10.8: Numerical Accuracy in FFTs

An important consideration in the implementation of any practical numerical algorithm is numerical accuracy: how quickly do floating-point roundoff errors accumulate in the course of the computation? Fortunately, FFT algorithms for the most part have remarkably good accuracy characteristics. In particular, for a DFT of length \(n\) computed by a Cooley-Tukey algorithm with finite-precision floating-point arithmetic, the worst-case error growth is \(O(\log n)\) and the mean error growth for random inputs is only \(O(\sqrt{\log n})\). This is so good that, in practical applications, a properly implemented FFT will rarely be a significant contributor to the numerical error.

The amazingly small roundoff errors of FFT algorithms are sometimes explained incorrectly as simply a consequence of the reduced number of operations: since there are fewer operations compared to a naive \(O(n^2)\) algorithm, the argument goes, there is less accumulation of roundoff error. The real reason, however, is more subtle than that, and has to do with the ordering of the operations rather than their number. For example, consider the computation of only the output \(Y[0]\) in the radix-2 algorithm of Pre ignoring all of the other outputs of the FFT. \(Y[0]\) is the sum of all of the inputs, requiring \(n-1\) additions. The FFT does not change this requirement, it merely changes the order of the additions so as to re-use some of them for other outputs. In particular, this radix-2 DIT FFT computes \(Y[0]\) as follows: it first sums the even-indexed inputs, then sums the odd-indexed inputs, then adds the two sums; the even- and odd-indexed inputs are summed recursively by the same procedure. This process is sometimes called cascade summation, and even though it still requires \(n-1\) total additions to compute \(Y[0]\) by itself, its roundoff error grows much more slowly than simply adding \(X[0], X[1], X[2]\) and so on in sequence. Specifically, the roundoff error when adding up \(n\) floating-point numbers in sequence grows as \(O(n)\) in the worst case, or as \(O(\sqrt{n})\) on average for random inputs (where the errors grow according to a random walk), but simply reordering these \(n-1\) additions into a cascade summation yields \(O(\log n)\) worst-case and \(O(\sqrt{\log n})\) average-case error growth.

However, these encouraging error-growth rates only apply if the trigonometric “twiddle” factors in the FFT algorithm are computed very accurately. Many FFT implementations, including FFTW and common manufacturer-optimized libraries,
therefore use precomputed tables of twiddle factors calculated by means of standard library functions (which compute trigonometric constants to roughly machine precision). The other common method to compute twiddle factors is to use a trigonometric recurrence formula—this saves memory (and cache), but almost all recurrences have errors that grow as \(O(\sqrt{n})\) or even \(O(n^2)\) which lead to corresponding errors in the FFT. For example, one simple recurrence is \(e^{i(k+1)\theta} = e^{ik\theta} e^{i\theta}\), multiplying repeatedly by \(e^{i\theta}\) to obtain a sequence of equally spaced angles, but the errors when using this process grow as \(O(n)\). A common improved recurrence is \(e^{i(k+1)\theta} = e^{ik\theta} + e^{ik\theta}(e^{i\theta} - 1)\) where the small quantity \(12(e^{i\theta} - 1) = \cos(\theta) - 1 + i\sin(\theta)\) is computed using \(\cos(\theta) - 1 = -2\sin^2(\theta/2)\), unfortunately, the error using this method still grows as \(O(\sqrt{n})\) far worse than logarithmic.

There are, in fact, trigonometric recurrences with the same logarithmic error growth as the FFT, but these seem more difficult to implement efficiently; they require that a table of \(\Theta(\log n)\) values be stored and updated as the recurrence progresses. Instead, in order to gain at least some of the benefits of a trigonometric recurrence (reduced memory pressure at the expense of more arithmetic), FFTW includes several ways to compute a much smaller twiddle table, from which the desired entries can be computed accurately on the fly using a bounded number (usually <3) of complex multiplications. For example, instead of a twiddle table with \(\Omega(n)\) entries \(\omega(n^k)\), FFTW can use two tables with \(\Omega(\sqrt{n})\) entries each, so that \(\omega(n^k)\) is computed by multiplying an entry in one table (indexed with the low-order bits of \(k\)) by an entry in the other table (indexed with the high-order bits of \(k\)).

There are a few non-Cooley-Tukey algorithms that are known to have worse error characteristics, such as the “real-factor” algorithm but these are rarely used in practice (and are not used at all in FFTW). On the other hand, some commonly used algorithms for type-I and type-IV discrete cosine transforms have errors that we observed to grow as \(O(\sqrt{n})\) even for accurate trigonometric constants (although we are not aware of any theoretical error analysis of these algorithms), and thus we were forced to use alternative algorithms.

Footnote

12 In an FFT, the twiddle factors are powers of \(\omega(n)\), so \(\theta\) is a small angle proportional to \(1/n\) and \(e^{i\theta}\) is close to 1.

Contributor

• ContribEEBurrus