points. Perhaps you have already guessed what they are: the *complex nth roots of unity*, that is, the n complex solutions to the equation $z^n = 1$.

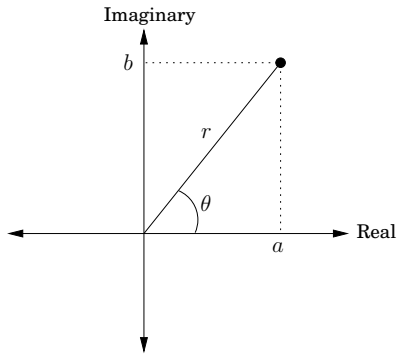
Figure 2.6 is a pictorial review of some basic facts about complex numbers. The third panel of this figure introduces the nth roots of unity: the complex numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2\pi i/n}$. If n is even,

1. The nth roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$.

2. Squaring them produces the $(n/2)$nd roots of unity.

Therefore, if we start with these numbers for some n that is a power of 2, then at successive levels of recursion we will have the $(n/2^k)$th roots of unity, for $k = 0, 1, 2, 3, \ldots$. All these sets of numbers are plus-minus paired, and so our divide-and-conquer, as shown in the last panel, works perfectly. The resulting algorithm is the fast Fourier transform (Figure 2.7).

Figure 2.6 The complex roots of unity are ideal for our divide-and-conquer scheme.



The complex plane

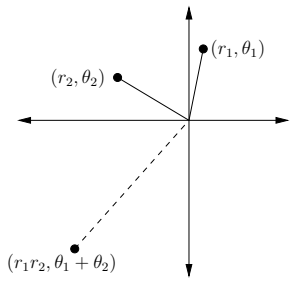
$z = a + bi$ is plotted at position (a, b).

Polar coordinates: rewrite as $z = r(\cos\theta + i\sin\theta) = re^{i\theta}$, denoted (r, θ).

- *length* $r = \sqrt{a^2 + b^2}$.
- *angle* $\theta \in [0, 2\pi)$: $\cos\theta = a/r, \sin\theta = b/r$.
- θ can always be reduced modulo 2π.

Examples:

Number	-1	i	$5 + 5i$
Polar coords	$(1, \pi)$	$(1, \pi/2)$	$(5\sqrt{2}, \pi/4)$

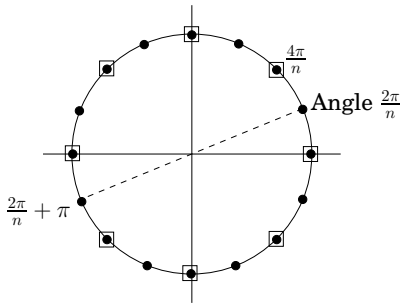


Multiplying is easy in polar coordinates

Multiply the lengths and add the angles:
$(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.

For any $z = (r, \theta)$,

- $-z = (r, \theta + \pi)$ since $-1 = (1, \pi)$.
- If z is on the *unit circle* (i.e., $r = 1$), then $z^n = (1, n\theta)$.



The nth complex roots of unity

Solutions to the equation $z^n = 1$.

By the multiplication rule: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$ (shown here for $n = 16$).

For even n:

- These numbers are *plus-minus paired*: $-(1, \theta) = (1, \theta + \pi)$.
- Their squares are the $(n/2)$nd roots of unity, shown here with boxes around them.

Divide-and-conquer step

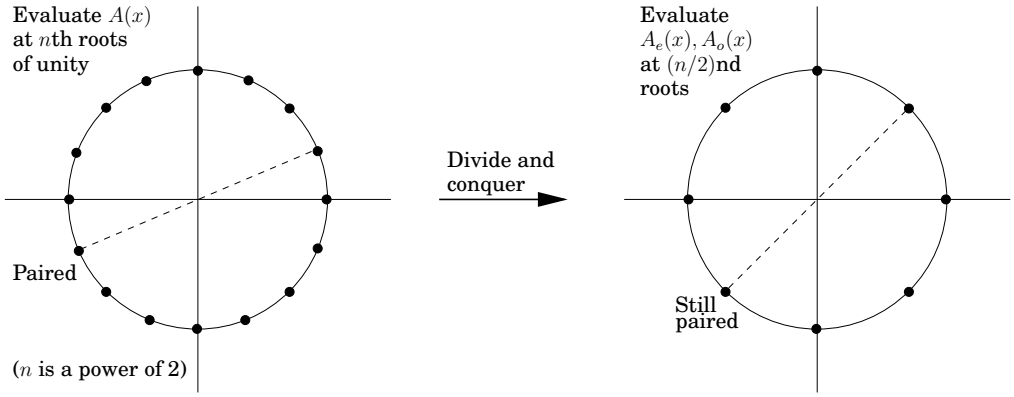


Figure 2.7 The fast Fourier transform (polynomial formulation)

