TAMPERE UNIVERSITY OF TECHNOLOGY

ATAUL GHALIB

ANALYSIS OF FIXED-POINT AND FLOATING-POINT QUANTIZATION IN FAST FOURIER TRANSFORM

Master of Science Thesis

Examiner: Professor Jarmo Takala
Examiner and topic approved in the Computing and Electrical Engineering Faculty Council meeting on 07.12.2011

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
GHALIB, ATAUL: Analysis of Fixed-Point and Floating-Point Quantization in Fast Fourier Transform
Master of Science Thesis, 56 pages, 9 enclosure pages.
June 2013
Major: Digital and Computer Electronics
Examiner: Prof. Jarmo Takala
Keywords: fixed-point quantization, floating-point quantization, fast Fourier transform, number system, digital signal processor, signal-to-noise ratio

Digital Signal Processors (DSPs) can be categorized on the basis of number system used for the arithmetic calculations. Either they can be fixed-point or floating-point processors. Some processors also support both fixed-point and floating-point number systems. Performing signal quantization is very necessary step in digital signal processing, either it is applied on the input signal to convert from analogue to digital domain or on the intermediate digital signal to keep it in the representable range to avoid overflow, it always introduces some error hence reducing the signal-to-noise ratio. Both number systems do not introduce the same amount of error when quantization is applied. So when implementing some DSP algorithm e.g. fast Fourier transform on a DSP processor, quantization analysis need to be performed for fixed-point and floating-point number system in order to have an optimized SNR.

In this thesis, we have presented such quantization analysis on double precision floating-point FFT model and optimized fixed-point and floating-point quantization for reference FFT model in order to generate the same SNR. For this purpose fixed-point and floating-point quantization models are generated and placed in the reference FFT model and experiments are performed with randomly generated complex stimulus. Results have shown that generally floating-point quantized FFT model shows better SNR results than fixed-point quantized FFT model, but at smaller number of exponent bits and higher number fractional bits floating-point and fixed-point results are almost the same.

PREFACE

This thesis work was completed in Department of Pervasive Systems, Tampere University of Technology.
First of all I would like to thank Almighty Allah for making me achieve this milestone in my life. I would also like to express my sincere gratitude to my parents especially to my mother Amatun Naseer (may Allah bless her soul) who had been the great source of inspiration for me for whole of my life for learning and advancing in academics, and it would never have been the same without her efforts. I would also like to thank my wife Samara Ghalib whose love and consistent support has made me accomplish this thesis work.
This piece of work has been completed under the supervision of Mr. Jarmo Takala and I would like to acknowledge his continued support and help during all the time of my thesis and thank him for this support. He has been very patient and continually supportive for me to complete this thesis work and provided all the necessary knowledge and technical support regarding work place and IT equipment.
I would also like to thank all my colleagues and friends in the university who helped me during this work technically or academically.
At the end I would like to thank all may family members for being supportive throughout my studies.

Tampere, May 15. 2013

Ataul Ghalib

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

DSP	Digital Signal Processing
CFT	Continues Fourier Transform
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
TTA	Transport Triggered Architecture
ADC	Analog to digital converter
PDF	Probability Density Function
SNR	Signal to Noise Ratio
SQNR	Signal to Quantization Noise Ration
LSb	Least Significant bit
LSB	Least Significant Byte
MSb	Most Significant bit
MSB	Most Significant Byte

1. INTRODUCTION

Quantization is performed when a signal is converted from continues to discrete domain. A continues time signal can have different value at any instance of time or frequency. The process of quantization defines a same value for a specific interval of time. This could also be understood that a signal in analogue domain would have infinite number values and would be impossible to represent it in digital domain where we have limited amount of bits for the representation. So quantization limits the signal to be represented using limited amount of bits which then introduces certain amount of error in the signal. Signal-to-noise ratio (SNR) could be used to measure the error introduced by quantization.

Mainly there are two different types of number representation, fixed-point number representation and floating-point number representation. Each representation system behaves differently when quantization is applied. Quantization error analysis helps to understand and optimize the behaviour of certain implementation for representation systems.

Fast Fourier Transform (FFT) is an algorithm very abundantly used in digital signal processing (DSP). Implementation of FFT on different hardware platforms introduces different design and optimization challenges. One of the biggest challenges is to select a hardware platform supporting a specific number representation system. So quantization analysis for fixed-point and floating-point representation system is very important in order to optimize an FFT implementation.

A double precision floating-point FFT model is used as reference and quantization analysis for fixed-point and floating-point representation systems of this reference model is the main scope of this research work.

For the purpose of quantization analysis two different quantization models are generated based on two number representation systems, fixed-point and floating-point quantization models. These quantization models are then placed in reference FFT model implementation to have two different FFT models for the analysis. Different sets of stimulus are generated and fed to these quantized FFT models along with reference FFT model to understand the affect of quantization when using fixed-point and floating-point number representations.

This thesis work is organized in such a way that initial chapters include the theoretical background. It is important to understand the complete number system and different representations used which are explained in chapter 2. Chapter 3 deals with quantization techniques used in the DSP domain and their affect on fixed and floating-point arithmetic. Then some background knowledge on Fourier Transform its properties and its types

e.g. Discrete Fourier Transform (DFT) is shared. This chapter also discusses the methods of calculating DFT, which is called FFT algorithm, which is to be used in later chapters as the base of our analysis. After discussing all the necessary background knowledge implementation and working model is discussed in chapter 5. It discusses in detail how the quantization models, which are used to perform quantization analysis on FFT implementation under consideration, are implemented. What types of quantization techniques are used, it also discusses the FFT implementation under consideration to have better understanding of our base implementation. This chapter is very important to understand the analysis performed in chapter 6. Analysis is performed based on different quantization parameters and selecting different quantization models generated.

2. NUMBER SYSTEMS

In this chapter, we will discuss different numeric representations used in processor design for arithmetic calculations. We will also try to highlight different positive and negative key points of these numeric representations and will present their comparison. Later in the chapter, we will briefly discuss quantization of signals, its need and possible affects of result. Quantization affects/errors will be discussed separately for fixed-point and floating-point numeric representations in the chapter 3.

2.1. Arithmetic Representation in Signal Processing

Numeric representation is quite an important issue for processor design. In digital signal processing (DSP) domain, there are a number factors, which define the type of DSP processor to be used during the design of an application-specific processor. Computational efficiency, ease of implementation, memory consumption, time to market and required precision are some of the key factors. Because of the enormous amount of computations to be performed in implementing any digital signal processing task, design and performance of a DSP processor is heavily affected by the numeric representation selected to be used. Its capacity to represent the dynamic range of results in lesser number of bits is the very key point for an arithmetic representation. Word width in the processor domain describes maximum size of one instruction and the maximum size of addressable memory, so representing the very dynamic range of numbers in the given word size would be one key characteristic required for choosing an arithmetic representation. Large dynamic range might be required in some cases where as simplicity and less computational overhead might be more useful in some other situations, so it is always a trade-off between the two. Figure 1 shows the types of DSP processors in based upon the numeric representation. The main two categories are fixed-point representation and floating-point representation. Both representations have their own pros and cons, before discussing those; we will first look into detail of these representations, their characteristics and types.

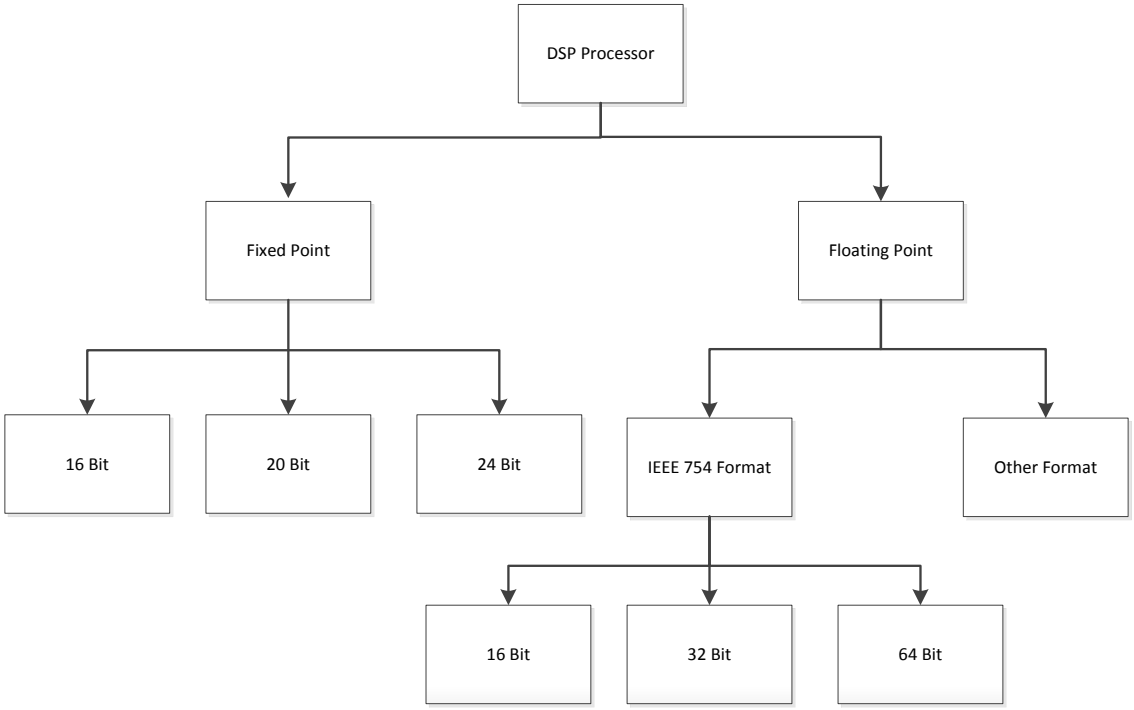


Figure 1 Types of DSP processors on the basis of numeric representations

2.2. Fixed-Point Arithmetic

Fixed-point arithmetic is generally used when cost, speed and complexity is important [5]. Since these factors are mostly of interest, therefore fixed-point arithmetic is quite heavily used in DSP processors. This representation is capable of dealing with positive and negative integers and whole numbers. Generally this type of arithmetic representation is used in control operations; address calculations etc. As the name of this arithmetic clarifies the place of decimal point is fixed in this representation and the step between the two representable numbers is always constant. The decimal point could be in the end or at a certain location e.g. a number *xxx.xx* represents fixed-point arithmetic having two bits after decimal point. The place of decimal point is selected according to the precession requirements; higher number of bits after decimal point would mean more precession. The bits before the decimal point are called integer bits and the ones after the decimal point are called fractional bits. Fixed-point could be further divided into integer and fractional representation. The major difference in the representation is place of decimal point, for integer binary representation the decimal point is always at the end, where as for fractional representation it can be anywhere depending upon the predefined number of fractional bits for the arithmetic. It should be kept in mind that decimal point is actually not stored in the memory this is just a way how the stored binary bits are interpreted. The data saved in the memory is always just in the form of bits either ‘0’ or ‘1’ and these different representations just allow us to manipulate those bits in different ways in order to achieve our requirements.

2.2.1. Integer Representations

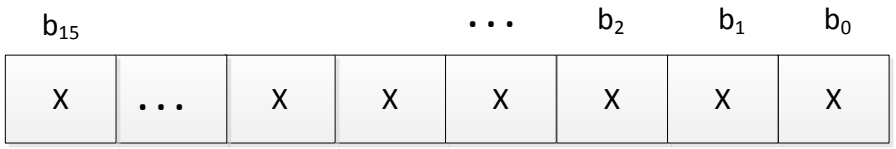


Figure 2 Bit pattern of integer binary representation

This is most straightforward and easy to understand representation. Most Significant bit (MSb) is on the most left hand side and Least Significant bit (LSb) is on the right hand side. If the representation of number is more than 8 bits (8 bits = 1 byte) then the byte orientation depends upon the endian of the representation. For Big-Endian format Most Significant Byte (MSB) is at the extreme left where as for Little-Endian format (mostly used in embedded hardware) MSB is at the extreme right side, the internal orientation of bits in every byte remains the same. In this way, weight of each bit can be readily calculated depending upon its position. Range of representable numbers depends upon the number of bits available. For *n* number of bits the numeric representation range could be found as

$$0 \leq x \leq 2^n - 1. \tag{1.1}$$

Figure 2 shows the bit pattern for integer binary representation where *n* = 16

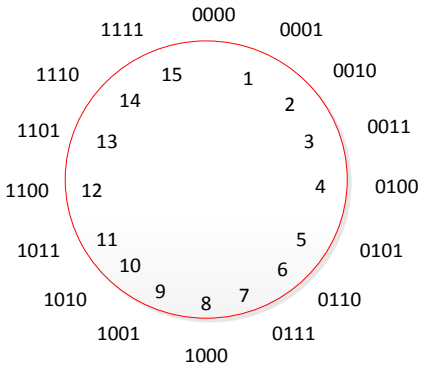


Figure 3 Number range for 4-bit unsigned integer representation

The decimal value for *n* number of binary bit pattern can be calculated by

$$X = 2^0 b_0 + 2^1 b_1 + \ldots + 2^{n-1} b_{n-1} = \sum_{i=o}^{n} 2^i b_i. \tag{1.2}$$

This representation could show numbers only in the positive range, so this is classified as unsigned binary representation and a full range for 4-bit unsigned integer representation is shown in Figure 3.

To be able to represent negative numbers signed binary representation is defined. Although the representable range remains the same but maximum representable positive number is reduced almost by half as shown in Figure 4. First bit is called sign bit and the rest of the bits show the original value. The number is negative if sign bit is 1 and

positive if 0. For *n* number of bits numeric representation in this range would be as follows,

$$-2^{n-1} \leq x \leq 2^{n-1} - 1. \quad (1.3)$$

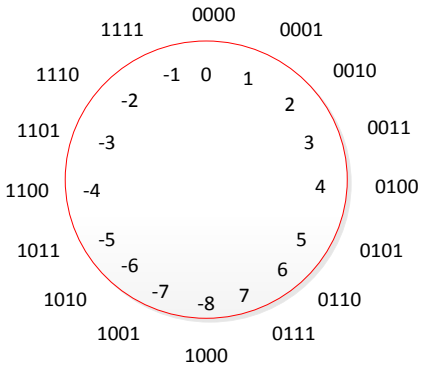


Figure 4 Number range for 4-bit signed integer representation

The decimal value for signed binary representation for *n* number of bits can be calculated as

$$X = -2^{n-1}b_{n-1} + \cdots + 2^1 b_1 + 2^0 b_0 = -(2^{n-1}b_{n-1}) + \sum_{i=o}^{n} 2^i b_i. \quad (1.4)$$

No fractional numbers can be represented in this format because of this reason it has very low precision and have high quantization errors.

2.2.1.1 Properties

Quantization step between two representable numbers is equal to one $Q_{step} = 1$.

Addition of two *n*-bit numbers would result in (*n*+1)-bit result. For N addition operations we need log2 (*N*) bits to store the result to avoid overflow.

Multiplication of two unsigned *n*-bit numbers will result in 2*n*-bit number.

Multiplication of two signed n bit numbers will result in (2*n* – 1)-bit number only the multiplication of (2*n*-1) by itself would result in a 2*n*-bit number. Normally for signed representation this operation, multiplication of two largest negative numbers, is not allowed because there is only one possible result, which could utilize the last bit.

Multiplication in integer binary representation would result in high overflow situations. To avoid we need n extra bits in every subsequent operation.

Any chain operation which has the final result, which could be represented in the given number of bits can be performed without overflow even if the overflow happens in the intermediate operations.

Because of begin highly overflow prone in non-linear arithmetic operations (e.g. multiplication) a lot scaling factors are required while using such arithmetic representation. Some examples proving previous properties are given below.

2.2.1.2 Arithmetic Operations

Consider 4-bit unsigned integer binary representation. Let us first look at addition operation.

```
  1001 = 9          in 4 bits
  1001 = 9          in 4 bits
 ----------
10010 = 18          in 5 bits

10010 → 0010   = 2    in 4 bits quantized result
```

Error in the result because of quantization to the representable range of bits is approximately 88%. This quantization error would be even higher for full scale operands (values utilizing full range of bits, e.g. in 4 bits 1111 is a full scale value) and would be low on the lower values close to zero. To avoid we must use one extra bit to be able to show the correct result.

Now consider a multiplication example with signed binary representation.

```
   0111 = 7          in 4 bits
   0111 = 7          in 4 bits
---------------
00000111
0000111
000111
00000
---------------
00110001 = 49          in 8 bits

00110001 → 0001 = 1        in 4 bits quantized result
```

To be able to show the correct result we need 8 bits and when the result is quantized down to 4 bits the error introduced is approximately 97% which is quite higher than that in addition operation, it would be even more for full scale values. This shows that integer binary representation is very overflow prone unless the result remains in the representable number of bits this representation is useful otherwise the results are completely not acceptable.

2.2.2. Fractional Representation

Although using the binary integer representation we can apply arithmetic operations addition, subtraction, multiplication etc. but in case of non-linear operations like multiplication the required amount of bits in every subsequent operation becomes double. But if we normalize the numbers in the range of [-1, 1) the result of every operation will not overflow. Such representation is called fractional representation.

Figure 5 shows an example of such representation with 16 bits. In this case, one bit is used for sign and 15 bits are used to represent the fractional value.

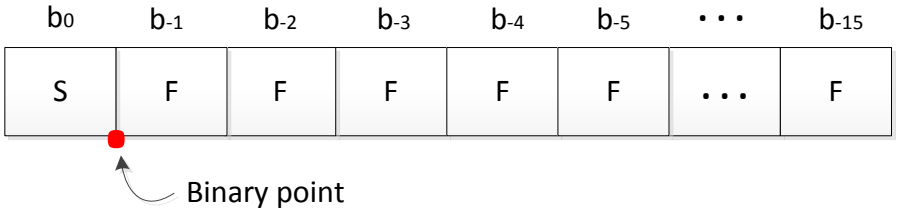


Figure 5 Q15 Fractional fixed-point representation

Figure 6 shows the range of fractional binary representation in case of 4 bits is used for fraction.

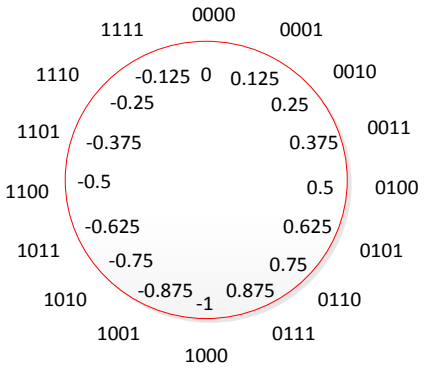


Figure 6 Number representation for 4-bit fractional representation

Fixed-point arithmetic combining both integer and fractional representations can be represented as $Q(M, F)$. Where M is the number bits in integer part and F is the number of bits in fraction. Only fractional representation can be considered a special case where number of bits in integer part is zero. So its notation is also known to be Qx, where x represents the number of fractional bits, as in Figure 5 $Q15$ representation is used.

Please note that some designers consider the sign bit inside and some don't. e.g. in $Q(4,4)$ fractional representation total number of bits required are 9, adding one bit for the sign where as some designers consider it 8 bits in total assigning one bit to sign from the integer part leaving actually 3 bits for integer part [6].

The positive maximum result 1 could be attained only when (-1 x -1) operation is performed. So just like in signed integer representation, here as well this operation is not allowed because the dynamicity attained by the last bit is wasted. So the range of representable numbers here becomes [-1, 1).

$$-1 \leq x \leq 1 - 2^{-(n-1)}. \quad (1.5)$$

Where *n* is the number fractional bits, here we have considered the integer part to be of 1 bit. The decimal value could be calculated as;

$$X = -b_{n-1} + 2^{-1}b_{n-2} + \ldots + 2^{-(n-1)}b_0. \quad (1.6)$$

2.2.2.1 Properties

The quantization step depends upon the number of fractional bits, $Q_{step} = 2^{-(n-1)}$ Where *n* = total number of bits including sign bit. Arithmetic operations like addition, subtraction, multiplication etc. do not go out of range. Quantization error reduces by increasing the number fractional bits. Step between two representable numbers remains same, as in integer representation.

