The Fourier transform is a conversion between two representations of a function: the list representation and the polynomial representation. Let's start with functions over the n-dimensional *Boolean cube* $\{-1,+1\}^n$. Its inputs are all 2^n bit strings of length n. We'll represent bits by the numbers $+1$ and -1. Here are two examples:

- The maximum of two bits $\max_2\colon \{-1,+1\}^2 \to \{-1,+1\}$ is

$$\max_2(-1,-1) = -1,\ \max_2(-1,+1) = +1,\ \max_2(+1,-1) = +1,\ \max_2(+1,+1) = +1. \tag{1}$$

- The majority of three bits $\mathrm{maj}_3\colon \{-1,+1\}^3 \to \{-1,+1\}$ given by

$$\mathrm{maj}_3(-1,-1,-1) = -1,\ \mathrm{maj}_3(-1,-1,+1) = -1,\ \mathrm{maj}_3(-1,+1,-1) = -1,\ \mathrm{maj}_3(-1,+1,+1) = +1$$
$$\mathrm{maj}_3(+1,-1,-1) = -1,\ \mathrm{maj}_3(+1,-1,+1) = +1,\ \mathrm{maj}_3(+1,+1,-1) = +1,\ \mathrm{maj}_3(+1,+1,+1) = +1.$$

These are the list representations of $\max_2$ and maj_3. Any function on n-bit inputs can be defined by listing its 2^n evaluations in some predetermined order. It can also be specified as a *polynomial*:

$$\begin{aligned} \max_2(x_1,x_2) &= \tfrac{1}{2} + \tfrac{1}{2}x_1 + \tfrac{1}{2}x_2 - \tfrac{1}{2}x_1x_2 \\ \mathrm{maj}_3(x_1,x_2,x_3) &= \tfrac{1}{2}x_1 + \tfrac{1}{2}x_2 + \tfrac{1}{2}x_3 - \tfrac{1}{2}x_1x_2x_3. \end{aligned} \tag{2}$$

The monomials that are used to built up these polynomials are the 2^n *parity functions* $1, x_1, x_2, x_1x_2, x_1x_2x_3$, etc. Every *real-valued* function $f(x_1,\ldots,x_n)$ can be uniquely represented as a linear combination of the 2^n possible parities in variables x_1 up to x_n.

The parities are indexed by the set S of variables that participate in it, e.g., x_1x_3 is indexed by the set $\{1,3\}$. We can also write down the polynomial representation by listing all 2^n coefficients $\hat{f}(S)$ of the parities, for example

$$\widehat{\max}_2(\varnothing) = \tfrac{1}{2}, \quad \widehat{\max}_2(\{1\}) = \tfrac{1}{2}, \quad \widehat{\max}_2(\{2\}) = \tfrac{1}{2}, \quad \widehat{\max}_2(\{1,2\}) = -\tfrac{1}{2}.$$

and

$$\begin{aligned} &\widehat{\mathrm{maj}}_3(\varnothing) = 0, && \widehat{\mathrm{maj}}_3(\{1\}) = \tfrac{1}{2}, && \widehat{\mathrm{maj}}_3(\{2\}) = \tfrac{1}{2}, && \widehat{\mathrm{maj}}_3(\{3\}) = \tfrac{1}{2} \\ &\widehat{\mathrm{maj}}_3(\{1,2\}) = 0, && \widehat{\mathrm{maj}}_3(\{1,3\}) = 0, && \widehat{\mathrm{maj}}_3(\{2,3\}) = 0, && \widehat{\mathrm{maj}}_3(\{1,2,3\}) = -\tfrac{1}{2}. \end{aligned}$$

These are the *Fourier representations* of $\max_2$ and maj_3, respectively. The Fourier transform is the conversion from the list representation of f to the Fourier representation $\hat{f}$.

This variant of the Fourier transform for functions over the Boolean cube is sometimes called the Fourier-Walsh transform. We'll talk in a bit about other types of functions.

1 The mathematics of the Fourier transform

Algebra

How did I calculate the Fourier transforms of $\max_2$ and maj_3 from their list representations? The key fact is that any function is a linear combination of *point functions* $\mathrm{point}_{\mathbf{a}}$, namely functions that evaluate to one at a specific input $\mathbf{x} = \mathbf{a}$ and zero at all other inputs $\mathbf{x} \neq \mathbf{a}$.

For example, the $\max_2$ function is a linear combination of the four point functions $\mathrm{point}_{(-1,-1)}$, $\mathrm{point}_{(-1,+1)}$, $\mathrm{point}_{(+1,-1)}$ and $\mathrm{point}_{(+1,+1)}$ with coefficients

$$\max_2 = -1 \cdot \mathrm{point}_{(-1,-1)} + 1 \cdot \mathrm{point}_{(-1,+1)} + 1 \cdot \mathrm{point}_{(+1,-1)} + 1 \cdot \mathrm{point}_{(+1,+1)}\,. \tag{3}$$

When we evaluate both sides at an input $\mathbf{x}$, say $\mathbf{x} = (-1, +1)$, only the corresponding point function $\text{point}_{(-1,+1)}(x)$ does not vanish. We can read off the value $\max_2(x)$ from its coefficient. This is precisely the list representation of $\max_2$, only written in different notation.

All we have to do now is figure out the polynomial/Fourier representation of the point functions. Once we have those we can add them up. The advantage of working with point functions is their multiplicativity. For example $\text{point}_{(-1,+1)}(x_1, x_2) = \text{point}_{-1}(x_1)\,\text{point}_{+1}(x_2)$. We reduced the problem to finding a polynomial for the *univariate* point function $\text{point}_a(x)$ (when a and x are single bits). This is the linear function

$$\text{point}_a(x) = \frac{1 + ax}{2}.$$

When a and x are equal, both sides are $(1 + 1)/2 = 1$. When they are different, they are $(1 - 1)/2 = 0$. By multiplicativity,

$$\text{point}_{a_1,a_2}(x_1, x_2) = \frac{1 + a_1x_1}{2} \cdot \frac{1 + a_2x_2}{2}$$

and all we have to do is plug this into (3)

$$\max_2(x_1, x_2) = -1 \cdot \frac{1 - x_1}{2} \cdot \frac{1 - x_2}{2} + 1 \cdot \frac{1 - x_1}{2} \cdot \frac{1 + x_2}{2} + 1 \cdot \frac{1 + x_1}{2} \cdot \frac{1 - x_2}{2} + 1 \cdot \frac{1 + x_1}{2} \cdot \frac{1 + x_2}{2}.$$

and simplify to obtain formula (2).

