Rounding and Scaling in Fixed-Point FFT Implementations

P. Kabal and B. Sayar

INRS-Télécommunications
3 Place du Commerce
Ile des Soeurs, Qué.
CANADA H3E 1H6

June 12, 1985

Rapport technique de l'INRS-Télécommunications no. 85-24[1]

Rounding and Scaling in Fixed-Point FFT Implementations

Abstract

The calculation of the discrete Fourier transform using a fast Fourier transform (FFT) algorithm with fixed-point arithmetic is considered. The input data is scaled to prevent overflow and to maintain accuracy. New conditions on the magnitudes of the input components to avoid overflow during the computation of the FFT are derived. Particular emphasis is placed on an implementation using a digital signal processing architecture based on a 16-bit fixed-point representation for the data and the provision for double precision accumulation of sums and products. Simulation results to assess the error performance (signal-to-noise ratio) are presented for various forms of the implementation. Algorithm variants as well as different rounding options are compared. Execution times for implementations based on a single chip signal processor (the Texas Instruments TMS320) are also given. These show that a considerable increase in accuracy can be obtained with only a small penalty in execution time, by applying an alternating form of rounding rather than truncation.

Contents

1. Introduction

The need for the computation of the discrete Fourier transform (DFT) arises in a variety of applications in speech, radar and sonar signal processing [1]. A DFT can be implemented either with special-purpose hardware, or as a program on a digital signal processing (DSP) chip. These DSP chips are characterized by a highly pipelined internal structure and a fast multiply/accumulate unit. This architecture is well suited for the computation of the DFT, either in direct form or using a fast Fourier transform (FFT) algorithm. A programmable approach is flexible and permits other functions to be performed with the same hardware. For instance, preprocessing steps to window the input data or postprocessing steps to further manipulate the Fourier coefficients can easily be accommodated the same programmable device.

Any computation of the DFT will have an accuracy limited by finite precision arithmetic. The objective of this report is to study the implementation of DFT's based on a *fixed-point* processor. A common feature both in DSP chips and standalone multipliers is a double precision accumulator. Specifically, the focus will be on 16-bit representation of data with judicious use of 32-bit accumulation of products and sums. This paper considers more general rounding schemes and variants of the FFT algorithms that do not conveniently fit into the analysis framework used in [2]. The accuracy and execution time of these computation procedures are the subject of this study. For execution time comparisons, attention is focussed on a widely available DSP chip, the Texas Instruments TMS320.

Three computation procedures are considered: 1) direct computation of the DFT, 2) decimation-in-time FFT algorithms, and 3) decimation-in-frequency FFT algorithms. The FFT algorithms are conventional algorithms which are important in practice since they can be implemented with compact hardware or memory efficient software. Furthermore, the ability of these algorithms to carry out the calculations in-place is an important asset for devices or configurations with limited data memory. Many of the scaling and rounding considerations which apply to conventional FFT's carry over to other fast DFT algorithms as well.

Evaluation of the different options for calculating the DFT will be in two steps. A simulation of the algorithm with fixed-point arithmetic on a general purpose computer is used to assess the error performance. This strategy allows flexible error performance monitoring in a data processing environment. The fixed-point arithmetic used in the simulation mimics that of a specific signal processor as far as word size and accuracy are concerned. As a consequence, the numerical results of the simulations are identical to that produced by an implementation on the signal processor. In addition, the execution time for the algorithms when implemented on the signal processor will be given. This allows one to select a technique with a good combination of error performance and speed of execution.

Section 2 introduces the basics of the fixed-point arithmetic system used for calculating the DFT. It also introduces sufficient conditions on the magnitudes of the input data to guarantee that

the output data and intermediate values are representable in fixed-point format. Relevant aspects of fast Fourier transform algorithms used to calculate the DFT are also discussed. The section closes with an examination of different rounding options that may be used in the FFT calculations.

Section 3 gives the results for simulations of the different forms of the fixed-point implementations. The main focus is on the error performance of the algorithms.

The implementation of the algorithms on a TMS32010 processor is considered in Section 4. Both the compatibility of the architecture with the different implementation options and the speed of execution are discussed.

The last section reviews the results of this study and gives a summary of the conclusions.

2. DFT Calculation with Fixed-Point Arithmetic

In this section, different methods of implementing the discrete Fourier transform with fixed-point arithmetic will be considered.

2.1 Fixed-Point Arithmetic

The arithmetic units in digital signal processing hardware usually employ fixed-point computations. It is convenient to consider data values which are fractions (magnitude less than unity). These fractional values are represented as fixed-point numbers. Negative numbers are represented in two's-complement notation. The advantage of fractional notation in signal processing is that products of fixed-point fractions remain fixed-point fractions. The possibility of overflow exists for sums of fixed-point fractions, generally necessitating additional scaling of the values. However, if the sum of a number of terms results in a value which can be represented in fractional notation, intermediate overflows can be ignored. This property will be exploited later.

A key feature of the arithmetic processor used is the availability of a double precision accumulator. Products of single precision values give double precision values which can be added to the accumulator. Stored single or double precision values can also be directly added to the accumulator. The contents of the accumulator can be stored either as single or double precision value.

The availability of double precision accumulation is of benefit in complex multiplication. Consider two complex numbers a_1 and a_2 which can be represented as

$$a_1 = u_1 + jv_1 \qquad \text{and} \qquad a_2 = u_2 + jv_2 \ . \tag{1}$$

The product of a_1 and a_2 is given by

$$a_1 a_2 = (u_1 u_2 - v_1 v_2) + j(u_1 u_2 + v_1 v_2) \ . \tag{2}$$

Note that each real multiplication yields a double precision result. With double precision accumulation, the products can be summed with no loss of precision. Furthermore, the real and imaginary parts of the result can be kept in double precision in anticipation of subsequent operations. On the other hand with single precision accumulation, precision is sacrificed at the intermediate steps even before forming the resultant values.

The simulations will focus on 16-bit representations, with a 32-bit accumulator. This representation is compatible with available signal processing elements.

2.2 Characteristics of a DSP Chip

An example of a single chip implementation of a signal processor with features as described above is the Texas Instruments TMS320 DSP chip. Since this chip has 16-bit wide input/output

paths, values are generally represented as 16-bit numbers. At the expense of multiple write or read cycles, 32-bit values can also be stored or retrieved from memory.

The DSP multiplier takes two 16-bit quantities (each represented by a sign bit with 15 data bits) to produce a 32-bit result, consisting of a sign extended 31-bit product (sign bit with 30 data bits). This product can be added to or loaded into the accumulator. Because of the extra bit of head room in the accumulator, two products can be summed without overflow. The arithmetic unit also provides for shifting 16-bit data values before adding to the accumulator and for the extraction of specific 16-bit fields from the accumulator. These shifting operations allow convenient scaling by (selected) powers of two.

2.3 Scaling for the Discrete Fourier Transform

Consider a complex sequence $\{x_n\}$ of length N. The discrete Fourier transform of $\{x_n\}$ is the complex sequence $\{X_k\}$ of length N, defined as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi nk/N} , \qquad k = 0, ..., N-1 . \tag{3}$$

The direct computation of the DFT simply consists of repeatedly performing the basic multiply/accumulate operations for each of the real and imaginary parts of the result. A double precision accumulator can be used to advantage, however care must be exercised to prevent overflow. FFT algorithms restructure the computation into a number of stages. For radix-2 algorithms, there are $\log_2 N$ stages in the computation. At each stage, N complex values are used to compute N new complex values.

The magnitude of any complex output value is less than or equal to the sum of the magnitudes of the complex input values. This means that the output magnitude is at most N times the maximum input magnitude. The magnitude of the output values is less than unity if the input data is scaled such that

$$|x_n| < \frac{1}{N} , \qquad 0 \le n \le N-1 . \tag{4}$$

This strategy results in a poor signal-to-noise ratio (SNR) since $\log_2 N$ bits of precision have been discarded at the beginning of the computation. To help maintain accuracy, a better approach is to postpone scaling as long as possible in the computation or to distribute the scaling in the processing steps.

For the remaining discussion, it will be assumed that the calculation of the DFT, whether in direct form or using an FFT algorithm, will include a scaling by $1/N$. With this assumption, $|x_n| < 1$, $n = 0, \ldots, N-1$, will guarantee a representable output value. Dynamic verification of this bound is not simple since calculation of the magnitude (squared) requires the calculation of the sum

of products. A test on the magnitudes of the real and imaginary components of the input sequence is easier to implement. Additionally, the complex DFT is often used as the kernel of other procedures. For instance, the calculation of the DFT of a real sequence [3] or the calculation of the discrete cosine transform (DCT) [4] can be broken into three steps. A preprocessing step involves rearrangement of the data storage order without any arithmetic operations on the values. This is followed by the computation of the complex DFT of the reordered data. The last step is a postprocessing of the DFT output data, which does involve arithmetic operations. For these types of algorithms, a test on the magnitudes of the input sequence is equivalent to a test on the magnitudes of the real and imaginary components of the input to the DFT part of the calculation.

Let the real and imaginary components of the input sequence satisfy

$$\left|\mathrm{Re}[x_n]\right| < a \qquad \text{and} \qquad \left|\mathrm{Im}[x_n]\right| < a \; . \tag{5}$$

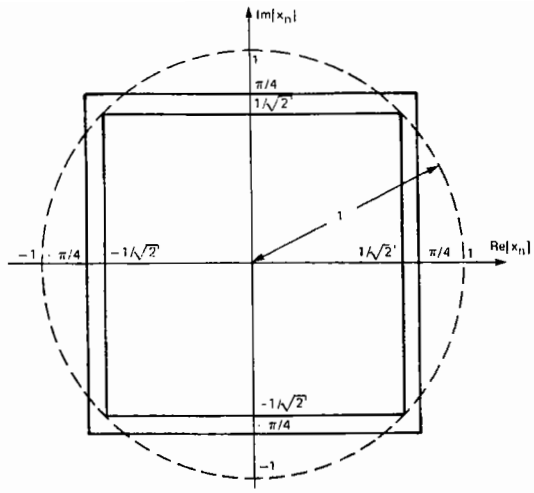
These bounds result in $|X_k| < \sqrt{2}Na$, $k = 0, 1, \ldots, N-1$. With scaling by $1/N$, and the requirement that the output magnitude be less than one, $a = 1/\sqrt{2}$. However, this is not the tightest bound on the real and imaginary components of the input data. In Appendix A it is shown that the magnitudes of the real and imaginary parts of the output of the complex DFT will be less than $Na\,4/\pi$, where a is the maximum magnitude of the components of the input. With scaling by $1/N$ and the requirement that the output components have magnitude less than one, $a = \pi/4$. Then the output components are representable in fractional notation if the real and imaginary parts of the input data satisfy

$$\left|\mathrm{Re}[x_n]\right| < \frac{\pi}{4} \qquad \text{and} \qquad \left|\mathrm{Im}[x_n]\right| < \frac{\pi}{4} \; . \tag{6}$$

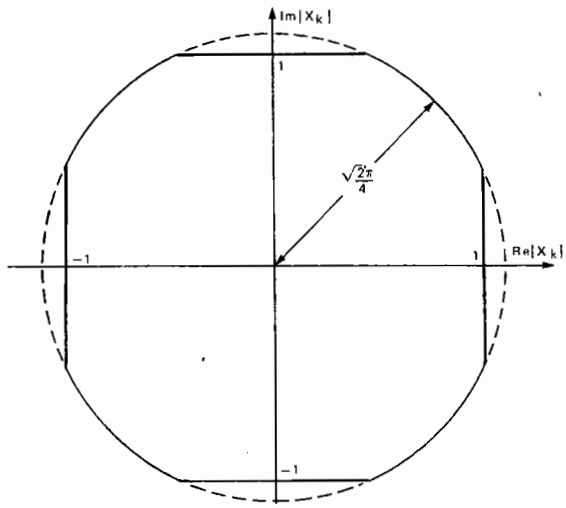
Note that for nearly equal real and imaginary components, these bounds can result in values of x_n and X_k which are larger than unity in magnitude (see Fig. 1).