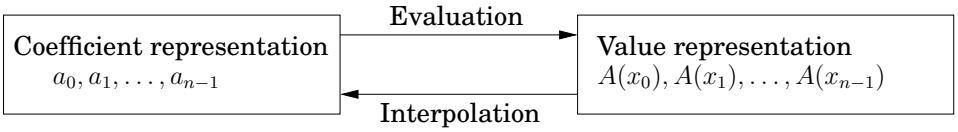
```
function FFT(A, ω)
Input:  Coefficient representation of a polynomial A(x)
        of degree ≤ n−1, where n is a power of 2
        ω, an nth root of unity
Output:  Value representation A(ω^0), ..., A(ω^(n−1))

if ω = 1:  return A(1)
express A(x) in the form A_e(x^2) + xA_o(x^2)
call FFT(A_e, ω^2) to evaluate A_e at even powers of ω
call FFT(A_o, ω^2) to evaluate A_o at even powers of ω
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))
```

2.6.3 Interpolation

Let's take stock of where we are. We first developed a high-level scheme for multiplying polynomials (Figure 2.5), based on the observation that polynomials can be represented in two ways, in terms of their *coefficients* or in terms of their *values* at a selected set of points.



The value representation makes it trivial to multiply polynomials, but we cannot ignore the coefficient representation since it is the form in which the input and output of our overall algorithm are specified.

So we designed the FFT, a way to move from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex nth roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

$$\langle \text{values} \rangle \;=\; \text{FFT}(\langle \text{coefficients} \rangle, \omega).$$

The last remaining piece of the puzzle is the inverse operation, interpolation. It will turn out, amazingly, that

$$\langle \text{coefficients} \rangle \;=\; \frac{1}{n}\,\text{FFT}(\langle \text{values} \rangle, \omega^{-1}).$$

Interpolation is thus 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. Meanwhile, our $O(n \log n)$ polynomial multiplication algorithm (Figure 2.5) is now fully specified.

A matrix reformulation

To get a clearer view of interpolation, let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$. They are both vectors of n numbers, and one is a linear transformation of the other:

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

Call the matrix in the middle M. Its specialized format—a *Vandermonde* matrix—gives it many remarkable 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 preceding matrix equation 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 that will give us the coefficients of $A(x)$ 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, using this for interpolation would still not be fast enough for us, so once again we turn to our special choice of points—the complex roots of unity.

Interpolation resolved

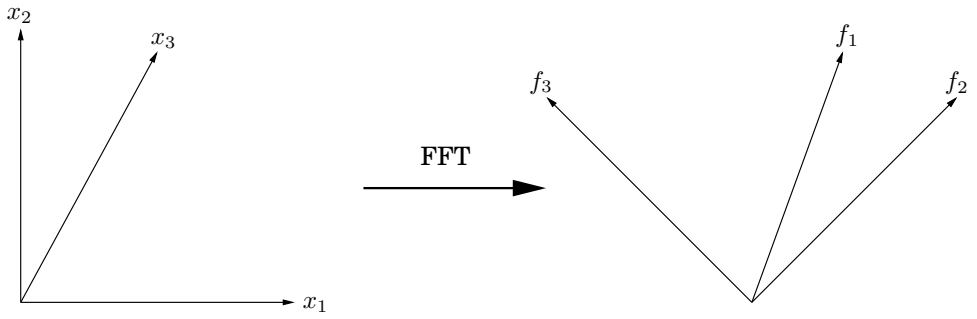
In linear algebra terms, the FFT multiplies an arbitrary n-dimensional vector—which we have been calling the *coefficient representation*—by the $n \times n$ matrix

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

where ω is a complex nth root of unity, and n is a power of 2. Notice how simple this matrix is to describe: its (j,k)th entry (starting row- and column-count at zero) is ω^{jk}.

Multiplication by $M = M_n(\omega)$ maps the kth coordinate axis (the vector with all zeros except for a 1 at position k) onto the kth column of M. Now here's the crucial observation, which we'll prove shortly: *the columns of M are orthogonal (at right angles) to each other*. Therefore they can be thought of as the axes of an alternative coordinate system, which is often called

Figure 2.8 The FFT takes points in the standard coordinate system, whose axes are shown here as x_1, x_2, x_3, and rotates them into the Fourier basis, whose axes are the columns of $M_n(\omega)$, shown here as f_1, f_2, f_3. For instance, points in direction x_1 get mapped into direction f_1.



the *Fourier basis*. The effect of multiplying a vector by M is to rotate it from the standard basis, with the usual set of axes, into the Fourier basis, which is defined by the columns of M (Figure 2.8). The FFT is thus a change of basis, a *rigid rotation*. The inverse of M is the opposite rotation, from the Fourier basis back into the standard basis. When we write out the orthogonality condition precisely, we will be able to read off this inverse transformation with ease:

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

But ω^{-1} is also an nth root of unity, and so interpolation—or equivalently, multiplication by $M_n(\omega)^{-1}$—is itself just an FFT operation, but with ω replaced by ω^{-1}.