This strategy works in general: The polynomial representation of a point function given by its 2^n evaluations $f(\mathbf{a})$ as $\mathbf{a}$ ranges over $\{-1, +1\}^n$ is obtained by simplifying the expression

$$f(\mathbf{x}) = \sum_{\mathbf{a} \in \{-1,+1\}^n} f(\mathbf{a})\,\text{point}_{\mathbf{a}}(\mathbf{x}) = \sum_{a_1,\ldots,a_n \in \{-1,+1\}} f(a_1, \ldots, a_n) \cdot \frac{1 + a_1x_1}{2} \cdots \frac{1 + a_nx_n}{2}. \tag{4}$$

Formula (4) gives one method of expanding f as a polynomial in its variables. Could other methods give different answers? Luckily, it turns out not: The polynomial representation in (4) is unique.

Geometry

To understand why we need to think geometrically. We can visualize the list representation (1) of $\max_2$ as the 4-dimensional vector $(-1, +1, +1, +1)$. Each "axis" in this vector space is labeled by an input: the first one by $(-1, -1)$, the second one by $(-1, +1)$, and so on. Formula (3) describes this vector as a linear combination of the standard basis vectors $\text{point}_{(-1,-1)} = (1, 0, 0, 0)$, $\text{point}_{(-1,+1)} = (0, 1, 0, 0)$, $\text{point}_{(+1,-1)} = (0, 0, 1, 0)$, and $\text{point}_{(-1,-1)} = (0, 0, 0, 1)$:

$$\begin{bmatrix} -1 \\ +1 \\ +1 \\ +1 \end{bmatrix} = -1 \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + 1 \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + 1 \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + 1 \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

The polynomial expansion (2) represents the same vector in a different basis: the basis of parity functions $1 = (+1, +1, +1, +1), x_1 = (-1, -1, +1, +1), x_2 = (-1, +1, -1, +1), x_1x_2 = (+1, -1, -1, +1)$. In this notation, (2) becomes

$$\begin{bmatrix} -1 \\ +1 \\ +1 \\ +1 \end{bmatrix} = \frac{1}{2} \cdot \begin{bmatrix} +1 \\ +1 \\ +1 \\ +1 \end{bmatrix} + \frac{1}{2} \cdot \begin{bmatrix} -1 \\ -1 \\ +1 \\ +1 \end{bmatrix} + \frac{1}{2} \cdot \begin{bmatrix} -1 \\ +1 \\ -1 \\ +1 \end{bmatrix} - \frac{1}{2} \cdot \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \end{bmatrix}.$$

From a geometric point of view, the Fourier transform is nothing but a change of basis formula! It converts from the the standard representation of f as a linear combination of the 2^n point functions

$\text{point}_{\mathbf{a}}\colon \mathbf{a} \in \{-1,+1\}^n$ to the polynomial representation of f as a linear combination of the 2^n parity functions $\text{parity}_S = \prod_{i\in S} x_i\colon S \subseteq \{1,\dots,n\}$. As the *dimension* of both bases is the same and we have conversion formulas in both directions, linear algebra tells us that the parities must be linearly independent and therefore produce a unique Fourier representation.

If you did not follow this argument do not despair because geometry reveals a much more interesting reason behind the linear dependence of the parities: They are *mutually orthogonal*. When $n = 2$, for example, $x_1 \cdot (x_1x_2) = (-1,+1,-1,+1)\cdot(+1,-1,-1,+1) = 0$, and the same goes for the other five pairs. There is an algebraic reason behind these cancellations: The dot product $x_1 \cdot (x_1x_2)$ is the sum of the products of the corresponding entries

$$x_1 \cdot (x_1x_2) = \sum_{a_1,a_2} a_1(a_1a_2) = \sum_{a_1,a_2} a_1^2a_2 = \sum_{a_1,a_2\in\{-1,+1\}} a_2 = 0$$

and the x_1-part cancels out because $x_1^2 = 1$. This is where the cumbersome representation of bits by -1s and $+1$s pays off: When we evaluate the dot product of any two distinct parities, we sum up the evaluations of a third parity. This third parity keeps shifting between -1 and $+1$ and is therefore zero.

Lemma 1. *The 2^n parity functions are mutually orthogonal when viewed as 2^n-dimensional vectors.*

Orthogonality in particular implies linear independence, but it tells us much more. First, the Fourier coefficients $\hat{f}(S)$ must equal the projections of f onto the parity determined by S, for example

$$\widehat{\text{maj}_3}(\{1,3\}) = \frac{\text{maj}_3\cdot(x_1x_3)}{\|x_1x_3\|^2} = \sum_{\mathbf{a}\in\{-1,+1\}^3} \frac{\text{maj}_3(a_1,a_2,a_3)a_1a_3}{\text{sum of } 2^3 \ (\pm1)^2} = \frac{1}{2^3}\sum_{\mathbf{a}\in\{-1,+1\}^n} \text{maj}_3(a_1,a_2,a_3)a_1a_3.$$

In this example you can verify that the sum zeroes out: $\text{maj}_3(a_1,a_2,a_3)$ and a_1a_3 are equal on four out of the eight inputs and different on the other four.

Probability

The last expression has a probabilistic interpretation: It is the average of the values obtained when f is shifted by the parity x_1x_3. In general,

$$\hat{f}(S) = \text{average value of } (f\cdot\text{parity}_S) = \frac{1}{2^n}\sum_{\mathbf{a}\in\{-1,+1\}^n} f(\mathbf{a})\,\text{parity}_S(\mathbf{a}). \tag{5}$$

When S is the empty set, $\text{parity}_\varnothing = 1$, so $\hat{f}(\varnothing)$ is nothing but the average value of f. If we had to summarize all of f by one number, this would be it. What do $\hat{f}(\{1\})$ and $\hat{f}(\{5,7\})$ tell us? We'll come back to this and more shortly. But now that we have a formula let's talk about algorithms.

Computation

Computing the Fourier Transform means taking the list of 2^n values $f(\mathbf{a})$ and producing the list of 2^n Fourier coefficients $f(S)$. Both the input and output have size $N = 2^n$ so it makes sense to express the complexity in terms of this N. Evaluating (5) would entail N^2 operations: It takes $N = 2^n$ additions to calculate each Fourier coefficient and there are $N = 2^n$ of them to calculate.