The bounds given above are sufficient to guarantee representable output values. As such, these results apply to the direct computation of a DFT. In an FFT computation, data is processed in stages. Even though the output may be known to be scaled properly, it is difficult in general to anticipate all combinations of possible overflow situations at the intermediate stages. However, as will be shown later, the above bounds also apply to the intermediate values, as well as the output values, in a decimation-in-time FFT algorithm.

Scaling of the data may be applied in several ways depending on the task at hand. Indeed, it may be possible to guarantee that the data to be transformed has a range which avoids overflow problems. Perhaps in a previous step, values have been scaled so as to limit the maximum magnitude of the components to be transformed. Consider a typical application in a signal processing environment. Data is collected from an analog-to-digital (A/D) converter and a tapered time window is applied to the data. In such a situation, the window coefficients can be scaled so as to ensure that the maximum data value is in the proper range.



(a) Input data bounds



(b) Output data bounds for $|\mathrm{Re}[x_n]| < \pi/4$ and $|\mathrm{Im}[x_n]| < \pi/4$

Fig. 1 Bounds on the input and output values

The situation cited above is an example of a fixed scaling of the data. In other cases, it is not a potential overflow that is the main concern, but instead it is desired to scale the data so that it occupies the available dynamic range. An example is in the use of a DFT in a frequency domain speech coder or in the front end processor of a speech recognition unit. One approach is to scan the input data for the element which has the largest magnitude. The entire data array is then scaled with the same scale factor to bring the largest value to near full scale. A particularly simple form of scaling is to shift the data to accomplish multiplication by powers of two until the largest magnitude element is less than either $\pi/4$ or $1/\sqrt{2}$ as appropriate. This can be considered to give a block floating point representation, with the number of bits shifted giving the exponent of the representation. In fact, this scaling can in some cases be absorbed into the the first stage of the

computation of the FFT.

The direct computation of the DFT involves the accumulation of products. To prevent accumulator overflow, a reasonable strategy is to scale the output of the multiplier by a power of two before accumulation. This scaling can be accomplished by shifting the product by $\lceil \log_2 N \rceil - 1$ bits prior to accumulation.† Even with this shift, for moderate values of N, the product is represented with considerable increased accuracy compared to a single precision representation. At the end of the computation of an output value, a single precision result can be extracted from the accumulator.

2.4 FFT Algorithms

Fast Fourier transform algorithms are efficient procedures for the computation of the DFT. Two versions, decimation-in-time and decimation-in-frequency are considered [5]. Attention will be restricted to basic radix-2 algorithms. As a result, the transform length N is assumed to be a power of two. Both forms of the algorithm can be characterized in terms of a basic computational unit commonly known as the butterfly. These butterflies take a pair of complex data values and process them to produce a new pair of complex data values which can occupy the same storage locations as the input data. The butterfly operations involve complex addition and multiplication by a complex exponential. At each of the $\log_2 N$ stages of the FFT, $N/2$ butterflies are computed. It is not possible (even without the constraint of in-place computation) to keep the output of the butterflies in more than single precision without a dramatic increase in storage requirements. The use of double precision accumulation *within* a butterfly is considered separately for the two forms of the FFT algorithm.

2.4.1 Decimation-in-Time

The basic relationship which forms the heart of the decimation-in-time algorithm is the doubling formula,

$$\begin{aligned} X_k &= \sum_{n=0}^{N/2-1} x_{2n} W_{N/2}^{-nk} + W_N^{-k} \sum_{n=0}^{N/2-1} x_{2n+1} W_{N/2}^{-nk} \\ &= A_k + W_N^{-k} B_k \ , \end{aligned} \tag{7}$$

where $W_N = \exp j2\pi/N$. This expression gives the DFT of length N in terms of two $N/2$ point DFT's, denoted by the sequences $\{A_k\}$ and $\{B_k\}$. Note that although each of the two $N/2$ point DFT's is periodic in k with period $N/2$, the twiddle factor W_N^{-k} makes the output sequence periodic with period N. Two output values will be formed from two intermediate values. For $k = 0, \ldots, N/2-1$, the computations are

$$\begin{aligned} X_k &= A_k + W_N^{-k} B_k \\ X_{N/2+k} &= A_k - W_N^{-k} B_k \ . \end{aligned} \tag{8}$$

† This takes into account the 1 bit of head room in the double precision product register.

A form of this basic butterfly computation is shown in Fig. 2. Recursive application of this formula gives the basic decimation-in-time (DIT) FFT algorithm. A flow graph of the complete computational procedure is shown in Fig. 3 for $N = 8$.

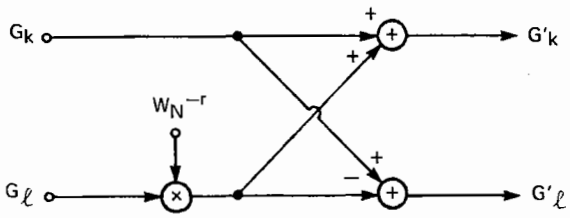


Fig. 2 Decimation-in-time butterfly

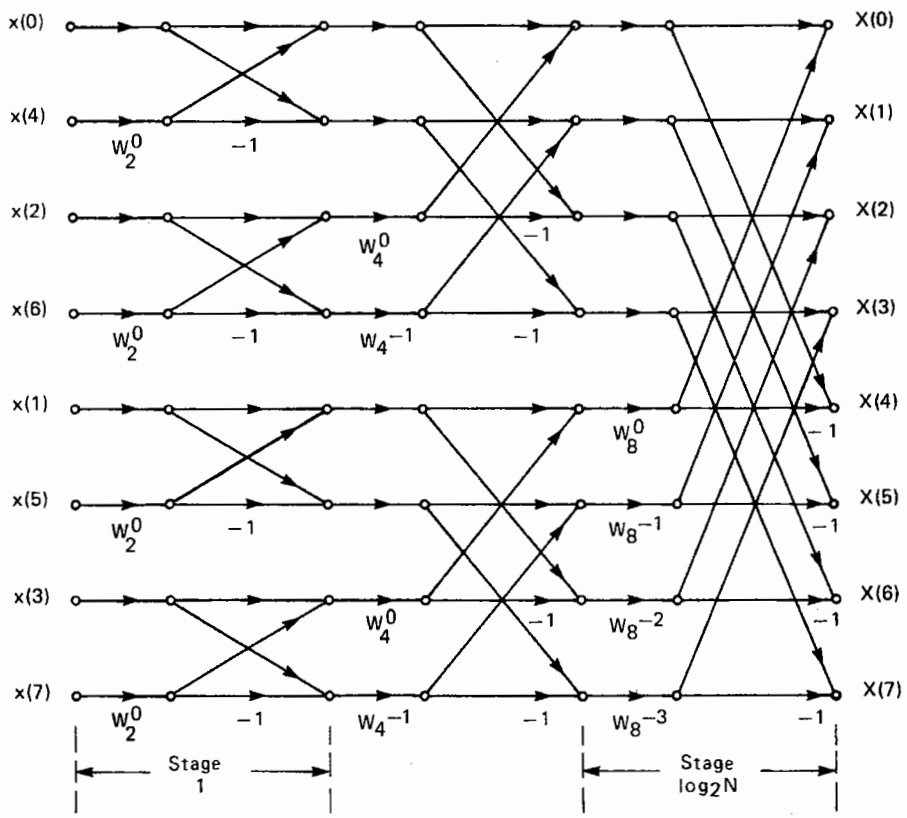


Fig. 3 Flow graph for the DIT FFT

As indicated above, scaling is needed to prevent overflows in the computation of the DFT. This scaling can be implemented at the butterflies of the FFT computation. The magnitude of the complex values grows by at most a factor of two across a butterfly,

$$\max(|G_k|, |G_l|) \leq \max(|G'_k|, |G'_l|) \leq 2\max(|G_k|, |G_l|) \; . \tag{9}$$

Shifting values by one bit to accomplish a scaling by $1/2$ is appropriate. The highest accuracy is preserved if this scaling is done only if an overflow is inevitable during the computation of the butterfly. However, for this strategy to be efficient, hardware traps have to be available to sense overflow conditions. These mechanisms are not available on current DSP chips. A compromise is to scale by a factor of $1/2$ at each of the $\log_2 N$ stages to result in the overall fixed scaling by $1/N$.

An important property of the DIT form of the FFT is that the intermediate results at the outputs of the butterflies are DFT's of subsequences of the input data. As a result, if the magnitudes of the real and imaginary parts of the input sequence are bounded by $\pi/4$, all of the intermediate results at the outputs of the butterflies, and of course the final output, will have real and imaginary parts which are less than one in magnitude.[†]

With a double precision accumulator, there are several options for the implementation of the DIT butterfly. In the lower branch, the result of the complex multiplication can be kept in double precision in anticipation of the subsequent addition. However, this requires some overhead in moving double precision values back and forth between the accumulator and temporary storage. The butterfly is completed with the addition of single precision values to the products. Because of the extra bit of head room in the product register, no overflow can occur with this strategy if the input data is scaled so as to guarantee representable output values. Scaling by the factor of $1/2$ is accomplished by shifting one bit position as results are stored from the accumulator. This last step is the point at which rounding can be applied. The options in terms of rounding will be described in more detail later.

An alternative strategy is to convert the result of the multiplication to a single precision quantity before addition. This reduces the number of interchanges between the accumulator and temporary storage. This version will be referred to as the single precision DIT algorithm as contrasted with the double precision DIT algorithm described in the previous paragraph. Note that the multiplication by W_N^{-r} in the butterfly can be thought of as a rotation of the input complex value. This means that if the magnitude of the input complex value is greater than one, it is possible for the real and imaginary parts of the product to exceed one. However scaling at this point can be avoided since intermediate overflows can be ignored if the butterfly output is known to be representable. This is the case if the real and imaginary parts of the original input sequence are less than $\pi/4$ in magnitude.

2.4.2 Decimation-in-Frequency

The relationship which forms the heart of the decimation-in-frequency (DIF) algorithm is an-

[†] The bounds given in Appendix A can be used to more tightly bound the intermediate values at the butterfly outputs, especially at the beginning stages of the FFT.

other doubling formula,

$$X_{2k} = \sum_{n=0}^{N/2-1} (x_n + x_{N/2+n}) W_{N/2}^{-nk}$$
$$X_{2k+1} = \sum_{n=0}^{N/2-1} (x_n - x_{N/2+n}) W_N^{-n} W_{N/2}^{-nk} \; . \tag{10}$$

This expression gives the DFT of length N in terms of two $N/2$ point DFT's. The first is the DFT of the sequence $\{x_n + x_{N/2+n}\}$, while the second is the DFT of the sequence $\{(x_n - x_{N/2+n})W_N^{-n}\}$. The form of the butterfly for the decimation-in-frequency algorithm is shown in Fig. 4, while the complete flow graph for a DIF FFT ($N = 8$) is shown in Fig. 5. Note that for the DIF algorithm, multiplication by the complex exponential term is performed at the output, rather than input stage of the butterfly (as in the DIT algorithm). The magnitude of the complex values grows by at most a factor of two across a butterfly (as in the DIT butterfly, c.f. Eq. (9)). Shifting values by one bit is used to scale by $1/2$ at each stage and gives an overall scale factor of $1/N$.

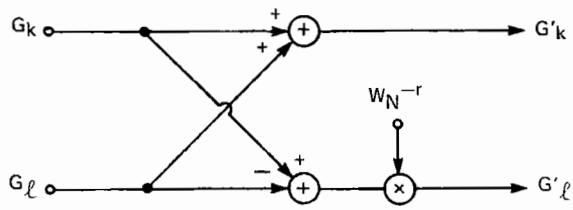


Fig. 4 Decimation-in-frequency butterfly

The butterfly computation involves sums of single precision quantities followed by a multiplication by a complex exponential. The result of the sum must be converted to a single precision quantity for use as input to the multiplier. The resulting product in the lower branch of the butterfly (Fig. 4) is converted to single precision for storage.