2.2.2.2 Arithmetic Operations

Addition operation might not be a very good candidate in order to show the difference between integer and fractional representations.

Let us take an example of multiplication of two numbers represented in *Q*3 format. The total number of bits required for *Q*3 format is 4, considering the sign bit inside. For the sake of comparison and simplicity let us take the same multiplication operands as in integer representation.

Multiplication:

0.111 = 0.875 in *Q*3
0.111 = 0.875 in *Q*3

00.000111
00.00111
00.0111
00.000

00.101001 = 0.765625 in *Q*6

00.101001 → 0.101 = 0.625 in *Q*3 quantized result

It should be kept in mind that the truncation of bits is not done from the higher bits and lower 4 bits are stored to memory instead this is done according to the *Q*3 format, which is sign bit before the binary point and after that rest of three bits.

Now the percentage error in this case is reduced down to approximately 20%. This result becomes more impressive when we think that the operands used here represent the positive full scale value in *Q*3 format, where as the full scale value arithmetic in singed integer representation introduced approximately 97% error.

The above mentioned two types of fixed-point arithmetic are used according to the need of implementation. There is always a tradeoff; fractional representation provides

accuracy and less quantization on the hand where as integer representation is fast, less CPU intensive and cost effective.

2.3. Floating-Point Arithmetic

Most of the times in DSP computations results need a very large dynamic range to be represented. One way to achieve this dynamicity is to use more and more bits to represent the largest and smallest numbers. This becomes waste of memory because a very wide range remains unused. The processing speed becomes lower because of accessing large memory areas and surely this would also increase the silicon cost [6].

The other way that is used by floating-point arithmetic is to introduce an exponent in the representation. This exponent increases the dynamic range and we are able to represent the very large and very small numbers. The quantization step thus does not remain constant as in the fixed-point representation and varies according to the exponent. Numbers having the same exponent have the same quantization step. The name floating-point comes from the fact that the decimal point is not fixed like fixed-point and it varies with the exponent value.

So floating-point representation consists of two main parts mantissa, the fractional part, in the floating-point representation, which introduces the accuracy and the exponent, the second part, introduces the dynamic range. If we increase the number of bits in the mantissa field more accuracy or resolution can be attained where as increasing the number of bits in the exponent field will provide more dynamicity. So floating-point arithmetic can be optimized in anyway required. But on the same hand floating-point is expensive to implement from hardware cost and CPU cycles point of view. Every floating-point operation will require much more machine cycle then that of fixed-point. Because of this drawback, floating-point is used only where it is very it cannot be avoided.

General floating-point representation can be described by the equation below;

$$x = sign(x)\, M\, \beta^{E}$$

where M is the mantissa, β is the base, and E is the exponent. Mantissa can be normalized as, $1/\beta \leq M < 1$. For binary base this would result the range of mantissa to be [0.5, 1) on the positive side and (-1, -0.5] on the negative side.

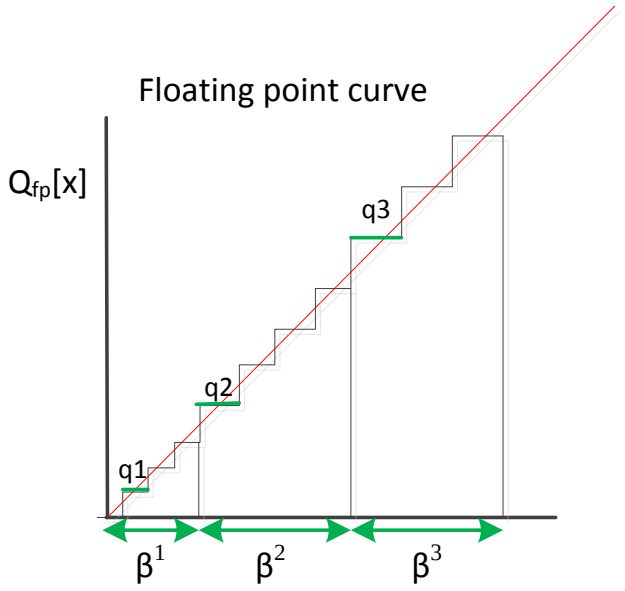


Figure 7 Floating-point quantization curve

So for storing the numbers in the form of floating-point arithmetic we need b_m+b_E+1 bits where b_m is number bits in mantissa, b_E is number of bits in exponent field and an additional bit for the sign is required. But if we look at the range of representation of mantissa in binary base after normalization we can notice that in mantissa the bit before the decimal point will always be 0 and the bit after the decimal point will always be 1.

$$x = \text{sign}(x)\ 0.1XXX\ 2^E$$

So these redundant bits can be removed from storage and thus reducing the total amount of bits required to b_m+b_E-1.

Let us take the example of subtraction operation in order to show the difference between fixed-point and floating-point arithmetic. Consider the subtraction of following two numbers with 5-bit mantissa;

A = $1.0011 \times 2^2 = 4.75_{10}$ in 5 bits mantissa
B = $1.0001 \times 2^1 = 2.125_{10}$ in 5 bits mantissa

There are four steps, which are needed to be performed in floating-point arithmetic;

1. Match the exponent by shifting smaller number to the right

A = 1.0011×2^2
B = 0.1000×2^2

2. Performing the arithmetic operation on mantissa

1.0011 – 0.1000 = 0.1011

3. Normalization of the result

$0.1011 \times 2^2 \rightarrow 1.0110 \times 2^1$

4. Rounding to appropriate number of bits

$1.0110 \times 2^1 = 2.75_{10}$ where as the original result is 2.625_{10} so the error introduced here is 0.125 which in actual is the largest error in this case as the quantization step is 0.125. The error here is 4.5%

2.3.1. Guard Bits

The loss of precision can be avoided in floating-point by using one extra bit in the mantissa. Using such bits in order to increase the precision are called guard bits.

If we use one guard bit in the above mentioned example;

$A = 1.00110 \times 2^2$

$B = 0.10000 \times 2^2$

Now after performing all the four steps the result produced will be 2.625_{10}, which is exactly the same what was expected, with 0% error. Normally by introducing just one guard bit is enough to dramatically reduce the error, as seen above and it is only required in those arithmetic operations where exponents does not match.

2.3.2. Floating-Point Standards

Standardization is quite an important issue and it is done for different purposes. Interoperability, portability, quality, reusability etc. are some of the main factors, which are improved a lot by standardization.

IEEE has developed a standard 754-1985, which explains the binary floating-point arithmetic [23]. This standard is quite widely accepted. This standard was introduced in 1985 and outdated in 2008, when a new standard 754-2008 for floating-point arithmetic was introduced [17]. It quite thoroughly explains different formats, dealing with abnormal situations, rounding methods. These concepts will be discussed shortly here.

The basic floating-point representation contains the three parts as shown in the Figure 8.

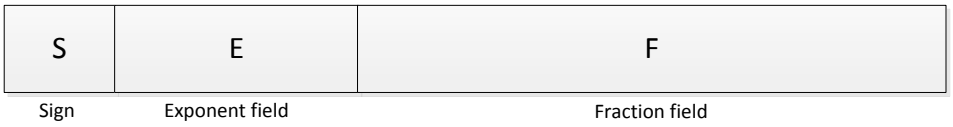


Figure 8 Basic floating-point representation

Here *S* is the sign bit, *E* is the exponent and *F* represents the number of bits in fraction. There are four different floating-point representation formats defined under this standard:

1. Basic single precision floating-point,
2. Extended single precision floating-point,
3. Basic double precision floating-point, and
4. Extended double precision floating-point.

These formats differ in the number of exponent *E*, and number of mantissa *F* bits, as shown in the table below.

Table 1 Floating-point standards

Parameter	Basic Single	Extended Single	Basic Double	Extended Double
Format width	32	43	64	79
Mantissa width (including sign bit)	24	32	53	64
Exponent width	8	11	11	15
Max exponent	+127	+1023	+1023	+16383
Min exponent	-128	-1024	-1024	-16384

The number of bits in the mantissa defines the resolution and the number of bits in the exponent field defines the dynamicity of the floating-point representation format. In the extended formats (single and double) the number of bits in exponent and mantissa can be equal to what mentioned or more. Number of bits mentioned for the mantissa field includes the sign bit, so in actual there are 23, 31, 52, and 63 bits for single, extended single, double and extended formats respectively, where extended formats show minimum number of bits not the exact amount of bits. Exponent bits are basically sign bits as it can be positive or negative, e.g. for basic single precision exponent field is 8 bit signed number. Sign of the mantissa represents the sign of the whole number, where sign of the exponent represents the magnitude of the whole number (positive exponent means higher magnitude and vice versa).

2.3.3. Biased Exponent

Since the exponent field is signed field so the smallest number (negative extreme) is represented by all bits one, e.g. 11111111 in basic single precision. This appears larger than positive extreme exponents. This problem is removed by using the biased exponent. So smallest exponent is represented by all zeros and largest exponent is represented by all ones. So negative exponents would be added with a bias amount which would depend upon the number of exponents e.g. for basic single precision it will be 127 as shown in Figure 9.

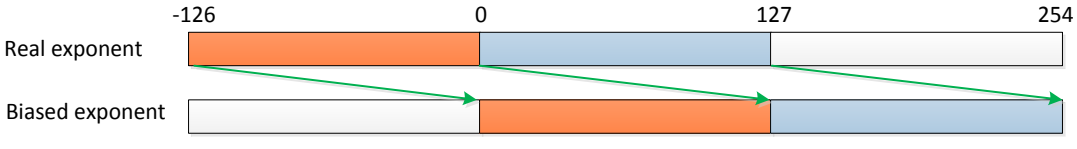


Figure 9 Biased exponent used in floating-point arithmetic

2.3.4. Denormal Numbers

Numbers that are not normal, either too small or too large in magnitude, to be representable using floating-point representation are called denormals. These numbers need special rules. For binary floating-point denormals are;

- $|x| < ½\ 2^{cmin}$
- $|x| > 2^{cmax}$
- +0, -0
- +∞, -∞

where *Cmin* and *Cmax* are the minimum and maximum exponents. In IEEE-754 standard bit patterns 000…0 and 111…1 are reserved for ±0 and ±∞. And a simple solution for the small numbers $(|x| < ½\ 2^{cmin})$ is to quantize numbers to zero. But this will raise a problem that two small non representable numbers subtraction will be zero (because both are flushed to zero) but they will not be equal.

2.3.5. Gradual Underflow

Another solution to the problem mentioned above is gradual underflow. It is done by shifting the fraction to the right. By using underflow smaller numbers use their precision gradually.

2.3.6. Rounding Modes

There are four rounding modes normally defined for IEEE 754 standard;

1. Round towards nearest – this is also called convergent rounding
2. Round towards zero – truncation of magnitude, since the whole magnitude is truncated.
3. Round towards - ∞
4. Round towards +∞

Round towards nearest is the most popular and where as round towards zero is more easy to implement and consumes less clock cycles. Rounding and truncation will be discussed in detail in chapter 3.

3. QUANTIZATION EFFECTS

The organization of this chapter would be that first we will discuss briefly about quantization and its types then quantization effects on fixed-point and floating-point arithmetic will be discussed. At the end we will talk about what parameters are used to analyze the quantization effects.

3.1. Quantization

Quantization is the process of conversion of signals from analog to digital domain. While conversion, representation of each analog value of a signal is not possible to be represented in digital domain, so the result needs to be quantized to a certain number of values, which is called quantization of signal. The amount of values represented in digital domain directly relates to number of bits used, e.g. 3 bits can represent $2^3 = 8$ values. Quantization also comes into play when computational results in the intermediate stages of DSP operations go beyond the representable range and are needed to be quantized to the available number bits. So in short quantization is required and necessary in the DSP computations. Because of the quantization of infinite samples of a signal to finite number of samples signal quality is reduced and introduces error. This is called quantization error and will be discussed in section 3.3. Figure 10 below shows the need of quantization in digital signal processing.

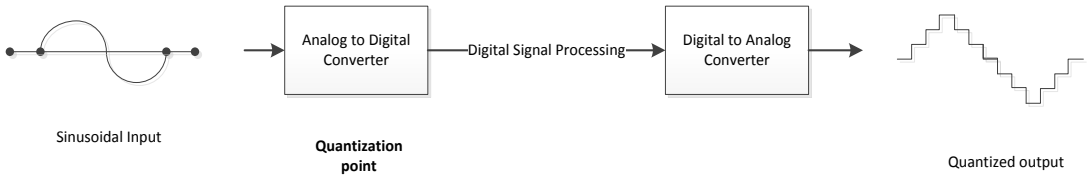


Figure 10 Quantization between analog and digital domain

3.2. Quantization Techniques

There are different quantization techniques. Let us take an example to understand why there is a need of different quantization techniques. Suppose in the analog to digital converter in the Figure 10 above we need quantization and the result could be represented maximum in 3 bits. The maximum values, which could be represented of the input signal is then $2^3 = 8$. So clearly we need to set a range of analog input data to be mapped to one point in digital domain, in order to compensate the infinite number of input val-

ues to just 8 values. Now the question arises here, that what range of values in analog domain should represent a value in quantized domain? This is where different quantization techniques come in. For different quantization techniques limits of the quantization step would be different. These techniques are discussed in the following sections 3.2.1 and 3.2.2.

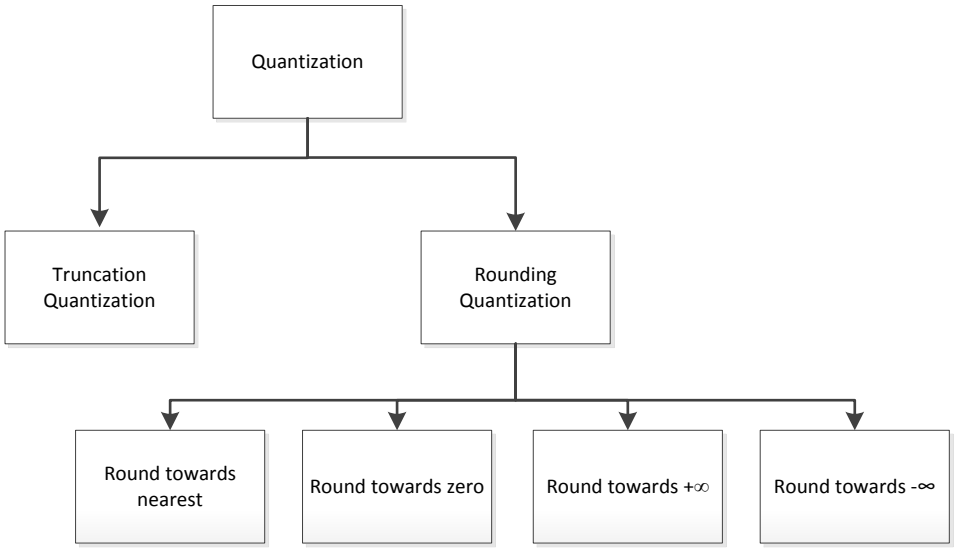


Figure 11 Quantization techniques

As shown in the Figure 11 quantization could be divided into two major groups, truncation and rounding. IEEE 754 standard has defined four rounding modes as shown in the figure. Quantization effects of round towards nearest mode will be discussed in section 3.3.

3.2.1. Truncation Quantization

Truncation is a process in which bits outside the representable range are simply dropped off from the LSb side regardless of the sing of number. Figure 12 explains this idea further.

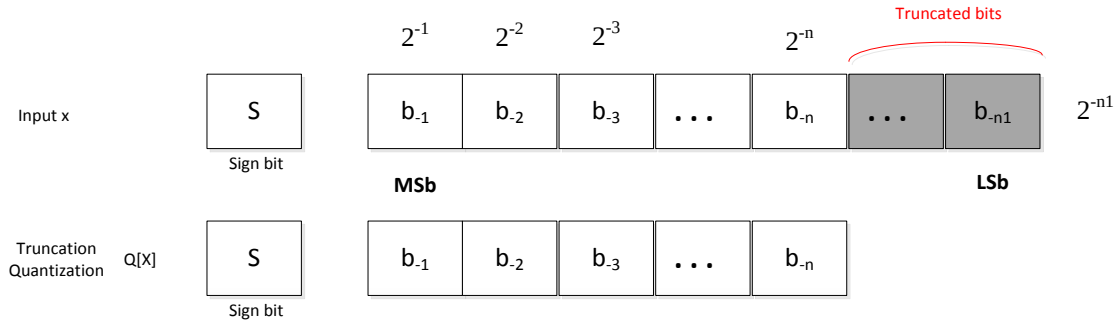


Figure 12 Truncation quantization

In Figure 12, x is a sequence and $Q[x]$ is its truncated version and total number of representable bits are b_n. so bits on the LSb side outside the representable range are

chopped off. If we draw the plot between original sequence and quantized one, the result would be as shown in Figure 13.

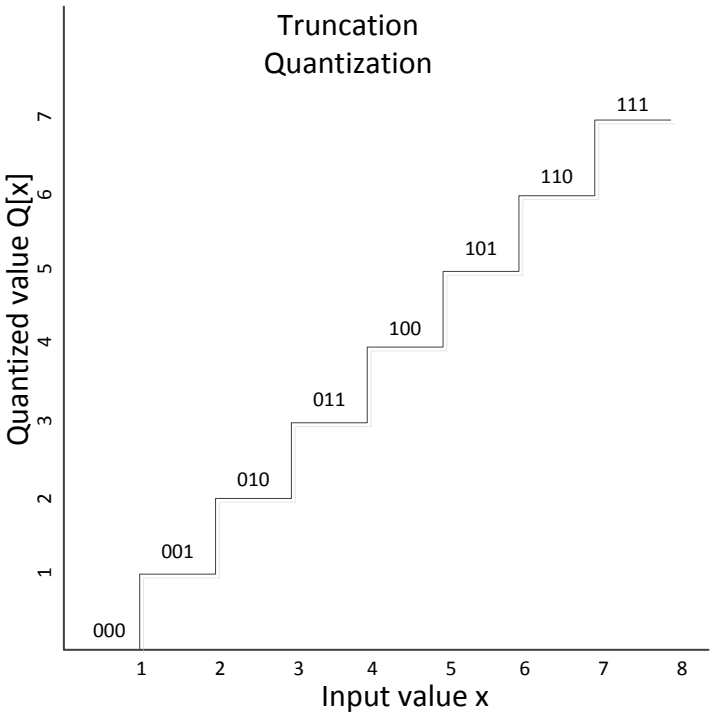


Figure 13 Graph of an input value with 3-bit truncation quantization

From Figure 13 it can be seen that all values before the next representable number are flushed back the previous representable number this introduces large quantization errors. These will further discussed in section 3.3.

Effects of quantization are discussed in the form of quantization error. For truncation quantization, quantization error is affected by the representation used for negative numbers. Representations used for negative numbers are sign magnitude, 1's complement and 2's complement and behavior of quantization error e_Q for these representations could be explained with the following equations as given in [2]

$$0 \leq e_Q \leq 2^{-n} - 2^{-n1} \; and \; |Q[x]| \leq x \tag{3.1}$$

$$0 \leq e_Q \leq 2^{-n} - 2^{-n1} \; and \; |Q[x]| \leq x \tag{3.2}$$

$$-(2^{-n} - 2^{-n1}) \leq e_Q \leq 0 \; and \; |Q[x]| \geq x \tag{3.3}$$

where eq. 3.1, eq. 3.2 and eq. 3.3 represent effects for sign magnitude, 1's complement and 2's complement representations for negative numbers. These three representation exhibit different behaviors for quantization error. E.g. quantization error would be opposite for both sign magnitude and 1's complement; if sequence is positive, error would be negative and vice versa. So error is correlated with the sequence. Where as in 2's complement error is always non positive, hence uncorrelated. Further quantization noise analysis on these representations will be discussed in section 3.3.