There is a faster recursive algorithm. It is easiest to describe it in the language of polynomials. Suppose that we have figured out the polynomial representations of the functions

$$f_-(x_1,\dots,x_{n-1}) = f(x_1,\dots,x_{n-1},-1) \qquad \text{and} \qquad f_+(x_1,\dots,x_{n-1}) = f(x_1,\dots,x_{n-1},+1).$$

How can we combine them into a single polynomial representation of f? What we want f to do is produce $f_-(x_1, \ldots, x_{n-1})$ when x_n is -1 and $f_+(x_1, \ldots, x_{n-1})$ when x_n is $+1$. We can resort to a tried and tested trick: Write f as a linear combination of two point functions.

$$\begin{aligned} f &= f_- \cdot \text{point}_{-1}(x_n) + f_+ \cdot \text{point}_{+1}(x_n) \\ &= f_- \cdot \frac{1 - x_n}{2} + f_+ \cdot \frac{1 + x_n}{2} \\ &= \frac{1}{2}(f_+ + f_-) + \frac{1}{2}(f_+ - f_-)x_n. \end{aligned}$$

The term $\frac{1}{2}(f_+ + f_-)$ contains those monomials that exclude x_n, while the term $\frac{1}{2}(f_+ - f_-)$ contains those monomials that include it. The corresponding formula for the monomial/Fourier coefficients is

$$\hat{f}(S) = \frac{1}{2}(\hat{f}_+(S) + \hat{f}_-(S)) \quad \text{and} \quad \hat{f}(S \cup \{n\}) = \frac{1}{2}(\hat{f}_+(S) - \hat{f}_-(S)) \tag{6}$$

for every subset S of $\{1, \ldots, n-1\}$.

Algorithm *FFW* (Fast Fourier-Walsh Transform)
Input: A list representation of $f\colon \{-1,+1\}^n \to \mathbb{R}$.
1 If $n = 0$, output the value $f()$.
2 Calculate $\hat{f}_+ = FFW(f_+)$ and $\hat{f}_- = FFW(f_-)$.
3 Calculate $\hat{f}$ using (6) and output it.

The correctness of the algorithm follows from induction by (6). The base case $n = 0$ holds because a function with no inputs is a constant, so its list and Fourier representations are identical.

Here is an example run of *FFW* when $f = \max_2$. The algorithm needs to calculate the Fourier-Walsh transforms of $f_+ = \max_2(x_1, 1)$ and $f_- = \max_2(x_1, -1)$ first. f_+ decomposes into $f_{++} = 1$ and $f_{+-} = 1$, obtaining $f_+ = \frac{1}{2}(f_{++} + f_{+-}) + \frac{1}{2}(f_{++} - f_{+-})x_1 = 1$. It similarly obtains $f_- = x_1$. In step 3 they are combined into

$$f = \tfrac{1}{2}(f_+ + f_-) + \tfrac{1}{2}(f_+ - f_-)x_2 = \tfrac{1}{2}(1 + x_1) + \tfrac{1}{2}(1 - x_1)x_2 = \tfrac{1}{2} + \tfrac{1}{2}x_1 + \tfrac{1}{2}x_2 - \tfrac{1}{2}x_1x_2.$$

2 Polynomials, complexity, and approximation

A key goal of the theory of computation is to characterize the *computational* complexity of various functions that come up. The Fourier representation is sometimes a helpful indicator of complexity.

One natural complexity measure of a polynomial is its degree. Low-degree polynomials have relatively few "degrees of freedom." A linear function over the n-dimensional Boolean cube is completely specified by its mean $\hat{f}(\varnothing)$ and its n level-1 Fourier coefficients $\hat{f}(\{1\}), \ldots, \hat{f}(\{n\})$. More generally a degree-d polynomial is determined by its Fourier coefficients of size d or less. There are at most $2^{nH(d/n)}$ of them, where $H(p) = -p \log p - (1-p)\log(1-p)$ is the binary entropy function. When d is smaller than n by some factor this is much less than the 2^n values it takes to specify the function as a whole.

The compactness of polynomial representations can be exploited by algorithms. Suppose we are given input-output examples $(x, f(x))$ for some unknown function f. If f is a degree-d polynomial we can try to reconstruct the polynomial representing f by interpolating its Fourier coefficients from these examples. This amounts to solving a linear system, which can be accomplished in time polynomial in $2^{nH(d/n)}$, provided there are sufficiently many linearly independent equations in the system.

Some simple types of computations are naturally captured by low-degree polynomials. As an example, the function

$$f(x, y, z) = \text{"if } x = 1 \text{ then } y \text{ else } z\text{"}$$

has degree 2. It is unlike the 72% of the other Boolean functions on 3 inputs whose degree is 3. It has degree 2 because all point functions that show up in its list representation

$$f = y \cdot \text{point}_1(x) + z \cdot \text{point}_{-1}(x) = y \cdot \frac{1+x}{2} + z \cdot \frac{1-x}{2}$$

have degree 2 or less.

A *decision tree* is an arbitrary nested if-then-else program of this form. The *depth* of the decision tree is the number of nested levels plus one. By the same reasoning any depth-d decision tree is a degree-d polynomial. Because it is a low-degree polynomial, some algorithmic tasks such as learning from random examples are easier for decision trees than for more complex functions.

A more complex type of function is a formula in disjunctive normal form, namely an AND of terms, each of which is an OR of literals (variables or negated variables) such as

$$(x \text{ AND } (\text{NOT } y) \text{ AND } z) \text{ OR } (x \text{ AND } w) \text{ OR } (u \text{ AND } w \text{ AND } (\text{NOT } z)).$$

Terms could represent qualification requirements. For example a student is eligible for a scholarship if they are in an upper year and in CS and have a GPA of 3 or more, or if they are Francophone and does not live too close to home, or if they are not in an upper year but won a contest. Such formulas do not have a low-degree representation. It turns out, however, that they can be "approximated" by polynomials of fairly low degree.