For the DIF algorithm, the intermediate results at the outputs of the butterflies are *not* DFT's of the original input data. The scaling requirement is that the magnitudes of the input real and imaginary parts each be less than $1/\sqrt{2}$, to guarantee that all butterfly outputs are less than one in magnitude. This is in contrast to the slightly larger limit of $\pi/4$ that can be used in the DIT algorithm. In fact, if the larger values are used, it is possible for overflow to occur at the very first stage of the FFT.

2.5 Rounding

As has been indicated, at one or more places in the computation of the butterflies, a double precision value must be converted to a single precision value. Truncation is a simple operation which

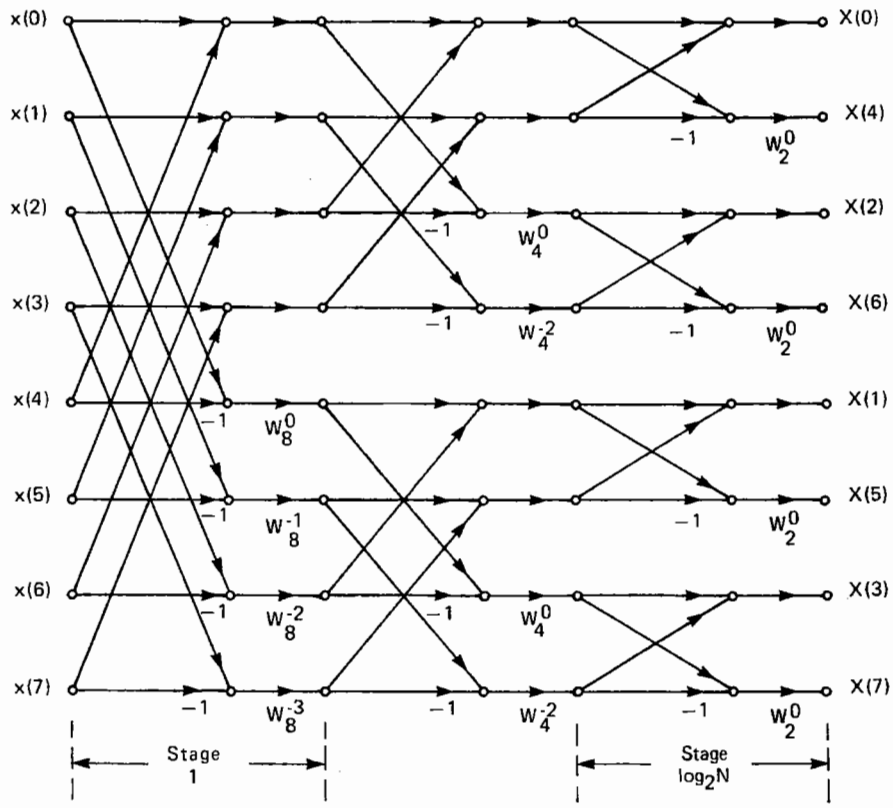


Fig. 5 Flow graph for the DIF FFT

discards the low-order part of the double precision value. This tends to shift the value, moving it towards zero for positive values and away from zero for negative values. On the other hand, rounding attempts to replace the double precision value with the nearest single precision value. However, an ambiguity arises whenever the value lies exactly halfway between two single precision values. This gives rise to a number of different methods of rounding, which differ only in the way they approximate the mid-way values. The different schemes are schematically illustrated in Fig. 6.

1) *Up-Rounding.* The mid-way points are shifted up to the next allowable value. Negative occurrences of mid-way points are moved towards zero.
2) *Down-Rounding.* The mid-way points are shifted down to the next allowable value. Negative occurrences of mid-way points are shifted away from zero.
3) *Magnitude Up-Rounding.* The mid-way points are moved away from zero for both positive and negative occurrences.
4) *Magnitude Down-Rounding.* The mid-way points are moved toward zero for both positive and negative occurrences.
5) *Value-alternate Rounding.* This is a deterministic scheme which moves some occurrences of the mid-way points up and some down. The specific implementation considered here moves the mid-way point above zero, towards zero. The next larger mid-way points move away from zero. An alternation of the direction of rounding applies to adjacent mid-way points. The direction in which rounding is applied is data dependent.

6) *Random Rounding.* The direction of rounding is determined in a pseudo-random fashion. The direction of rounding does not depend on the data values.

7) *Stage-alternate Rounding.* In this new scheme which applies directly to FFT computations, the direction of rounding alternates at each stage of the FFT. The direction of rounding does not depend on the data values.

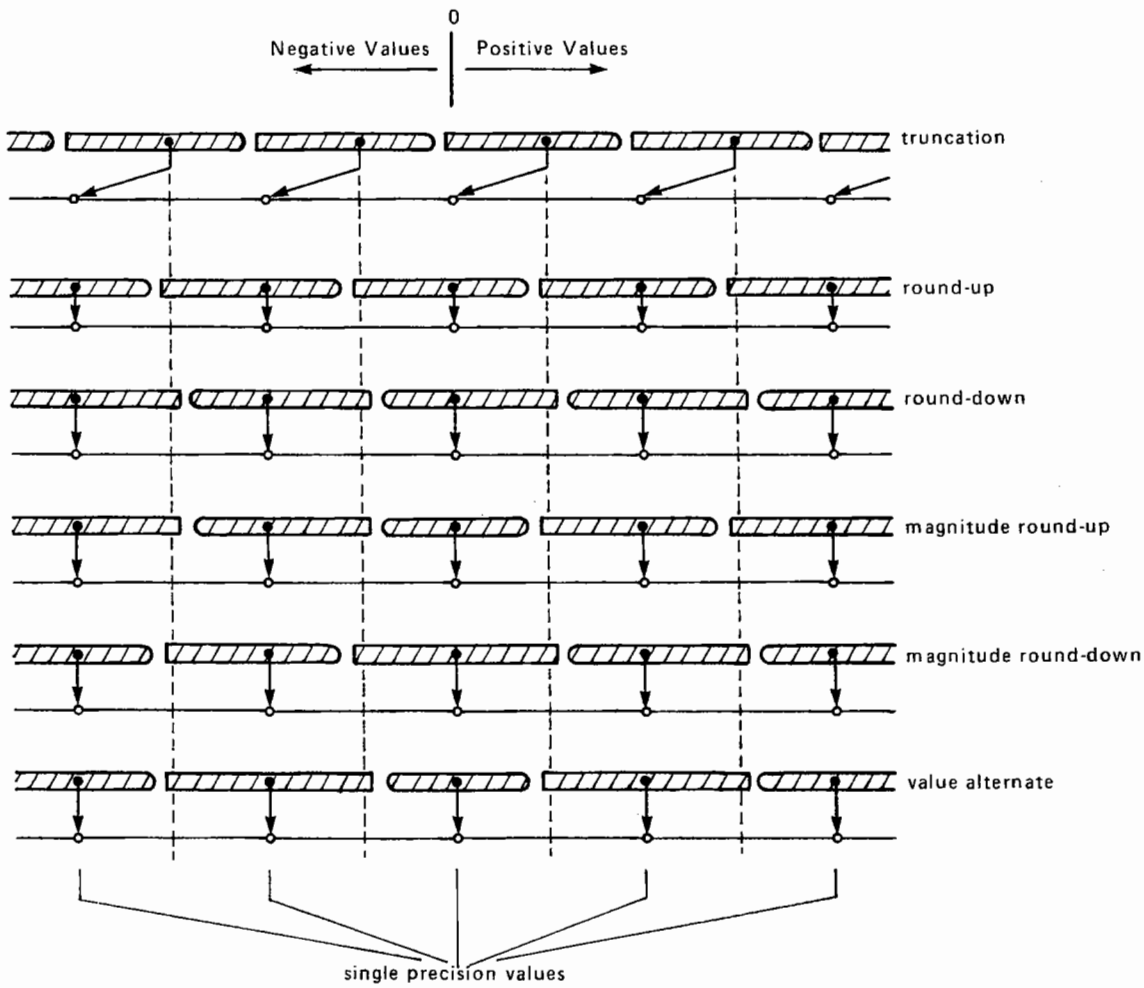


Fig. 6 Rounding methods

The most common method of rounding is up-rounding. It is simple to implement. A value corresponding to half of the weight of the least significant bit of single precision values is added to the double precision accumulator. Truncation of this sum gives up-rounding. Down-rounding is also simple to implement. The value to be added before truncation is slightly less than the value applied for up-rounding. Specifically the value to be added is equal to half of the weight of the least significant bit of single precision values, less the weight of the least significant bit of double precision values. Intuitively both of these methods will tend to bias the output values by shifting the mean.

Magnitude rounding is more complex to implement than either up- or down-rounding since it requires a test of the sign of the contents of the accumulator to determine whether rounding up or down is to be applied. Intuitively, magnitude rounding techniques tend to alter the overall gain, but not alter the mean. For instance, magnitude down-rounding will tend to decrease the magnitude of the output values.

The last three rounding schemes could be grouped together under the rubric of unbiased techniques. Value-alternate and random rounding attempt to avoid both the change in gain and the shift in mean, but succeed only if the data exercises an equal number of up- and down-rounding steps. Stage-alternate rounding has the same goals, but attempts to cancel biases in one stage of the FFT by biases in the other stages.

Differences in the rounding techniques will manifest themselves only if the probability of hitting the mid-way points is significant. Consider first the product of two single precision quantities. The number of bits to be discarded in converting the double precision product to single precision is large. Thus the probability of the mid-way point is small and differences in the results for the various forms of rounding should be small. On the other hand, adding two single precision values gives a value with only one extra bit of precision. When storing the sum scaled by a factor $1/2$ as a single precision quantity, the mid-way point can be expected to occur about half of the time. For this situation, the rounding methods are no better than truncation in terms of the magnitude of the error in each operation. It is for this case that the unbiased rounding schemes are useful. These schemes impart errors in different directions at the different stages and thereby try to eliminate the overall bias.

The efficacy of these approaches in cancelling the bias is discussed in the next section. The next section will also deal with the computational accuracy of each of the DFT algorithms described earlier.

3. Simulation Results

The DFT algorithms described in the previous section have been simulated using fixed-point arithmetic. The implementation uses routines which simulate the multiplication of two single precision quantities, the addition of both single and double precision quantities to a double precision value and the conversion of a double precision value to single precision by extraction of a 16-bit field.

The goal of this section is the assessment of the performance of various forms of the DFT. To this end, a gain and mean compensated SNR is introduced. This measure is useful in interpreting the error performance of the different rounding options.

Results for both one-way and two-way (forward DFT followed by inverse DFT) computations are given. The two-way SNR is important for evaluating fixed-point transforms in some filtering and transform coding applications. For instance, a transform coder for speech or video has a transform stage followed by a quantizer and coder. The matching decoder utilizes an inverse transform. With fixed-point arithmetic, the errors introduced in the computations of the transform and the inverse transform limit the performance of such a coding system.

3.1 Experimental Setup

In the simulations, the input data were uniformly distributed, pseudo-random fixed-point fractional values between $-1/\sqrt{2}$ and $+1/\sqrt{2}$. This guarantees no overflow of the intermediate results of the FFT computations (butterfly outputs), although care must still be exercised inside each butterfly. Note that the use of fixed-point data means that initial approximation errors are not included in the resulting error. The statistical stability of the results was assured by averaging a number of runs (typically 10). Experimentally, the SNR figures vary only a small fraction of a dB for averages taken with different pseudo-random input sequences.

The root-mean square (rms) value of the random test input is approximately $1/\sqrt{6}$ or -7.78 dB relative to full scale. Note that this type of input sequence tends to make SNR figures look comparatively good. The input occupies nearly the full dynamic range and yet has a relatively large rms value. For more typical input signals the ratio of peak value to rms value is significantly larger. For instance for speech signals this ratio is often taken to be 4 (referred to as 4σ loading). The SNR for such a signal would be about 7.3 dB worse than for uniformly distributed data with an equal peak value.

