The classical and quantum Fourier transform

Ronald de Wolf

February 22, 2011

1 The classical Fourier transform

1.1 The discrete Fourier transform

The Fourier transform occurs in many different versions throughout classical computing, in areas ranging from signal-processing to data compression to complexity theory.

For our purposes, the Fourier transform is going to be an $N \times N$ unitary matrix, all of whose entries have the same magnitude. For $N = 2$, it's just our familiar Hadamard transform:

$$F_2 = H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Doing something similar in 3 dimensions is impossible with real numbers: we can't give three orthogonal vectors in $\{+1, -1\}^3$. However, using *complex* numbers[1] allows us to define the Fourier transform for any N. Let $\omega_N = e^{2\pi i/N}$ be an N-th root of unity ("root of unity" means that $\omega_N^k = 1$ for some integer k, in this case $k = N$). The rows of the matrix will be indexed by $j \in \{0, \ldots, N-1\}$ and the columns by $k \in \{0, \ldots, N-1\}$. and define the (j,k)-entry of the matrix F_N by $\frac{1}{\sqrt{N}}\omega_N^{jk}$:

$$F_N = \frac{1}{\sqrt{N}} \begin{pmatrix} & \vdots & \\ \cdots & \omega_N^{jk} & \cdots \\ & \vdots & \end{pmatrix}$$

Note that F_N is a unitary matrix, since each column has norm 1, and any pair of columns (say those indexed by k and k') is orthogonal:

$$\sum_{j=0}^{N-1} \frac{1}{\sqrt{N}}(\omega_N^{jk})^* \frac{1}{\sqrt{N}}\omega_N^{jk'} = \frac{1}{N}\sum_{j=0}^{N-1} \omega_N^{j(k'-k)} = \begin{cases} 1 & \text{if } k = k' \\ 0 & \text{otherwise} \end{cases}$$

Since F_N is unitary and symmetric, the inverse $F_N^{-1} = F_N^*$ only differs from F_N by having minus signs in the exponent of the entries. For a vector $v \in \mathbb{R}^N$, the vector $\widehat{v} = F_N v$ is called the Fourier transform of v.[2] Doing the matrix-vector multiplication, its entries are given by $\widehat{v}_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{jk} v_k$.

[1] A complex number is of the form $c = a + bi$, where $a, b \in \mathbb{R}$, and i is the imaginary unit, which satisfies $i^2 = -1$. Such a c can also be written as $c = re^{i\phi}$ where $r = |c| = \sqrt{a^2 + b^2}$ is the *magnitude* of c, and $\phi \in [0, 2\pi)$ is the angle that c makes with the positive horizontal axis when we view it as a point (a, b) in the plane. Note that complex numbers of magnitude 1 lie on the unit circle in this plane. We can also write those as $e^{i\phi} = \cos(\phi) + i\sin(\phi)$. The complex conjugate c^* is $a - ib$, equivalently $c^* = re^{-i\phi}$.

[2] The literature on Fourier analysis usually talks about the Fourier transform of a *function* rather than of a vector,

1.2 The Fast Fourier Transform

The naive way of computing the Fourier transform $\widehat{v} = F_N v$ of $v \in \mathbb{R}^N$ just does the matrix-vector multiplication to compute all the entries of $\widehat{v}$. This would take $O(N)$ steps (additions and multiplications) per entry, and $O(N^2)$ steps to compute the whole vector $\widehat{v}$. However, there is a more efficient way of computing $\widehat{v}$. This algorithm is called the *Fast Fourier Transform* (FFT), and takes only $O(N \log N)$ steps. This difference between the quadratic N^2 steps and the near-linear $N \log N$ is tremendously important in practice, and is the main reason that Fourier transforms are so widely used.

