EURECOM
COMPUTATIONAL METHODS
2018

Laboratory Report

Fixed Point FFT

Professor: Raymond Knopp

Authors:

Simili Michele

Contents

1
Introduction

1.1 Objective of the Laboratory

All the work discussed in this report is available at my Github page at `https://github.com/mikilauda95/comp_meth`

The objective of this laboratory is to test the effect of using a fixed point arithmetic when computing FFT radix 4 algorithm. In particular we will test two configurations:

1. Fixed point with 16 bits input operands and results in 16 bits (for both additions and multiplication)
2. Fixed point with multiplications between 25 bits and 18 bits input operands resulting in a 25 bits result and additions between 25 bits.

In order to study the error introduced by this quantization, the results will be compared with the floating point which would represent the case with an infinite precision.

2
Simulation of the fixed point

2.1 Overall of the simulation

The simulation of the FFT is performed in the following steps:

1. **Signal creation**
   We need a signal on which to perform the FFT-radix4 algorithm. In particular, 3 types of signal are considered:

   - Cosine
   - 16QAM signals
   - White noise signal

   It is possible set as a parameter, the number of points to generate. However this number of points must be a power of 4 in order to perform optimally the FFT-radix4.
2. **Preparation of the data**
   Once the signal is created it must be converted in the notation to be tested. This will be better discussed in the next section.
3. **Application of the algorithm**
   The algorithm can finally be performed. It is tested against signal power going from -40 dB to 0 dB at steps of 0.1 dB.
4. **Error computation**
   Once obtained the transformed signal, it is compared with the ideal case (using the floating point). The comparison is performed through the mean square error of all the points. This corresponds to a distortion given by the use of fixed point notation. More details are provided in next sections.

2.1.1 White Noise Generation

In the code provided, the White Noise signal generation was missing. In order to create such a signal, some functions provided have been used.

The waveform is then generated using the following lines of codes:

Listing 2.1: *White Noise Signal*

```
for (i=0; i<N; i++)
data[i].r=pow(10,.05*input_dB)*gaussdouble(0.0, 1)/sqrt(2);
data[i].i=pow(10,.05*input_dB)*gaussdouble(0.0, 1)/sqrt(2);
```

Where the *gaussdouble* function generates numbers in a Gaussian distribution with 0 mean and standard deviation equal to 1.

2.2 Simulation of the Fixed point arithmetic

In order to simulate the fixed point behavior in C, we need to use particular integer types and define the arithmetic operations.

In both the cases, we need to take into account the limited range of the 16 bits and 25 bits that could lead to some overflows. There could be two options to manage this:

1. Not managing the overflow,
2. Saturate in case of overflow

For the kind of algorithm implemented, the second option is the most convenient (this is true in general) as in case of overflows the distance between the ideal result and the limited range one would be limited.

So for the 16 bits case (15 bits module), the *int16_t type* is used, while for the 25x18 bits case the *int32_t* is used.

Here is a snapshot of the two data structures used for this purpose. Note that signals will be complex numbers.

Listing 2.2: *Complex data structures*

```
struct complex
{
  double r;
  double i;
};

struct complex16
{
  int16_t r;
  int16_t i;
};

struct complex32
{
  int r;
  int i;
};
```

Another aspect that needs to be managed is the conversion of the signal from a floating point signal to the respective 16 bits. In the fixed point representation, the module of the number must be be between 0 and 1. at that point the number is represented as 0.XXXXX where the X represents the number in the fixed point notations. So the representable numbers are in the range

$$(-1, 1 - 2^{-(n-1)})$$

with a resolution of $2^{-(n-1)}$ while the unsigned value of the fixed point goes from

$$(-2^{-(n-1)}, 2^{-(n-1)} - 1)$$

To obtain the translation in the fixed point, we need to make the number fall in this range. Then the fixed point notation is returned with the casting to the target type. This is done by the following lines of code:

Listing 2.3: *data_conversion*

```
                data16[i].r = (int16_t)(data[i].r*32767.0);
                data16[i].i = (int16_t)(data[i].i*32767.0);
                data32[i].r = (int)(data[i].r*32767.0);
                data32[i].i = (int)(data[i].i*32767.0);
```