Transform lengths ranging from 16 to 256 complex values are considered. This range is consistent with the transform sizes that can be accommodated using the limited internal data memory available in current generation DSP chips.

For the two-way results, the inverse DFT is computed using the fixed-point (forward) DFT. If $\{x_n\}$ and $\{X_k\}$ are a transform pair, indicated symbolically by

$$\{X_k\} = \mathrm{DFT}_N\{x_n\} \ , \tag{11}$$

the inverse transform relationship giving $\{x_n\}$ in terms of $\{X_k\}$ is

$$\{x_n^*\} = \mathrm{DFT}_N\left\{\frac{X_k^*}{N}\right\} \ . \tag{12}$$

The factor $1/N$ is already included in the computation of the fixed-point DFT's. The operations involve complex conjugation of the input sequence, calculation of the (forward) DFT, and conjugation of the output sequence.

For two-way computations, the input sequence itself serves as a reference for measuring SNR. For one-way SNR measurements, a double precision floating point DFT is used as the benchmark. The two-way SNR for this floating point computation is over 320 dB for all sizes of DFT quoted in this study.

A reference fixed-point result is also computed. If a fixed-point DFT is calculated with an arbitrarily large precision at intermediate stages, errors will occur only when the final result is converted to single precision accuracy. This situation is simulated for the one-way case by taking the results of the double precision floating point reference, scaling them by $1/N$ and converting them to fixed-point representation by rounding. For the two-way case, the original input data is scaled by $1/N$ and converted to single precision fixed-point representation by rounding. The SNR for these fixed-point results is referred to in the plots as the ideal fixed-point DFT.

3.2 Assessing Bias

In attempting to assess and explain the behaviour of truncation and the various forms of rounding, a modified SNR definition may be used. The conventional SNR can be expressed in the form

$$\mathrm{SNR} = \frac{\sum_{n=0}^{N-1} |X_k|^2}{\sum_{n=0}^{N-1} |E_k|^2} \ . \tag{13}$$

This conventional form of SNR measures error as

$$E_k = \hat{X}_k - X_k \ , \tag{14}$$

where $\hat{X}_k$ is the output of the fixed-point DFT.[†] In the experimental results, $\{X_k\}$ is the pseudo-random input sequence. This error expression penalizes changes in gain as well as shifts in the mean. For instance, an output which is merely scaled by a constant gain will exhibit a low SNR.

[†] It is assumed that the factor of $1/N$ which is present in the fixed-point computations has been taken appropriately into account.

Consider a modified error term,

$$E_k = a(\hat{X}_k - b) - X_k \ . \tag{15}$$

This expression includes the constants a and b which compensate for gain and mean respectively. Four cases can be identified.

1) *Conventional SNR.* $a = 1$, $b = 0$.
2) *Gain Compensated SNR.* a chosen to maximize the SNR, $b = 0$.
3) *Mean Compensated SNR.* $a = 1$, b chosen to maximize the SNR.
4) *Gain and Mean Compensated SNR.* Both a and b chosen to maximize the SNR.

The SNR for cases 2) or 3) is larger than the conventional SNR. Likewise, the combined gain and mean compensated SNR is larger than the other SNR figures. Techniques to calculate the constants a and b are discussed in Appendix B. The mean offset b is restricted to have equal real and imaginary components. The offset to either the real or imaginary component will be referred to simply as the mean offset. The main interest will be in looking at the gain and mean needed to get the best match between the true DFT and the fixed-point DFT. The increased SNR that results in choosing the gain and mean optimally will indicate the degree of match of the "shape" of the DFT. This indeed may the appropriate criterion for certain applications. For instance, returning to a speech recognition example, gain normalization is an inherent part of comparing speech segments against stored frequency response templates. In this situation a non-unity gain factor in the FFT computation does not affect the results.

3.3 Truncation Results

The resulting SNR[†] for the two forms of the DIT FFT (single precision intermediate values and double precision intermediate values) and the DIF FFT are shown in Fig. 7 for both one-way and two-way computations. The SNR for the direct computation of the DFT using fixed-point arithmetic is also given in the graphs. These results are for conversion of the contents of the double precision accumulator to single precision by truncation. The SNR values for the ideal fixed-point DFT shown in the graphs may be interpreted as upper bounds on the achievable performance.

A perhaps initially surprising result is that the single precision DIT FFT algorithm outperforms the double precision DIT FFT algorithm. Further investigation into the details of the butterfly computation reveal that the biases due to truncation of the product and the subsequent truncation of the differences tend to cancel in the single precision version. Theoretically, the mean offset at the output of each single precision butterfly is half that of the double precision version. The mean offset

[†] To avoid ambiguity, all SNR figures quoted are conventional SNR; modified SNR figures are given as increases over conventional SNR.

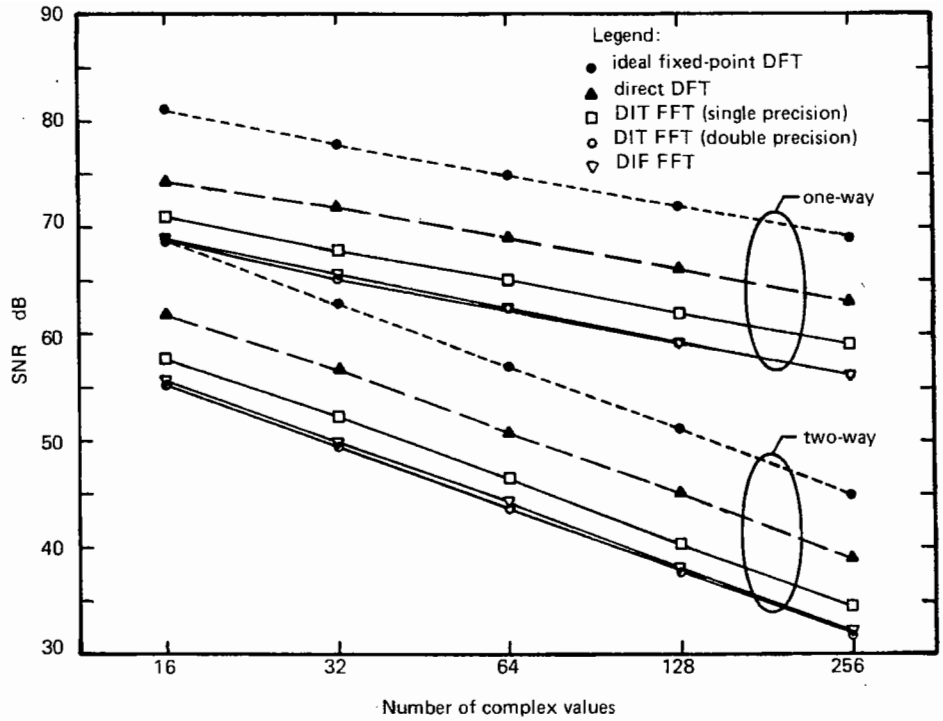


Fig. 7 SNR for truncation

computed using the modified SNR definition verifies this analysis. For the one-way computation, the single precision DIT algorithm has a mean offset of approximately $-Nv_{\text{lsb}}/2$, while the double precision DIT algorithm has a mean offset of approximately $-Nv_{\text{lsb}}$, where v_{lsb} is the weight of the least significant bit of a single precision value ($v_{\text{lsb}} = 1/32768$). The increase in SNR with mean compensation is about 1.5 dB for the single precision version and about 4 dB for the double precision version.

The mean offset for the DIF algorithm is about the same as the double precision DIT algorithm. As can be seen in Fig. 7, the conventional SNR for the DIF algorithm is also about the same as that of the double precision DIT algorithm. With mean compensation, the SNR increases about 4.5 dB. This means that both forms of the DIT algorithm and the DIF algorithm have about the same mean compensated SNR. Gain compensation does not help either DIT algorithm but does increase the SNR for the DIF algorithm by a further 1.5–2 dB.

The direct computation of the DFT has a mean offset of about $Nv_{\text{lsb}}/2$. The mean compensated SNR is about 6 dB better than the conventional SNR and essentially the same as the conventional SNR of the ideal fixed-point DFT.

3.4 Computational Error Model

Auxiliary experiments were conducted to assess the effect of varying the range of the input data. Fig. 8 shows these results for a 128 point DFT with truncation used for the fixed-point implementations. The input data levels are given in terms of the rms value of the pseudo-random data relative to unity. Thus the data range of $-1/\sqrt{2}$ to $+1/\sqrt{2}$ used for the main results corresponds to the -7.78 dB point on these plots. For a given transform size, the SNR varies directly with the input level, at least for useful signal-to-noise ratios. Results for rounding and for other transform lengths show the same variation with input level. This behaviour indicates that the error can be modelled as being additive and independent of the signal level.

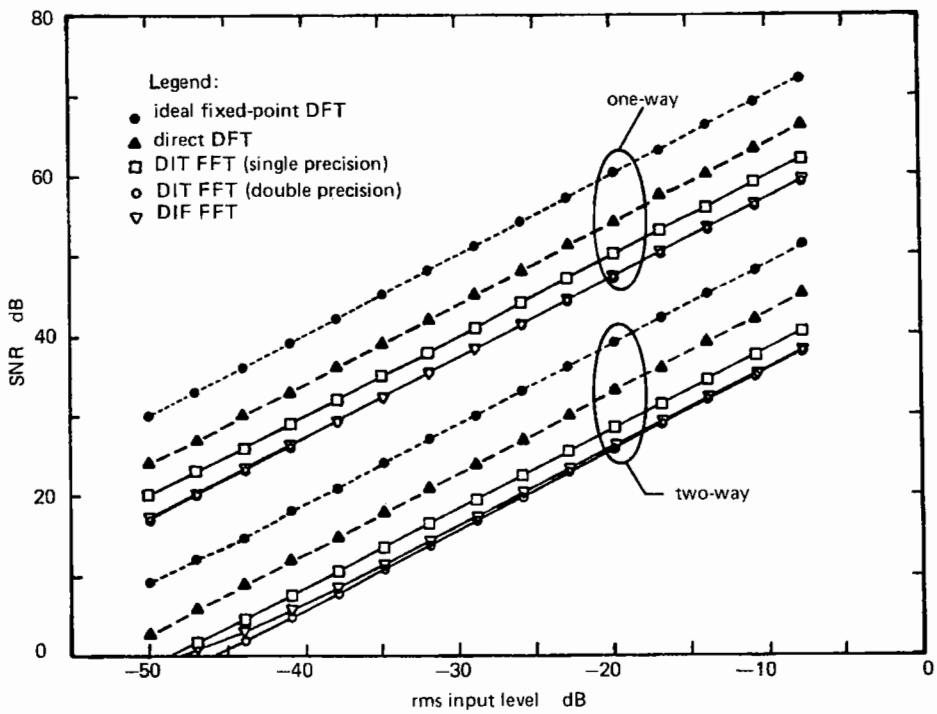


Fig. 8 SNR as a function of the input signal level ($N = 128$, truncation)

Given an additive error component which is uncorrelated with the data and which is independent and identically distributed for each sample, the SNR for a two-way computation can be expressed in terms of the SNR for a one-way computation. Consider the model shown in Fig. 9. If the output of the forward DFT is connected to the input of the inverse DFT, the overall output has two error components, that due to the forward DFT and that due to the inverse DFT. The complex conjugation operations do not affect the mean-square value of the error components. Parseval's

relationship for the DFT is

$$N \sum_{n=0}^{N-1} |e_n|^2 = \sum_{k=0}^{N-1} |E_k|^2 \ , \tag{16}$$

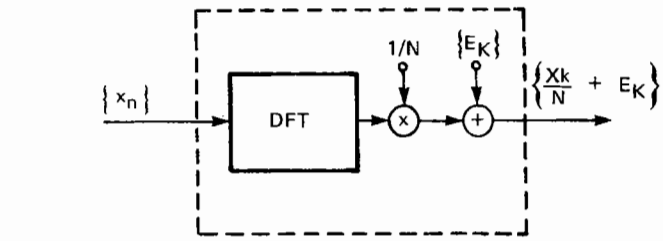
where $\{e_n\}$ is the inverse DFT of $\{E_k\}$. The mean-square error at the output of the first stage is

$$\varepsilon_1 = \overline{|E_k|^2} \ . \tag{17}$$

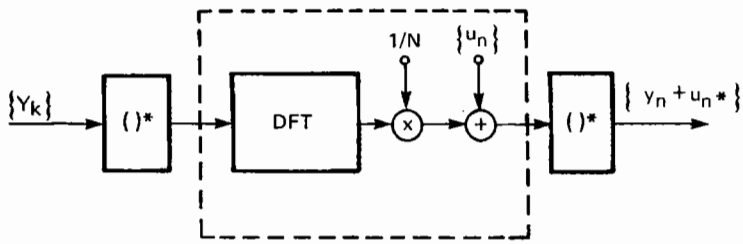
The mean-square error at the output of the second stage is

$$\begin{aligned} \varepsilon_2 &= \overline{|u_n|^2} + \overline{|e_n|^2} \\ &= \overline{|u_n|^2} + \frac{1}{N}\overline{|E_k|^2} \\ &= \frac{N+1}{N}\varepsilon_1 \ . \end{aligned} \tag{18}$$

The last equality comes from the fact that the energy of the complex error is the same for the inverse and forward DFT's, i.e. $\overline{|E_k|^2} = \overline{|u_n|^2}$.



