The Fast Fourier Transform[1]

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 (Figure 1.1(a)). This signal might, for instance, capture a human voice by measuring the fluctuations of air pressure close to the speaker's mouth. Or it might capture the conformation of stars in the night sky, by measuring brightness across a range of spatial coordinates. In order to extract information from the signal, 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 designing a system, there is an endless diversity of possible transformations to choose from. The most convenient ones to deal with are those which satisfy two particular 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.

Any system with these properties can be described very concisely. In fact, 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

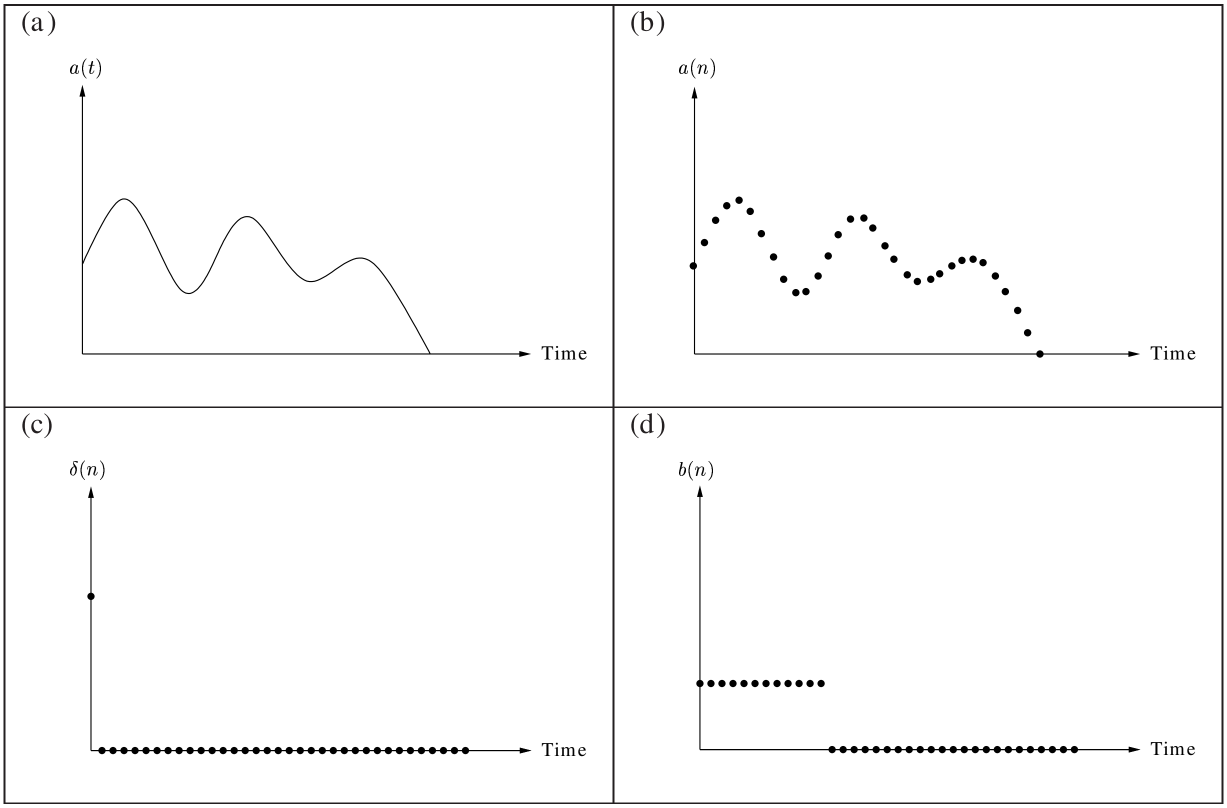
$$c(n) \;=\; \sum_{i=-\infty}^{\infty} a(i)b(n-i),$$

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

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.



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 $2T-2$:

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

This basic primitive is very important to compute efficiently. Each $b(i)$ takes $O(T)$ steps[2], so it looks like the overall computation time must be $O(T^2)$. Remarkably, the fast Fourier transform does it in just $O(T \log T)$ steps, and 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,

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

[2]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.

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*.

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. We could obtain $C(x)$ by computing these coefficients one-by-one, but this is no different from the quadratic-time baseline scheme for convolution.