To illustrate how this works let's look at the formula

$$f = (x_1 \text{ AND } x_2) \text{ OR } (x_3 \text{ AND } x_4) \text{ OR } (x_5 \text{ AND } x_6).$$

Its complete polynomial representation is too long to write down. The lowest two levels are

$$f = -.15625 + .28125(x_1 + x_2 + x_3 + x_4 + x_5 + x_6) + \text{higher order terms.}$$

What if we discard the higher order terms? Let's start with the single degree-zero term $\hat{f}(\varnothing) = -.15625$. This is the mean value of f. Suppose we used this constant $\hat{f}(\varnothing)$ as an approximation of $f(\mathbf{x})$. This approximation is not too good: f evaluates to -1 or $+1$ at any input, and $\hat{f}(\varnothing)$ is quite far from both -1 and $+1$.

This approximation, however, is best possible on average among all constants c. The Fourier coefficient $\hat{f}(\varnothing)$ minimizes the average of $(f(\mathbf{x}) - c)^2$. Moreover, there is a formula for this minimum square average error. It is precisely $1 - \hat{f}(\varnothing)^2$, which is about 0.975 in our example. If we want a decent approximation we have to look beyond constants.

Let's now include the degree-1 parts of f in the approximation. We obtain the linear function

$$\ell(\mathbf{x}) = \hat{f}(\varnothing) + \hat{f}(\{1\})x_1 + \cdots + \hat{f}(\{6\})x_6 = -.15625 + .28125(x_1 + x_2 + x_3 + x_4 + x_5 + x_6).$$

Here is a comparison of $f(\mathbf{x})$ versus $\ell(\mathbf{x})$ on ten randomly chosen $\mathbf{x}$s:

```
x      f(x)  l(x)
000111 -1    -0.15625
011011 -1    -0.71875
011011 -1    -0.71875
101100 -1    -0.15625
011101 -1    -0.71875
010010 +1     0.40625
101011 -1    -0.71875
110101 -1    -0.71875
010100 +1     0.40625
001001 +1     0.40625
```

The empirical square error $\sum(f(\mathbf{x}) - \ell(\mathbf{x}))^2$ is about 2.877, or less than 30% per run. Among all linear functions, the one that we chose minimizes this error when averaged over all 64 possible inputs $\mathbf{x}$. There is again a formula for the error:

$$\text{average of } (f(\mathbf{x}) - \ell(\mathbf{x}))^2 = 1 - \hat{f}(\varnothing)^2 - \hat{f}(\{1\})^2 - \cdots - \hat{f}(\{6\})^2.$$

In our example this is about 0.499. The empirical estimate of 30% was optimistic but not by much. A 50% error for a linear approximation of a degree-6 ± 1-valued polynomial is not too bad!

In general, the degree-d approximation of any (not necessarily Boolean-valued) function f that minimizes the mean-squared error is the sum of the first d Fourier levels $\sum_{S:|S|\leq d} \hat{f}(S)\,\text{parity}_S$. This follows from orthogonality of the parities. Just like adding terms to the spectral decomposition of gave successively better approximations of a matrix, taking higher degree terms in the Fourier decomposition of a function also improves the approximation. The Fourier representation is in fact a special case of spectral decomposition.

For a general function f, the mean-square error of the degree-d approximation equals

$$\big(\text{average of } f(\mathbf{x})^2\big) - \sum_{S:|S|\leq d} \hat{f}(S)^2.$$

When d equals n, there is no approximation error, and we obtain *Parseval's identity*

$$\text{average of } f(\mathbf{x})^2 = \sum_{S} \hat{f}(S)^2. \tag{7}$$

For a ± 1 valued function f, $f(\mathbf{x})^2$ always equals one and the squared Fourier coefficients add up to one.

One weakness of polynomial degree as a measure of complexity is that dense parity functions, such as the parity of all the bits, do not qualify as simple. This is contrary to experience as parities are easy to compute in practice.

3 The modular Fourier transform

The Fourier transform for functions on the Boolean cube is sometimes useful for reasoning about computational processes but hasn't found much applications outside of theory. There is another type of Fourier transform whose practical significance is indisputable.

Let's start with Boolean functions $f\colon \{-1,+1\} \to \mathbb{R}$ with a one-bit input ($n = 1$, $N = 2$ in the notation of Section 1). Their Fourier transform is

$$f(x) = \hat{f}(0) + \hat{f}(1)x, \qquad \hat{f}(0) = \frac{f(1) + f(-1)}{2}, \quad \hat{f}(1) = \frac{f(1) - f(-1)}{2}.$$

I snuck in a change of notation here: Instead of indexing the Fourier coefficients by the *sets* $\varnothing$ and $\{1\}$, I used the *numbers* 0 and 1.

One important feature of the parity basis $1 = (1,1)$ and $x = (1,-1)$ is that it is not only orthogonal as a basis of functions, but it is also a *group*: The pointwise product of two basis functions is a basis function. The multiplication table for 1 and x looks like this:

$\cdot$	1	x
1	1	x
x	x	1

This is precisely the same as the multplication rule for the numbers $+1$ and -1:

$\cdot$	$+1$	-1
$+1$	$+1$	-1
-1	-1	$+1$

(8)

The monomials 1 and x in the Fourier representation "play the same role" as the values $+1$ and -1 that we represent bits by.

How does this reasoning generalize to functions that take three input values, i.e., trits? Let's name these trits ω_0, ω_1, and ω_2. We want to represent functions f over domain $\{\omega_0, \omega_1, \omega_2\}$ by polynomials. By dimension-counting f should have three "degrees of freedom", suggesting a quadratic representation

$$f(x) = \hat{f}(0) + \hat{f}(1)x + \hat{f}(2)x^2. \tag{9}$$

How should the multiplication table of 1, x, and x^2 look like? In analogy to the bit setting, they should multiply exactly like the corresponding numbers $\omega_0, \omega_1, \omega_2$. Can we make up a "multiplication table" like (8) out of some three numbers $\omega_0, \omega_1, \omega_2$? A moment's thought should convince you that this is quite challenging. It is in fact impossible if we insist on *real* numbers.