(a) Forward transform



(b) Inverse transform

Fig. 9 Model for error in computing the DFT

The signal component of the output of the DFT is $\{X_k/N\}$, while the signal component of the two-way output is $\{x_n/N\}$. The energies in these components are related by Parseval's relationship (c.f. Eq. (16)). Combining this with Eq. (17) and Eq. (18), the SNR for a two-way computation (SNR_2) can be related to the SNR for a one-way computation (SNR_1),

$$\mathrm{SNR}_2 = \frac{\mathrm{SNR}_1}{N+1} \ . \tag{19}$$

It can be verified from Fig. 7 that this result is very well approximated by the experimental data. For the subsequent results, only one-way SNR's are presented — the two-way SNR's may be derived from Eq. (19).

3.5 Rounding Results

The various forms of rounding were simulated to assess their effect on accuracy for the different DFT algorithms. In view of the many combinations of algorithm variants and rounding options, the two classes of rounding techniques will be considered separately to simplify the discussion.

3.5.1 Conventional Rounding

First consider the DIT FFT algorithm with double precision intermediate values. Rounding occurs as the last step in a butterfly. The DIT FFT improves 8–10 dB with rounding, with the larger improvements occuring with the longer transform lengths. The form of rounding has almost no effect on the SNR. The optimal gain is essentially unity and the mean offset is small. This agrees with the notion that the mid-way values which are rounded differently for the different rounding options occur with very small probability.

For the DIT FFT with single precision intermediate values and the DIF FFT, rounding can be applied at two different places. The first is in the rounding of the double precision products to single precision. Here, the exact form of rounding is unimportant since the mid-way values occur with small probability. The second place is the point at which the sum of two single precision values is stored in single precision. For this rounding operation, the conventional rounding schemes (up-rounding, down-rounding, magnitude up-rounding and magnitude down-rounding) perform similarly in terms of SNR ($\sim$ 1 dB spread). In fact for this rounding operation, in which only a single extra bit is available on which to base the rounding, down-rounding and truncation are identical. However, it should be noted that the biases for the different rounding schemes are not the same. As anticipated earlier, the magnitude rounding schemes tend to result in an optimal gain different from unity but a mean offset nearly equal to zero. The gain factor deviates from unity by about 0.08% for $N = 256$. The resulting gain compensated SNR is about 4 dB larger than the conventional SNR. For up- and down-rounding, the gain is essentially unity, but the magnitude of the mean offset is about $Nv_{\mathrm{lsb}}/2$. The resulting mean compensated SNR is about 4 dB larger than conventional SNR. Based on the different behaviour of rounding for sums and for products, a selective rounding scheme in which rounding (of any form) is applied only to the product and truncation used for the sum performs as well as any of the conventional rounding schemes. Compared to truncation, rounding applied at the product gives a 2 dB increase in SNR for the single precision DIT algorithm and a 5 dB increase for the DIF algorithm. Table 1 shows SNR values for the FFT algorithms for a transform length of 128 using various forms of rounding at the products and sums.

In the direct computation of the DFT, rounding occurs as the last step in the process. Indeed, extended precision results may be obtained with the only penalty being the larger storage space required. For single precision results, the exact form of rounding does not affect the SNR. The

$N = 128$	$\times$ truncation + truncation	$\times$ up-rounding + truncation	$\times$ up-rounding + up-rounding	$\times$ stage-alternate + stage-alternate
dp DIT	59.3 dB	68.6 dB	68.6 dB	68.6 dB
sp DIT	62.0 dB	64.3 dB	64.1 dB	68.2 dB
DIF	59.2 dB	64.5 dB	64.4 dB	68.6 dB

Table 1 SNR for the double precision (dp) DIT, single precision (sp) DIT and DIF algorithms for the indicated forms of rounding at the products ($\times$) and sums ($+$)

rounded results are essentially the same as that for the ideal integer DFT and about 6 dB better than truncation.

3.5.2 Unbiased Rounding

For the direct calculation of the DFT and the double precision DIT FFT algorithm, unbiased rounding performs the same as conventional rounding. However for the single precision DIT FFT and the DIF FFT, the unbiased rounding schemes offer a benefit at the sum step. The differences between the unbiased schemes are not large, but stage-alternate rounding stands out as being slightly better on the average ($\sim$ 1 dB) than value-alternate rounding, which is about the same as random rounding. The stage-alternate rounding scheme is about 4 dB better than conventional up-rounding (e.g. see Table 1).

The unbiased schemes give a useful increase in SNR for the DIF FFT and the single precision DIT FFT. The ease of implementation is an important consideration in the use of such a scheme. Of the unbiased schemes, stage-alternate rounding stands out as being the easiest to implement. The rounding operation consists of adding an appropriate offset before truncating. This offset is changed only outside the main computation loop in the FFT and hence is altered only $\log_2 N$ times. The best unbiased rounding scheme also comes with very little increase in computation or program size compared to conventional up-rounding.

As anticipated, these rounding schemes tend to result in optimal gain factors close to unity. The magnitude of the mean offset for stage-alternate rounding is about 1/3 of that for conventional up-rounding.[†] The mean offsets for the other two unbiased schemes are essentially zero. In stage-alternate rounding, the main contribution to the mean offset is due to the last stage in which all output values are biased in the same direction. This is consistent with the observation that the mean offset changes sign depending on whether an even number of stages or an odd number of stages are used. For the other two unbiased schemes, the data in the output stage is not all rounded in the same direction. This suggests yet another alternative unbiased rounding scheme: stage-

[†] Stage-alternate rounding is not truly unbiased. Mean compensation increases the SNR by about 0.5 dB.

alternate magnitude rounding. For this scheme, magnitude up- and magnitude down-rounding are alternated. In terms of conventional SNR, this scheme performs about the same as the original stage-alternate rounding. The disadvantage of stage-alternate magnitude rounding is the extra complexity of magnitude rounding.

The improvements in SNR due to the use of rounding are summarized in Fig. 10 (note the expanded vertical scale). The rounding results give the conventional SNR for up-rounding applied to the double precision DIT and the direct DFT algorithms, and stage-alternate rounding applied to the single precision DIT and the DIF algorithms.

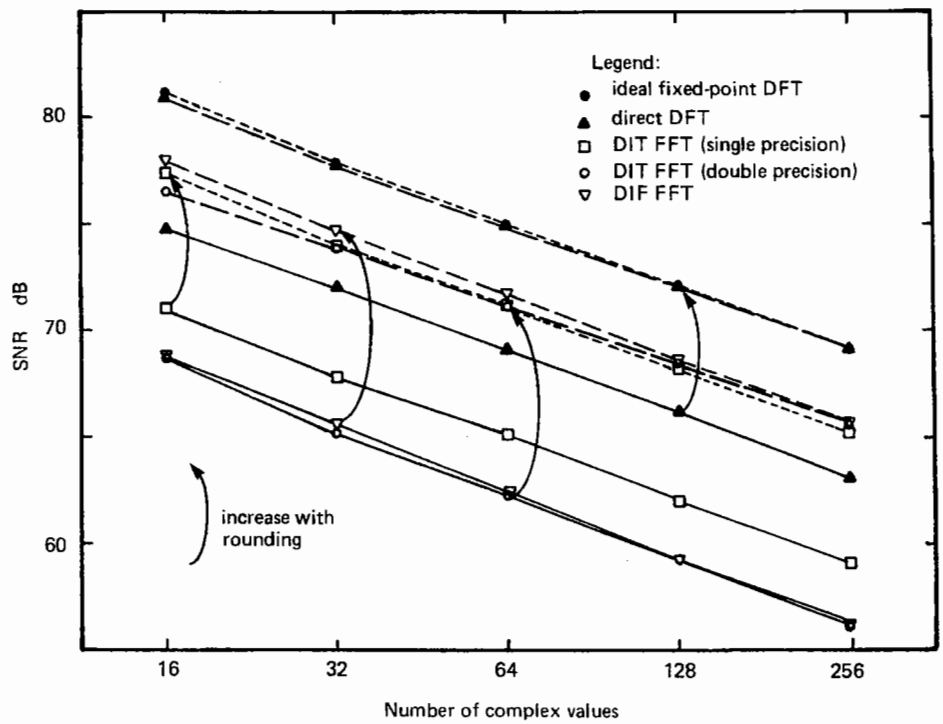


Fig. 10 SNR for truncation and rounding

The high accuracy of stage-alternate rounding applied to the FFT algorithms with single precision intermediate values has important ramifications in both hardware and software requirements. For custom hardware or custom chip implementations, significant reduction in complexity is possible if single precision registers are used rather than double precision registers. The implications of stage-alternate rounding on the FFT execution time on a single chip DSP are examined in Section 4.

3.6 Cosine Table

An inherent part of the computation of the DFT is multiplication by a complex exponential. For this purpose, the fixed-point implementations use a table of stored cosine values. The previously

generated results were given using a table of cosine values which are rounded to the nearest 16-bit value. With fractional notation, $+1$ is not representable, although -1 is representable. Only the first quadrant of the cosine table is stored. The values which would have been rounded to $+1$ are stored as the largest representable positive number. The cosine in other quadrants are determined by suitable changes in sign. Values of the sine are determined by appropriate offsets into the table and suitable changes in sign. Note that this strategy results in the value -1 also not being represented exactly.

Two experiments were conducted to examine alternate strategies. In the first, the cases where the sine or cosine takes on values of ± 1 were trapped in the code and the implied multiplications were avoided. This is equivalent to exact representation of the values ± 1. In an FFT computation the first two or last two stages (DIT or DIF algorithms respectively) which involve only factors of ± 1 and $\pm j$ can be implemented as separate loops avoiding multiplications. This gets rid of most of the multiplies by ± 1 but still leaves some embedded in the remaining stages. With the full avoidance of multiplies by ± 1, the SNR results for the range of transform sizes considered earlier do not increase by more than a small fraction of a dB.

One fixed-point implementation that we have seen, scales all the cosine table values downward such that the largest cosine value is just equal to the largest representable positive fraction. Theoretically this strategy is not justified. In the butterfly, one branch becomes scaled while the other is not. However, even these cosine table values do not degrade the SNR by more than a small fraction of a dB.

These results are consistent with comments in [2] which indicate that coefficient quantization is not as important as computational error.