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

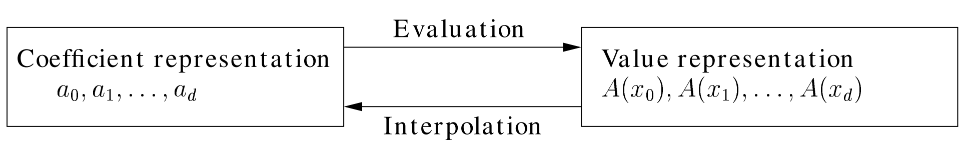
Fact. A degree-d polynomial is uniquely characterized by its values at *any* $d+1$ distinct points.

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 figure out, just $A(x)$ 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.

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

2.2 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 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, 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, $x_0^2, \ldots, x_{n/2-1}^2$.

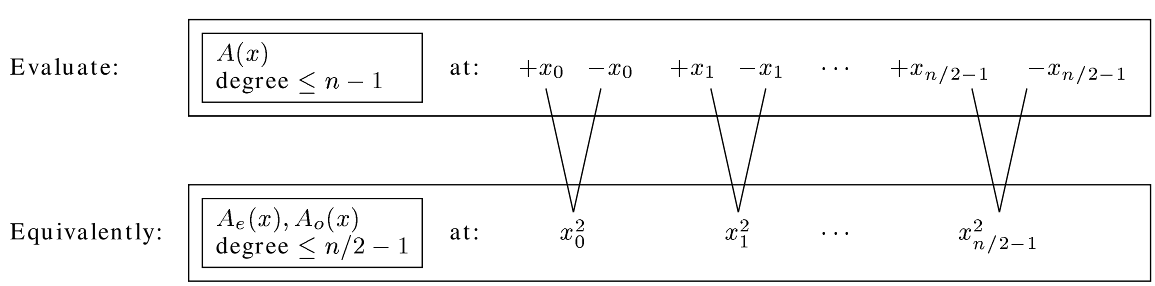
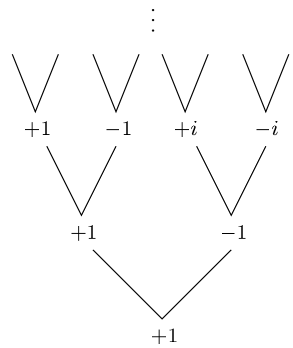


Figure 2.2 Reverse-engineering a good initial set of points.



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 are not yet done. For the divide-and-conquer to work, we need the points to be paired at *every* level of the recursion, not just initially. One way to arrange this is to "reverse-engineer" it. Here's what we mean. 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, $\pm\sqrt{1} = \pm 1$ (Figure 2.2). 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 n^{th} *complex roots of unity*, that is, the n complex solutions to the equation $z^n = 1$.

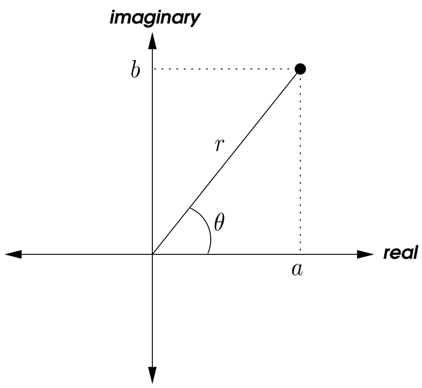
Figure 2.3 is a pictorial review of some basic facts about complex numbers. The third panel of this figure introduces the n^{th} roots of unity: the complex numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2\pi i/n}$ (in polar coordinates, $(1, 2\pi/n)$). If n is even,

1. the n^{th} roots are paired, $\omega^{n/2+j} = -\omega^j$; and
2. squaring them produces the $(n/2)^{nd}$ roots of unity.

Therefore, if we start with these numbers for some n which is a power of two, 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 paired, and so our divide-and-conquer, as shown in the last panel, works perfectly. The resulting algorithm is the fast Fourier transform (Figure 2.4).

Incidentally, why are we allowed to assume that n is a power of two? Well, our polynomial multiplication scheme lets us choose any value of n we like, as long as it is more than $2d$. For the sake of efficiency we would like to keep n small, but 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 2.3 The complex roots of unity are ideal for our divide-and-conquer scheme.



The complex plane

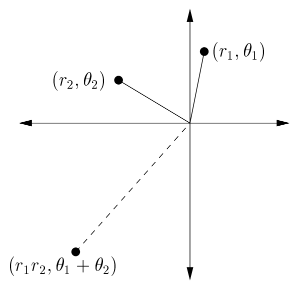
$z = a + bi$ is plotted at position (a, b).