There is a beautiful solution if we are willing to tolerate complex numbers. Associate the monomial x^j by the complex root of unity $\omega_j = e^{2\pi ij/3} = \cos(2\pi j/3) + i\sin(2\pi j/3)$. These can be genuinely multiplied:

$\cdot$	ω_0	ω_1	ω_2
ω_0	ω_0	ω_1	ω_2
ω_1	ω_1	ω_2	ω_0
ω_2	ω_2	ω_0	ω_1

Geometrically, the monomials $1, x, x^2$ are now represented by the complex basis vectors

$$1 = \begin{bmatrix} 1 \\ \omega_0 \\ \omega_0^2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad x = \begin{bmatrix} 1 \\ \omega_1 \\ \omega_1^2 \end{bmatrix} = \begin{bmatrix} 1 \\ e^{2\pi i/3} \\ e^{4\pi i/3} \end{bmatrix} \qquad x^2 = \begin{bmatrix} 1 \\ \omega_2 \\ \omega_2^2 \end{bmatrix} = \begin{bmatrix} 1 \\ e^{4\pi i/3} \\ e^{8\pi i/3} \end{bmatrix} = \begin{bmatrix} 1 \\ e^{4\pi i/3} \\ e^{2\pi i/3} \end{bmatrix}.$$

If we take the "dot product" of x and x^2 naively we would get the result $1 \cdot 1 + \omega_1\omega_2 + \omega_1^2\omega_2^2 = 3$ so it appears that we lost orthogonality. The issue is not with orthogonality but with the definition of dot product for vectors with complex entries: When calculating $\mathbf{a} \cdot \mathbf{b}$ we need to conjugate the entries of $\mathbf{b}$ before evaluating the sum of products:

$$\mathbf{a} \cdot \mathbf{b} = (a_1, \cdots, a_n) \cdot (b_1, \cdots, b_n) = a_1\overline{b_1} + \cdots + a_n\overline{b_n}.$$

Under this correct definition of complex dot product you can verify that 1, x, and x^2 are orthogonal, and each has squared length 3.

In summary, the Fourier transform modulo 3 is defined for complex-valued functions f over the third roots of unity $\{1, e^{2\pi i/3}, e^{4\pi i/3}\}$, i.e., $f\colon \{1, e^{2\pi i/3}, e^{4\pi i/3}\} \to \mathbb{C}$. These functions have a unique quadratic polynomial representation (9). In analogy to (5), the Fourier coefficients $\hat{f}(0), \hat{f}(1), \hat{f}(2)$ are

$$\hat{f}(j) = \text{average of } \big(f \cdot \overline{x^j}\big) = \text{average of } (f \cdot x^{-j}) = \frac{1}{3}\big(f(1) \cdot 1 + f(e^{2\pi i/3}) \cdot e^{-2\pi ij/3} + f(e^{4\pi i/3}) \cdot e^{-4\pi ij/3}\big).$$

It is convenient to give the name ω to the "primitive" root of unity $e^{2\pi i/3}$. In this notation, the domain of f is $\{1, \omega, \omega^2\}$ and its Fourier transform is

$$\hat{f}(j) = \frac{1}{3}\big(f(1) + f(\omega)\omega^{-j} + f(\omega^2)\omega^{-2j}\big).$$

These formulas generalize to arbitrary modulus $N \geq 2$. We set ω to $e^{2\pi i/N}$. Functions $f\colon \{1, \omega, \ldots, \omega^{N-1}\} \to \mathbb{C}$ have a unique degree-$(N-1)$ polynomial representation

$$f(x) = \hat{f}(0) + \hat{f}(1)x + \cdots + \hat{f}(N-1)x^{N-1} \tag{10}$$

with the Fourier coefficients given by

$$\hat{f}(j) = \frac{1}{N}\sum_{k=0}^{N-1} f(\omega^k)\omega^{-kj} = \text{average value of } (f(x)x^{-j}). \tag{11}$$

The basis functions $x^j = (1, \omega^j, \omega^{2j}, \cdots, \omega^{(N-1)j})$ are orthogonal as vectors under the complex dot product. Their role is analogous to that of the parity functions parity_S in the Fourier transform over the Boolean cube. The size of S corresponds to the magnitude of j: The zeroth coefficient $\hat{f}(0)$ is the average value of f, which is the best approximation among all constants c with respect to the average of $|f(x) - c|^2$. The first-level truncation $\hat{f}(0) + \hat{f}(1)x$ is the best approximation among all linear functions $\ell(x)$ with respect to the average of $|f(x) - \ell(x)|^2$, and so on.

4 The Fast Fourier Transform

Calculating the modular Fourier representation with formula (11) takes time quadratic in N, just like for the Fourier-Walsh transform over the Boolean cube. Again, there is a faster algorithm, provided N is a power of two. The strategy is to separately calculate the even and odd coefficients in (10). To be specific, let's take $N = 8$. Then f can be split into even and odd parts:

$$f(x) = \big(\hat{f}(0) + \hat{f}(2)x^2 + \hat{f}(4)x^4 + \hat{f}(6)x^6\big) + \big(\hat{f}(1)x + \hat{f}(3)x^3 + \hat{f}(5)x^5 + \hat{f}(7)x^7\big).$$

Those parts are the respective polynomial representations of $(f(x) + f(-x))/2$ and $(f(x) - f(-x))/2$:

$$\begin{aligned}\frac{f(x) + f(-x)}{2} &= \hat{f}(0) + \hat{f}(2)x^2 + \hat{f}(4)x^4 + \hat{f}(6)x^6 \\ \frac{f(x) - f(-x)}{2} &= \hat{f}(1)x + \hat{f}(3)x^3 + \hat{f}(5)x^5 + \hat{f}(7)x^7 \\ &= x(\hat{f}(1) + \hat{f}(3)x^2 + \hat{f}(5)x^4 + \hat{f}(7)x^6).\end{aligned}$$

This suggests setting up the functions

$$f_+(x^2) = \frac{f(x) + f(-x)}{2} \qquad \text{and} \qquad f_-(x^2) = \frac{f(x) - f(-x)}{2x}, \tag{12}$$

calculating their polynomial expansions, and adding up the resulting polynomials.