3.7 Input Precision

Experiments were conducted to assess the importance of data precision on the resultant SNR. With reduced input precision, the input data retains the same range as before but lower order bits are set to zero. For instance, this may be the situation when data is gathered from an A/D converter with less than 16 bits of resolution. For the range of transform lengths considered earlier, the SNR for the FFT algorithms varies only slightly ($\sim$ 0.5 dB) for the input precision decreasing to 4 bits. This is further confirmation that the error energy is relatively independent of the form of the input signal.

4. Speed of Execution on the TMS32010

In this section, the trade-offs between speed and accuracy are examined for a specific single chip digital signal processor, the TMS32010. Efficient programs written in TMS32010 assembly language [6] have been implemented for each of the computation procedures previously described.

4.1 FFT Algorithms

An important feature of the FFT algorithms considered here is that computations can be performed in-place — the output values can overlay the input values. This requires bit-reverse reordering at the input (DIT algorithm) or at the output (DIF algorithm). With in-place computations, a transform length of up to 64 complex points ($N = 64$) can be handled by the TMS32010 using on-chip random access memory (RAM) to store the data.

The bit reversal portion of the algorithms is implemented using straight-line code for maximum speed (a listing of the program is included in Appendix C). The execution times for bit reversal are 20, 39 and 90 μs for $N = 16$, 32 and 64 respectively.

The main part of the FFT algorithms, the butterfly computations, is implemented using looped coding (see Appendix C). In each case, the sine and cosine values are read from a table stored in read-only-memory (ROM). Since the argument of the sine and cosine values lies between 0 and π, only the values for three quarter cycles of the sine function need to be stored in program ROM for fast table look-up (the cosine values are read with a quarter-cycle memory address offset). Data values reside in the first page (128 words) of RAM; all constants, variables, pointers and indices lie in the second page (16 words).

The execution times (excluding bit reversal time) for the double precision DIT are shown in Table 2. In this table rounding refers to conventional up-rounding. This is the only type of rounding that need be applied to the double precision DIT algorithm since all rounding schemes give the same SNR. In Table 3, rounding combinations of interest for the DIF and single precision DIT algorithms are shown. The single precision DIT algorithm has the same execution time as the DIF algorithm. Selective rounding refers to the use of truncation at the sums and up-rounding (or down-rounding) at the products. The figures listed under rounding refer to up-rounding, down-rounding or stage-alternate rounding applied at both sums and products. Note that stage-alternate rounding can be implemented with essentially no speed penalty compared to conventional rounding. The figures for magnitude rounding are for magnitude up- or down-rounding applied at the sums and conventional up-rounding at the products.

From these results, several interesting observations can be made. First, rounding is relatively inexpensive in terms of execution time, yet increases the SNR by a considerable amount in all cases. For example, the processing time for the double precision DIT algorithm with rounding is increased

	Execution time μs	
N	Truncation	Rounding
16	429	453
32	1018	1082
64	2371	2523

Table 2 Execution time for double precision DIT

	Execution time μs			
N	Truncation	Sel. round.	Rounding	Mag. round.
16	378	391	404	442
32	890	922	954	1050
64	2067	2143	2219	2447

Table 3 Execution time for DIF and single precision DIT

by only 6% compared with truncation. For DIF and single precision DIT, this increase is less than 4% for selective rounding, about 7% for stage-alternate rounding, and about 18% for magnitude rounding.

In Section 3, stage-alternate rounding applied either to the single precision DIT or the DIF algorithm stood out as giving a high SNR, about the same as that for the double precision DIT algorithm with rounding. When comparing the speed figures, it can be seen that the schemes using stage-alternate rounding are somewhat faster ($\sim$ 12%) than the double precision DIT algorithm.

4.2 Direct Computation

Earlier it was found that of the methods considered, direct computation of the DFT has the highest accuracy. With rounding it performs the same as the ideal fixed-point DFT. The main drawback of this method is the number of computations, which grows in proportion to the square of the transform length.

An important problem with the direct method is that the computations cannot be done in-place, i.e., the input and output must be stored in separate locations in memory. In the case of the TMS32010, this limits the maximum length of the DFT to 32 complex points. Another problem is that the argument of the sine and cosine values needed for the multiplications can lie outside the range of 0 to 2π. Therefore, modulo 2π resolution of the argument is required, which for general N can be time consuming. However, if the transform length N is a power of two, modulo 2π resolution becomes equivalent to a bit masking operation. For fast table look-up, values for one and a quarter

cycles of the sine function are stored in read-only memory (ROM). Another factor which slows down execution is the operation of arithmetic right shift on a 32-bit quantity (for scaling), which cannot be implemented efficiently on the TMS32010. The direct computation is coded on the TMS32010 using looped coding (see the program listing in Appendix C).

For efficient coding of the inner loop, auxiliary registers are used as both pointers and loop counters. Data values occupy the first page of RAM; while constants, variables, pointers and indices are all located in the second page.

Table 4 shows the execution time of a direct DFT computation for $N = 16$ and 32 for both rounding and truncation. The execution time is approximately equal to $10.2N^2 + 4N\mu$s if N is a power of two.

	Execution time μs	
N	Truncation	Rounding
16	2677	2690
32	10575	10600

Table 4 Execution time of direct DFT computation

The results show that rounding is a small fraction of the execution time, since it is done only once for each output point. The time required to perform a 16-complex-point DFT (2.7 ms) surpasses that of a 64-complex-point DFT implemented with the double precision DIT algorithm (2.5 ms, with rounding). In general, the direct DFT is to be only recommended when high accuracy is needed.

In some applications a large fraction of the input points are known to be zero, or only some of the output points are needed. In this case the execution time can become proportional to the product of the number of non-zero input points and the number of desired output points, rather than N^2. These changed circumstances might warrant a reappraisal of the relative merits of the direct DFT and the FFT algorithms.

5. Summary and Conclusions

In this report, a new scaling bound for scaling DFT computations has been developed. The bound is in terms of the easily tested magnitudes of the real and imaginary parts of the input sequence. This bound can be used to guarantee that all intermediate and output values in the decimation-in-time FFT algorithm are representable in fixed-point notation.

A systematic method to measure the bias in arithmetic operations has been developed. The resulting gain and mean compensated signal-to-noise ratio is a measure of the match in the spectral shape. This allows the comparison of different versions of the FFT algorithms and different forms of rounding taking into account these biases.

In some applications such as transform coding of speech and video, the two-way performance of fixed-point transforms is important. In this report, the signal-to-noise ratio for a two-way computation has been shown to be simply related to that for a one-way transform. Experimental data shows that this relationship accurately predicts the two-way performance.

Several rounding methods for use with the FFT algorithms have been examined. Truncation is of course the fastest of the options. However, rounding significantly improves the accuracy of all of the implementations considered. The improvement seems worthwhile for all forms of the DFT in light of the moderate speed penalty on a TMS32010 or similar signal processor.

In this report, several procedures for implementing the discrete Fourier transform of a set of samples on a 16-bit fixed-point processor have been examined. If speed is paramount, the DIF algorithm with truncation is a good choice. It is one of the two fastest schemes, yet its mean compensated SNR is high, indicating that it preserves the signal shape. Indeed, a simple modification to add in a constant value at the last stage of the computation might be considered to implement compensation for the mean offset.

If both accuracy and speed are important considerations, the single precision DIT algorithm with a new form of rounding, dubbed stage-alternate rounding, is a good candidate for implementation. It has a high SNR, the rounding involves only a small increase in computation time over truncation, and it offers an increased input dynamic range (input scaling to $\pi/4$ rather than $1/\sqrt{2}$ as in the DIF algorithm).

Stage-alternate rounding gives high accuracy without the need for full double precision accumulation. This translates to reduced execution time for a single chip DSP implementation or reduced complexity for hardware and custom chip implementations. In addition, stage-alternate rounding can be applied profitably to other fast DFT algorithms. Most such algorithms have in common the partitioning of the computation into stages and modules roughly analagous to the stages and butterflies of a conventional FFT. As such, the use of single precision intermediate values in these modules can be beneficial in reducing execution time or complexity.

Appendix A. Maximum Magnitude of the DFT Components

This appendix develops bounds on the maximum magnitudes of the real and imaginary parts of a complex sequence that result in the magnitude of the real and imaginary parts of the corresponding DFT being less than a given value.

A.1 Maximum Output Values

The output of the DFT can be written as

$$\begin{aligned} \mathrm{Re}[X_k] &= \sum_{n=0}^{N-1} \left(\mathrm{Re}[x_n] \cos \frac{2\pi nk}{N} - \mathrm{Im}[x_n] \sin \frac{2\pi nk}{N} \right) , \\ \mathrm{Im}[X_k] &= \sum_{n=0}^{N-1} \left(\mathrm{Im}[x_n] \cos \frac{2\pi nk}{N} + \mathrm{Re}[x_n] \sin \frac{2\pi nk}{N} \right) . \end{aligned} \tag{A.1}$$

Let the input be bounded as follows,

$$\left|\mathrm{Re}[x_n]\right| \le a \qquad \text{and} \qquad \left|\mathrm{Im}[x_n]\right| \le a \; . \tag{A.2}$$

With these input constaints, the magnitude of $\mathrm{Re}[X_k]$ for a particular k is maximized when the real and imaginary parts of x_n are chosen to have a magnitude a and a sign chosen to match that of the sine and cosine terms, viz.,

$$\begin{aligned} \mathrm{Re}[x_n] &= a \, \mathrm{sgn}\left[\cos \frac{2\pi nk}{N}\right] , \\ \mathrm{Im}[x_n] &= -a \, \mathrm{sgn}\left[\sin \frac{2\pi nk}{N}\right] . \end{aligned} \tag{A.3}$$

Then the real part of the output is bounded as follows

$$\left|\mathrm{Re}[X_k]\right| \le a \sum_{n=0}^{N-1} \left(\left|\cos \frac{2\pi nk}{N}\right| + \left|\sin \frac{2\pi nk}{N}\right| \right) . \tag{A.4}$$

The maximum magnitude of $\mathrm{Im}[X_k]$ is obtained in a similar manner and has the same expression.

Consider the normalized sum,

$$S_{k,N} = \frac{1}{N} \sum_{n=0}^{N-1} \left(\left|\cos \frac{2\pi nk}{N}\right| + \left|\sin \frac{2\pi nk}{N}\right| \right) . \tag{A.5}$$

The maximum output values can be expressed as

$$\left|\mathrm{Re}[X_k]\right| \le N a S_{k,N} \; . \tag{A.6}$$

A.2 Bounds on the Normalized Sum

There are three cases to consider in calculating $S_{k,N}$.

1) $k = 0$. Then $S_{0,N} = 1$.
2) k divides N. Let $K = \mathrm{GCD}(k, N)$ be the greatest common divisor of k and N. There are N/K distinct values of the arguments of the sine and cosine terms, each visited K times. Then $S_{k,N} = S_{1,N/K}$.
3) k relatively prime to N. The arguments of the sine and cosine terms step through all values $2\pi l/N$, $l = 0, \ldots, N-1$. Then $S_{k,N} = S_{1,N}$.

These relationships mean that only terms of the form $S_{0,M}$ or $S_{1,M}$ need be evaluated. Consider the term $S_{1,M}$, where

$$S_{1,M} = \frac{1}{M} \sum_{n=0}^{M-1} \left(\left|\cos \frac{2\pi n}{M}\right| + \left|\sin \frac{2\pi n}{M}\right| \right) . \tag{A.7}$$

Consider the case that M is a multiple of 4. Then the two terms in the sum contribute equally to the overall result, and the sum for $n < M/2$ is equal to the sum for $n \geq M/2$. Let $M = 4K$,

$$\begin{aligned} S_{1,4K} &= \frac{1}{K} \sum_{n=0}^{2K-1} \sin \frac{\pi n}{K} \\ &= \frac{1}{K} \mathrm{Im}\Big[\sum_{n=0}^{2K-1} W_{4K}^{n} \Big] \\ &= \frac{1}{K} \frac{\cos \frac{\pi}{4K}}{\sin \frac{\pi}{4K}} \end{aligned} \tag{A.8}$$

It can be shown that $S_{1,4K}$ increases monotonically from 1 to $4/\pi$ with increasing K,

$$1 \leq S_{1,4K} < \frac{4}{\pi} . \tag{A.9}$$

It remains to be shown that the same bounds hold for M not a multiple of 4. If the number of terms M is even but not a multiple of 4, the sum defining $S_{1,2M}$ can be separated into a sum over the even numbered terms and a sum over the odd numbered terms. This tack can be used to show that $S_{1,M}$ is equal to $S_{1,2M}$. Similarly for M odd, by separating even and odd terms, it can be shown that $S_{1,M}$ is equal to $S_{1,2M}$, which in turn is equal to $S_{1,4M}$. This completes the equivalences: the sum with an odd number of terms is the same as the sum with $4M$ terms; the sum with an even number (but not a multiple of 4) of terms is the same as the sum with $2M$ terms. Fig. A.1 shows the values of $S_{1,M}$ as a function of M. It can be seen that this sum quickly approaches its upper bound for moderate values of M.