We will assume $N = 2^n$, which is usually fine because we can add zeroes to our vector to make its dimension a power of 2 (but similar FFTs can be given also directly for most N that aren't a power of 2). The key to the FFT is to rewrite the entries of $\widehat{v}$ as follows:

$$\begin{aligned}
\widehat{v}_j &= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{jk} v_k \\
&= \frac{1}{\sqrt{N}} \left(\sum_{\text{even } k} \omega_N^{jk} v_k + \sum_{\text{odd } k} \omega_N^{jk} v_k \right) \\
&= \frac{1}{\sqrt{2}} \left(\frac{1}{\sqrt{N/2}} \sum_{\text{even } k} \omega_{N/2}^{jk/2} v_k + \frac{\omega_N^j}{\sqrt{N/2}} \sum_{\text{odd } k} \omega_{N/2}^{j(k-1)/2} v_k \right)
\end{aligned}$$

Note that we've rewritten the entries of the N-dimensional Fourier transform $\widehat{v}$ in terms of two $N/2$-dimensional Fourier transforms, one of the even-numbered entries of v, and one of the odd-numbered entries of v.

This suggest a recursive procedure for computing $\widehat{v}$: first separately compute the Fourier transform $\widehat{v_{\text{even}}}$ of the $N/2$-dimensional vector of even-numbered entries of v and the Fourier transform $\widehat{v_{\text{odd}}}$ of the $N/2$-dimensional vector of odd-numbered entries of v, and then compute the N entries

$$\widehat{v}_j = \frac{1}{\sqrt{2}} (\widehat{v_{\text{even}}}_j + \omega_N^j \widehat{v_{\text{odd}}}_j).$$

Strictly speaking this is not well-defined, because $\widehat{v_{\text{even}}}$ and $\widehat{v_{\text{odd}}}$ are just $N/2$-dimensional vectors. However, if we define $\widehat{v_{\text{even}}}_{j+N/2} = \widehat{v_{\text{even}}}_j$ (and similarly for $\widehat{v_{\text{odd}}}$) then it all works out.

The time $T(N)$ it takes to implement F_N this way can be written recursively as $T(N) = 2T(N/2) + O(N)$, because we need to compute two $N/2$-dimensional Fourier transforms and do $O(N)$ additional operations to compute $\widehat{v}$. This works out to time $T(N) = O(N \log N)$, as promised. Similarly, we have an equally efficient circuit for the *inverse* Fourier transform $F_N^{-1} = F_N^*$.

1.3 Application: multiplying two polynomials

Suppose we are given two real-valued polynomials p and q, each of degree at most d:

$$p(x) = \sum_{j=0}^{d} a_j x^j \text{ and } q(x) = \sum_{k=0}^{d} b_k x^k$$

but on finite domains that's just a notational variant of what we do here: a vector $v \in \mathbb{R}^N$ can also be viewed as a function $v : \{0, \ldots, N-1\} \to \mathbb{R}$ defined by $v(i) = v_i$. Also, in the classical literature people sometimes use the term "Fourier transform" for what we call the inverse Fourier transform.

We would like to compute the product of these two polynomials, which is

$$(p \cdot q)(x) = \left(\sum_{j=0}^{d} a_j x^j\right)\left(\sum_{k=0}^{d} b_k x^k\right) = \sum_{\ell=0}^{2d} (\underbrace{\sum_{j=0}^{2d} a_j b_{\ell-j}}_{c_\ell}) x^\ell,$$

where implicitly we set $b_{\ell-j} = 0$ if $j > \ell$. Clearly, each coefficient c_ℓ by itself takes $O(d)$ steps (additions and multiplications) to compute, which suggests an algorithm for computing the coefficients of $p \cdot q$ that takes $O(d^2)$ steps. However, using the fast Fourier transform we can do this in $O(d \log d)$ steps, as follows.