3.2.2. Rounding Quantization

Rounding is technique of quantization through which we do not just chop off the extra bits, whereas we round the number towards the nearest representable number of b_n bits. Figure 14 shows the graph between the round quantized sequence $Q[x]$ and x.

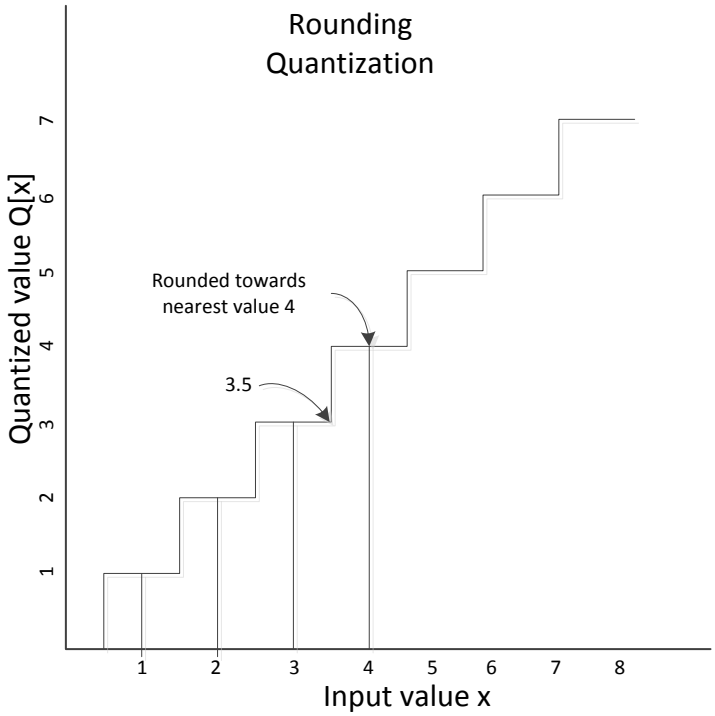


Figure 14 Graph of an input value with 3-bit rounding quantization

From Figure 14 it can be seen that rounding method introduces less quantization error since lower half of the values are flushed down to previous representable number and upper half is rounded to the next representable number. This quantization technique is very desirable due to less quantization error but on the same time it may also produce $Q[x]$ to be larger than x sometimes, e.g. according to the Figure 14 if value of $x = 1.6$ then $Q[x] = 2.0$. This produces erroneous affects in DSP computations as explained in [2].

Error of rounding quantization is independent of representations as we saw the case in truncation quantization [4]. Quantization error here is equally distributed on both positive and negative sides and is always uncorrelated to sequence, means it does not get affected from the sign of input sequence.

3.3. Quantization Effects

Effect of quantization, also known as quantization error is a very important measure in DSP domain. There have been many methods introduced in the literature to reduce them, e.g. [16] describes quantization effects on coefficient of digital filters. But techniques for reducing quantization effects are specific to the applications where quantiza-

tion is used. So we will just focus on discussing these effects on fixed-point and floating-point arithmetic rather than discussing the techniques to reduce them.

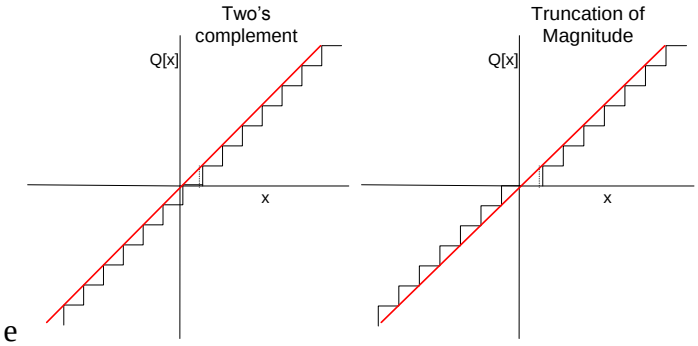
Let us say we have a signal x and the quantized version of this signal is represented as $Q[x]$ then quantization error e_Q could be defined as;

$$e_Q = x - Q[x] \tag{3.4}$$

Quantization error has different behaviors for different arithmetic; some of them are already discussed in section 3.2. Here we will try to analyze in detail quantization effects on the two types of number representations, fixed-point and floating-point.

3.3.1. Effects of Quantization in Fixed-Point Arithmetic

We will discuss the effects of quantization on fixed-point arithmetic by using three above mentions techniques of quantization i.e. 2's complement truncation, magnitude truncation and rounding.



e

Figure 15 Fixed-point arithmetic quantization curves

Let x be the sequence of fixed-point arithmetic and $Q[x]$ be its quantized output then Figure 15 shows the quantization curves of all three quantization techniques. Round towards nearest is the most favorable technique but on the same hand its implementation is more complex than truncation techniques.

Quantization errors of these techniques as discussed in section 3.2 can be drawn as;

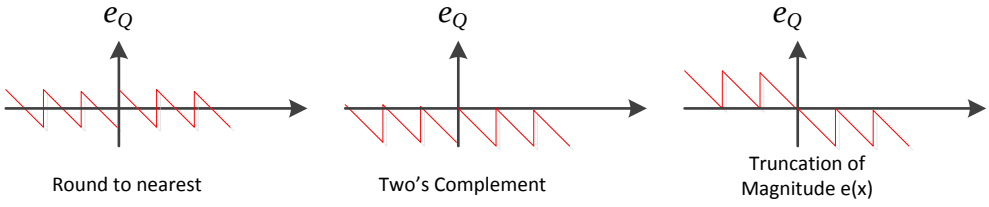


Figure 16 Fixed-point arithmetic quantization errors

Figure 16 shows that quantization error of rounding is equally distributed on positive and negative sides and is uncorrelated with the sign of input. Whereas quantization noise is always negative in case of 2's complement truncation and again it is also uncorrelated. Only magnitude truncation noise is correlated with the sign of input and is always opposite.

Probability density functions (PDF) of quantization noises of these three representations could be drawn as follows;

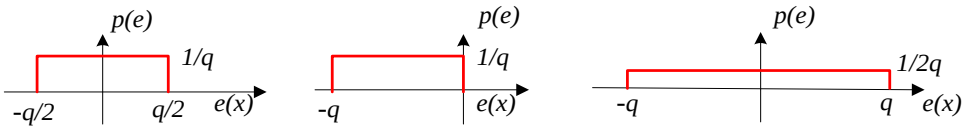


Figure 17 Probability Density Function of fixed-point quantization errors

Figure 17 shows PDF of rounding quantization is equally distributed and zero mean function which means that in DSP applications this noise is easy to remove as it can be modeled as additive white noise as discussed in [22], where as non zero mean noise PDF of 2's complement is difficult to remove. The additive noise could be expressed from the following equation;

$$Q[x] = x + e_Q. \tag{3.5}$$

There are different techniques developed in the literature to suppress quantization noises of fixed-point arithmetic as discussed in [22], [10] and [11] but we will keep our discussion focused just on the effects rather than suppressing them.

3.3.2. Effects of Quantization in Floating-Point Arithmetic

Floating-point representation of numbers is already discussed in chapter 2 and we know that quantization step does not remain constant in this representation, so quantization curve for floating-point arithmetic could be shown as in Figure 18.

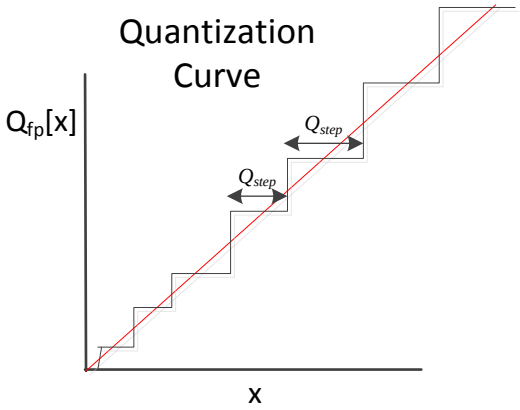


Figure 18 Floating-point arithmetic quantization curve

Because of continuous increase in the quantization step with the increment in exponent as shown in Figure 7, the quantization error increases also as the input grows large. Quantization error can be plotted as follows;

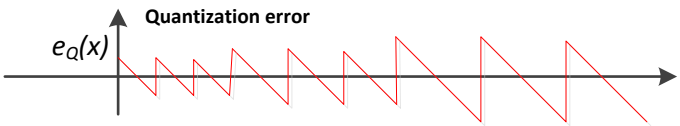


Figure 19 Quantization error of floating-point arithmetic

It is better to analyze relative quantization error, which is shown in the Figure 19 as well. Quantization error can be modeled as multiplicative error in case of floating-point arithmetic and can be expressed as;

$$Q[x] = x(1 + r) = x + xr \tag{3.6}$$

where r is the relative quantization error. For the floating-point quantization it becomes more difficult to suppress the noise components since the noise is multiplicative. Using block floating-point might be one way to achieve same efficiency as floating-point but with reduced complexity, as discussed in [12].

3.3.3. Comparison of Fixed-Point and Floating-Point Quantization

Affects of fixed-point and floating-point quantization are presented in previous section. In this section we will try to summarize some affects in tabular form.

Table 2 Fixed-point and floating-point quantization comparison

Quantization affects on fixed-point	**Quantization affects on floating-point**
Quantization noise is additive	Quantization noise is multiplicative
Quantization noise is constant power noise	Quantization noise power is not constant, relative error is taken into consideration
Quantization noise is independent of signal level	Quantization noise is related to signal level
Quantization noise can be removed easily	Removing / reducing noise is quite difficult
Noise level is defined by number of bits	Noise level is defined by number of bits in mantissa

Being multiplicative and dependent of signal level it is always difficult to reduce quantization noise in floating-point arithmetic. But reducing quantization noise in fixed-point is easier, that could be one reason why floating-point arithmetic is avoided as much as possible.

3.3.4. Analyzing Quantization Affects

SNR can be used to measure the noise level introduced in the signal. To measure SNR we need to have quantization error and its root mean square (RMS) value. Quantization error could be calculated by subtracting the quantized signal from the original signal as shown in the start of section 3.3.

Let us say we have a signal x which is quantized to N number of discrete levels and $Q[x]$ is its quantized version then root mean square (rms) value of e_Q could then be calculated as given in [21],

$$rms(e_Q) = \sqrt{\frac{1}{N}\sum |e_Q|^2} \tag{3.7}$$

where e_Q is considered to be a vector of N values. Now we can calculate the SNR of signal x as given in [24] as follows,

$$SNR = 20 \times \log_{10}\left(\frac{rms\,(x)}{rms\,(e_Q)}\right). \tag{3.8}$$

4. FOURIER TRANSFORM

This chapter starts with a general introduction about Fourier Transform its types and its properties and then some more concepts relating to Fourier Transform are presented. Then a fast algorithm developed by Cooley and Tukey to implement Fourier Transform mathematically known as Fast Fourier Transform (FFT) will be discussed, as well as its different types are discussed.

4.1. History of Fourier Transform

Transforms are mainly used to reduce the complexity of mathematical equations. Through such transforms complex differential and integral equations can be converted to normal algebraic equations. Fourier Transform is one such transform. It is most widely used and has various applications in digital signal processing [7].

Fourier Transform, named after Joseph Fourier (1768 – 1830), is the extension of Fourier series. It basically shows the signals in time domain as function of frequency domain. In other words it shows the spectrum of a signal or signal harmonics.

Fourier Transform is defined for continues time signals but DSP systems manipulates discrete time signals so Discrete Fourier Transform (DFT) is the one mostly used in DSP [7].

Based upon the form of the input signal (continues or discrete) it can be divided into two sub-categories, Discrete Fourier Transform (DFT) and Continues Fourier Transform (CFT).

4.2. Types of Fourier Transform

4.2.1. Continuous Fourier Transform

Consider we have a continues time signal $x(t)$ its Fourier transform can be represented as $X(\omega)$ and can be defined as,

$$X(\omega) = \sqrt{1/2\pi} \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt. \tag{4.1}$$

Since the portion $e^{-j\omega t}$ can be converted to the sum of *sine* and *cosines*, according to Euler's identity

$$e^{j\pi} = \cos\pi + i \sin\pi. \tag{4.2}$$

So alternatively we can say that Fourier Transform of any signal $x(t)$ is its decomposition into *sine* and *cosine* components. These sinusoidal components are also called harmonics. Decomposition of any signal into its fundamental harmonics through Fourier

transform has a very vast application is DSP and gives a lot of information, e.g. it tells at which harmonic the signal has the most strength, how many basic frequency components this signal is composed of? This helps in filter designing to filter the noise components etc.

4.2.2. Discrete Fourier Transform

As stated earlier DSP mostly deals with discrete signals, so the discrete form of above mentioned Fourier Transform (DFT) can be defined as follows.

Consider there is an *N* point data sequence$x(n)$, which ranges in between [0, *N*-1]. N-point DFT of this sequence can be defined as [3]

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi nk/N} \quad 0 \leq k < N. \tag{4.3}$$

The $e^{-j2\pi nk/N}$ in the above equation can be simplified as follows [7];

$$W_N = e^{-j2\pi/N}. \tag{4.4}$$

This W_N notation is called twiddle factor and $e^{-j2\pi}$ is the root of unity (unity can be obtained by expanding $e^{-j2\pi}$ using Euler's identity) and $e^{-j2\pi/N}$ is the *n-th* root of unity. So by using twiddle factor N-point DFT equation reduces to

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{nk}. \tag{4.5}$$

4.3. Properties of Fourier Transform

We will discuss some basic properties of Fourier transform for the reader to have better understanding and to know its capacity of being widely used in the field of DSP; further detailed discussion could be found in [13].

4.3.1. Linearity

Fourier transform is linear. It poses the property of homogeneity, which means that if a signal changes its amplitude in one domain the same amount of change occurs in other domain as well.

Let us take example of discrete signals. Let $x[n] \leftrightarrow X(k)$ then mathematically it can be shown as

$$x[n] + y[n] = z[n]; \tag{4.6}$$

$$X(k) + Y(k) = Z(k). \tag{4.7}$$

The above equations show that if *z*[*n*] is the sum of *x*[*n*] and y[*n*] then *Z*(*k*) can be obtained with the sum of frequency domain signals of *x*[*n*] and *y*[*n*].

4.3.2. Phase Characteristics

A time domain shift in the signal introduces a linear change in the phase of the frequency domain where as magnitude of the signal in frequency domain remains unchanged.

Mathematically it can be described as;

$$x[n] \leftrightarrow Re[X(k)] + \text{Im[X(k)]}; \tag{4.8}$$

$$x[n+a] \leftrightarrow Re[X(k)] + \text{Im[X(k)]} + 2\pi af. \tag{4.9}$$

4.3.3. Periodic Nature of DFT

DFT considers all the signals to be periodic. Normally in real world in DSP experiments signals taken into use are not periodic but DFT takes this signal to be one period of an infinitely long signal and considers the input signal to be periodic. Figure 20 helps to more clarify this property;

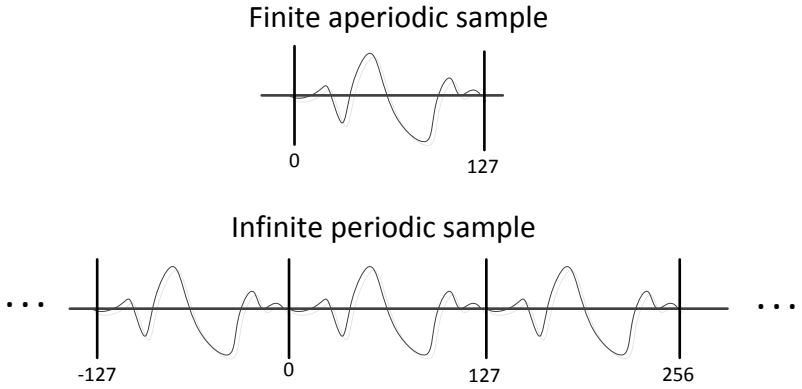


Figure 20 Periodic nature of DFT

4.3.4. Compression and Expansion

Compression of signal in one domain results in the expansion in the other domain and vice versa.

Let $X(k)$ be the Fourier transform of $x[n]$, then

$$x[mn] \leftrightarrow \frac{1}{m} X\left(\frac{k}{m}\right). \tag{4.10}$$

4.3.5. Multiplication

Multiplication of signals in one domain results in the convolution of signals in other domain:

$$x[n] \times y[n] \leftrightarrow X(k) * Y(k). \tag{4.11}$$

This is very useful property, which is used in the computation a lot. Since convolution is computation expensive so by using Fourier Transform and multiplying those in other domain will output the same result with less computation overhead.

4.3.6. Parseval's Relation

Since a signal can be represented in time domain as well as frequency domain so energy of a signal is conserved in both domains. The normalized energy in the expression is given by [15];

$$\sum_{n=0}^{N-1} x^2[n] = \frac{1}{N}\sum_{k=0}^{N-1} |x(k)|^2 \qquad (4.12)$$

4.4. Fast Fourier Transform

Generally the Fourier Transform is calculated by decomposing it into its trivial components. This leads to a lot of calculation overhead and in the field of DSP where Fourier Transforms of very large inputs are required such computations grow very large. But luckily there are some fast algorithms developed for this purpose, which reduce the complexity of calculation in logarithmic manner one such algorithm developed and widely used is Cooley-Tukey decomposition.

4.4.1. Cooley – Tukey Decomposition

The first ever algorithm developed for the FFT calculation was by Cooley and Tukey in 1965. The main principle utilized by them was "divide and conquer". Divide the *N* point sequence into smaller number sequence until you get the prime factors of the original sequence then compute the DFT of those small prime sequence multiply them with twiddle factor, reorder and sum up. In this way you will get the complexity reduced to *N log N*.

So to summarize the steps, Cooley-Tukey decomposition method can be described in following points;

1. Divide / decimate the sequence into smaller prime sequences
2. Compute the DFT of these sequences
3. Multiply with twiddle factors
4. Re-order the sequence

If the length of the input sequence can be expressed as the power of *R*, then decomposition of the length of the final small prime sequence is always equal and is *R*. In this case it is called Radix – *R* FFT. But otherwise if the length of the small prime sequence is not same it is called mix radix FFT computation [7].

4.4.2. DFT Computation by Cooley – Tukey Decomposition

Let us discuss the Radix-2 decimation in time FFT calculation in detail, which was the basic FFT algorithm proposed by Cooley and Tukey at the start and later on extended to other Radixes.

Consider we have a sequence *x(n)*. By looking at the N point DFT formula we can understand that to compute it we need *NxN* multiplications and *N* addition. So before applying Cooley-Tukey decomposition the complexity of this computation is $O(N^2)$.

Now suppose that we divide this sequence into two sequences, each of length *N/2*. And the length of the sequence can be expressed as powers of 2.

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{nk} \tag{4.13}$$

After diving this into two equal length sequences

$$X(k) = \sum_{l=0}^{N/2-1} g(l)\, W_N^{2lk} + \sum_{l=0}^{N/2-1} h(l)\, W_N^{(2l+1)k}. \tag{4.14}$$

This division is done so that all the even indices of *x(n)* become $g(l)$ and all the odd indices become $h(l)$.