A.3 Application to the DFT

A bound on the magnitude of the real and imaginary parts of the output of a DFT of length N is found by calculating all values of $S_{k,N}$ for $k = 0, \ldots, N-1$. The maximum over k of $S_{k,N}$ is

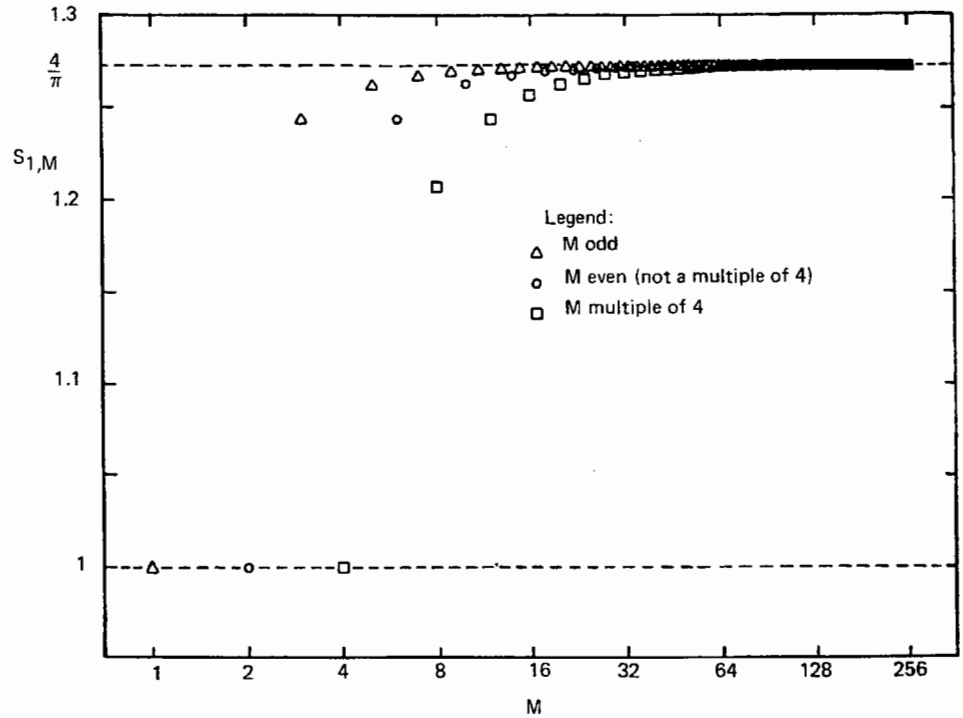


Fig. A.1 Value of the sum $S_{1,M}$

easily shown to be equal to $S_{1,N}$. Note however that the maximum value can occur for values of k other than one. The bounds on $S_{k,N}$ lead to bounds on the DFT output values (see Eq. (A.6)). For $k = 0, 1, \ldots, N-1$,

$$\begin{aligned} \left|\mathrm{Re}[X_k]\right| &\leq NaS_{k,N} \leq NaS_{1,N} < Na\frac{4}{\pi} , \\ \left|\mathrm{Im}[X_k]\right| &\leq NaS_{k,N} \leq NaS_{1,N} < Na\frac{4}{\pi} . \end{aligned} \tag{A.10}$$

For particular values of N and k, the value of $S_{k,N}$ provides a bound on the DFT value. This bound is tight in the sense that a worst case input sequence can be devised such that either $\left|\mathrm{Re}[X_k]\right|$ or $\left|\mathrm{Im}[X_k]\right|$ takes on the value $NaS_{k,N}$. For a particular N, $S_{1,N}$ provides a bound on all output values. For all but very small values of N, the rightmost bound gives a convenient and reasonably tight bound on all output values.

If the input sequence is purely real, separate bounds on the real and imaginary parts of the output sequence can be found. The techniques to generate the bounds are similar to those used above. The real part of the output for a real input sequence is bounded by Na, while the imaginary part of the output is bounded by $Na\,2/\pi$.

Appendix B. Gain and Mean Compensated Error

Consider a difference of complex values of the form

$$D_k = a(Y_k - b) - X_k \ . \tag{B.1}$$

The real factor a and the complex factor b may be used to compensate for differences in gain and mean value between the reference signal X_k and the test signal Y_k. In general, the real and imaginary parts of b may be chosen independently. However in many practical situations involving the use of the DFT for a complex sequence, it is more reasonable to require that the mean offset for the real and imaginary components be the same. With this requirement, the resulting formulation is the same as that which results when the real and imaginary parts of the data are considered to be concatenated into a sequence of real data of twice the length. This being so, for ease of exposition, the discussion that follows assumes real data, with a real gain factor a and a real mean offset b.

The sum of the squared difference terms can be expressed as

$$\varepsilon = \sum_{k=0}^{N-1} D_k^2 \ . \tag{B.2}$$

A conventional squared error is obtained with $a = 1$ and $b = 0$. The difference term for this case will be denoted by E_k where $E_k = Y_k - X_k$. The following subsections will consider optimized values for a and b. However, first a shorthand notation for some sums is introduced,

$$S_x = \sum_{k=0}^{N-1} X_k \ , \qquad S_y = \sum_{k=0}^{N-1} Y_k \ , \qquad S_e = \sum_{k=0}^{N-1} E_k \ . \tag{B.3}$$

Sums of cross-products and squared terms can be similarly denoted as follows,

$$\begin{aligned} S_{xx} &= \sum_{k=0}^{N-1} X_k^2 \ , \qquad S_{xy} = \sum_{k=0}^{N-1} X_k Y_k \ , \\ S_{yy} &= \sum_{k=0}^{N-1} Y_k^2 \ , \qquad S_{ee} = \sum_{k=0}^{N-1} E_k^2 \ , \\ S_{ey} &= \sum_{k=0}^{N-1} E_k Y_k \ . \end{aligned} \tag{B.4}$$

B.1 Mean Compensated Squared Difference

If $a = 1$, then the optimal b is easily found as a difference in sample means

$$\begin{aligned} b &= \frac{S_y - S_x}{N} \\ &= \frac{S_e}{N} \ , \end{aligned} \tag{B.5}$$

and the resulting mean compensated squared difference as

$$\varepsilon = S_{ee} - \frac{S_e^2}{N} \ . \tag{B.6}$$

The first term represents the squared error for a conventional definition, while the last term represents the decrease with mean compensation.

B.2 Gain Compensated Squared Difference

If $b = 0$, the optimal a can be expressed as a normalized cross-correlation sum

$$\begin{aligned} a &= \frac{S_{xy}}{S_{yy}} \\ &= 1 - \frac{S_{ey}}{S_{yy}} , \end{aligned} \tag{B.7}$$

and the resulting gain compensated squared difference can be written as

$$\begin{aligned} \varepsilon &= S_{xx} - \frac{S_{xy}^2}{S_{yy}} \\ &= S_{ee} - \frac{S_{ey}^2}{S_{yy}} . \end{aligned} \tag{B.8}$$

The last term in the last line represents the decrease with gain compensation. For small differences, the squared difference as expressed in the first line of the equation involves the difference between two terms of almost equal magnitude. This can lead to numerical problems. For this reason, a formulation using the error sums as in the last line of the equation is preferred for computations.

B.3 Gain and Mean Compensated Squared Difference

Some observations will help in simplifying the analysis for this case. Given the gain a, the value b which minimizes $\sum_k D_k^2$ will set the mean of the $\sum_k D_k$ to zero,

$$\begin{aligned} \sum_{k=0}^{N-1} D_k &= aS_y - S_x - abN \\ &= 0 . \end{aligned} \tag{B.9}$$

The factor b can be considered to consist of two components — one which compensates for the mean of X_k and another which compensates for the mean of Y_k. With this choice of b, the factor a can be chosen to minimize the sum of the squared difference of the two zero mean sequences. Consider a new set of zero mean variables

$$\begin{aligned} X_k' &= X_k - \frac{S_x}{N} , \qquad Y_k' = Y_k - \frac{S_y}{N} , \\ E_k' &= E_k - \frac{S_e}{N} . \end{aligned} \tag{B.10}$$

The gain compensated case can now be applied to this new set of variables. The optimal a is given by

$$a = 1 - \frac{S_{e'y'}}{S_{y'y'}} , \tag{B.11}$$

where the sums of the primed variables are defined as

$$S_{y'y'} = S_{yy} - \frac{S_y^2}{N} , \qquad S_{e'y'} = S_{ey} - \frac{S_e S_y}{N} . \tag{B.12}$$

The resulting gain and mean compensated squared difference is

$$\varepsilon = S_{e'e'} - \frac{S_{e'y'}^2}{S_{y'y'}} , \tag{B.13}$$

where

$$S_{e'e'} = S_{ee} - \frac{S_e^2}{N} . \tag{B.14}$$

This expression have been arranged such that the computations can be carried out entirely in terms of the sums S_e, S_y, S_{ee}, S_{yy}, and S_{ey}. This formulation tends to avoid numerical difficulties in calculating the sum of the squared differences when the error is small. The value of b which sets the mean of the error to zero can be expressed in terms of a (see Eq. (B.9)). Replacing a by its optimal value,

$$b = \frac{1}{N}\left[\frac{S_e S_{yy} - S_y S_{ey}}{S_{y'y'} - S_{e'y'}}\right] . \tag{B.15}$$

Appendix C. TMS320 Routines

This appendix contains program sections for implementing the various DFT algorithms on a TI TMS320 processor.