For the implementation of the operations, instead, we cannot use the default multiplication and addition. They are emulated with the following functions:

Listing 2.4: *Saturated multiplication*

```
//Q15 multiply
// x and y are 16-bit integers (Q15)
int16_t FIX_MPY(int x, int y)  {
        return ((int16_t)(((int)x * (int)y)>>15));
}

//25-bit by 18-bit multiply, with output scaled to 25-bits, Q24xQ17
// x is 25-bits stored as 32, y is 18-bits stored as 32
int FIX_MPY25by18(int x,int y) {

  return ((int)(((long long)x * (long long)y)>>17));
}
```

Listing 2.5: *Saturated addition*

```
//Saturated addition for Q15
int16_t SAT_ADD16(int16_t x,int16_t y) {
  if ((int)x + (int)y > 32767)
    return(32767);
  else if ((int)x + (int)y < -32767)
    return(-32768);
  else
    return(x+y);
}

//Saturated addition for Q24
int SAT_ADD25(int x,int y) {

  if ((int)x + (int)y > (1<<24)-1)
    return((1<<24)-1);
  else if ((int)x + (int)y < -(1<<24))
    return(-(1<<24));
  else
    return(x+y);
}
```

So, implementing these functions on the computation of the FFT we have (Q24 example, for Q15 it is analog) :

Listing 2.6: *FFT fixed point implementation Q24*

```
// Radix 4 Butterfly
bfly[0].r = SAT_ADD25(x[n2].r ,SAT_ADD25( x[N2+n2].r,\
SAT_ADD25( x[2*N2+n2].r,x[3*N2+n2].r )));
bfly[0].i = SAT_ADD25(x[n2].i ,SAT_ADD25( x[N2+n2].i,\
SAT_ADD25( x[2*N2+n2].i,x[3*N2+n2].i )));

bfly[1].r = SAT_ADD25(x[n2].r ,SAT_ADD25( x[N2+n2].i, \
SAT_ADD25( -x[2*N2+n2].r,-x[3*N2+n2].i )));
bfly[1].i = SAT_ADD25(x[n2].i ,SAT_ADD25( -x[N2+n2].r, \
SAT_ADD25( -x[2*N2+n2].i, x[3*N2+n2].r )));

bfly[2].r = SAT_ADD25(x[n2].r ,SAT_ADD25( -x[N2+n2].r, \
SAT_ADD25( x[2*N2+n2].r,-x[3*N2+n2].r )));
bfly[2].i = SAT_ADD25(x[n2].i ,SAT_ADD25( -x[N2+n2].i, \
SAT_ADD25( x[2*N2+n2].i, -x[3*N2+n2].i )));

bfly[3].r = SAT_ADD25(x[n2].r ,SAT_ADD25( -x[N2+n2].i, \
SAT_ADD25( -x[2*N2+n2].r, x[3*N2+n2].i )));
bfly[3].i = SAT_ADD25(x[n2].i ,SAT_ADD25( x[N2+n2].r, \
SAT_ADD25( -x[2*N2+n2].i, -x[3*N2+n2].r )));

// In-place results
/*x[n2].r = bfly[0].r;*/
/*x[n2].i = bfly[0].i;*/

for (k1=0; k1<N1; k1++)
{
twiddle_fixed_Q17(&W, N, (double)k1*(double)n2);
x[n2 + N2*k1].r = SAT_ADD25(FIX_MPY25by18(bfly[k1].r, W.r),\
-FIX_MPY25by18(bfly[k1].i, W.i));
x[n2 + N2*k1].i = SAT_ADD25(FIX_MPY25by18(bfly[k1].i, W.r),\
FIX_MPY25by18(bfly[k1].r, W.i));
}
```

The last point that we need to consider is when the fixed point notation must be interpreted. This happens during the calculation of the error where we subtract the result obtained using the floating point with the result of the fixed point. However this passage can be easily done rescaling the number by $2^{(n-1)}$.