As x ranges over the 8-th complex roots of unity $e^{2\pi ij/8}$, x^2 ranges over the 4-th roots $e^{2\pi ij/4}$. The functions f_+ and f_- must therefore have the intended Fourier expansions even modulo 4, namely

$$\begin{aligned}f_+(y) &= \hat{f}(0) + \hat{f}(2)y + \hat{f}(4)y^2 + \hat{f}(6)y^3, \qquad &\text{or } \hat{f}_+(j) = \hat{f}(2j) \\ f_-(y) &= \hat{f}(1) + \hat{f}(3)y + \hat{f}(5)y^2 + \hat{f}(7)y^3, \qquad &\text{or } \hat{f}_-(j) = \hat{f}(2j+1).\end{aligned}$$

Conversely,

$$\hat{f}(j') = \begin{cases} \hat{f}_+(j'/2), & \text{if } j' \text{ is even,} \\ \hat{f}_-((j'-1)/2), & \text{if } j' \text{ is odd.} \end{cases} \tag{13}$$

Algorithm *FFT* (The Fast Fourier Transform)
Input: N (a power of two) and the list $f(1), f(\omega), \ldots, f(\omega^{N-1})$.
1 If $N = 1$, output the value $f(1)$.
2 Set $\omega = e^{2\pi i/N}$.
3 Calculate the list representations of $f_+, f_- \colon \{1, \omega^2, \omega^4, \ldots, \omega^{2(N-1)}\} \to \mathbb{C}$ using (12).
4 Calculate $\hat{f}_+ = FFT(N/2, f_+)$ and $\hat{f}_- = FFT(N/2, f_-)$.
5 Construct $\hat{f} \colon \{0, \ldots, N-1\} \to \mathbb{C}$ using (13) and output it.

The FFT algorithm calculates the Fourier transform of f using $O(N \log N)$ operations with complex numbers (additions, subtractions, and scalings).

Example Let $N = 4$. The fourth roots of unity are 1, $\omega = i$, $\omega^2 = -1$ and $\omega^3 = -i$. Let's take the function f over domain $\{1, i, -1, -i\}$ whose list representation is $f(1) = 1, f(i) = 0, f(-1) = -2, f(-i) = 1$. We want to derive the polynomial representation

$$\begin{aligned} f(x) &= \hat{f}(0) + \hat{f}(1)x + \hat{f}(2)x^2 + \hat{f}(3)x^3 \\ &= (\hat{f}(0) + \hat{f}(2)x^2) + x(\hat{f}(1) + \hat{f}(3)x^2) \\ &= f_+(x^2) + xf_-(x^2), \end{aligned} \tag{14}$$

where $f_+(y)$ and $f_-(y)$ are functions over the second roots of unity 1 and -1. Their list representations are calculated in line 3 as

$$f_+(1) = \frac{f(1) + f(-1)}{2} = -.5 \qquad f_-(1) = \frac{f(1) - f(-1)}{2} = 1.5$$
$$f_+(-1) = \frac{f(i) + f(-i)}{2} = .5 \qquad f_-(-1) = \frac{f(i) - f(-i)}{2i} = .5i.$$

Even though f was real-valued, f_- is complex-valued.

In step 4 the FFT algorithm recursively calculates the polynomial representations of f_+ and f_-. As the domain of these functions is $-1, 1$ the outcome is the same as for the Fourier-Walsh transform and the representations are

$$f_+(y) = -.5y, \qquad f_-(y) = (.75 + .25i) + (.75 - .25i) \cdot y.$$

Finally, in step 5 these representations are plugged into (14) to obtain the polynomial representation of f:

$$\begin{aligned} f(x) &= -.5x^2 + x \cdot \big((.75 + .25i) + (.75 - .25i) \cdot x^2\big) \\ &= (.75 + .25i) \cdot x - .5 \cdot x^2 + (.75 - .25i) \cdot x^3, \end{aligned}$$

or $\hat{f}(0) = 0, \hat{f}(1) = .75 + .25i, \hat{f}(2) = -.5, \hat{f}(3) = .75 - .25i$.

5 Variants

The Cosine Transform

In signal processing applications, the function $f(\omega^t)$ might represent a signal like the amplitude of a sound sampled at times $t = 0$, 1, up to $N - 1$:

$$\text{signal at time } t = f(\omega^t) = \hat{f}(0) + \hat{f}(1)\omega^t + \cdots + \hat{f}(N-1)\omega^{t(N-1)}, \quad \omega = e^{2\pi i/n}.$$

There are two annoyances with this representation. First, even though the signal is real-valued, the Fourier coefficients and the basis functions are complex. Second, the function $f(\omega^t)$ is periodic modulo n, but signals like sound waves are not naturally periodic. It is likely that there will be a discontinuity when the signal wraps around from $t = N - 1$ to $t = N \equiv 0$ (see Figure 1). Discontinuities create unnatural artifacts in the Fourier expansion.

There is a beautiful trick that eliminates both problems. Before taking its Fourier transform, concatenate the signal with its mirror image (see Figure 1). All the Fourier coefficients become real, and all basis functions reduce to their real part, which is a vector of cosines:

$$g(\omega^t) = 2\hat{g}(0) + 4\hat{g}(1) \cdot \cos\Big(\frac{\pi t}{2N}\Big) + 4\hat{g}(1) \cdot \cos\Big(\frac{3\pi t}{2N}\Big) + \cdots + 4\hat{g}(N-1) \cdot \cos\Big(\frac{(2N-1)\pi t}{2N}\Big),$$

where $\omega = e^{\pi it/2N}$ and t is odd modulo $4N$. The index j in $\hat{g}(j)$ represents the *frequency* of the corresponding cosine wave (see Figure 2). The low-frequency parts (small js) capture the stable part of the signal, while the high-frequency ones (large js) capture the oscillating part.

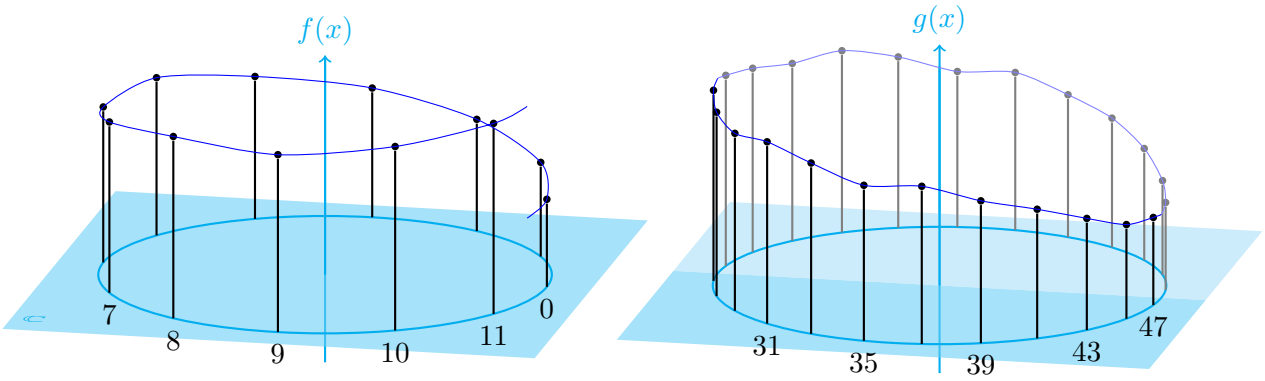


FIGURE 1: The cosine transform. (a) Evaluations of $f(e^{2\pi it/12})$, $t = 0, \ldots, 11$. The signal exhibits a discontinuity as it wraps from $t = 11$ to $t = 12 \equiv 0$. (b) g is obtained by concatenating f with its mirror image and shifting it so that it is symmetric about the real axis. The discontinuities are smoothed out and g is symmetric with respect to complex conjugation ($g(\overline{x}) = g(x)$).

