14.1 Complexity of the DFT

Let us recall the previously derived formula (4.32) for the N-point DFT

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk}$$

where the N^{th} root of unity is defined as

$$W_N \equiv e^{-\mathrm{i}\frac{2\pi}{N}} = \cos\left(\frac{2\pi}{N}\right) - \mathrm{i}\,\sin\left(\frac{2\pi}{N}\right)$$

How many calculations must we perform to find one X_k from a set of N time domain values x_n? Assume that the complex constant W_N and its powers W_N^j have all been precalculated and stored for our use. Looking closely at (4.32) we see N complex multiplications and $N-1$ complex additions are performed in the loop on n. Now to find the *entire* spectrum we need to do N such calculations, one for each value of k. So we expect to have to carry out N^2 complex multiplications, and $N(N-1)$ complex additions.

This is actually a slight overestimate. By somewhat trickier programming we can take advantage of the fact that $W_N^0 = 1$, so that each of the $X_{k>0}$ takes $N-1$ multiplications and additions, while X_0 doesn't require any multiplications. We thus really need only $(N-1)^2$ complex multiplications.

A complex addition requires the addition of real and imaginary parts, and is thus equivalent to two real additions. A complex multiplication can be performed as four real multiplications and two additions $(a+\mathrm{i}b)(c+\mathrm{i}d) = (ac-bd) + \mathrm{i}(bc+ad)$ or as three multiplications and five additions $(a+\mathrm{i}b)(c+\mathrm{i}d) = a(c+d) - d(a+b) + \mathrm{i}\left(a(c+d) + c(b-a)\right)$. The latter form may be preferred when multiplication takes much more time than addition, but can be less stable numerically. Other combinations are possible, but it can be shown that there is no general formula for the complex product with less than three multiplications. Using the former, more common form, we find that the computation of the entire spectrum requires $4(N-1)^2$ real multiplications and $2(N-1)(2N-1)$ real additions.

Actually, the calculation of a single X_k can be performed more efficiently than we have presented so far. For example, Goertzel discovered a method of transforming the iteration in equation (4.32) into a recursion. This has the effect of somewhat reducing the computational complexity and also saves the precomputation and storage of the W table. Goertzel's algorithm, to be presented in Section 14.8, still has asymptotic complexity of order $O(N)$ per calculated X_k, although with a somewhat smaller constant than the direct method. It thus leaves the complexity of calculation of the entire spectrum at $O(N^2)$, while the FFT algorithms to be derived presently are less complex. Goertzel's algorithm thus turns out to be attractive when a single, or only a small number of X_k values are needed, but is not the algorithm of choice for calculating the entire spectrum.

Returning to the calculation of the entire spectrum, we observe that both the additions and multiplications increase as $O(N^2)$ with increasing N. Were this direct calculation the only way to find the DFT, real-time calculation of large DFTs would be impractical. It is general rule in DSP programming that only algorithms with linear asymptotic complexity can be performed in real-time for large N. Let us now see why this is the case.

The criterion for real-time calculation is simple for algorithms that process a single input sample at a time. Such an algorithm must finish all its computation for each sample before the next one arrives. This restricts the number of operations one may perform in such computations to the number performable in a sample interval t_s. This argument does not directly apply to the DFT since it is inherently a block-oriented calculation. One cannot perform a DFT on a single sample, since frequency is only defined for signals that occupy some nonzero interval of time; and we often desire to process large blocks of data since the longer we observe the signal the more accurate frequency estimates will be.

For block calculations one accumulates samples in an array, known as a buffer, and then processes this buffer as a single entity. A technique known as *double-buffering* is often employed in real-time implementations of block calculations. With double-buffering two buffers are employed. While the samples in the *processing buffer* are being processed, the *acquisition buffer* is acquiring samples from the input source. Once processing of the first buffer is finished and the output saved, the buffers are quickly switched, the former now acquiring samples and the latter being processed.