Now by using the property of roots of unity (Periodicity property) $W_N^{2lk} = W_{N/2}^{lk}$ the above equation can be transformed to

$$X(k) = \sum_{l=0}^{N/2-1} g(l)\, W_{N/2}^{lk} + W_N^k \sum_{l=0}^{N/2-1} h(l)\, W_{N/2}^{lk} \tag{4.15}$$

This can be expressed as sum of two DFTs as;

$$X(k) = G(k) + W_N^k\, H(k) \qquad k = 0, 1, 2 \ldots N-1 \tag{4.16}$$

Where $G(k)$ and $H(k)$ are the *N*/2-point DFTs of $g(l)$ and$h(l)$. Now the complexity of computing these DFTs would be $(N/2)^2 + (N/2)^2 \rightarrow O\,(N^2/2)$. So complexity is reduced to half just by dividing an *N* point sequence into two equal *N*/2 point sequences.

Now again if the length *N*/2 is even and can be divided by two, both $g(l)$ and$h(l)$ could be further divided to 4 *N*/4 length sequences which will further reduce its complexity to half.

Consider we want to calculate an 8-point DFT. So we can keep on diving the sequence length to equal pieces until we get 2 point DFTs which are not an even number anymore. So the complexity of the algorithm is reduced exponentially and finally the whole DFT computation can be done with the complexity of $O\,(N \log_2 N)$.

Block diagram of an 8-point DFT model using Radix-2 algorithm as given in [7] is shown in the Figure 21. Figure 22 shows the signal flow graph of the same model.

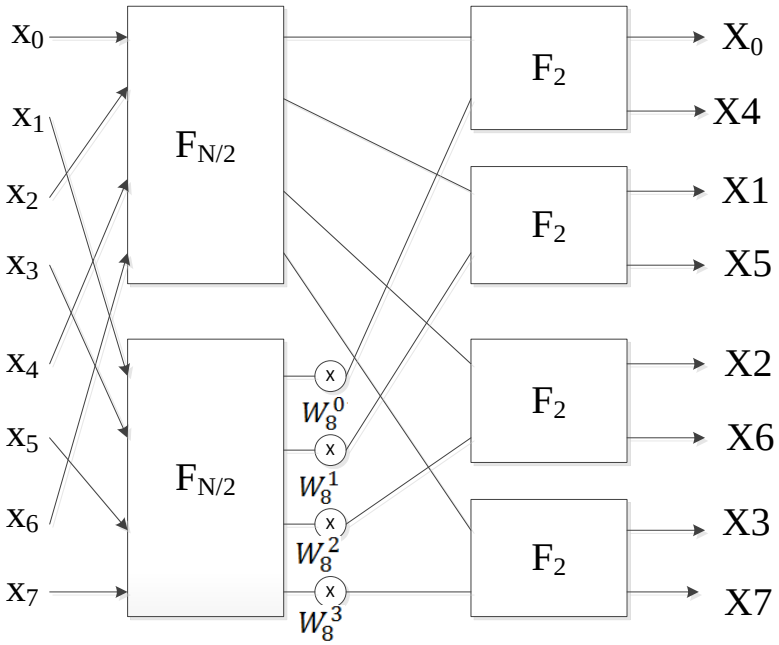


Figure 21 Block diagram of an 8-point radix-2, in-order input, DFT

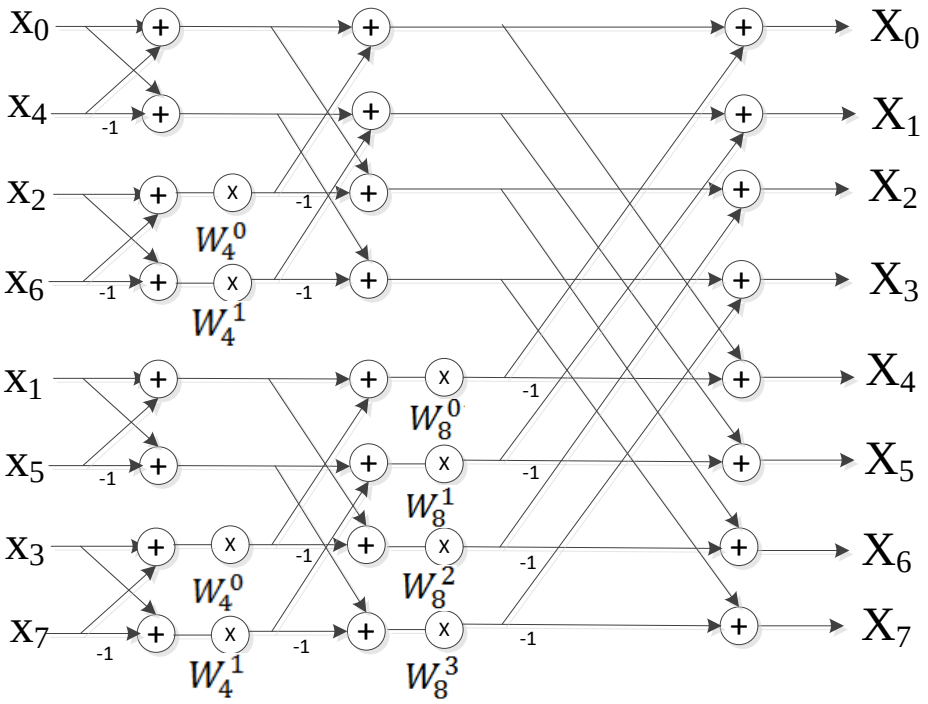


Figure 22 Signal flow graph of 8-point radix-2, in-order output, DFT

We can see from the signal flow graph of the 8-point DFT shown in Figure 22, at the last stage the computation performed consists of one addition and one multiplication operation to attain each coefficient. This is called the butterfly computation and will be discussed in next section.

The above case is presented when the length of the input sequence is power of 2. But it can also be extended to the sequences which can be expressed as powers of 4 e.g. sequence of length 16. Computation of such FFT will be radix-4 FFT. Similarly both of these radices could be mixed as well by using radix-4 at the stage where the length of

the sequence can be expressed as powers of 4 and using radix-2 where the length could be expressed as powers of 2. Although mixed radix FFT is not as efficient as single radix, but different methods have been developed and are under constant development to make it efficient. One such research work is presented in [1].

4.4.3. Some Basic Concepts about FFT

4.4.3.1 Radix

Radix of an FFT can be defined as the size of its last stage decomposition. If the FFT is single radix (not mixed radix) then its length should be representable in power of some number, where as mixed radix FFT's length can be represented as power of different numbers at different stages. E.g. for radix-2 FFT length of the sequence could be 4, 8, 16, 32, etc.

4.4.3.2 Decimation In Time And Frequency

Decomposition/dividing of FFT into even and odd points of the signal in time domain is called Decimation in time, similarly when this decomposition is applied in frequency domain it is called Decimation in Frequency.

4.4.3.3 Twiddle Factor

A factor used in DFT representation of a sequence $x(n)$ and represented as W_N for an N-point DFT. Twiddle factor is also called the root of unity since $W_N = e^{-j\,2\pi/N}$ for an N-point sequence.

4.4.3.4 Butterfly

The simplest 2-point FFT calculation, which is the basic unit of FFT calculation, is composed of one addition and one multiplication operation. It can be further explained by the signal flow graph shown in Figure 23.

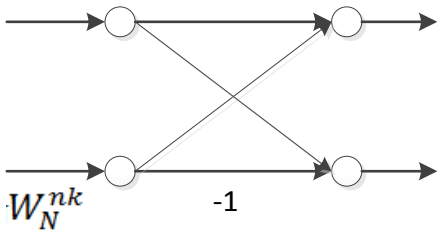


Figure 23 Butterfly diagram, basic unit of FFT computation

4.4.3.5 Bit Reversal

Input and output indexes of FFT are not same and they need to be calculated. But luckily for radix-2 FFT calculation there exist a pattern that input and output indexes are related in the bit reversed fashion. It means if the input index is 1 (001_2) the output index would be 4 (100_2) for an 8-point DFT calculation. A table for 8-point DFT as given by [3] is shown below.

Table 3 Bit reversal pattern for 8 point DFT calculation

Input index	Binary	Bit reversed	Output index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

4.4.4. Applications of FFT

Fast Fourier Transform is one of the top ten algorithms in 20^{th} century [19]. Because of its reduction of computation calculation of DFT it is enormously used almost everywhere in DSP. Applications of FFT are described in detail in [14]. Discussing the applications of FFT is out of the scope of our thesis but to point out some names, it is used in Biomedical engineering, mechanical analysis, analysis of stock market data, geophysical analysis, and the conventional radar communications field and many more.

5. WORKING MODEL

To explain the results of quantization analysis on the reference FFT model under consideration it is important to explain the simulation model of FFT as well as the quantization models. This chapter is dedicated to explain the quantization models in detail and some description of the FFT model used. Some discussion about the stimuli generation will also be presented.

Figure 24 presents an overview of the quantization analysis model.

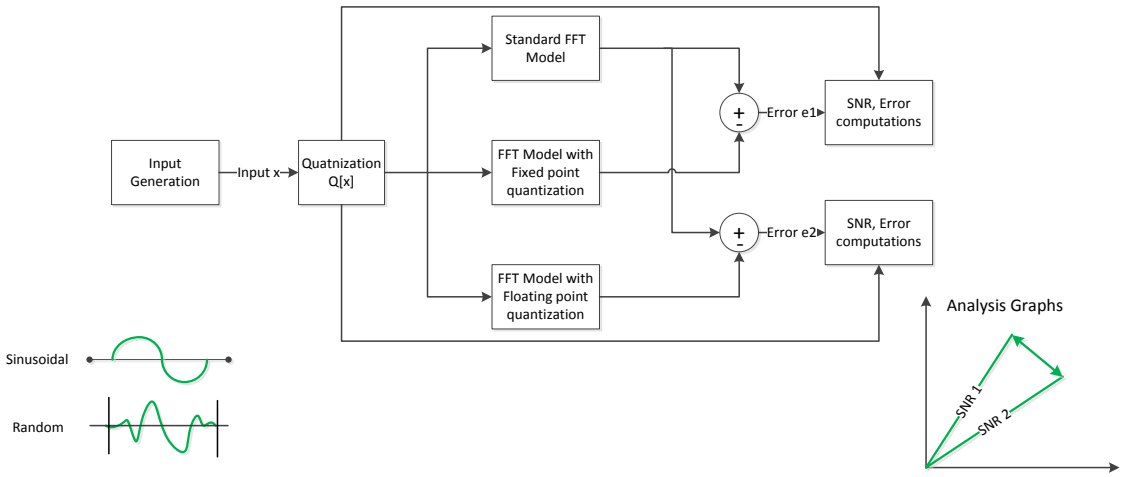


Figure 24 Quantization analysis model

There are two quantization models developed to analyze fixed-point and floating-point quantization in the given reference FFT model. So there are three outputs, fixed-point quantized FFT, floating-point quantized FFT and reference FFT model (which is double-precision floating-point FFT model). Error is calculated with reference to the double-precision model of FFT we have. Signal to Noise Ratio (SNR) is calculated for each model in order to optimize different parameters for fixed-point and floating-point quantized FFT model to get optimized SNR.

Different parameters are taken into consideration for variation e.g. various generated stimulus, word length, FFT length, quantization technique etc. Variation parameters for fixed-point and floating-point models will be discussed while analyzing the results in chapter 6.

5.1. Quantization Models

We should keep in mind that numbers are represented in double precision floating-point format in Matlab so analysing fixed-point quantization requires converting the input from double precision floating to fixed-point format first. This will be discussed in section 5.1.2. There are two different quantization techniques used in each model, quantization of two's complement and round towards nearest.

Quantization models are implemented in C language, whereas the reference FFT model (double precision FFT model) used is implemented in Matlab. So in order to integrate C and Matlab code CMEX support of Matlab is required.

Before going into the detail of the quantization models, let us first have a look at CMEX feature of Matlab.

5.1.1. CMEX Support

MEX programming in Matlab allows you to compile functions from C/C++ or FORTRAN source code. There are a specific set of compilers which can be used with Matlab to compile the source code, a list can be found in [18]. A small discussion about how to use MEX files to call C files is given below.

5.1.1.1 Using MEX Files to Call C Files

The command used to compile the C files with Matlab is

mex *test_file.c test_file_mex.c*

This shows that for every C file, which you want to compile, you need a mex file. A specific format for writing the mex files for Matlab is defined. There are some specific steps defined for a mex file as described in [20]

- Creating Gateway Functions
- Declaring Data Structures
- Managing Inputs and Outputs
- Validating Inputs
- Allocating and Freeing Memory
- Manipulating Data
- Displaying Messages to the User
- Handling Errors

After compiling C files with MEX files you get the MEX binary which then can be called by Matlab. So the process could be summarized as; write the functions in C, write the gateway function (MEX function), build the MEX binary using the Matlab mex command and binary MEX file is ready to be used like Matlab functions.

5.1.1.2 MEX Files and Matlab Interface

An interface of MEX files, C files and Matlab could be explained by Figure 25.

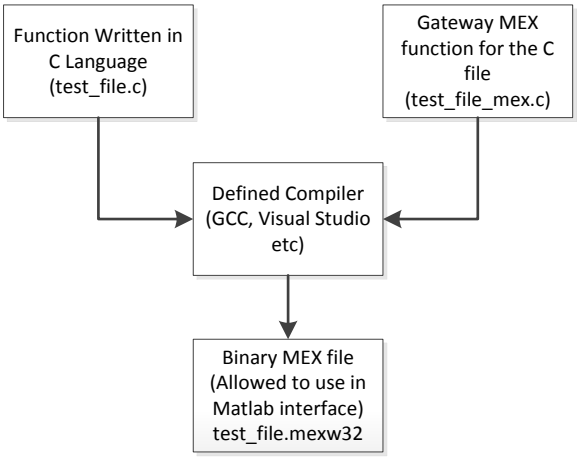


Figure 25 MEX file generation

test_file.c C file as shown in the Figure 25 is compiled along with *test_file_mex.c* mex file by the defined compiler for Matlab using command shown in the previous section.

This command generates the binary file *test_file.mexw32* or *test_file.mexw64* depending upon the operating system (32 bit or 64bit). The generated binary MEX file then can be called from Matlab or **.m* file in the same way as Matlabe function.
Following Figure 26 explains the process of calling of binary MEX files by Matlab.

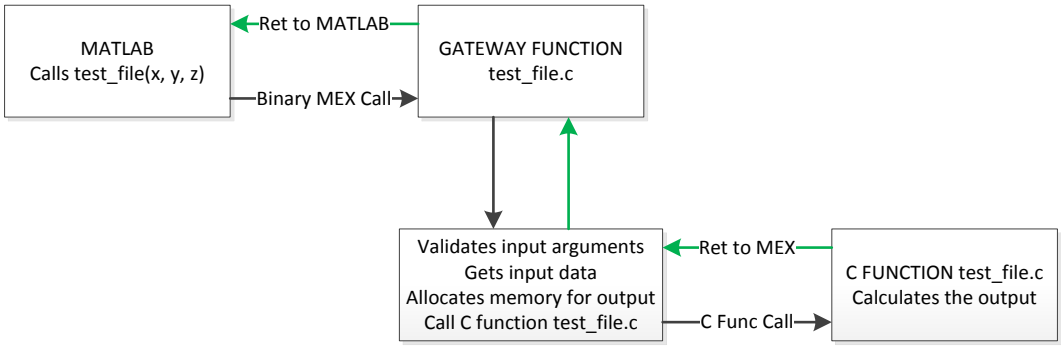


Figure 26 MEX file and MATLAB interface

Figure 26 explains that when such function is called form Matlab the gateway function first passes the parameters after performing its check routines to the C function which after computing the results places them in the output vectors generated by gateway function. And then results are available for the Matlab.

5.1.2. Fixed-Point Quantization Model

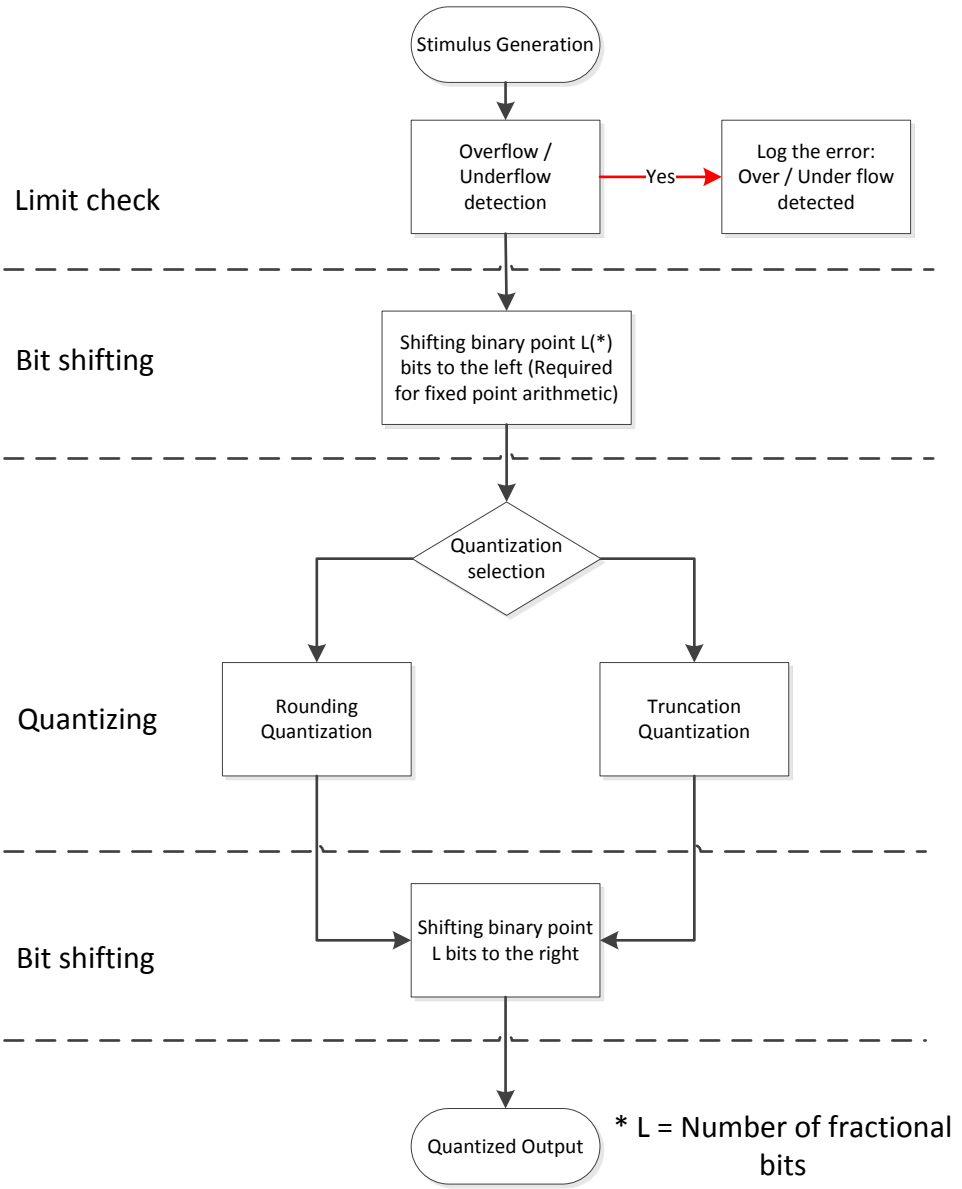


Figure 27 Fixed-point quantization model

As described earlier, in Matlab numbers are represented with double precision floating-point format. For fixed-point quantization there are no exponent bits required, in other words exponent width is zero. So the input vector from Matlab will be in double precision floating-point format and they are needed to be converted to fixed-point format. Normal fixed-point representation generally needs two parameters "Number of integer bits" and "number of fractional bits" but since we are using fractional fixed-point representation as explained in section 2.2.2 Figure 5 and Figure 6 so we will be concerned just with number of fractional bits and number of integer bits will be considered zero. As described in the Figure 5 total number of bits for fractional fixed-point representation will be;

$$\textit{Total number of bits} = L + 1 \qquad \textit{where } L = \textit{number of fractional bits}$$

One extra bit is used for sign as shown in Figure 5. As explained before numbers of fractional bits are taken input as a parameter and number of fractional bits will be varied while analyzing the output SNR of fixed-point quantized FFT model in chapter 6.