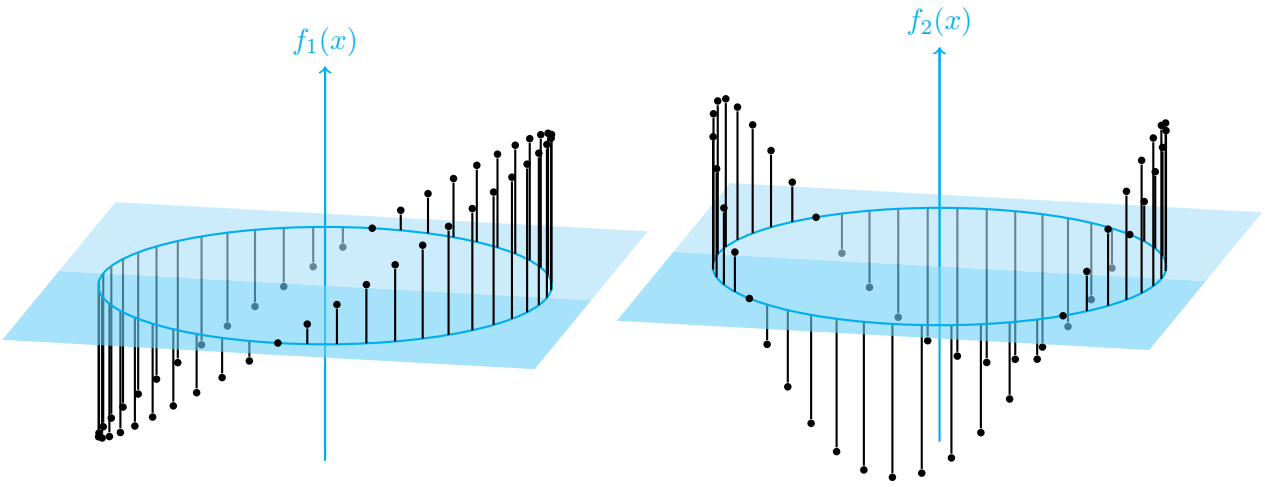


FIGURE 2: The basis functions (a) $g_1(\omega^t) = \cos(\pi t/2N)$ and (b) $g_2(\omega^t) = \cos(2\pi t/2N)$.

Two-dimensional transforms

There are Fourier transforms that apply to two and more-dimensional signals like images. They are constructed by taking the "product" of one-dimensional Fourier transfoms. Suppose we know a Fourier representation for functions in variable x and we also know a Fourier representation for functions in y. Then functions in x and y have a unique representation

$$f(x, y) = \sum_{i,j} \hat{f}(i, j) x^i y^j.$$

The embeddings of the monomials $x^i \cdot y^j$ are obtained by taking the outer product of the embeddings of x^i and y^j individually: the (s, t)-th entry of the vector representing $x^i y^j$ is the product of the s-th entry of x^i times the t-th entry of y^j. The Fourier coefficients are calculated by a formula analogous to (11): $\hat{f}(i, j)$ is the average of $f(x, y)x^{-i}y^{-j}$ for x, y ranging over their respective domains.

If x and y range over the N-th and M-th roots of unity, respectively, the Fourier representation of $f(x, y)$ has the form

$$\text{signal at position } (s, t) = f(e^{2\pi is/N}, e^{2\pi it/M}) = \sum_{j,k} \hat{f}(j, k) e^{2\pi i(sj/N + tk/M)}$$

with j ranging from 0 to $N-1$ and k ranging from 0 to $M-1$. The Fourier coefficient $\hat{f}(j,k)$ is obtained by averaging $f(e^{2\pi is/N}, e^{2\pi it/M})e^{-2\pi i(sj/N+tk/M)}$ over $s \in \{0,\ldots,N-1\}$ and $t \in \{0,\ldots,M-1\}$.

The Fourier-Walsh transform over the Boolean cube $\{-1,1\}^n$ is compatible with this construction: It is the n-th power of the Fourier transform for functions on a single bit $\{-1,1\}$.

Continuous transforms

It is sometimes useful to think of the signal as a continuous function. The amplitude of a sound wave can in principle be measured at any continuous instant t. Discrete-time approximations that are amenable to data processing can be obtained by sampling the signal at regularly spaced points. The more frequent the sampling, the more precise the approximation is. We would expect the same to be true for its Fourier transform.

Indeed, as N gets larger and larger, the Fourier transform modulo N approaches a continuous Fourier transform. Its inputs are functions f that take values on the continuous unit circle. Any "reasonable" f of this type can be expanded as a possibly infinite *Fourier series* $f(\omega) = \sum \hat{f}(j)\omega^j$. Here ω denotes any point on the complex unit circle and j ranges over all integers. The Fourier coefficient $\hat{f}(j)$ is the average of $f(\omega)\omega^{-j}$ with ω sampled uniformly from the unit circle.

Continuous Fourier representations are insightful for understanding properties like sensitivity to continuous noise: By how much does the value of f change typically if its input is perturbed by a small amount? Just like for functions over the Boolean cube, the higher frequency terms in the Fourier expansion yield more precise but also more complex approximations of f.

For "standard" mathematical functions f like the pulse and the Gaussian curve the Fourier coefficients can be calculated exactly via integration, often leading to insightful infinite series representations.

6 Quantum Fourier sampling

A quantum computer is an imaginary computational device. It is believed that if one is ever built, it will be dramatically more effective than any existing, classical computer for certain types of problems. Much of this excitement comes from the conjectured ability of quantum computers to "perform" Fourier transforms exponentially faster than classical computers can.