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:

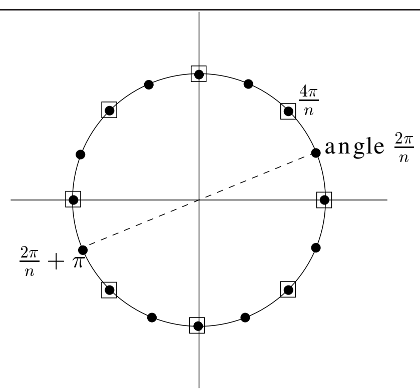
Number	-1	i	$5+5i$
Polar coords	$(1,\pi)$	$(1,\pi/2)$	$(5\sqrt{2},\pi/4)$



Multiplying is easy in polar coordinates

$(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.

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 n^{th} complex roots of unity

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

By 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 *paired*.
- Their squares are the $(n/2)^{nd}$ roots of unity, shown here with boxes around then.

Divide-and-conquer step

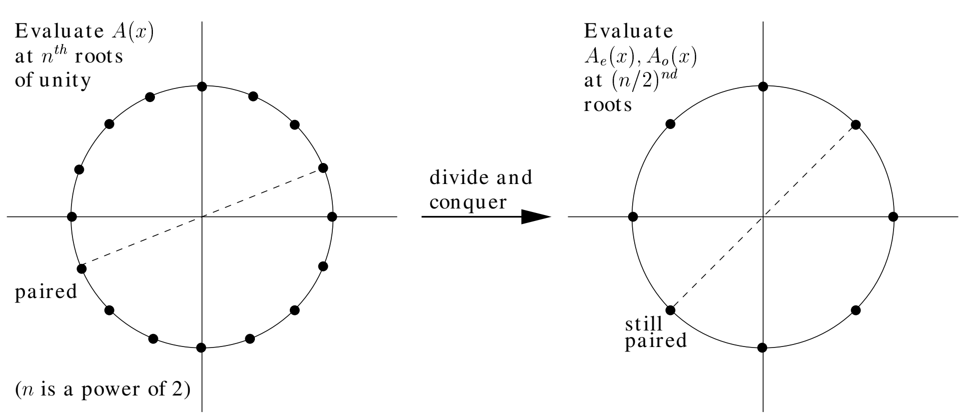


Figure 2.4 The fast Fourier transform (polynomial formulation)

`function FFT(`A,ω`)`
`Input:` Coefficient representation of a polynomial $A(x)$
of degree $\leq n-1$, where n is a power of two
ω, an n^{th} root of unity
`Output:` Value representation $A(\omega^0), \ldots, A(\omega^{n-1})$

`if` $\omega = 1$ `then return` $A(1)$
`rewrite` $A(x) = A_e(x^2) + xA_o(x^2)$
`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` $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$

`return` $A(\omega^0), \ldots, A(\omega^{n-1})$

2.3 Interpolation

We are well on our way towards efficiently realizing the grand scheme of Figure 2.1. The last remaining piece of the puzzle is interpolation, the inverse of evaluation:

$$\begin{tikzcd}
\boxed{\begin{array}{l}\text{Coefficient representation}\\ \quad a_0, a_1, \ldots, a_{n-1}\end{array}} \arrow[r, shift left, "\text{Evaluation}"] & \boxed{\begin{array}{l}\text{Value representation}\\ A(x_0), A(x_1), \ldots, A(x_{n-1})\end{array}} \arrow[l, shift left, "\text{Interpolation}"]
\end{tikzcd}$$

Our two polynomial representations are vectors of the same length; denote the coefficient representation simply by A, and the value representation by $\tilde{A}$. The FFT takes us from A to $\tilde{A}$ when the points $\{x_i\}$ are complex roots of unity,

$$\tilde{A} \;=\; \mathrm{FFT}(A, \omega).$$

For the reverse direction, here's a crazy idea, an idle shot in the dark: how about trying $\mathrm{FFT}(\tilde{A}, \omega^{-1})$? Amazingly, as we will very soon see, it works!

$$A \;=\; \frac{1}{n}\mathrm{FFT}(\tilde{A}, \omega^{-1}).$$

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 2.1) is now fully specified.