How can we tell if block calculations can be performed in real-time? As for the single sample case, one must be able to finish all the processing needed in time. Now 'in time' means completing the processing of one entire buffer, before the second buffer becomes full. Otherwise a condition known as *data-overrun* occurs, and new samples overwrite previously stored, but as yet unprocessed, ones. It takes $N\Delta t$ seconds for N new samples to arrive. In order to keep up we must process all N old samples in the processing buffer before the acquisition buffer is completely filled. If the complexity is linear (i.e., the processing time for N samples is proportional to N), then $C = qN$ for some q. This q is the *effective time per sample* since each sample effectively takes q time to process, independent of N. Thus, the selection of buffer size is purely a memory issue, and does not impact the ability to keep up with real-time. However, if the complexity is superlinear (for example, $T_{processing} = qN^p$ with $p > 1$), then as N increases we have less and less time to process each sample, until eventually some N is reached where we can no longer keep up, and data-overrun is inevitable.

Let's clarify this by plugging in some numbers. Assume we are acquiring input at a sample rate of 1000 samples per second (i.e., we obtain a new sample every millisecond) and are attempting to process blocks of length 250. We start our processor, and for one-quarter of a second, we cannot do any processing, until the first acquisition buffer fills. When the buffer is full we quickly switch buffers, start processing the 250 samples collected, while the second buffer of length 250 fills. We must finish the processing within a quarter of second, in order to be able to switch buffers back when the acquisition buffer is full. When the dependence of the processing time on buffer length is strictly linear, $T_{processing} = qN$, then if we can process a buffer of $N = 250$ samples in 250 milliseconds or less, we can equally well process a buffer of 500 samples in 500 milliseconds, or a buffer of $N = 1000$ samples in a second. Effectively we can say that when the single sample processing time is no more than $q = 1$ millisecond per sample, we can maintain real-time processing.

What would happen if the buffer processing time depended quadratically on the buffer size—$T_{processing} = qN^2$? Let's take q to be 0.1 millisecond per sample squared. Then for a small 10-millisecond buffer (length $N = 10$), we will finish processing in $T_{processing} = 0.1 \cdot 10^2 = 10$ milliseconds, just in time! However, a 100-millisecond buffer of size $N = 100$ will require $T_{processing} = 0.1 \cdot 100^2$ milliseconds, or one second, to process. Only by increasing our computational power by a factor of ten would we be able to maintain real-time! However, even were we to increase the CPU power to accommodate this buffer-size, our 250-point buffer would still be out of our reach.

As we have mentioned before, the FFT is an algorithm for calculating the DFT more efficiently than quadratically, at least for certain values of N. For example, for powers of two, $N = 2^k$, its complexity is $O(N \log_2 N)$. This is only very slightly superlinear, and thus while *technically* the FFT is not suitable for real-time calculation in the asymptotic $N \to \infty$ limit, in practice it *is* computable in real-time even for relatively large N. To grasp the speed-up provided by the FFT over direct calculation of (4.29), consider that the ratio between the complexities is proportional to $\frac{N}{\log_2 N}$. For $N = 2^4 = 16$ the FFT is already four times faster than the direct DFT, for $N = 2^{10} = 1024$

it is over one hundred times faster, and for $N = 2^{16}$ the ratio is 4096! It is common practice to compute $1K$- or $64K$-point FFTs in real-time, and even much larger sizes are not unusual.

The basic idea behind the FFT is the very exploitation of the N^2 complexity of the direct DFT calculation. Due to this second-order complexity, it is faster to calculate a lot of small DFTs than one big one. For example, to calculate a DFT of length N will take N^2 multiplications, while the calculation of two DFTs of length $\frac{N}{2}$ will take $2(\frac{N}{2})^2 = \frac{N^2}{2}$, or half that time. Thus if we can somehow piece the two partial results together to one spectrum in less than $\frac{N^2}{2}$ time then we have found a way to save time. In Sections 14.3 and 14.4 we will see several ways to do just that.

EXERCISES

14.1.1 Finding the maximum of an N-by-N array of numbers can be accomplished in $O(N^2)$ time. Can this be improved by partitioning the matrix and exploiting the quadratic complexity as above?

14.1.2 In exercise 4.7.4 you found explicit equations for the $N = 4$ DFT for $N = 4$. Count up the number of *complex* multiplications and additions needed to compute X_0, X_1, X_2, and X_3. How many *real* multiplications and additions are required?

14.1.3 Define temporary variables that are used more than once in the above equations. How much can you save? How much memory do you need to set aside? (Hint: Compare the equations for X_0 and X_2.)