Conversion of floating-point to fixed-point is based on the fact that by specifying the number of fractional bits as an input parameter we are fixing the binary point at a specific location. And the bits after that location in the fraction are chopped off or rounded depending upon the quantization technique.

Figure 27 shows the implemented fixed-point quantization model and the source code for this model could be found in Appendix A.

Input to the quantization model is provided by stimulus generator and will be discussed later in this chapter. To explain the quantization model consider an input vector of length *n* is generated by stimulus generator

$$X = x_1 + x_2 + x_3 + \cdots + x_n. \tag{5.1}$$

Where *X* is a vector of complex number normalized in the range of [-1, 1]. Quantization steps are performed separately on each scalar value.

First overflow / underflow detection is performed. Input vector is checked to be in the maximum (on positive and negative side) and minimum (minimum representable number / quantization step) limits. Positive and negative limits are different in since sign bit is included in the magnitude. So the input range on positive and negative side can be shown as follows;

$$-1 \leq x\ F_{max}. \tag{5.2}$$

$$F_{max} = 1 - Q_{step}. \tag{5.3}$$

Where $Q_{step} = 2^{-L}$. Values outside this range are considered to be overflow values and maximum value is used instead. For underflow detection, values below the minimum representable number are detected as underflow values,

$$\forall\ |x| < Q_{step}; x = 0 \tag{5.4}$$

If the input value goes below the quantization step, it is considered to be zero as shown in the above equation.

If the value is inside the range of representable numbers then second step of moving the binary point to the required amount of bits is performed. Bit moving / shifting is used for fixed-point quantization to shift the binary point by *L* bits to the left as shown in Figure 27. This makes it very easy to quantize the number according to the required technique. Bit shifting could be explained with the following equation.

Consider x_r is the intermediate value after the first step then,

$$x_m = x_r \times 2^L \tag{5.5}$$

After bit shifting the result undergoes quantization in third step. Since we are using two methods, truncation (2's complement quantization) and rounding towards nearest. Truncation quantization is performed by simply chopping off the bits after binary point as explained in section 3.2.1. In practice this is performed by storing the fractional number as integer and compiler does the copping of fractional bits. For round towards

nearest number should be incremented or decremented to closest representable number, detail can be found in section 3.2.2. In practice, this is performed by adding a factor of 0.5 to the number and then bits after binary point are chopped off.

Finally in the last step binary point is shifted back to the left by "total number of fractional bits".

Consider *y* to be the final result of quantizing *x* and x_q is the result after quantization step then,

$$y = x_q \times 2^{-L} \tag{5.6}$$

Left and right shifting of the binary point is actually scaling the input up and down by the amount of number of bits shifted.

Let us now take an example and go through step by step in order to show the fixed-point quantized output using the implemented fixed-point quantization model.

Consider the total number of fractional bits = 10. So quantization step as explained in section 2.2.2.1 will be

$$Q_{step} = 2^{-(10-1)} = 0.001953 = F_{min} \tag{5.7}$$

$$F_{max} = 1 - 0.001953 = 0.998047 \tag{5.8}$$

Where,F_{min} is minimum representable value and F_{max} is maximum representable value as shown in section 2.2.2.

Lets us suppose that a random complex input, 0.2119 – *j* 0.3320 is generated by stimulus generator. Real and imaginary parts will be quantized separately so moving forward with only real part would be enough to understand the fixed-point quantization.

First overflow and underflow check is performed. There is no underflow or over-flow in case of this input since

$$F_{min} < 0.2119 < F_{max} \tag{5.9}$$

Then shifting of binary point by 16 (number of fractional bits) results in 108.492800. Let's use round towards nearest quantization, which produces the result as 108.000000. At the last step the binary point is shifted back to get the final result, which is 0.2109.

Quantization error for this fixed-point quantization model could be calculated by formula discussed in section 3.3.4.

$$e_Q = |0.2109 - 0.2119| = 0.001 \tag{5.10}$$

5.1.3. Floating-point quantization model

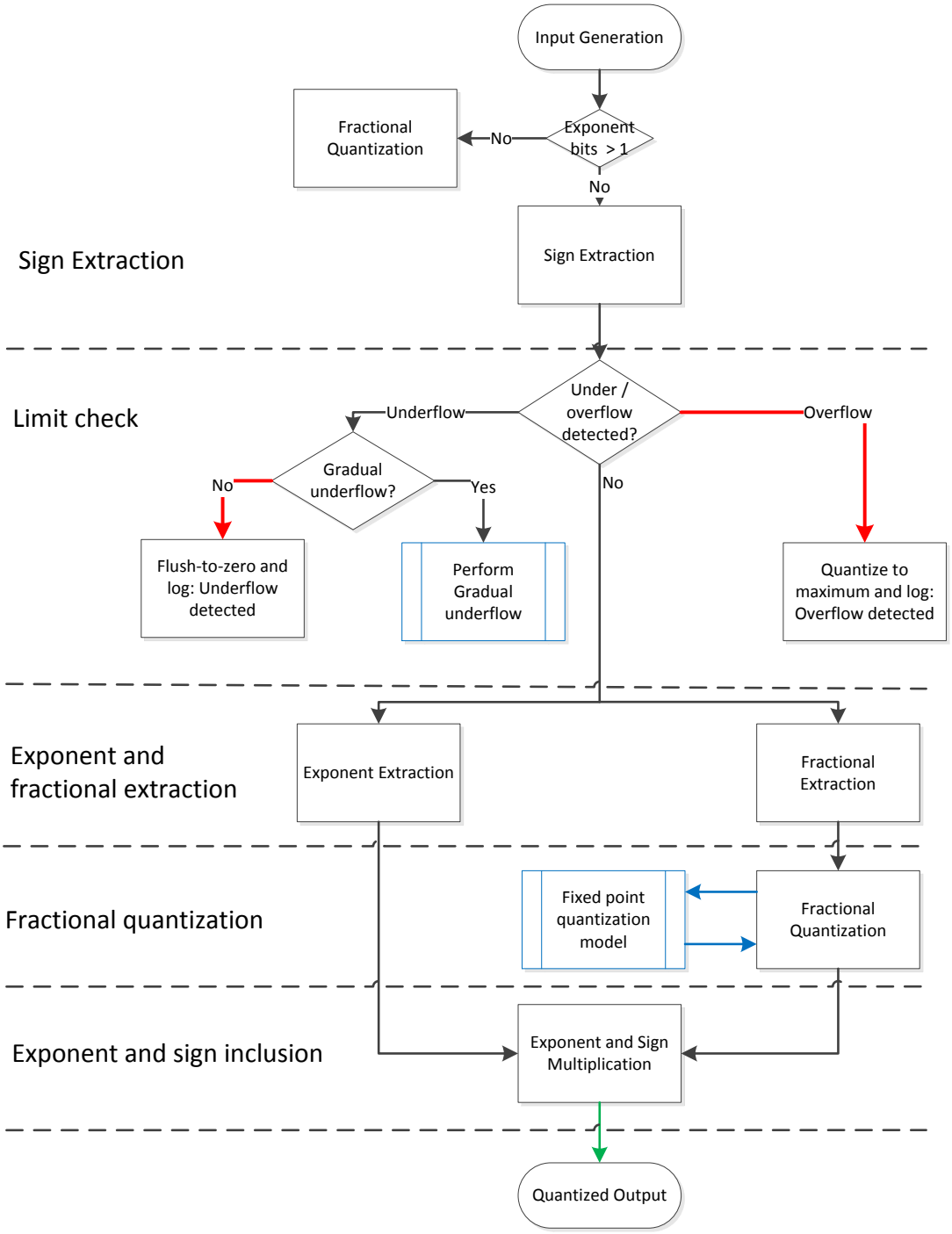


Figure 28 Floating-point quantization model

Since Matlab uses double precision floating-point representation which is described in Table 1. Table 1 show that this standard of floating-point representation uses 53 bits in mantissa (including the sign bit) and 11 bits in exponent field, so total width of this representation is 64 bits. This representation is also shown in the Figure 29 for better visual understanding.

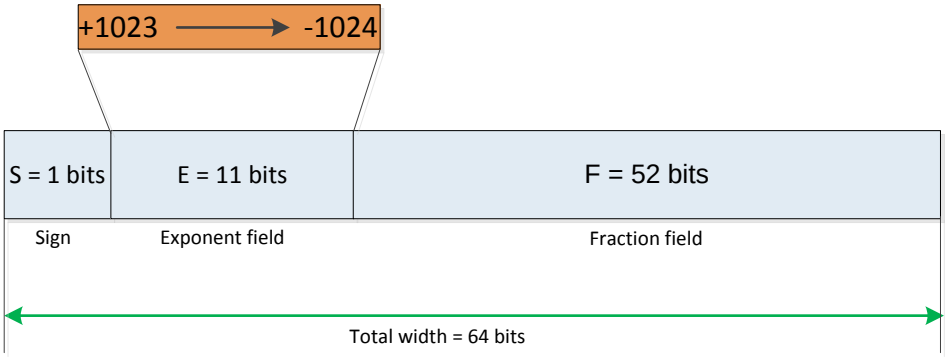


Figure 29 Double precision floating-point representation

Detailed steps of floating-point quantization model are shown in Figure 28. Initially the input parameter number of exponent bits is checked if it is equal to 0 then the input is considered to be fixed-point and fixed-point quantization is applied otherwise floating-point quantization is applied. Steps used in this quantization are explained below.

5.1.3.1 Sign Extraction

Since sign bit is separate in floating-point representation so overflow and underflow detection could be performed after sign extraction. So sing is extracted at the first step and further calculations are performed on positive values.

5.1.3.2 Overflow / Underflow Detection

Limit detection is required in this quantization model as well, just like in fixed-point quantization model but as explained in first step maximum representable number on the positive and negative side is same because sing bit is not considered to be included in magnitude. So maximum representable number in IEEE 754 double precision floating-point standard, after which overflow would occur could be shown as;

$$Max = Fmax \times Cmax. \tag{5.11}$$

Where

$$Fmax = 1.0 - 2^{-L} \tag{5.12}$$

$$Cmax = 2^{C-1} - 1 \tag{5.13}$$

L = number of fractional bits

C = number exponent bits

Similarly minimum representable number for underflow detection could be shown as;

$$Min = Fmin \times Cmin. \tag{5.14}$$

Where

$$Fmin = 0.5 \tag{5.15}$$

$$Cmin = -2^{C-1} \tag{5.16}$$

If underflow is detected and flag for gradual underflow is set then gradual underflow is performed. We will analyze results of gradual underflow in chapter 6 as well.

5.1.3.3 Exponent and Fractional Extraction

Since quantization is only required for the fractional part of the input so exponent and fractional part is separated. This extraction of the exponent part is done by taking *log2* of the input and flooring down the value. As shown below;

$$exp = \lfloor log2(input) \rfloor \tag{5.17}$$

And fractional part is calculated by dividing the input with the quantity 2^{exp}

$$frac = \frac{input}{2^{exp}} \tag{5.18}$$

Exponent part is kept to be added in later steps and fractional part is quantized.

5.1.3.4 Fractional Quantization

Then quantization of the fractional part is carried out based upon the quantization method selected. Fractional part is quantized by fixed-point quantization model. Since exponent has already been extracted from the input therefore it could be easily fed to fixed-point quantization model to be quantized as shown in the Figure 28. Selection of quantization method (Truncation of 2's complement or Rounding towards nearest) is provided to fixed-point quantization via input parameter.

5.1.3.5 Exponent and Sign Multiplication

Finally floating-point quantized output is generated by inclusion of sign and extracted exponent to the quantized fractional part.

For further detailed study code for the floating-point quantization model could be found in Appendix B.

Let us now take an example and go through the above mentioned steps and show the output of floating-point quantized model.

Consider the same input used in fixed-point quantization model $0.2119 - j\ 0.3320$. Other parameters are as follows;

Number of fractional bits = 5

Number of exponent bits = 4

Round towards nearest quantization used

Let us now calculate *Max* and *Min* first.

$$Max = 124.000000$$

$$Min = 0.001953$$

First step is to extract sign. Since our value 0.2119 is already positive so no action is required. Also no overflow or underflow is occurring as,

$$0.001953 < 0.2119 < 124.000000.$$

After extracting the exponent

$$Fraction = 1.695200$$

$$\exp = -3.0000.$$

After performing the quantization on fractional part

$$Fraction_{Quant} = 1.687500.$$

And the final result after exponent and sign multiplication is 0.2188. Quantization error e_Q and SNR calculated to be,

$$e_Q = |0.2109 - 0.2119| = 0.001$$

5.2. Stimulus Generation

There are different types of stimulus used to generate and compare the results from fixed-point and floating-point quantized FFT models.

A complex input is generated using *rand* function of matlab and scaled between [-1, 1]. Input is also quantized and according to the Figure 24 same quantized input is fed to all three models in order to analyze the results. Results have been analyzed on random input including full scale values, random excluding full scale values.

Input stimulus will also be mentioned in chapter 6 while discussing the results of quantization of reference FFT model.

5.3. Reference FFT Model

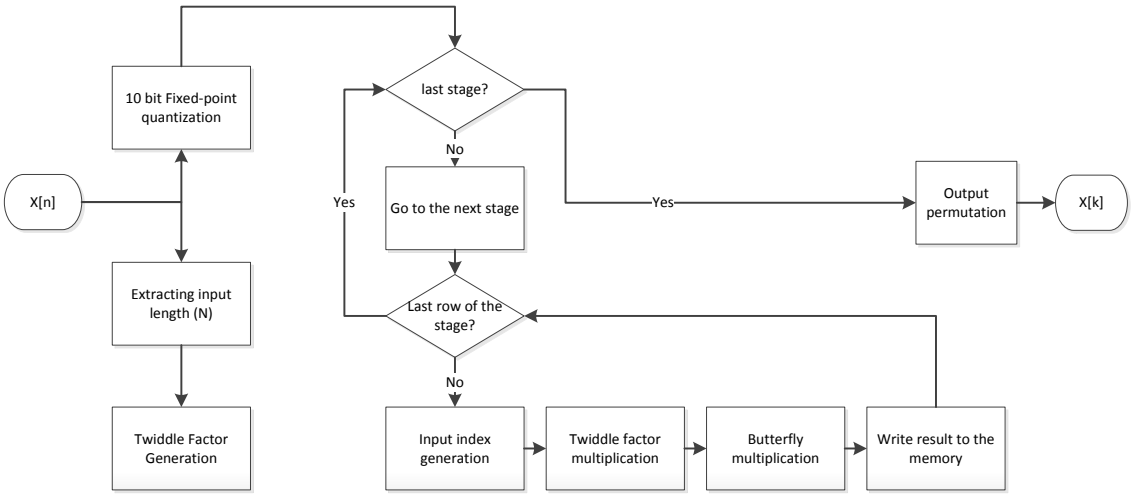


Figure 30 Block diagram of FFT model used

Figure 30 shows the block diagram of the FFT model used. The blocks shown in the Figure 30 are the basics steps of any FFT implementation but the main computational block referred in the figure as "FFT Kernel Computations" varies depending upon the implementation. As explained in the section 4.4.2 mainly FFT kernel computation consists of twiddle factor multiplication and addition of different input indexes (collectively called as butterfly computation). To compute one single output entry two computations (one addition and one multiplication) is required. So the working of "FFT kernel computation" part could better be explained with the signal flow graph.

Signal flow graph of an example input of length $N = 8$ is given in Figure 31.

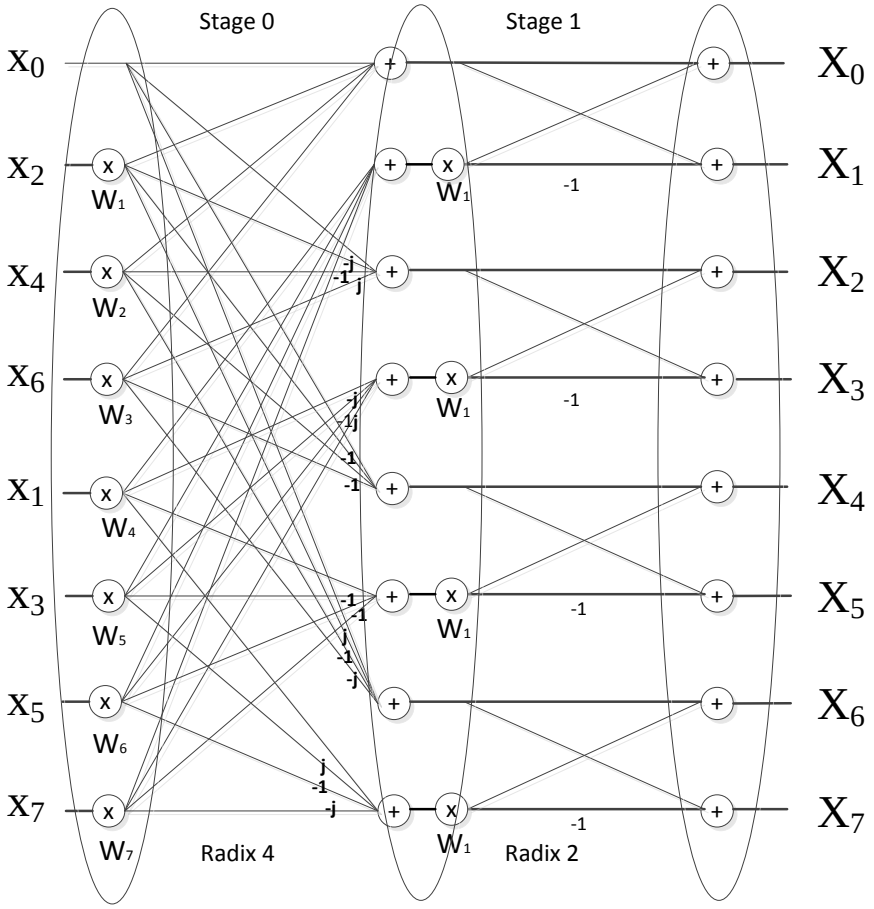


Figure 31. Signal flow graph of 8-point radix-4/2 FFT implementation model

Since length of the input vector cannot be expressed as powers of 4, so it is going to be selected as mix radix (4, 2) FFT computations. This is a mix radix (4, 2) FFT implementation and is such that it analysis the length of the input and decides which radix is going to be used, radix 2, radix 4 or both. So e.g. for N = 32 mix radix is going to be used where as for N = 16 radix 4 is going to be utilized.

To calculated 8 point DFT using radix 2 FFT algorithm it takes log2(8) = 3 stages where as using this such mix radix approach DFT could be calculated using just two stages. Twiddle factors shown in Figure X above W_1, W_2, W_3 can be expressed as;

$$W_1 = W_N^k, W_2 = W_N^{2k}, W_3 = W_N^{3k} \ldots \quad (5.19)$$

Where N = 8 for the give example.

As seen from the Figure 31 two different radices are used to compute 8 point DFT so the basic computation unit (butterfly) for radix 4 DFT is computed using following four equations;

$$X_0 = x_0 + W_1 x_1 + W_2 x_2 + W_3 x_3 \quad (5.20)$$

$$X_1 = x_0 - jW_1 x_1 - W_2 x_2 + W_3 x_3 \quad (5.21)$$

$$X_2 = x_0 - W_1 x_1 + W_2 x_2 - W_3 x_3 \quad (5.22)$$

$$X_3 = x_0 + jW_1 x_1 - W_2 x_2 - jW_3 x_3 \quad (5.23)$$

where x_0, x_1, x_2, x_3 are considered to be the inputs and X_0, X_1, X_2, X_3 are the outputs of any intermediate stage of FFT calculations.