2.4 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 interesting properties, of which the following (see exercise) 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 *coefficient representation* – by the $n \times n$ matrix

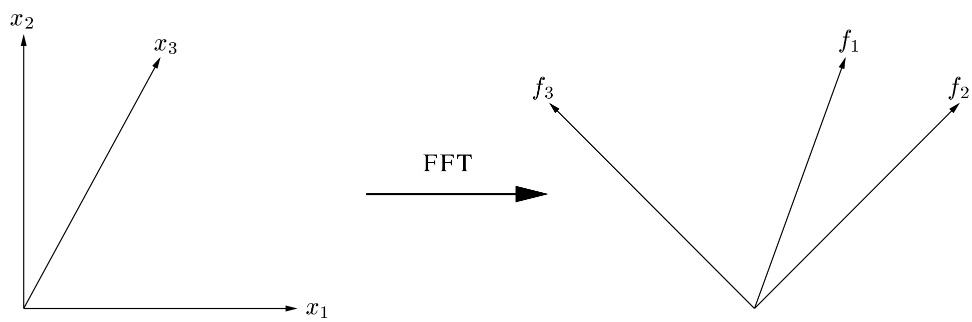
$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & \vdots & & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & \vdots & & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots &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

where ω is any complex n^{th} root of unity, and n is a power of two. (Notice how simple this matrix is to describe: its $(j, k)^{th}$ entry is ω^{jk}.) We now examine the geometry of this transformation, through which it will become clear that the inverse of M_n looks a lot like M_n itself:

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

But ω^{-1} is also an n^{th} root of unity, and this is why interpolation – or equivalently, multiplication by $M_n(\omega)^{-1}$ – is itself just a single FFT.

Figure 2.5 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 rows of $M_n(\omega)$, shown here as f_1, f_2, f_3. For instance, points in direction x_1 gets mapped into direction f_1.



For simplicity, take ω to be $e^{2\pi i/n}$ and abbreviate $M_n(\omega)$ by M. It is helpful to think of the rows of M as vectors in $\mathbf{C}^n$, and to understand their spatial layout: specifically, the angles between them. The angle between any two vectors $u = (u_0, \ldots, u_{n-1})$ and $v = (v_0, \ldots, v_{n-1})$ in $\mathbf{C}^n$ is a function of 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[3] of z. This is maximized when the vectors lie in the same direction, and is zero when the vectors are orthogonal (at right angles) to each other.

Taking the inner product of rows j and k of matrix M, we get a geometric series

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

The usual formula (see exercise) tells us that its sum is n if $j = k$ and is zero otherwise. In other words, *the rows of the matrix are orthogonal to each other*. They can therefore be thought of as the axes of an alternative coordinate system, which is often called 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 rows of M (Figure 2.5). The FFT is thus a change of basis, or to put it more geometrically, a *rigid rotation*.

The inverse of M is the opposite rotation, from the Fourier basis back into the standard basis. Expressing the orthogonality of M's rows in the form $MM^* = nI$, we see that this reverse transformation is

$$M^{-1} \;=\; \tfrac{1}{n} M^*.$$

Let's take a closer look at M^*. Its $(j,k)^{th}$ entry is the complex conjugate of the corresponding entry of M, which means it must be ω^{-jk}. Whereupon $M^* = M_n(\omega^{-1})$. Interpolation, the final piece of our polynomial multiplication algorithm, is therefore just multiplication by $M_n(\omega^{-1})$, a single FFT operation.

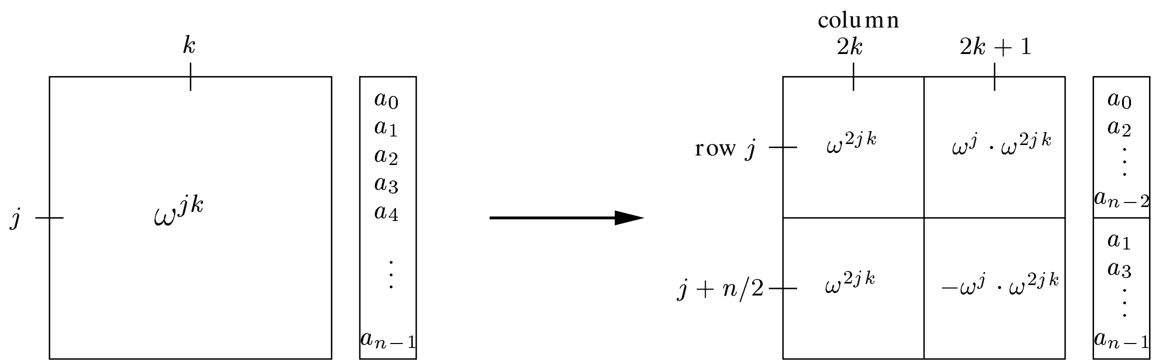
Let's step back and take a geometric view of the overall algorithm. The key idea is that the task we need to perform, polynomial multiplication (or equivalently, convolution), is a