14.1.4 Up to now we have not taken into account the task of finding the trigonometric W factors themselves, which can be computationally intensive. Suggest at least two solutions, one that requires a large amount of auxiliary memory but practically no CPU, and one that requires little memory but is more CPU intensive.

14.1.5 A computational system is said to be 'real-time-oriented' when the time it takes to perform a task can be guaranteed. Often systems rely on the weaker criterion of *statistical real-time*, which simply means that on-the-average enough computational resources are available. In such cases double buffering can be used in the acquisition hardware, in order to compensate for peak MIPS demands. Can hardware buffering truly make an arbitrary system as reliable as a real-time-oriented one?

14.1.6 Explain how double-buffering can be implemented using a single circular buffer.

14.2 Two Preliminary Examples

Before deriving the FFT we will prepare ourselves by considering two somewhat more familiar examples. The ideas behind the FFT are very general and not restricted to the computation of equation (14.1). Indeed the two examples we use to introduce the basic ideas involve no DSP at all.

How many comparisons are required to find the maximum or minimum element in a sequence of N elements? It is obvious that N–1 comparisons are absolutely needed if all elements are to be considered. But what if we wish to simultaneously find the maximum *and* minimum? Are twice this number really needed? We will now show that we can get away with only $1\frac{1}{2}$ times the number of comparisons needed for the first problem. Before starting we will agree to simplify the above number of comparisons to N, neglecting the 1 under the asymptotic assumption $N \gg 1$.

A fundamental tool employed in the reduction of complexity is that of splitting long sequences into smaller subsequences. How can we split a sequence with N elements

$$x_0,\ x_1,\ x_2,\ x_3,\ \ldots\ x_{N-2},\ x_{N-1}$$

into two subsequences of half the original size (assume for simplicity's sake that N is even)? One way is to consider pairs of adjacent elements, such as x_1, x_2 or x_3, x_4, and place the smaller of each pair into the first subsequence and the larger into the second. For example, assuming $x_0 < x_1$, $x_2 > x_3$ and $x_{N-2} < x_{N-1}$, we obtain

$$\begin{array}{llll} x_0\ x_3 & \ldots\ \min(x_{2l}, x_{2l+1})\ \ldots & x_{N-2} \\ x_1\ x_2 & \ldots\ \max(x_{2l}, x_{2l+1})\ \ldots & x_{N-1} \end{array}$$

This splitting of the sequence requires $\frac{N}{2}$ comparisons. Students of sorting and searching will recognize this procedure as the first step of the *Shell sort.*

Now, the method of splitting the sequence into subsequences guarantees that the minimum of the entire sequence must be one of the elements of the first subsequence, while the maximum must be in the second. Thus to complete our search for the minimum and maximum of the original sequence, we must find the minimum of the first subsequence and the maximum of the second. By our previous result, each of these searches requires $\frac{N}{2}$ comparisons. Thus the entire process of splitting and two searches requires $\frac{N}{2} + 2\frac{N}{2} = \frac{3N}{2}$ comparisons, or $1\frac{1}{2}$ times that required for the minimum or maximum alone.

Can we further reduce this factor? What if we divide the original sequence into adjacent quartets, choosing the minimum and maximum of the four? The splitting would then cost four comparisons per quartet, or N comparison altogether, and then two $\frac{N}{4}$ searches must be carried out. Thus we require $N + 2\frac{N}{4}$ and a factor of $1\frac{1}{2}$ is still needed. Indeed, after a little reflection, the reader will reach the conclusion that no further improvement is possible. This is because the new problems of finding only the minimum *or* maximum of a subsequence are simpler than the original problem.