The *convolution* of two vectors $a, b \in \mathbb{R}^N$ is a vector $a * b \in \mathbb{R}^N$ whose ℓ-th entry is defined by $(a*b)_\ell = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} a_j b_{\ell-j \bmod N}$. Let us set $N = 2d+1$ (the number of nonzero coefficients of $p \cdot q$) and make the $(d+1)$-dimensional vectors of coefficients a and b N-dimensional by adding d zeroes. Then the coefficients of the polynomial $p \cdot q$ are proportional to the entries of the convolution: $c_\ell = \sqrt{N}(a*b)_\ell$. It is easy to show that the Fourier coefficients of the convolution of a and b are the products of the Fourier coefficients of a and b: for every $\ell \in \{0, \ldots, N-1\}$ we have $\left(\widehat{a * b}\right)_\ell = \widehat{a}_\ell \cdot \widehat{b}_\ell$. This immediately suggests an algorithm for computing the vector of coefficients c_ℓ: apply the FFT to a and b to get $\widehat{a}$ and $\widehat{b}$, multiply those two vectors entrywise to get $\widehat{a * b}$, apply the inverse FFT to get $a*b$, and finally multiply $a*b$ with $\sqrt{N}$ to get the vector c of the coefficients of $p \cdot q$. Since the FFTs and their inverse take $O(N \log N)$ steps, and pointwise multiplication of two N-dimensional vectors takes $O(N)$ steps, this whole algorithm takes $O(N \log N) = O(d \log d)$ steps.

Note that if two numbers $a_d \cdots a_1 a_0$ and $b_d \cdots b_1 b_0$ are given in decimal notation, then we can interpret the digits as coefficients of polynomials p and q, respectively, and the two numbers will be $p(10)$ and $q(10)$. Their product is the evaluation of the product-polynomial $p \cdot q$ at the point $x = 10$. This suggests that we can use the above procedure (for fast multiplication of polynomials) to multiply two numbers in $O(d \log d)$ steps, which would be a lot faster than the standard $O(d^2)$ algorithm for multiplication that one learns in primary school. However, in this case we have to be careful since the steps of the above algorithm are themselves multiplications between numbers, which we cannot count at unit cost anymore if our goal is to implement a multiplication between numbers! Still, it turns out that implementing this idea carefully allows one to multiply two d-digit numbers in $O(d \log d \log \log d)$ elementary operations. This is known as the Schönhage-Strassen algorithm. We'll skip the details.

2 The quantum Fourier transform

Since F_N is an $N \times N$ unitary matrix, we can interpret it as a quantum operation, mapping an N-dimensional vector of amplitudes to another N-dimensional vector of amplitudes. This is called the quantum Fourier transform (QFT). In case $N = 2^n$ (which is the only case we will care about), this will be an n-qubit unitary. Notice carefully that this quantum operation does something different from the classical Fourier transform: in the classical case we are given a vector v, written on a piece of paper so to say, and we compute the vector $\widehat{v} = F_N v$, and also write the result on a piece of paper. In the quantum case, we are working on *quantum states*; these are vectors of amplitudes, but we don't have those written down anywhere—they only exist as the amplitudes in a superposition. We will see below that the QFT can be implemented by a quantum circuit using $O(n^2)$ elementary

gates. This is exponentially faster than even the FFT (which takes $O(N \log N) = O(2^n n)$ steps), but it achieves something different: computing the QFT won't give us the entries of the Fourier transform written down on a piece of paper, but only as the amplitudes of the resulting state.

2.1 An efficient quantum circuit

Here we will describe the efficient circuit for the n-qubit QFT. The elementary gates we will allow ourselves are Hadamards and controlled-R_s gates, where $R_s = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^s} \end{pmatrix}$. Note that $R_1 = Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ and $R_2 = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$. Since the QFT is linear, it suffices if our circuit implements it correctly on basis states $|k\rangle$, i.e., it should map

$$|k\rangle \mapsto F_N|k\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \omega_N^{jk} |j\rangle.$$