C.1 Bit Reversal

This program section performs bit-reversed reordering on a complex array (real and imaginary arrays) of data. This is implemented in straight line code for the particlar transform length of interest. The example which follows gives the code for bit reversal for the case of a 16-point radix-2 transform. The data is asumed to reside in RAM. The constants `XO` and `YO` denote the starting memory address of the real and imaginary arrays respectively.

```
* Bit reverse X array (real part)
        SWAP      XO+1,XO+8
        SWAP      XO+2,XO+4
        SWAP      XO+3,XO+12
        SWAP      XO+5,XO+10
        SWAP      XO+7,XO+14
        SWAP      XO+11,XO+13

* Bit reverse Y array (imaginary part)
        SWAP      YO+1,YO+8
        SWAP      YO+2,YO+4
        SWAP      YO+3,YO+12
        SWAP      YO+5,YO+10
        SWAP      YO+7,YO+14
        SWAP      YO+11,YO+13

        END

*-------Macro for swapping content of two memory locations-------
SWAP    $MACRO    A,B
        LARK      AR0,:A:
        LARK      AR1,:B:
        LAC       *,0,1       ;load accumulator with A
        SACL      TEMP        ;store in temporary location
        LAC       *,0,0       ;load accumulator with B
        SACL      *,0,1       ;store in A
        LAC       TEMP        ;retrieve A from temporary location
        SACL      *,0,0       ;store in B
        $END
```

C.2 Double Precision DIT FFT

This program section implements a radix-2 decimation-in-time FFT of a set of fixed-point complex values. The butterfly computations employ double precision computations as far as possible. The input values are assumed to be stored in bit-reversed order.

```
        LACK    1
        SACL    LE              ;LE=1
        SACL    L               ;L=1
        LACK    FFTSIZ          ;FFT size
        SACL    DIVLE1

D0260   LAC     LE,1
        SACL    LE1             ;LE1=2*LE

        LAC     DIVLE1,15       ;DIVLE1=FFTSIZ/LE1
        SACH    DIVLE1

        LACK    0
        SACL    K               ;K=0

D0240   LACK    COS0
        LT      K
        MPY     DIVLE1
        APAC
        SACL    SIN
        TBLR    COS             ;read cosine

        LACK    SIN0-COS0
        ADD     SIN
        TBLR    SIN             ;read sine

        LAC     K
        SACL    I               ;I=K
        ADD     ONE
        SACL    K               ;K=K+1

* Butterfly loop start
D0220   LAC     I
        ADD     LE
        SACL    J               ;J=I+LE

* set up pointers for Y(I) and Y(J)
        LACK    Y0              ;start of y-array
        ADD     I               ;x-array starts at zero
        SACL    PYI
        LACK    Y0
        ADD     J
        SACL    PYJ
```

```
LAR     AR0,J
LT      *,1
MPY     COS
PAC
LAR     AR1,PYJ
LT      *,0
MPY     SIN
SPAC
SACH    HBIT1
SACL    LBIT1           ;X(J)*cos-Y(J)*sin

MPY     COS
PAC
LT      *,1
MPY     SIN
APAC
SACH    HBIT2
SACL    LBIT2           ;Y(J)*cos+X(J)*sin

LAR     AR1,I
LAC     *,15,0
SUBH    HBIT1
SUBS    LBIT1
ADD     ONE,15          ;rounding
SACH    *,0,1

LAC     *,15
ADDH    HBIT1
ADDS    LBIT1
ADD     ONE,15          ;rounding
SACH    *

LAR     AR1,PYI
LAC     *,15,0
SUBH    HBIT2
SUBS    LBIT2
ADD     ONE,15          ;rounding
LAR     AR0,PYJ
SACH    *,0,1

LAC     *,15
ADDH    HBIT2
ADDS    LBIT2
ADD     ONE,15          ;rounding
SACH    *,0,0

LAC     I
ADD     LE1
SACL    I               ;I=I+LE1
```

```
        LACK    NM1
        SUB     I
        BGEZ    DO220           ;IF (I.LE.N-1) GO TO DO220

        LAC     K
        SUB     LE
        BLZ     DO240           ;IF (K.LE.LE-1) GO TO DO240

        LAC     LE1
        SACL    LE              ;LE=LE1

        LACK    1
        ADD     L
        SACL    L               ;L=L+1

        LACK    LOG2N
        SUB     L
        BGEZ    DO260           ;IF (L.LE.LOG2N) GO TO DO260

        END
```

C.3 Single Precision DIT FFT

This program section implements a radix-2 decimation-in-time FFT of a set of fixed-point complex values. The input values are assumed to be stored in bit-reversed order.

```
        LACK    1
        SACL    LE              ;LE=1
        SACL    L               ;L=1
        LACK    FFTSIZ          ;FFT size
        SACL    DIVLE1

DO260   LAC     LE,1
        SACL    LE1             ;LE1=2*LE

        LAC     DIVLE1,15       ;DIVLE1=FFTSIZ/LE1
        SACH    DIVLE1

        LACK    0
        SACL    K               ;K=0
```

```
D0240   LACK    COSO
        LT      K
        MPY     DIVLE1
        APAC
        SACL    SIN
        TBLR    COS             ;read cosine

        LACK    SINO-COSO
        ADD     SIN
        TBLR    SIN             ;read sine

        LAC     K
        SACL    I               ;I=K
        ADD     ONE
        SACL    K               ;K=K+1

* Butterfly loop start
D0220   LAC     I
        ADD     LE
        SACL    J               ;J=I+LE

* set up pointers for Y(I) and Y(J)
        LACK    YO              ;start of y-array
        ADD     I               ;x-array starts at zero
        SACL    PYI
        LACK    YO
        ADD     J
        SACL    PYJ

        LAR     ARO,J
        LT      *,1
        MPY     COS
        PAC
        LAR     AR1,PYJ
        LT      *,0
        MPY     SIN
        SPAC
        ADD     ONE,14          ;rounding
        SACH    UR,1            ;X(J)*cos-Y(J)*sin

        MPY     COS
        PAC
        LT      *,1
        MPY     SIN
        APAC
        ADD     ONE,14          ;rounding
        SACH    UI,1            ;Y(J)*cos+X(J)*sin
```

```
LAR     AR1,I
LAC     *,15,0
SUB     UR,15
ADD     ONE,15          ;rounding
SACH    *,0,1

ADDH    UR
SACH    *,0

LAR     AR0,PYJ
LAR     AR1,PYI
LAC     *,15,0
SUB     UI,15
ADD     ONE,15          ;rounding
SACH    *,0,1

ADDH    UI
SACH    *,0,0

LAC     I
ADD     LE1
SACL    I               ;I=I+LE1

LACK    NM1
SUB     I
BGEZ    D0220           ;IF (I.LE.N-1) GO TO D0220

LAC     K
SUB     LE
BLZ     D0240           ;IF (K.LE.LE-1) GO TO D0240

LAC     LE1
SACL    LE              ;LE=LE1

LACK    1
ADD     L
SACL    L               ;L=L+1

LACK    LOG2N
SUB     L
BGEZ    D0260           ;IF (L.LE.LOG2N) GO TO D0260

END
```

C.4 DIF FFT

This program section implements a radix-2 decimation-in-frequency FFT of a set of fixed-point complex values. The butterfly computations employ double precision computations as far as possible. The output values are stored in bit-reversed order.

```
        LACK    FFTSIZ          ;FFT size
        SACL    LE              ;LE=FFTSIZ
        LACK    1
        SACL    DIVLE
        SACL    L               ;L=1

D0260   LAC     LE,15
        SACH    LE1             ;LE1=LE/2

        LACK    0
        SACL    K               ;K=0

D0240   LACK    COSO
        LT      K
        MPY     DIVLE
        APAC
        SACL    SIN
        TBLR    COS             ;read cosine

        LACK    SINO-COSO
        ADD     SIN
        TBLR    SIN             ;read sine

        LAC     K
        SACL    I               ;I=K
        ADD     ONE
        SACL    K               ;K=K+1

* Butterfly loop start
D0220   LAC     I
        ADD     LE1
        SACL    J               ;J=I+LE1

* set up pointers for Y(I) and Y(J)
        LACK    YO              ;start of y-array
        ADD     I               ;x-array starts at zero
        SACL    PYI
        LACK    YO
        ADD     J
        SACL    PYJ
```

```
LAR     AR0,I
LAR     AR1,J

LAC     *,15,1
SUB     *,15
ADD     ONE,15          ;rounding
SACH    UR

ADDH    *,0
SACH    *

LAR     AR0,PYI
LAR     AR1,PYJ

LAC     *,15,1
SUB     *,15
ADD     ONE,15          ;rounding
SACH    UI

ADDH    *,0
SACH    *,0,1

LT      UI
MPY     COS
PAC
LT      UR
MPY     SIN
APAC
ADD     ONE,14          ;rounding
SACH    *,1

LAR     AR1,J

MPY     COS
PAC
LT      UI
MPY     SIN
SPAC
ADD     ONE,14          ;rounding
SACH    *,1,0

LAC     I
ADD     LE
SACL    I               ;I=I+LE

LACK    NM1
SUB     I
BGEZ    D0220           ; IF (I.LE.N-1) GO TO D0220

LAC     K
SUB     LE1
BLZ     D0240           ;IF (K.LE.LE1-1) GO TO D0240
```

```
        LAC     LE1
        SACL    LE              ;LE=LE1

        LAC     DIVLE,1
        SACL    DIVLE

        LACK    1
        ADD     L
        SACL    L               ;L=L+1

        LACK    LOG2N
        SUB     L
        BGEZ    D0260           ;IF (L.LE.LOG2N) GO TO D0260

        END
```

C.5 Direct DFT

This program section implements the direct computation of the DFT of fixed-point complex values. Although this program is designed to work for N equal to a power of 2, it will also work for any other value of N (less than 32) provided that the modulo N resolution code is modified accordingly.

```
        LAC     NM1
        SACL    I               ;I=N-1

D0100   ZAC
        SACL    SUM1H
        SACL    SUM1L           ;SUM1=0
        SACL    SUM2H
        SACL    SUM2L           ;SUM2=0

        LARK    AR0,XI15        ;pointer to last element of XI
        LARK    AR1,YI15        ;pointer to last element of YI

D0200   SAR     AR0,K

        MODN    I,K,NM1,IKMODN  ;IKMODN=MOD(I*K,N)
        COSIN   IKMODN,SINO,COSO,COS,SIN ;get sine and cosine values
```

```
LT        *,1
MPY       COS
PAC
LT        *-,0
MPY       SIN
SPAC                        ;ACCUM=X(K)*cos-Y(K)*sin

RSHIFT    LG2NM1,MASK       ;shift product LOG2N-1 bits

ADDH      SUM1H
ADDS      SUM1L
SACH      SUM1H
SACL      SUM1L             ;SUM1=SUM1+ACCUM

MPY       COS
PAC
LT        *
MPY       SIN
APAC                        ;ACCUM=Y(K)*cos+X(K)*sin

RSHIFT    LG2NM1,MASK       ;shift product

ADDH      SUM2H
ADDS      SUM2L
SACH      SUM2H
SACL      SUM2L             ;SUM2=SUM2+ACCUM

BANZ      DO200             ;IF (K.NE.0) GO TO DO200

LAR       AR0,PX0
ZALH      SUM1H             ;rounding
ADDS      SUM1L
ADD       ONE,15
SACH      *-,0,1
SAR       AR0,PX0           ;store X0(I)

LAR       AR1,PY0
ZALH      SUM2H             ;rounding
ADDS      SUM2L
ADD       ONE,15
SACH      *-,0,0
SAR       AR1,PY0           ;store Y0(I)

LAC       I
SUB       ONE
SACL      I                 ;I=I-1
BGEZ      DO100             ;IF (I.GE.0) GO TO 100

END
```

```
*-------Macro for modulo N (power of two) resolution-------
MODN    $MACRO    I,K,NM1,IKMODN
        LT        :I:
        MPY       :K:
        PAC
        AND       :NM1:
        SACL      :IKMODN:
        $END

*-------Macro for reading sine & cosine values from table-------
COSIN   $MACRO    IKMODN,SINO,COSO,COS,SIN
        LACK      :SINO:
        ADD       :IKMODN:
        TBLR      :SIN:
        LACK      :COSO:
        ADD       :IKMODN:
        TBLR      :COS:
        $END

*-------Macro to perform arithmetic right shift -------
*----------on 32-bit content of accumulator------------
RSHIFT $MACRO     SHIFT,MASK
        SACH      TRH
        SACL      TRL
        LAC       TRL,16-:SHIFT:
        SACH      TRL
        LAC       :MASK:
        AND       TRL
        ADD       TRH,16-:SHIFT:
        $END
```

References

1. A. V. Oppenheim, Ed., *Applications of Digital Signal Processing*, Prentice-Hall, 1978.
2. T.-Thông and B. Liu, "Fixed-point fast Fourier transform error analysis", *IEEE Trans. Acoustics, Speech, Signal Processing*, vol. ASSP-24, pp. 563–573, Dec. 1976.
3. J. W. Cooley, P. A. Lewis, and P. D. Welch, "The fast Fourier transform algorithm: Programming considerations in the calculation of sine, cosine and Laplace transforms", *J. Sound Vib.*, vol. 12, pp. 315–337, July 1970.
4. M. J. Narashimha and A. M. Peterson, "On the computation of the discrete cosine transform", *IEEE Trans. Commun.*, vol. COM-26, pp. 934–936, June 1978.
5. A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975.
6. Texas Instruments, *TMS32010 User's Guide*, Texas Instruments, 1983.