Now let's get into the details. Take ω to be $e^{2\pi i/n}$ for convenience, and think of the columns of M as vectors in $\mathbb{C}^n$. Recall that the *angle* between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v = (v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their *inner product*

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \cdots + u_{n-1} v_{n-1}^*,$$

where z^* denotes the complex conjugate[4] of z. This quantity is maximized when the vectors lie in the same direction and is zero when the vectors are orthogonal to each other.

The fundamental observation we need is the following.

Lemma *The columns of matrix M are orthogonal to each other.*

Proof. Take the inner product of any columns j and k of matrix M,

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

This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio $\omega^{(j-k)}$. Therefore it evaluates to $(1 - \omega^{n(j-k)})/(1 - \omega^{(j-k)})$, which is 0—except when $j = k$, in which case all terms are 1 and the sum is n. ∎

[4]The *complex conjugate* of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$. The complex conjugate of a vector (or matrix) is obtained by taking the complex conjugates of all its entries.

The orthogonality property can be summarized in the single equation

$$MM^* = nI,$$

since $(MM^*)_{ij}$ is the inner product of the ith and jth columns of M (do you see why?). This immediately implies $M^{-1} = (1/n)M^*$: we have an inversion formula! But is it the same formula we earlier claimed? Let's see—the (j,k)th entry of M^* is the complex conjugate of the corresponding entry of M, in other words ω^{-jk}. Whereupon $M^* = M_n(\omega^{-1})$, and we're done.

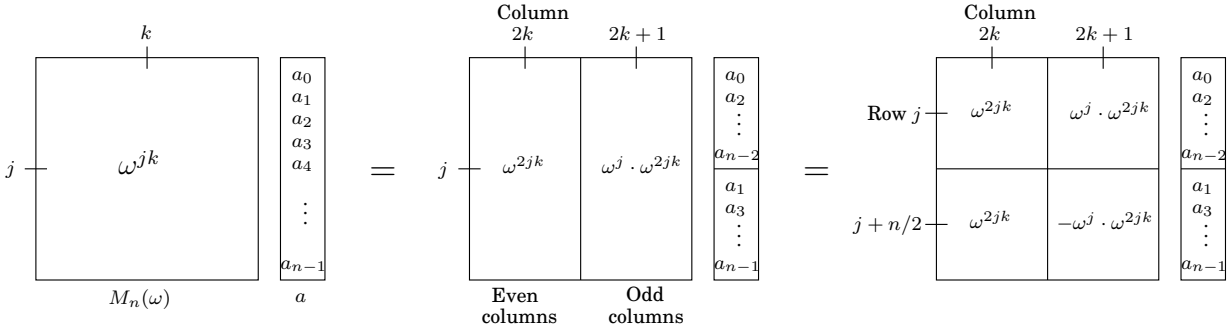
And now we can finally step back and view the whole affair geometrically. The task we need to perform, polynomial multiplication, is a lot easier in the Fourier basis than in the standard basis. Therefore, we first rotate vectors into the Fourier basis (*evaluation*), then perform the task (*multiplication*), and finally rotate back (*interpolation*). The initial vectors are *coefficient representations*, while their rotated counterparts are *value representations*. To efficiently switch between these, back and forth, is the province of the FFT.

2.6.4 A closer look at the fast Fourier transform

Now that our efficient scheme for polynomial multiplication is fully realized, let's hone in more closely on the core subroutine that makes it all possible, the fast Fourier transform.

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex nth roots of unity. It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$, which has (j,k)th entry (starting row- and column-count at zero) ω^{jk}. The potential for using divide-and-conquer in this matrix-vector multiplication becomes apparent when M's columns are segregated into evens and odds:



In the second step, we have simplified entries in the bottom half of the matrix using $\omega^{n/2} = -1$ and $\omega^n = 1$. Notice that the top left $n/2 \times n/2$ submatrix is $M_{n/2}(\omega^2)$, as is the one on the bottom left. And the top and bottom right submatrices are almost the same as $M_{n/2}(\omega^2)$, but with their jth rows multiplied through by ω^j and $-\omega^j$, respectively. Therefore the final product is the vector