Similarly for radix-2 these equations get simplified as follows;

$$X_0 = x_0 + x_1 \tag{5.24}$$

$$X_1 = x_0 - x_1 \tag{5.25}$$

$$X_2 = x_2 + x_3 \tag{5.26}$$

$$X_3 = x_2 - x_3 \tag{5.27}$$

5.3.1. Quantization in Place

Quantization is required to be placed after every 2-operand operation in the signal flow graph shown in the Figure 31. Locations where quantization is required are marked with elliptical shapes in Figure 31. Four operand operations in computation of Radix-4 FFT stages are broken down to two 2-operand operations and quantized after each operation.

Resultant signal flow graph after putting the quantization models in place can be seen in Figure 32. Figure also shows the larger view of quantization models to understand the idea of requirement of quantization.

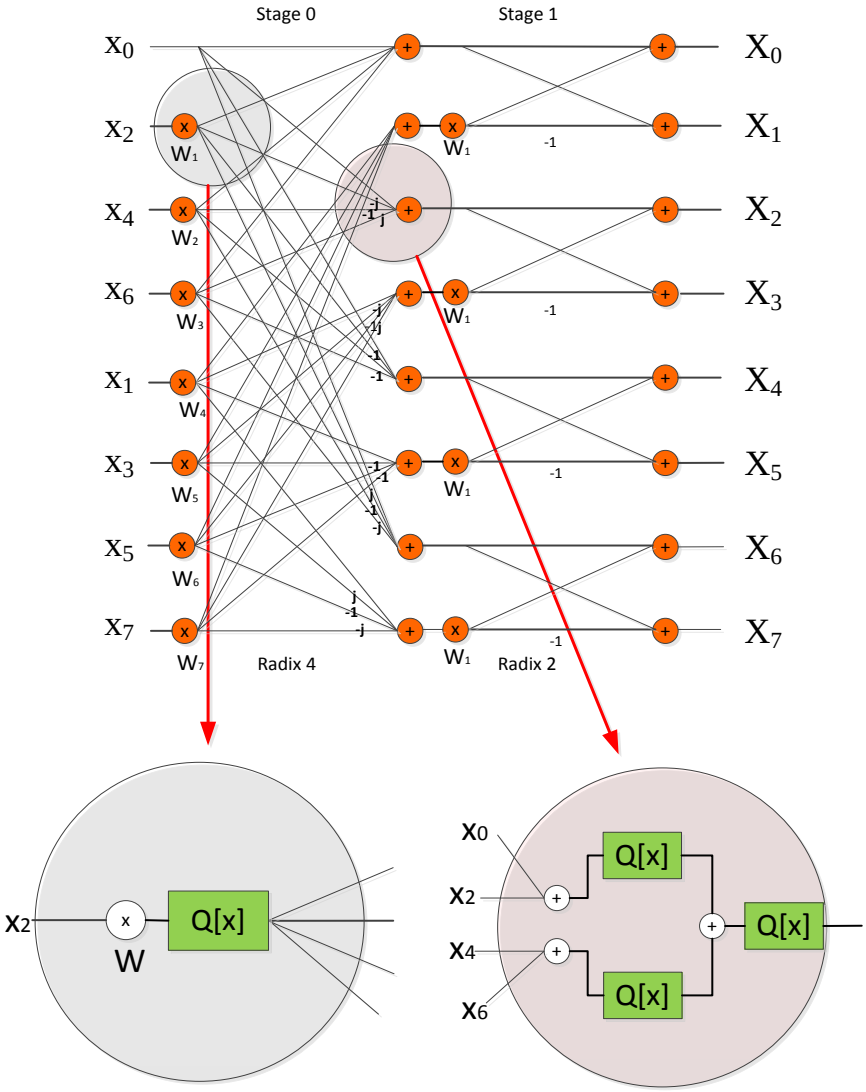


Figure 32. 8-point radix-4/2 FFT with quantization model

All the arithmetic operations shown with orange background require quantization before feeding them to the next stage.

Figure 32 shows that quantization is applied only on arithmetic operations, but quantization is also applied on the input and generated twiddle factor before feeding them for the kernel computations, this is shown in the Figure 30 showing the block diagram of reference FFT model used. For further detailed study source code for fixed-point quantized FFT and floating-point quantized FFT models are presented in Appendix D and E.

6. ANALYSIS AND RESULTS

This chapter is dedicated to analyze the results, which are generated by using the quantization models (fixed-point quantization model and floating-point quantization model) with the reference FFT model computed with double precision floating-point arithmetic as described in the previous chapter. Signal to noise ratio (SNR) analysis is done to compare the efficiency of quantized FFT models. Before discussing the SNR graphs of the FFT models we would describe the experiment methodology, what is done in order to get the SNR for vectors, what quantization parameters are used for generating the graphs and what kind of stimulus is used for the experiment and how they are generated. Analysis on the results of fixed-point and floating-point quantization model is done considering various different parameters, e.g., word width, FFT length, and quantization methods.

6.1. SNR Analysis for Quantized FFT Model

SNR (Signal to noise ratio) is one of the very important measure to be analyzed in DSP domain. Quantization models, when placed in the reference FFT model, will introduce some quantization noise, which can be compared by using SNR. High SNR values will depict less quantization noise compared to the signal level and vice versa.

6.1.1. Stimulus Generation

The stimulus used in the experiment is randomly generated (using *rand()* function of Matlab) complex vector and quantized with 10 bits fixed-point quantization. Quantization of 2's complement is used for stimulus quantization. Matlab's *rand()*function generates values in the range of [0,1], so the generated complex vector is scaled to cover the full range [-1, 1].

Multiple sets of stimulus vectors needs to be generated in order to analyze the models. Each set of stimulus contains 50 stimulus vectors. This is done to repeat the experiment for the same kind of stimulus 50 times and average of the result is used in order to average out the unwanted outputs. SNR is also observed on scaled stimulus values. Scaling factors used 0.25, 0.5, 1, 1.5, 2, 2.5, 3 and 3.5 which served the purpose of scaling in both up and down directions and observe the behavior on SNR respectively.

Initially large stimulus vectors are generated each vectors contains 2048 complex values (to serve as the input for 2048 point FFT) but general analysis is performed on 1024 points, so initial 1024 values of stimulus vectors are taken into use. Same stimulus vectors are used to observe affect on SNR with the variation in FFT length, by selecting

different number of complex values from the vector according the observed FFT lengths. E.g. 8, 16, 32 values are selected for 8, 16, and 32 point FFT calculations accordingly. Stimulus generation is explained in further detail in the following Figure 33

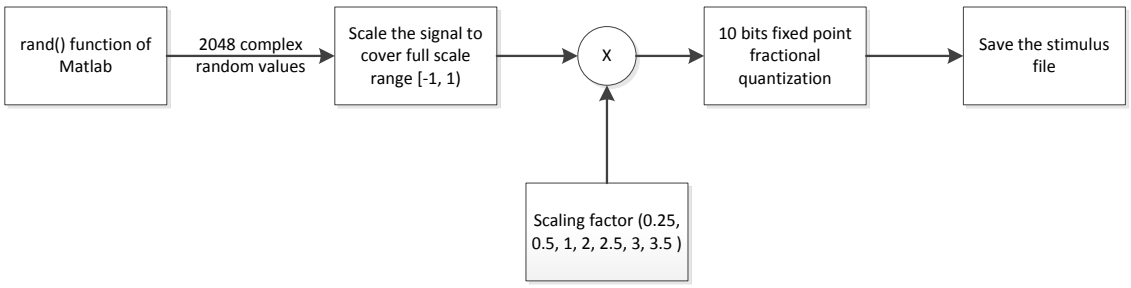


Figure 33 Stimulus generation

6.1.2. Experiment Methodology

FFT calculation for fixed-point quantized FFT model is done for number of fractional bits varying from 10 to 25 and SNR is calculated compared to reference FFT model. For each number of fractional bits amount experiment is repeated 50 times and average SNR is used in order to attain one SNR value on the graph. SNR for vectors is calculated using the formula mentioned in eq. 3.8. For calculation of SNR we need *rms* which is calculated according to eq. 3.7 and quantization error vector e_Q is defined in eq. 3.4. The *QuantizedFFT* is a vector obtained as output of reference FFT model with quantization model (fixed-point or floating-point) in place and *StandardFFT* vector is calculated using our reference model as explained in section 5.4.

The procedure described above for the floating-point quantized FFT model is then repeated for different number of exponent bits. General FFT computations are performed on length of 1024 point, but the behavior on SNR for changing the FFT length is also observed at the end, as mentioned in previous section. Figure 34 below shows in more detail how SNR values are calculated for the stimulus vectors.

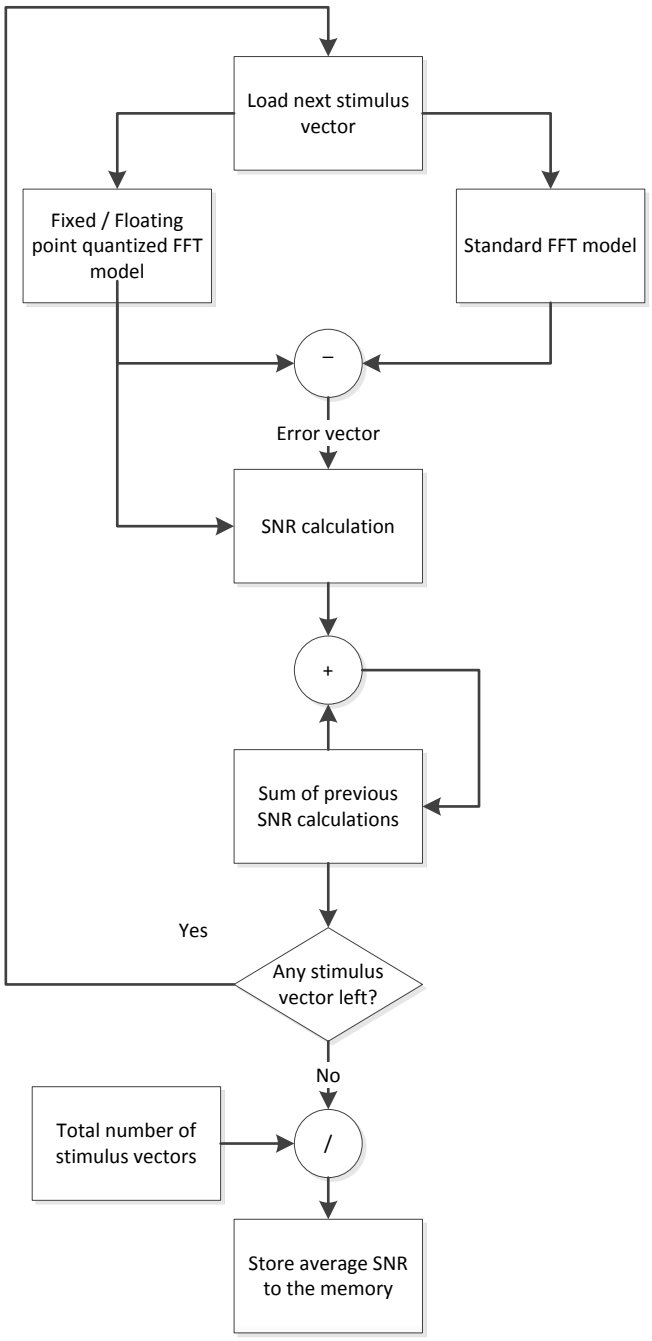


Figure 34 Experiment methodology for SNR calculation

6.1.3. SNR with Fixed-Point Quantized FFT Model

First we tested the system by feeding full scale signal, i.e., signal range is [-1,1). Since *rand* function of Matlab generates values in between [0, 1], for this analysis stimulus is scaled to cover the complete number range of [-1, 1). The results of this test can be seen in Figure 35.

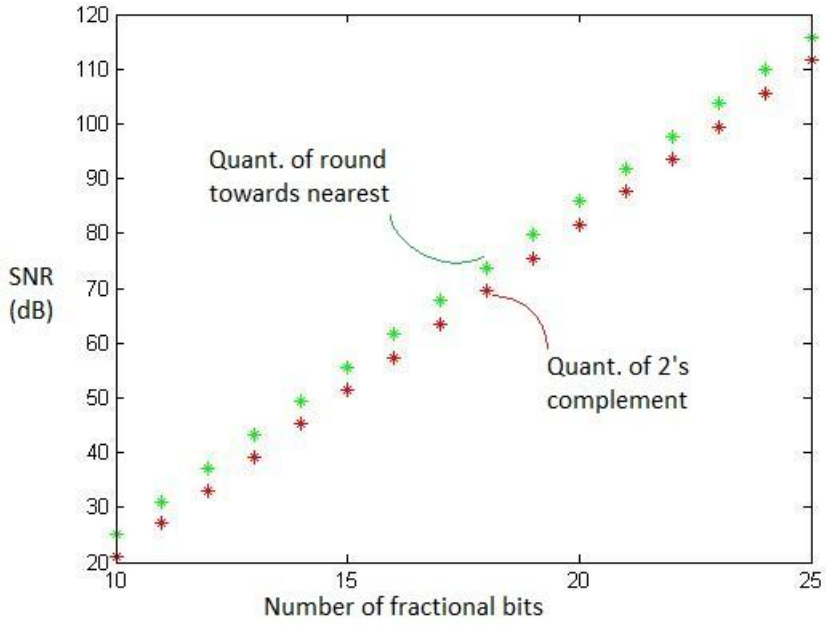


Figure 35 Fixed-point SNR with full scale stimulus

By increasing the number of fractional bits SNR increases hence they are directly proportional. Round towards nearest quantization have higher SNR values as compared to its counterpart two's complement quantization as was expected.

For the following analysis full scale values are excluded from the stimulus, by using the following method. In case of 10 bit fractional quantization, which is used to quantize the stimulus, the range of full scale signal is [-1, 1 – Q_{step}] where $Q_{step} = 2^{-(10-1)}$. Values equal to full scale values are further reduced by one quantization step. The results can be seen in Figure 36.

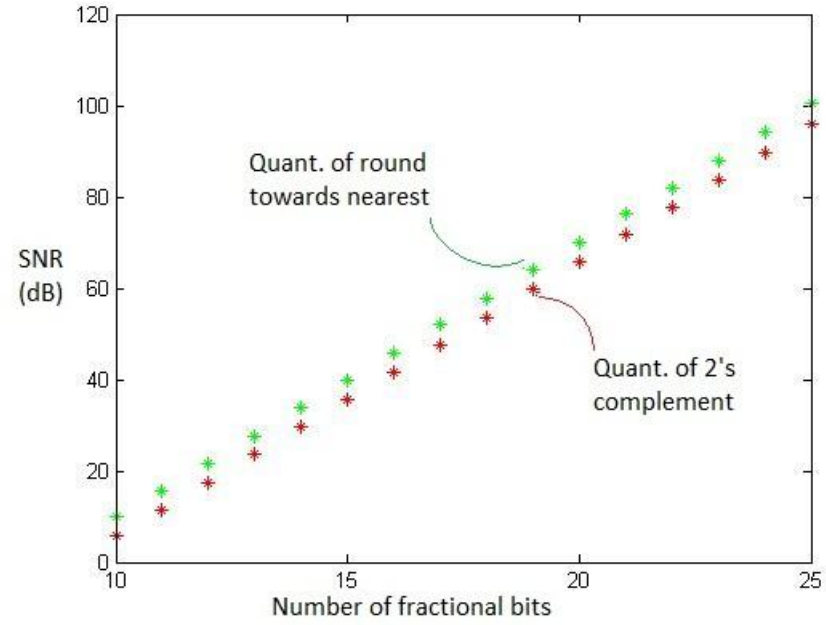


Figure 36 Fixed-point SNR with normal range stimulus

Normal signal stimulus curves are following the full scale stimulus curves behavior in both round towards nearest and two's complement quantization but notable thing is that normal signal values result in less SNR as compared to full scale values on the same number of fractional bits. Which depicts that SNR is affected with the strength of the stimulus as well, and is directly proportional in the given scenario.

6.1.4. SNR with Floating-Point Quantized FFT Model

We tested the behavior of flush-to-zero quantization by varying the number of fractional bits on different values of exponent bits, as explained in the experiment methodology above. Flush to zero quantization is obtained by simply flushing all underflow values (values below minimum limit) to zero level. The results are shown in Figure 37.

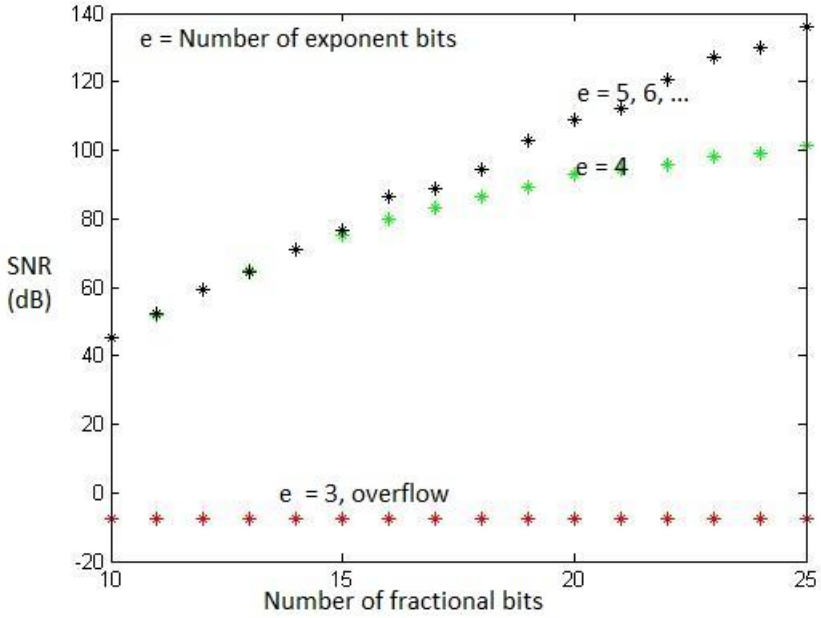


Figure 37 Floating-point, flush to zero, SNR at different number of exponent bits

General behavior of increment in SNR on increasing the fractional bits is also observed here. Increasing the number of exponent bits also affects the SNR curve, but only at low values e.g. see the difference in the curves of $e = 4$ and $e = 5$. This is also notable that considerable SNR difference starts to appear on larger number of fractional bits (e.g. => 15) in comparison $e = 4$ and $e = 5$ curves because SNR saturation starts to occur. Since the stimulus is limited to [-1, 1] where as dynamic range of floating-point FFT is larger than this limit and most of the values lying below the minimum limit are flushed to zero hence saturating the SNR. This problem seems to go away at high values such e => 5. For e = 3 we see the considerable difference in the behavior which is due to over flow occurring at such low number of exponent bits.

Then gradual underflow situation is tested at different number of exponent bits. Gradual underflow is implemented by not flushing the values smaller than minimum limit to zero but floating-point quantization is performed for these values as well. The result of this analysis is shown in Figure 38.