When a problem *can* be reduced recursively to subproblems *similar* to the original, the process may be repeated to attain yet further improvement. We now discuss an example where such recursive repetition is possible. Consider multiplying two $(N+1)$-digit numbers A and B to get a product C using long multiplication (which from Section 6.8 we already know to be a convolution).

$$\begin{array}{cccccccc}
 & & & A_N & A_{N-1} & \cdots & A_1 & A_0 \\
 & & & B_N & B_{N-1} & \cdots & B_1 & B_0 \\
\hline
 & & & B_0A_N & B_0A_{N-1} & \cdots & B_0A_1 & B_0A_0 \\
 & & B_1A_N & B_1A_{N-1} & & \cdots & B_1A_0 & \\
 & & & & & \vdots & & \\
B_NA_N & \cdots & B_NA_1 & B_NA_0 & & & & \\
\hline
C_{2N} & \cdots & C_{N+1} & C_N & C_{N-1} & \cdots & C_1 & C_0
\end{array}$$

Since we must multiply every digit in the top number by every digit in the bottom number, the number of one-digit multiplications is N^2. You are probably used to doing this for decimal digits, but the same multiplication algorithm can be utilized for N-bit binary numbers. The hardware-level complexity of straightforward multiplication of two N-bit numbers is proportional to N^2.

Now assume N is even and consider the left $\frac{N}{2}$ digits and the right $\frac{N}{2}$ digits of A and B separately. It does not require much algebraic prowess to convince oneself that

$$\begin{aligned}
A &= A_L 2^{\frac{N}{2}} + A_R \\
B &= B_L 2^{\frac{N}{2}} + B_R \qquad (14.1)\\
C &= A_L B_L 2^N + (A_L B_R + A_R B_L) 2^{\frac{N}{2}} + A_R B_R \\
&= A_L B_L (2^N + 2^{\frac{N}{2}}) + (A_L - A_R)(B_R - B_L) 2^{\frac{N}{2}} + A_R B_R (2^{\frac{N}{2}} + 1)
\end{aligned}$$

involving only three multiplications of $\frac{N}{2}$-length numbers. Thus we have reduced the complexity from N^2 to $3(\frac{N}{2})^2 = \frac{3}{4}N^2$ (plus some shifting and adding operations). This is a savings of 25%, but does not reduce the asymptotic form of the complexity from $O(N^2)$. However, in this case we have only just begun! Unlike for the previous example, we have reduced the original multiplication problem to three similar but simpler multiplication problems!

We can now carry out the three $\frac{N}{2}$-bit multiplications in equation (14.1) similarly (assuming that $\frac{N}{2}$ is still even) and continue recursively. Assuming N to have been a power of two, we can continue until we multiply individual bits. This leads to an algorithm for multiplication of two N-bit numbers, whose asymptotic complexity is $O(N^{\log_2(3)}) \approx O(N^{1.585})$. The slightly more sophisticated Toom-Cook algorithm divides the N-bit numbers into more than two groups, and its complexity can be shown to be $O(N\log(N)2^{\sqrt{2\log(N)}})$. This is still not the most efficient way to multiply numbers. Realizing that each column sum of the long multiplication in equation (14.1) can be cast into the form of a convolution, it turns out that the best way to multiply large numbers is to exploit the FFT!

EXERCISES

14.2.1 The reader who has implemented the Shell sort may have used a different method of choosing the pairs of elements to be compared. Rather than comparing adjacent elements x_{2l} and x_{2l+1}, it is more conventional to consider elements in the same position in the first and second halves the sequence, x_k and $x_{\frac{N}{2}+k}$ Write down a general form for the new sequence. How do we find the minimum and maximum elements now? These two ways of dividing a sequence into two subsequences are called *decimation* and *partition.*

14.2.2 Devise an algorithm for finding the *median* of N numbers in $O(N\log N)$.

14.2.3 The product of two two-digit numbers, ab and cd, can be written $ab * cd = (10 * a + b) * (10 * c + d) = 100ac + 10(ad + bc) + bd$. Practice multiplying two-digit numbers in your head using this rule. Try multiplying a three-digit number by a two-digit one in similar fashion.

14.2.4 We often deal with complex-valued signals. Such signals can be represented as vectors in two ways, *interleaved*

$$\Re(x_1)\Im(x_1), \Re(x_2), \Im(x_2), \ldots \Re(x_N), \Im(x_N)$$

or *separated*

$$\Re(x_1)\Re(x_2), \ldots \Re(x_N), \Im(x_1), \Im(x_2), \ldots \Im(x_N)$$

Devise an efficient in-place algorithm changing between interleaved and separated representations. Efficient implies that each element accessed is moved immediately to its final location. In-place means here that if extra memory is used it must be of constant size (independent of N). What is the algorithm's complexity?