The Fast Fourier Transform algorithm is already very efficient. It computes a Fourier Transform of a function with N values in time $O(N \log N)$. Intuitively no algorithm can do better than time N because the least it needs to do it inspect all values of f. Quantum algorithms, just like classical ones, are subject to this limitation. There is little room for improvement in algorithms for *computing* the Fourier transform. (Having said that, the discovery of a linear-time classical or quantum algorithm for the Fourier Transform would be a tremendous breakthrough!)

There is a different problem related to Fourier transforms called Fourier Sampling. The starting point is Parseval's identity (7). Assume f is a function whose average square value is 1. Boolean ± 1-valued functions are like that. Then the squared Fourier coefficients specify a probability distribution over subsets S of $\{1,\ldots,n\}$: Set S is picked with probability $\hat{f}(S)^2$. For example, the distribution induced by maj_3 is uniform over the sets $\{1\}$, $\{2\}$, $\{3\}$, and $\{1,2,3\}$.

Fourier Sampling: Given the list representation of f, output set S with probability $\hat{f}(S)^2$.

We would still expect that to solve Fourier Sampling, a computer would have to at the least inspect all N values of f. This is indeed the case for classical computers, but no longer true for quantum computers. The Quantum Fourier Sampling algorithm solves the problem using $\log N$ elementary quantum steps!

It may be difficult to understand this exponential speedup without talking about quantum computers in detail but let me give a try. The state of a classical computer is determined by the contents of its memory. A computer with n bits of memory must be in one of the 2^n possible memory states. For example if $n = 2$ the possible states are 00, 01, 10, and 11. In contrast, a quantum computer can be in a *superposition* of

these states. Mathematically, a superposition is any vector of unit norm in the 4-dimensional vector space spanned by the standard basis vectors labeled by the four memory states. These four vectors are usually denoted by $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ in quantum computing books. In general, a (pure) quantum state of an n-qubit quantum computer is a superposition of the 2^n standard basis states $|00\cdots0\rangle, |00\cdots1\rangle, |11\cdots1\rangle$.

Given a list representation of f, a quantum computer can create the quantum superposition state $|s\rangle = \frac{1}{2}(f(++)|00\rangle + f(-+)|10\rangle + f(+-)|01\rangle + f(--)|11\rangle)$ in one step. This is a valid quantum state, namely a unit vector, because its length is precisely the squared average of f. In general, the quantum computer should start in state

$$|s\rangle = \frac{1}{\sqrt{N}} \cdot \sum_{x \in \{\pm1\}^n} f(x)|x \text{ in 0/1 representation}\rangle \tag{15}$$

Suppose it could process this state into the Fourier basis state $|t\rangle = \hat{f}(\varnothing)|00\rangle + \hat{f}(\{1\})|10\rangle + \hat{f}(\{2\})|01\rangle + \hat{f}(\{1,2\})|11\rangle$. Quantum physics tells us that the desired outcome—sample S with probability $\hat{f}(S)^2$—will be achieved merely by *observing* the quantum state: A measurement of a quantum state collapses it to a single (observable) outcome with probability proportional to the square of the coefficient in front of it (the amplitude).

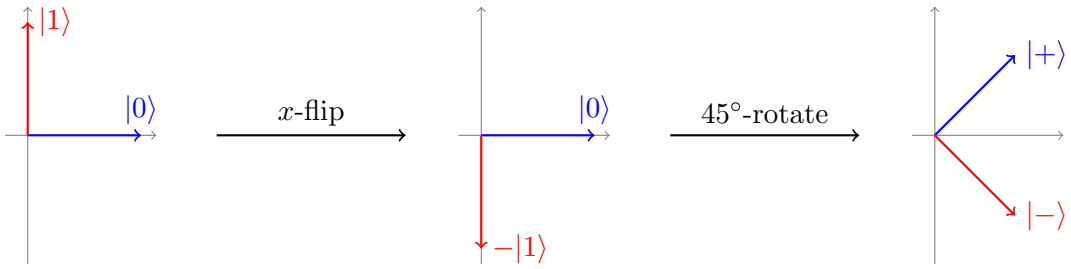
What remains to do is to instruct the quantum computer to move from state $|s\rangle$ to state $|t\rangle$. By the product nature of the Fourier transform, this can be achieved by moving every qubit independently. (In linear algebra language, the n-dimensional Fourier transform is the n-th tensor power of the one-dimensional Fourier transform.) When $n = 1$, this means transitioning from the list-of-values basis

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

to the Fourier basis

$$|+\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}+1\\+1\end{bmatrix} \qquad |-\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}+1\\-1\end{bmatrix}.$$

The instructions of a quantum computer are *unitary transformations*, namely linear transformations that preserve unit length. Examples include rotations and flips. The desired change of basis can be effected by a flip about the x-axis followed by a 45 degree rotation:



The algebraic specification of the instruction is multiplication by the *Walsh-Hadamard matrix*

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}+1 & +1\\+1 & -1\end{bmatrix}.$$

Algorithm QFS (Quantum Fourier Sampling)
Input: List representation of $f\colon \{\pm1\}^n \to \{\pm1\}$.
1 Prepare the quantum start state (15).
2 Apply H to each of the n qubits.
3 Measure the state and report the outcome.

A variant of Quantum Fourier Sampling works for the modular Fourier transform. One important application of it is period finding. Suppose the input function f is periodic modulo some unknown number q. The period shows in the Fourier transform of f: Most of its "weight" is concentrated around multiples of N/q, provided q is sufficiently smaller than N. The output of Quantum Fourier Sampling on such functions is likely to be very close to some multiple of N/q. It is possible to find a good approximation of q itself from a few such samples.

The most spectacular application of unknown period recovery is factoring. Given a product $N = pq$ of two primes it is possible to cook up a function that is almost periodic modulo one of the prime factors of N. Quantum Fourier Sampling then recovers one of the prime factors of N up to some small error. In 1994 Peter Shor showed how the error can be eliminated. His discovery dramatically raised the stakes in quantum computing because the ability to factor efficiently would break most existing encryption infrastructure.

Despite enormous well-funded efforts since then, a quantum computer is not yet in the cards. Many scientists believe it will one day be built. This quantum computer will be able to run the precise instructions required by algorithms like Quantum Fourier Sampling on inputs of arbitrarily large scale.

There are also researchers who conjecture there are fundamental obstacles that forbid the physical realization of quantum computers. Mathematician Gil Kalai is among the most outspoken ones. His main argument is that quantum systems are too susceptible to noise to do anything useful. How does he argue his case? You guessed it—the proof is in the Fourier Transform.