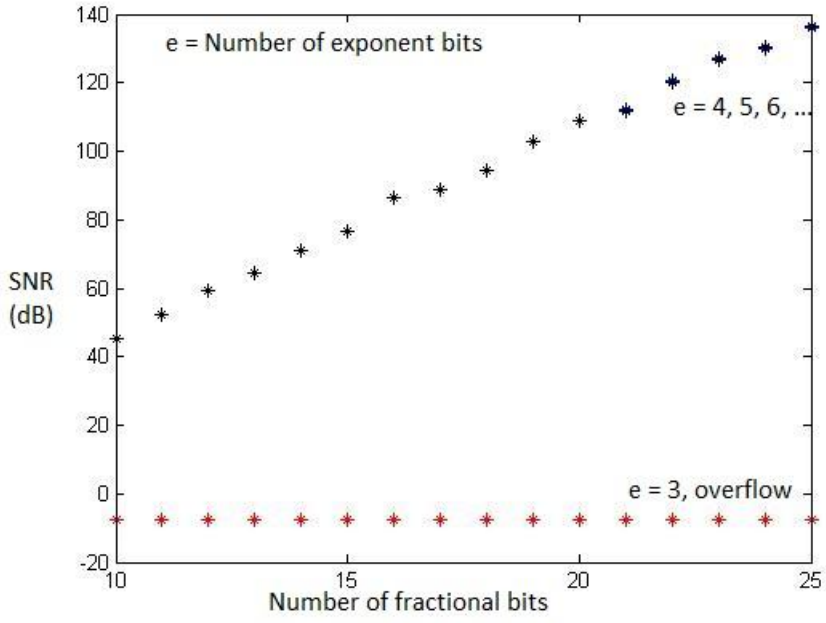


Figure 38 Floating-point, gradual underflow, SNR at different number of exponent bits

In flush-to-zero situation values below the minimum level are flushed to zero level and a considerable part of the signal is lost and due to that at smaller *e* values we observed saturation in SNR curve as explained in previous section. No such behavior is observed here as expected, SNR level is improved and no saturation occurs at *e=4* because values close to zero not flushed to zero anymore and quantization is performed for them as well. Increasing *e* values also seem to have no affect on SNR which is expected because increasing *e* values increases dynamic range and if no overflow or saturation (in case of flush-to-zero) is occurring then increasing *e* values will not have any effect on SNR curve.

6.1.4.1 SNR with Scaled Stimulus

Previous observations are preformed on normal stimulus. It is also worth observing the change in SNR curves when scaled stimulus is applied. Note that this scaling is done before the quantization of stimulus. Scaling is done by multiplying or dividing with a constant factor. Scaling behavior is observed at different number of exponent bits.

As explained in the previous section two types of floating-point quantization (flush-to-zero and gradual underflow) are observed so scaled stimulus will be applied to both as well.

First we performed experiment with scaled stimulus on floating-point flush-to-zero quantized FFT model at e = 3.Since SNR values are already showing overflow at exponent bits = 3 as shown in the Figure 37 so the stimulus is scaled down for this experiment. Overflow is still observed but SNR values are improved to some extent as shown in the Figure 39 below.

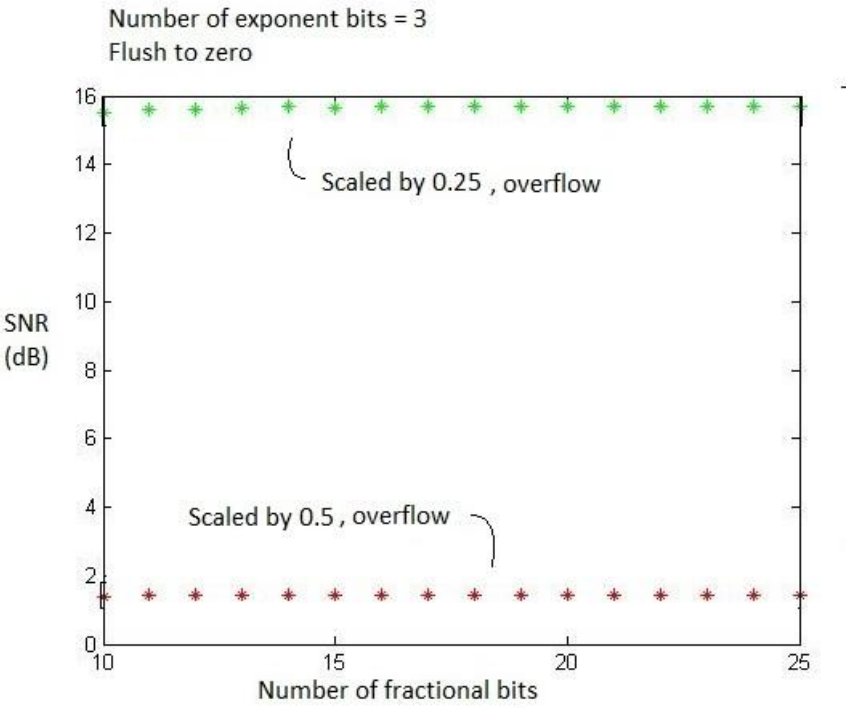


Figure 39 Floating-point, flush to zero, SNR with scaled stimulus at *e* = 3

Scaled by 0.5 curve shows less improvement than scaled by 0.25, when compared with $e = 3$ curve in Figure 37. Scaling down the signal further does not improve SNR further.

The above experiment is then repeated with $e = 4$ where we observed saturation as shown in the Figure 37 in $e = 4$ curve. The result of this experiment is show in Figure 40 below.

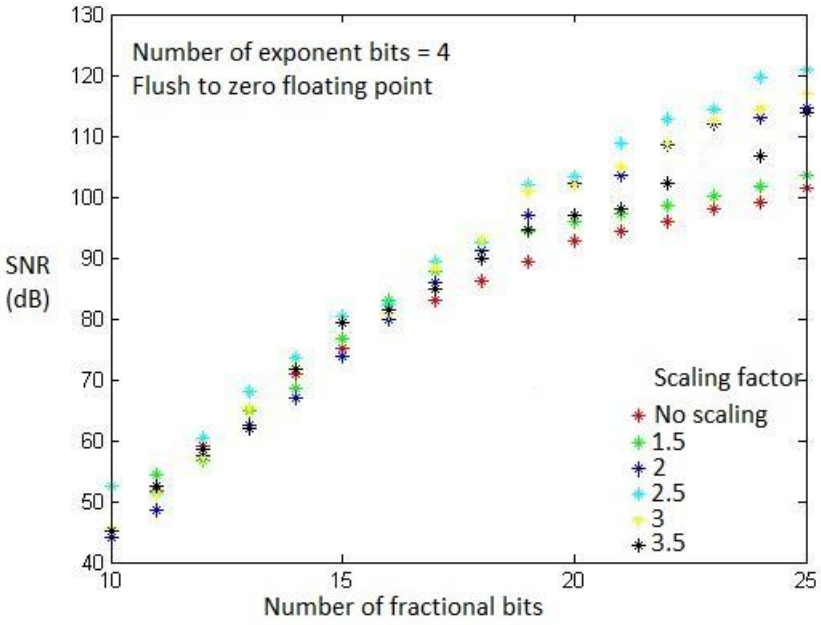


Figure 40 Floating-point, flush to zero, SNR with scaled stimulus at *e* = 4

General trend of increased SNR values is observed by increasing the scaling factor as shown by red dots (No scaling) and sky blue dots (scaled by 2.5). But as we increase the scaling factor higher than 2.5 SNR curves start degrade as shown by curve of black dots (scaled by 3.5). So scaling factor 2.5 is optimum to obtain best SNR results at $e = 4$ floating-point flush-to-zero quantized FFT.

Same experiment is repeated on higher values of $e = 5$ and $e = 6$ and results are shown in Figure 41 and Figure 42 below.

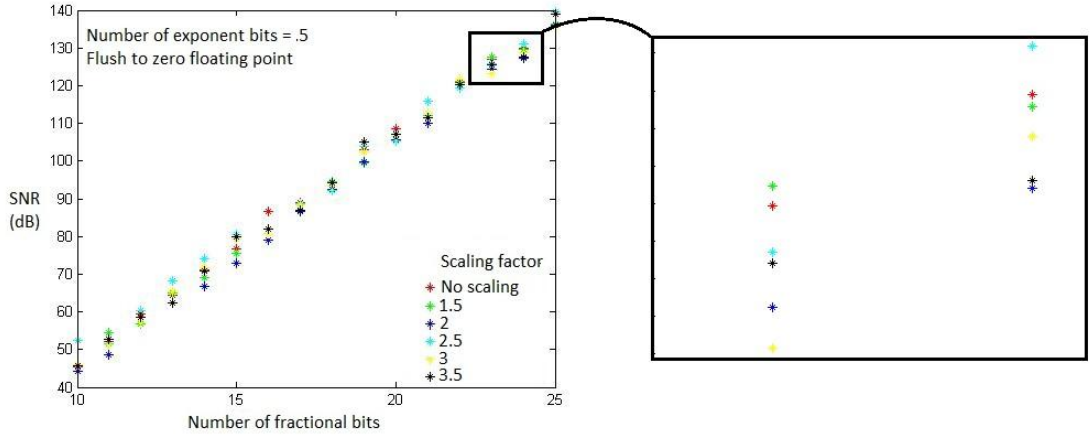


Figure 41 Floating-point, flush to zero, SNR with scaled stimulus at $e = 5$

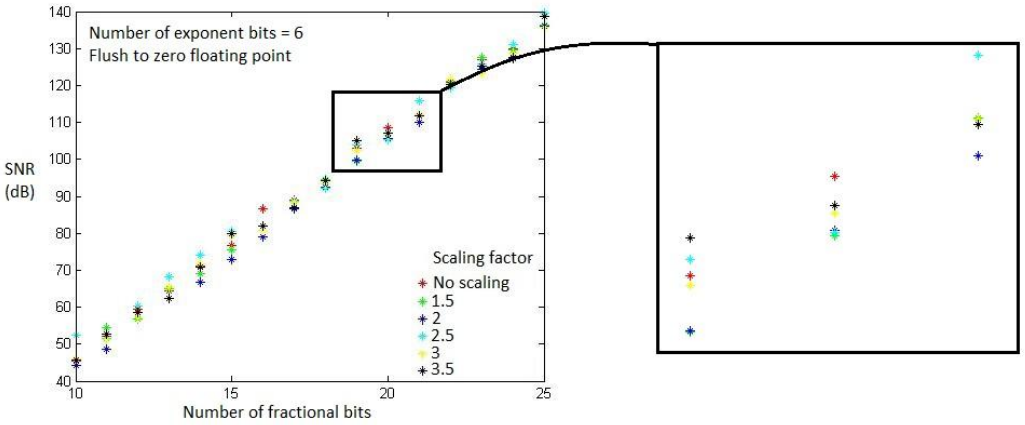


Figure 42 Floating-point, flush to zero, SNR with scaled stimulus at $e = 6$

No specific behavior on scaled stimulus at higher values of e is observed as shown by above figures. But the optimum scaling factor 2.5 as observed in Figure 40 is also showing the best SNR curve among both in Figure 41 and Figure 42 as shown by sky blue dots.

Then scaled stimulus experiments are repeated with gradual underflow floating-point quantized FFT model. Gradual underflow shows a better behavior than flush-to-zero, which is expected.

First experiment is performed on $e = 3$ and results are shown in Figure 43 below.

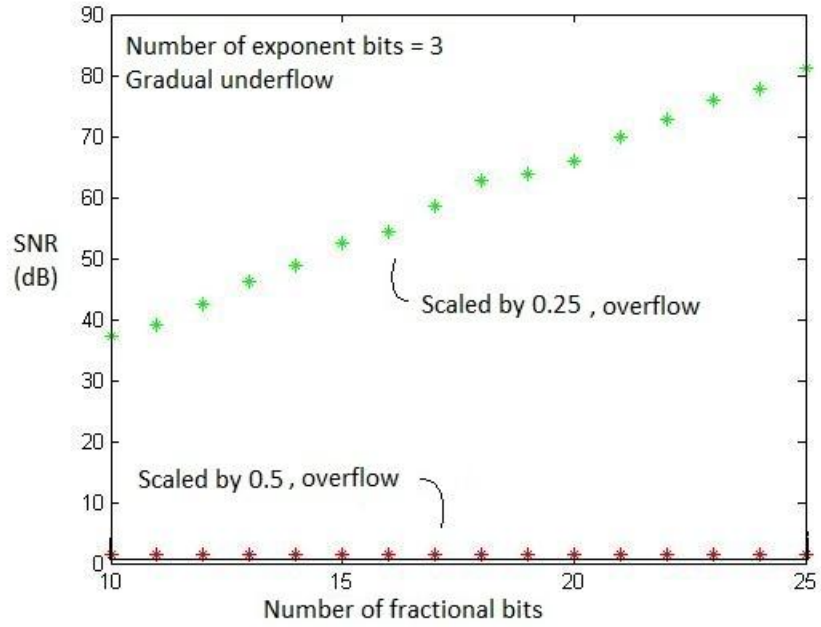


Figure 43 Floating-point, gradual underflow, SNR with scaled stimulus at *e* = 3

Result for scaled stimulus by 0.25 have better SNR values and also following the general trend of increase with the increment of number of fractional bits. Scaled by 0.5 shows poor results but better than non-scaled curves as shown in Figure 38. Overflow is observed in both scaling.

Next the experiment is repeated for $e = 4$, $e = 5$ and $e = 6$. Results for these experiments are show in Figure 44, Figure 45, and Figure 46.

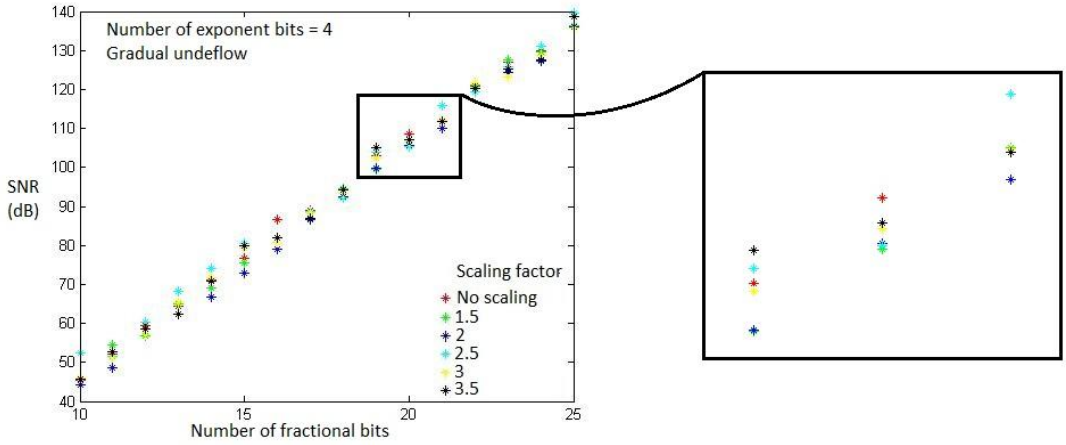


Figure 44 Floating-point, gradual underflow, SNR with scaled stimulus at *e* = 4

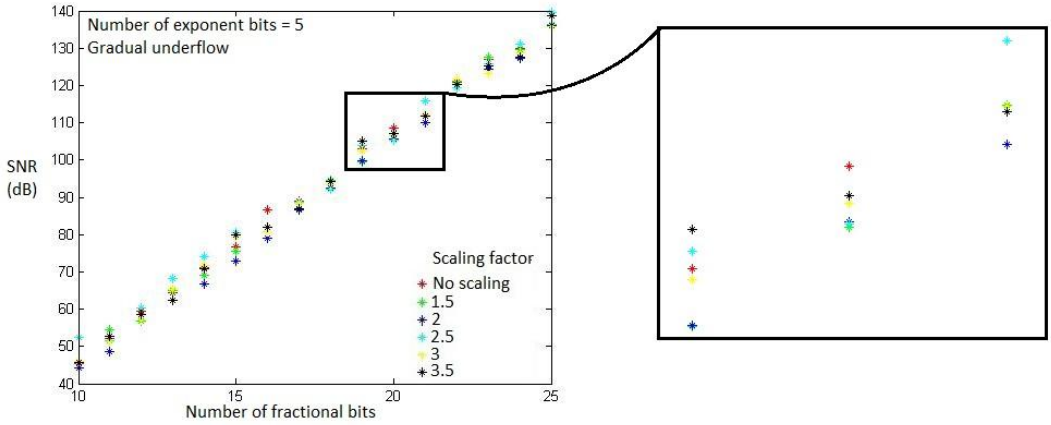


Figure 45 Floating-point, gradual underflow, SNR with scaled stimulus at *e* = 5

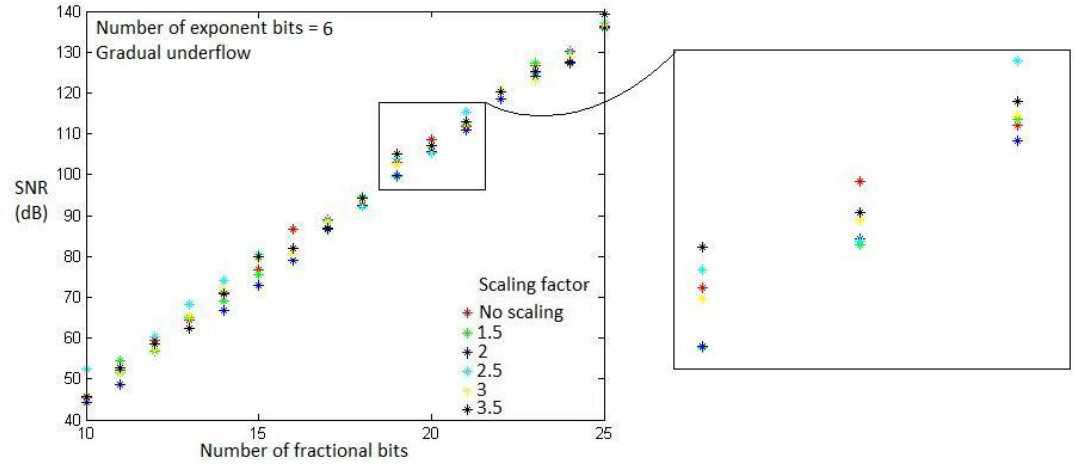


Figure 46 Floating-point, gradual underflow, SNR with scaled stimulus at *e* = 6

Since there is no saturation observed at $e = 4$ curve in the case of gradual underflow as show in Figure 38, there is no specific improvement observed at $e = 4$ in Figure 44 with scaled stimulus as observed in the case of flush-to-zero scenario. But scaling factor 2.5 is showing the best SNR curve here as well, proving to be the best scaling factor for gradual underflow floating-point quantized FFT model as well.

Now if we compare the fixed-point quantized FFT model results and floating-point quantized FFT model results we can see that generally floating-point model shows better results at higher values of e in comparison to the fixed-point. But at lower values of e e.g. at $e = 4$ in flush-to-zero SNR value at number of fractional bits = 20, is around 90 dB which is comparable to the full scale fixed-point SNR value at number of fractional bits = 20, which is also around 90 dB. Thus for our analysis model floating-point quantized FFT will show nearly same results for $e = 4$ and number of fractional bits ≥ 20.

7. CONCLUSIONS AND FUTURE WORK

In this thesis work we have generated fixed-point and floating-point quantization models and placed those in the reference double precision floating-point FFT to analyze the behavior of quantization on fixed-point and floating-point FFT models.

Analysis on both models is performed with different types of stimuli e.g. normal randomly generated, scaled stimulus and full scale stimulus etc. Two different quantization techniques, two's complement and round towards nearest, has been used and SNR curves has been obtained by varying the number of fractional bits for fixed-point and the numbers of fractional bits and the number of exponent bits for floating-point quantized FFT models and results are presented in detail in chapter 6.