The key to doing this efficiently is to rewrite $F_N|k\rangle$, which turns out to be a *product state*. Let $|k\rangle = |k_1 \dots k_n\rangle$ (k_1 being the most significant bit). Note that for integer $j = j_1 \dots j_n$, we can write $j/2^n = \sum_{\ell=1}^{n} j_\ell 2^{-\ell}$. For example, binary 0.101 is $1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 5/8$. We have

$$\begin{aligned}
F_N|k\rangle &= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i jk/2^n} |j\rangle \\
&= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i (\sum_{\ell=1}^{n} j_\ell 2^{-\ell})k} |j_1 \dots j_n\rangle \\
&= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \prod_{\ell=1}^{n} e^{2\pi i j_\ell k/2^\ell} |j_1 \dots j_n\rangle \\
&= \bigotimes_{\ell=1}^{n} \frac{1}{\sqrt{2}} \left(|0\rangle + e^{2\pi i k/2^\ell} |1\rangle \right).
\end{aligned}$$

Note that $e^{2\pi i k/2^\ell} = e^{2\pi i 0.k_\ell \dots k_n}$: the $\ell - 1$ most significant bits of k don't matter for this.

As an example, for $n = 3$ we have the 3-qubit product state

$$F_8|k_1k_2k_3\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.k_3}|1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.k_2k_3}|1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.k_1k_2k_3}|1\rangle).$$

This example suggests what the circuit should be. To prepare the first qubit of the desired state $F_8|k_1k_2k_3\rangle$, just apply a Hadamard to $|k_3\rangle$, giving state $\frac{1}{\sqrt{2}}(|0\rangle + (-1)^{k_3}|1\rangle)$ and observe that $(-1)^{k_3} = e^{2\pi i 0.k_3}$. To prepare the second qubit of the desired state, apply a Hadamard to $|k_2\rangle$, giving $\frac{1}{\sqrt{3}}(|0\rangle + e^{2\pi i 0.k_2}|1\rangle)$, and then conditioned on k_3 (before we apply the Hadamard to $|k_3\rangle$) apply R_2. This multiplies $|1\rangle$ with a phase $e^{2\pi i 0.0k_3}$, producing the correct qubit $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.k_2k_3}|1\rangle)$. Finally, to prepare the third qubit of the desired state, we apply a Hadamard to $|k_1\rangle$, apply R_2 conditioned on k_2 and R_3 conditioned k_3. This produces the correct qubit $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.k_1k_2k_3}|1\rangle)$. We have now produced all three qubits of the desired state $F_8|k_1k_2k_3\rangle$, *but in the wrong order*: the

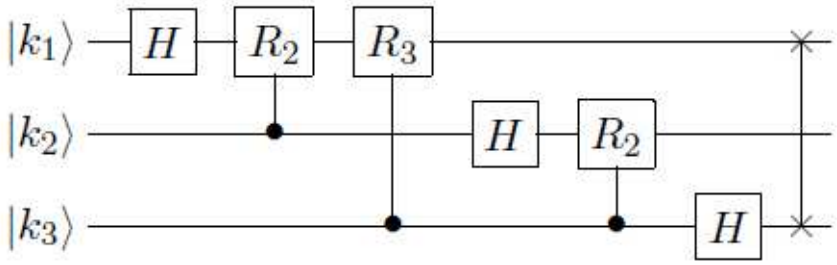

Figure 1: The circuit for the 3-qubit QFT

first qubit should be the third and vice versa. So the final step is just to swap qubits 1 and 3. The picture illustrates the circuit in the case $n = 3$. The general case works analogously: starting with $\ell = 1$, we apply a Hadamard to $|k_\ell\rangle$ and then "rotate in" the additional phases required, conditioned on the values of the later bits $k_{\ell+1} \dots k_n$. Some swap gates at the end then put the qubits in the right order.

Since the circuit involves n qubits, and at most n gates are applied to each qubit, the overall circuit uses at most n^2 gates. In fact, many of those gates are phase gates R_s with $s \gg \log n$, which are very close to the identity and hence don't do much anyway. We can actually omit those from the circuit, keeping only $O(\log n)$ gates per qubit and $O(n \log n)$ gates overall. Intuitively, the overall error caused by these omissions will be small (a homework exercise asks you to make this precise). Finally, note that by inverting the circuit (i.e., reversing the order of the gates and taking the adjoint U^* of each gate U) we obtain an equally efficient circuit for the inverse Fourier transform $F_N^{-1} = F_N^*$.