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

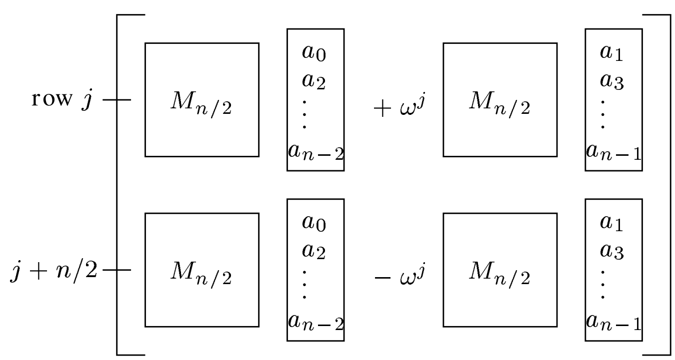
lot easier in the Fourier basis than in the standard basis. Therefore, we first rotate vectors into the Fourier basis (*evaluation*), then perform the task (*multiplication*), and finally rotate back (*interpolation*). The initial vectors are *coefficient representations*, while their rotated counterparts are *value representations*. To efficiently switch between these, back and forth, is the province of the FFT. We now take a closer look at how this is achieved.

3 The definitive algorithm

The FFT multiplies vectors by the matrix $M_n(\omega)$, whose $(j,k)^{th}$ entry (starting row- and column-count at zero) is ω^{jk}. The potential for divide-and-conquer becomes apparent when M's columns are segregated into evens and odds.



Here we have simplified entries using $\omega^{n/2} = -1$ and $\omega^n = 1$. Notice that the top left $n/2 \times n/2$ submatrix is simply $M_{n/2}(\omega^2)$. The other submatrices are closely related. Therefore the final product is the vector



This leads to the definitive FFT algorithm of Figure 3.1.

4 The fast Fourier transform unraveled

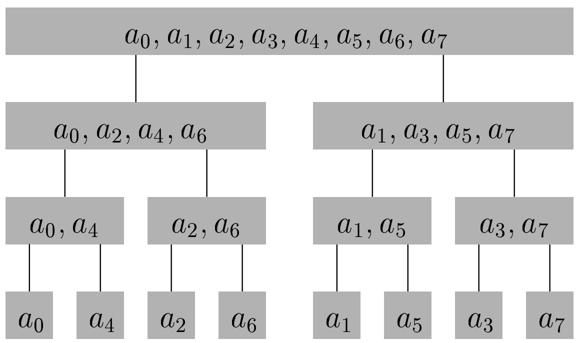
Through all our discussions so far, the fast Fourier transform has remained tightly cocooned within the strictures of a divide-and-conquer formalism. To fully expose its structure, we need to unbind it, to unravel the recursion.

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.

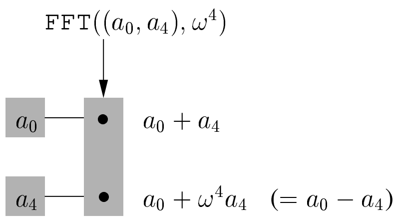
Figure 3.1 The fast Fourier transform

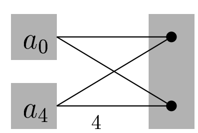
function FFT(a,ω)
Input: A vector $a = (a_0, a_1, \ldots, a_{n-1})$, for n a power of two
A primitive n^{th} root of unity, ω
Output: $M_n(\omega)\,a$

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



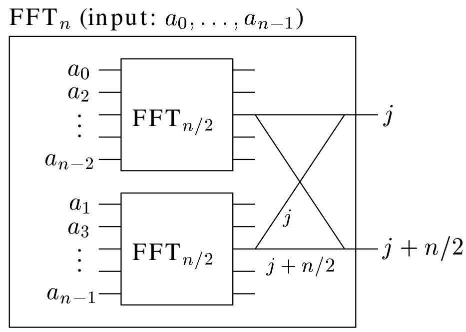
The intermediate computations are extremely simple. What happens with (a_0, a_4), for instance, is shown below on the left.





This kind of computational structure 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.

When written in terms of these circuit elements, the divide-and-conquer step has a very simple form. Here is how a problem of size n is reduced to two subproblems of size $n/2$ (for clarity, one pair of outputs $(j, j + n/2)$ is singled out):



The complete FFT circuit for 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.

Figure 4.1 The fast Fourier transform circuit.