For the comparison between normal, scaled and full scale stimulus it can be concluded that full scale values have higher SNR than non-scaled values for fractional fixed-point quantized FFT model, whereas scaling the stimulus improve SNR for floating-point quantized FFT model only at lower number of exponent bits. In the case of non-scaled stimulus, floating-point quantized FFT output SNR is lower for lower number of exponent bits, because saturation in the result, and higher at higher number of exponent bits, while keeping the number of fractional bits constant. Floating-point analysis is performed on flush-to-zero and gradual underflow floating-point models. Results have shown that gradual underflow has always shown higher SNR values as compared to flush-to-zero floating-point FFT, even at lower number of exponent bits no saturation occurs which is evident in flush-to-zero.

From the analysis presented in this thesis work this could be concluded that floating-point quantization shows better results if number of bits allocated for the exponent filed are higher but for lower word widths where number of exponent bits cannot be large enough fixed-point quantization would be our choice.

This research could be improved by extending the analysis on developed quantization models. Analysing the SNR curves based upon different FFT lengths could be one important measure as well.

Possible future work could be to analyze the efficiency of developed quantization models on hardware. Since floating-point computations are always CPU intensive so an optimization between fixed-point and floating-point quantization models could be done on this basis.

Apart from our reference FFT model, developed quantization models could be applied to any DSP application where optimization between fixed-point and floating-point quantization is required.

REFERENCES

[1] Singleton, R. C. An Algorithm for Computing the Mixed Radix Fast Fourier Transform. IEEE Transactions on Audio and Electroacoustics, Vol. 17, No. 2, June 1969.

[2] Elliott, D. F. Handbook of Digital Signal Processing: Engineering Applications. Academic Press January 11, 1988.

[3] Hayes, M. H. Schaum's Outline of Theory and Problems of Digital Signal Processing. McGraw-Hill companies 1999.

[4] Mitra, S. K. Digital Signal Processing: A Computer Based Approach. McGraw-Hill College, 2nd edition July 2001.

[5] Bich Ngoc, D. T. & Ogawa, M. Overflow and Roundoff Error Analysis via Model Checking. Seventh IEEE International Conference on Software Engineering and Formal Methods 2009.

[6] Liu, D. Embedded DSP Processor Design. Morgan Kaufmann 1st edition June 13, 2008.

[7] Mäkinen, R. Fast Fourier Transform on Transport Triggered Architectures. Master of Science Thesis, Tampere University of Technology.

[8] Chang, W. H. & Nguyen, T. Q. On the Fixed-point Accuracy Analysis of FFT Algorithms. IEEE Transactions on Signal Processing, Vol. 56, No. 10, October 2008.

[9] Wong, P.W. Quantization and Round off Noises in Fixed-point FIR Digital Filters. IEEE Transactions on Signal Processing, Vol. 39, No. 7, July 1991.

[10] Claasen, Theo A.C.M. Mechlenbrauker, W.F.G. & Peek, J.B.H.Quantization Noise Analysis for Fixed-point Digital Filters using Magnitude Truncation for Quantization. IEEE Transactions on Circuits and Systems, Vol. CAS-22, No. 11, November 1975.

[11] Parashar, K., Menard, D., Rocher, R. & Senieys, O. Shaping Probability Density Function of Quantization Noise in Fixed-point Systems. Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), 2010.

[12] Kalliojarvi, K. & Astola, J. Roundoff Errors in Block-floating-point Systems. IEEE Transaction on Signal Processing, Vol. 44, No. 4, April 1996.

[13] Smith, S. W. The Scientist and Engineer's Guide to Digital Signal Processing. California Technical Pub. 1st edition 1997.

[14] Brigham, E. Fast Fourier Transform and Its Applications. Prentice Hall, 1st edition April 8, 1988.

[15] Ifeachor, E. C. & Jervis, B. W. Digital Signal Processing: A Practical approach. Addison-Wesley, 1993.

[16] Oppenheim, A. V., Schafer, R. W. & Buck, J. R. Discrete Time Signal Processing. Prentice Hall, 2nd edition January 10, 1999.

[17] http://standards.ieee.org/findstds/standard/754-2008.html [accessed on May 3, 2013 at 00:58 Finland time]

[18] http://www.mathworks.se/support/compilers/R2012b/win64.html [accessed on November 17, 2012 at 17:32 Finland time]

[19] http://dsec.pku.edu.cn/~tieli/notes/num_meth/lect11.pdf [accessed on November 11, 2012 at 20: 58 Finland time]

[20] http://www.mathworks.se/help/matlab/matlab_external/using-mex-files-to-call-c-c-and-fortran-programs.html#brgxbch-1 [accessed on November 17, 2012 at 18:00 Finland time]

[21] http://www.mathworks.se/help/dsp/ref/dsp.rmsclass.html [accessed on April 21, 2013 at 19:28 Finland time]

[22] http://steve.hollasch.net/cgindex/coding/ieeefloat.html [accessed on April 27, 2013 at 19:51 Finland time]

[23] http://standards.ieee.org/findstds/standard/754-1985.html [accessed on May 3, 2013 at 00:55 Finland time]

[24] http://www.analog.com/static/imported-files/tutorials/MT-001.pdf [accessed on May 12, 2013 at 16:42 Finland time]

APPENDIX A

Source code for fixed-point quantization model

```
/*****************************************************************/
/*This function returns the quantized value of given input depending
upon
 * the number of fractional bits given as argument.
 * INPUT:
 * x = Input vector to be quantized
 * frac = Number of fractional bits
 * length = length of the input array
 * Q = Quantization technique:
 *     Two's complement
 *     Round towards nearest
 * warn = Warning to be displayed for overflow and underflow
 *
 * OUTPUT:
 * y = output vector to store the result

*****************************************************************/

#include <math.h>
#include <stdio.h>

double Quant_fraction(double x, int frac, int Q);

void QuantFrac(double* y, double* x, int length, int frac, int Q, int
warn) {

    int counter, r2;
    double r1, r3, q_step, max, min;

    q_step = pow(2.0, 1.0-(double) frac);
    max = 1.0 - q_step;
    min = -1.0;

    mexPrintf("frac:%d, qstep: %f\n", frac, q_step);

    for(counter = 0 ; counter < length ; counter++) {
        //get the input from the input vector
        r1 = *(x + counter);
        if(Q==0) {
            r3=r1;
        }
        else {
            //checking for overflow
            if(r1 < min){
                r3 = min;
                if (warn == 1)
                    mexPrintf("overflow at %f (%f)\n", r1, min);
            }
            //checking for the overflow
            else if(r1 > max){
                r3 = max;
                if (warn == 1)
                    mexPrintf("overflow at %f (%f)\n", r1, max);
```

```
            }
            else{
                r3 = Quant_fraction(r1, frac, Q);
                //multiply the sign
                //r3 = r3 * sign;
            }
        }
        *(y + counter) = r3;
    }
}

double Quant_fraction(double r1, int frac, int Q) {
    double r3;
    double r2;
    double r2temp;

    if (Q == 1 || Q==2 || Q==3) {
        //move the binary point to required amount of bits
        r2temp = r1 * ((double) (1 << (frac - 1)));
        mexPrintf("r2temp: %f\n", r2temp);
        if (Q==2)
            // ROUND TO NEAREST
            r2temp += 0.5;
        if (Q==3 && r1<0) {
            // TRUNCATION OF MAGNITUDE//
            r2 = ceil(r2temp);
        }
        else
            r2 = floor(r2temp);
        mexPrintf("r2: %f\n", r2);
        //scale back to the original number
        r3 =  r2 * pow(2.0, 1.0 - (double)frac);
    }
    else
        //no quantization
        r3=r1;
    return(r3);
}
```

APPENDIX B

Source code for floating-point quantization model

```
/*****************************************************************/
/*This function returns the quantized value of given input depending
upon
 * the number of fractional bits and number of exponent bits given as
argument.
 * INPUT:
 * x = Input vector to be quantized
 * exp_bits = Number of exponent bits
 * frac = Number of fractional bits
 * length = length of the input array
 * Q = Quantization technique, 1,2: Two's complement,Round towards
nearest
 * gflow = gradual uncerflow, on / off: 1 / 0
 * warn = Warning to be displayed for overflow and underflow
 *
 * OUTPUT:
 * y = vector to store the result

*****************************************************************/

#include <math.h>
#include <stdio.h>
#include <mex.h>

void QuantFloat(double* y, double* x, int length, int frac, int
exp_bits, int Q, int gflow, int warn) {

    // execute only if number of exponent bits are greate than one
    if(exp_bits) {

        int Cmin = -(1 << (exp_bits - 1));
        int Cmax = (1 << (exp_bits - 1)) - 1;
        int counter;
        int exp = 0;
        double r1, r3, fraction;
        double maxF, minF, expF;

        minF = 0.5 * pow(2.0, (double) Cmin);
        maxF = (1.0 - pow(2.0, (double) -1*frac));
        maxF = maxF * pow(2.0, (double) Cmax);

        for(counter = 0 ; counter < length ; counter++) {
            int sign = 1;
            r1 = *(x + counter);
            //extracting sign
            if(r1 < 0) {
                sign = -1;
                r1 = -1 * r1;
            }
            //checking for the overflow
            if(r1 > maxF) {
                r3 = sign * maxF;
                if (warn >= 1)
                    mexPrintf("overflow %f\n", r1);
```

```
            }
            else {
                //extract the exponent
                expF = log2(r1);
                expF = floor(expF);
                if(r1 < minF) {
                    if (warn == 1)
                        mexPrintf("underflow %d\n",warn);
                    if (gflow == 0) {
                        //flush-to-zero
                        r3 = 0.0;
                    }
                    else {
                        //graduale underflow
                        //extract the fraction
                        fraction = r1 * pow(2.0, (double)(1-Cmin) );
                        // Perform the fractional quantization, on
fractional part.
                        fraction = Quant_fraction((fraction), frac,
Q);
                        //compute final values
                        r3 = sign*fraction*pow(2.0, (double)(Cmin-1));
                    }
                }
                else {
                    //extract the fraction
                    fraction = r1 * pow(2.0, -expF);
                    // Perform the fractional quantization, on frac-
tional part.
                    fraction = Quant_fraction((fraction), frac, Q);
                    //compute final values
                    r3 = sign*fraction*pow(2, expF);
                }
            }
            //Store results
            *(y + counter) = r3;
        }
    }
    else {
        mexErrMsgTxt("Exponent field width is zero!");
    }
}
```

APPENDIX C

Source code for mex file for Matlab compilation

```
#include <stdio.h>
#include <math.h>
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[],
         int nrhs, const mxArray *prhs[])
{
   int frac_bits, gflow, Q, exp_bits;
   double* oPtr;
   double* iPtr;
   int m, n, k, warn, length;

   if(nrhs != 6)
   {
      mexErrMsgTxt("Usage: x = data vector to be quantized \nfrac:
Number of fractional bits\nexp_bits: number of exponent bits \n Q:
methode of quantization to be used \n gflow: gradual underflow\n Disp:
Display warning if overflow \n");
   }
   else if (nlhs > 1) {
      mexErrMsgTxt("Output of the function cannot be more than 1");
   }
   /* Check that input parameters are in correct format*/
   if( !mxIsDouble(prhs[0]) || !mxIsDouble(prhs[1]) ||
!mxIsDouble(prhs[2]) || !mxIsDouble(prhs[3]) || !mxIsDouble(prhs[4])
|| !mxIsDouble(prhs[5]))
      mexErrMsgTxt("Inputs must be real.\n");

   /* read input parameters */
   m = mxGetM(prhs[0]);
   n = mxGetN(prhs[0]);
   length = m*n;
   iPtr = mxGetPr(prhs[0]);

   frac_bits = (int) mxGetScalar(prhs[1]);
   exp_bits = (int) mxGetScalar(prhs[2]);
   Q = (int) mxGetScalar(prhs[3]);
   gflow = (int) mxGetScalar(prhs[4]);
   warn = (int) mxGetScalar(prhs[5]);
   plhs[0] = mxCreateDoubleMatrix(m,n, mxREAL);

   oPtr = mxGetPr(plhs[0]);
   if (Q==0) {
       for(k=0;k<length;k++)
           *oPtr++ = *iPtr++;
       }
   else if (exp_bits==0)
        QuantFrac(oPtr, iPtr, length, frac_bits, Q, warn);
   else
        QuantFloat(oPtr, iPtr, length, frac_bits, exp_bits, Q, gflow,
warn);
}
```

APPENDIX D

Source code for fixed-point quantized FFT model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DESCRIPTION: This function returns the n-point FFT defined by the
length of the input vector x. Out FFT is fixed point quantized depend-
ing upon the Q_word_width and quantization method
%
% INPUT:
%         x: Input vector for the FFT calculation
%         Q_word_width: Number of fractional bits for quantization
%         Q_method: Quantization method
%         Q_warning: Selection of displaying the warning
% OUTPUT:
%         X: Fixed-point quantized FFT vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function X = fftinplr42_Fixed(x, Q_word_width, Q_method, Q_warning)
s = size(x);
if (s(2) ~= 1)
    error('Input sequence must be a column vector');
end;
N=s(1);
type = rem(log2(N),2);
if  type == 0
    %radix-4 algorithm
    RADIX4=1;
else if type == 1
        %mixed-radix (4-2) algorithm
        RADIX4=0;
    else
        error('Length of input sequence must be a power of two');
    end
end
bN=ceil(log2(N)/2); % Number of butterfly columns

xx=x;

%TWIDDLE FACTORS
[W3,Wk3]=twid3(N);

%Quantize the calculated Twiddle factor
W3 = Quant_mex(real(W3)/2, Q_word_width,0, 2,0,1) + i *
Quant_mex(imag(W3)/2, Q_word_width, 0, 2, 0, 1);

% quantize this result as well
B=zeros(4,1);
%KERNEL COMPUTATIONS
F4 = [1  1  1  1
      1 -i -1  i
      1 -1  1 -1
      1  i -1 -i];
F2=[1 1  0 0
    1 -1 0 0
    0 0  1 1
    0 0  1 -1];
```

```
% scaling factors are for integer computations
scaleF2=1; % check it could be 1
scaleF4=0.5; % check it could be 0.5

r2flag=0;
for stage=0:bN-1
    if RADIX4==0 && stage == bN-1
        r2flag=1;
    end
    blen = N/(4^(stage));
    for k=0:N/4-1
        base1 = floor(4*k/blen)*blen;
        base2 = floor((4*k+2)/blen)*blen;

        idx1 = base1+stride_idx(mod(4*k,blen),blen,ceil(blen/4)) ;
        idx2 = base1+stride_idx(mod(4*k+1,blen),blen,ceil(blen/4)) ;
        idx3 = base2+stride_idx(mod(4*k+2,blen),blen,ceil(blen/4)) ;
        idx4 = base2+stride_idx(mod(4*k+3,blen),blen,ceil(blen/4)) ;

        %Twiddle Factors
        % scaling (/2) is required in fixed point input

        B(1) = (W3(4*k+1,stage+1)*xx(idx1+1));
        B(2) = (W3(4*k+2,stage+1)*xx(idx2+1));
        B(3) = (W3(4*k+3,stage+1)*xx(idx3+1));
        B(4) = (W3(4*k+4,stage+1)*xx(idx4+1));

        B = Quant_mex(real(B),Q_word_width,0,Q_method,0,Q_warning)+
i*Quant_mex(imag(B),Q_word_width,0,Q_method,0,Q_warning);

        % Butterfly (4 or 2-point DFT)
         if r2flag==0
            B=scaleF4*F4*B;
        else
            B=scaleF2*F2*B;
        end
        B =
Quant_mex(real(B),Q_word_width,0,Q_method,0,Q_warning)+i*Quant_mex(ima
g(B), Q_word_width,0, Q_method,0, Q_warning);

        %Write results to memory
        xx(idx1+1) = B(1);
        xx(idx2+1) = B(2);
        xx(idx3+1) = B(3);
        xx(idx4+1) = B(4);

    end

end

%OUTPUT PERMUTATION
XX=zeros(N,1);
for k=0:N-1
    XX(k+1) = xx(r42out_idx(k,N)+1);
end
X = XX*N;
```

APPENDIX E

Source code for floating-point quantized FFT model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DESCRIPTION: This function returns the n-point FFT defined by the
length of the input vector x. Out FFT is floating-point quantized de-
pending upon the number of fractional and exponent bits and quantiza-
tion method
%
% INPUT:
%         x: Input vector for the FFT calculation
%         Q_word_width: Number of fractional bits for quantization
%         Q_exp_width: Number of fractional bits for quantization
%         Q_method: Quantization method
%         Q_gflow: Gradual underflow selection
%         Q_warning: Selection of displaying the warning
% OUTPUT:
%         X: Floating-point quantized FFT vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function X = fftinplr42_Float(x,  Q_word_width, Q_exp_width, Q_method,
Q_gflow, Q_warning)
s = size(x);
if (s(2) ~= 1)
    error('Input sequence must be a column vector');
end;
N=s(1);
type = rem(log2(N),2);
if  type == 0
    %radix-4 algorithm
    RADIX4=1;
else if type == 1
        %mixed-radix (4-2) algorithm
        RADIX4=0;
    else
        error('Length of input sequence must be a power of two');
    end
end
bN=ceil(log2(N)/2); % Number of butterfly columns

xx=x;

%TWIDDLE FACTORS
[W3,Wk3]=twid3(N);

%Quantize the calculated Twiddle factor
W3 = Quant_mex(real(W3), Q_word_width, Q_exp_width, 2, Q_gflow,
Q_warning) + i * Quant_mex(imag(W3), Q_word_width, Q_exp_width, 2,
Q_gflow, Q_warning);

% quantize this result as well
B=zeros(4,1);
%KERNEL COMPUTATIONS
F4 = [1  1  1  1
      1 -i -1  i
      1 -1  1 -1
```

```
      1  i -1 -i];
F2=[1 1  0 0
    1 -1 0 0
    0 0  1 1
    0 0  1 -1];

r2flag=0;
Q_warning = 2;
for stage=0:bN-1
    if RADIX4==0 && stage == bN-1
        r2flag=1;
    end
    blen = N/(4^(stage));
    for k=0:N/4-1
        base1 = floor(4*k/blen)*blen;
        base2 = floor((4*k+2)/blen)*blen;

        idx1 = base1+stride_idx(mod(4*k,blen),blen,ceil(blen/4)) ;
        idx2 = base1+stride_idx(mod(4*k+1,blen),blen,ceil(blen/4)) ;
        idx3 = base2+stride_idx(mod(4*k+2,blen),blen,ceil(blen/4)) ;
        idx4 = base2+stride_idx(mod(4*k+3,blen),blen,ceil(blen/4)) ;

        %Twiddle Factors
        % */2 is not needed in floating point computations
        B(1) = (W3(4*k+1,stage+1)*xx(idx1+1));
        B(2) = (W3(4*k+2,stage+1)*xx(idx2+1));
        B(3) = (W3(4*k+3,stage+1)*xx(idx3+1));
        B(4) = (W3(4*k+4,stage+1)*xx(idx4+1));

        B = Quant_mex(real(B), Q_word_width, Q_exp_width, Q_method,
Q_gflow, Q_warning)+ i * Quant_mex(imag(B), Q_word_width, Q_exp_width,
Q_method, Q_gflow, Q_warning);
        if r2flag==0
            B = F4*B;
        else
            B = F2*B;
        end
        B = Quant_mex(real(B), Q_word_width, Q_exp_width, Q_method,
Q_gflow, Q_warning)+ i * Quant_mex(imag(B), Q_word_width,Q_exp_width,
Q_method, Q_gflow, Q_warning);

        %Write results to memory
        xx(idx1+1) = B(1);
        xx(idx2+1) = B(2);
        xx(idx3+1) = B(3);
        xx(idx4+1) = B(4);

    end

end

%OUTPUT PERMUTATION
XX=zeros(N,1);
for k=0:N-1
    XX(k+1) = xx(r42out_idx(k,N)+1);
end
X = XX;
```