Here are some extracts:

Listing 2.7: *Reconversion*

```
mean_error += pow((data[i].r-((double)data32[i].r/(32767.0))),2) \
+ pow((data[i].i-((double)data32[i].i/(32767.0))),2);
```

Last thing to consider when applying the FFT-radix4 algorithm is when to apply a proper rescaling. In fact, given the formula:

$$X_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_k * W_N^{jk}. \tag{2.1}$$

We can see that a division by $\sqrt{N}$ is performed. As N is always a power of 4, the division is always by a power of 2 that corresponds to a shifting. Then, it must be noticed that it is not necessary divide always at the end of the algorithm (that could lead to overflow during the execution of the algorithm), but can be done at any stage. This *scaling* must be so that the total number of shifts is always equal to $\log_2 \sqrt{N}$.

This is accomplished with the following code (just the 64 bits example):

Listing 2.8: *Choice of optimal rescaling*

```
case 64:
for (scale[0]=0;scale[0]<=MAXSHIFT;scale[0]++)
for (scale[1]=0;scale[1]<=MAXSHIFT-scale[0];scale[1]++)
for (scale[2]=0;scale[2]<=MAXSHIFT-scale[0]-scale[1];scale[2]++) {
scale[6]=MAXSHIFT-scale[0]-scale[1]-scale[2];
fft_distortion_test(N,test,configuration,input_dB,\
scale,&maxSNR,maxscale,data,data16,data32);
}
printf("%f, %f, %% Optimum Scaling : %d %d %d %d %d %d %d\n",input_dB,maxSN
maxscale[3],maxscale[4],maxscale[5],maxscale[6]);
```

Here we see that the optimal result is reached in an exhaustive way trying all the different combinations of *scaling* at each stage.

2.3 Simulation results

In order to have smoother plots for the QPSK and white noise tests, a Montecarlo simulation has been adopted (only 10 repetitions resulted enough to have an acceptable results). A python script helped to launch the program several times and to plot the graphs.

Plots are parametrized with the number of points used to compute the FFT. The horizontal line represent the value of 50db which I suppose to be the minimum SNR value in order to identify dome operating ranges.

2.3.1 Cosine signal

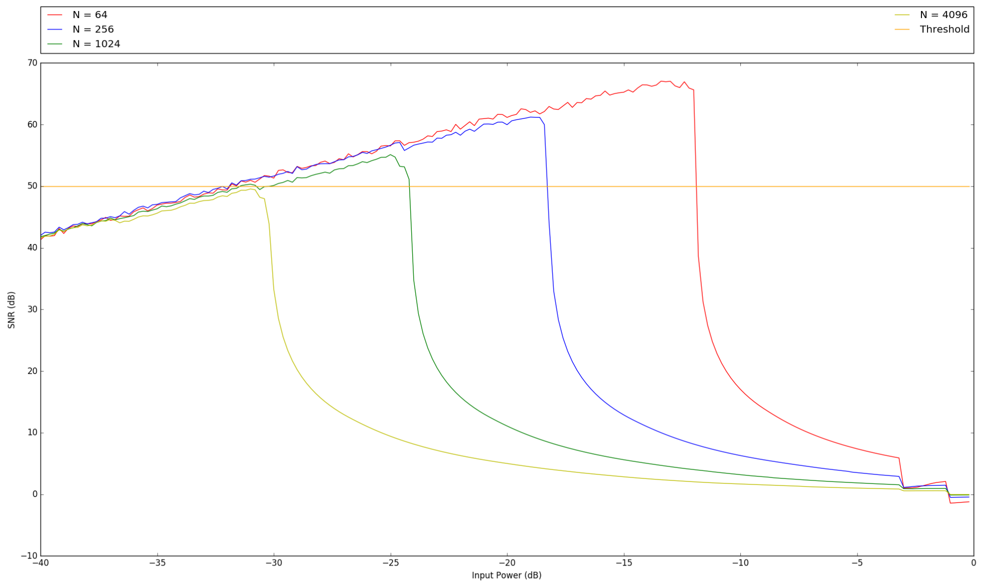

Figure 2.1: *Cosine_ Q15*

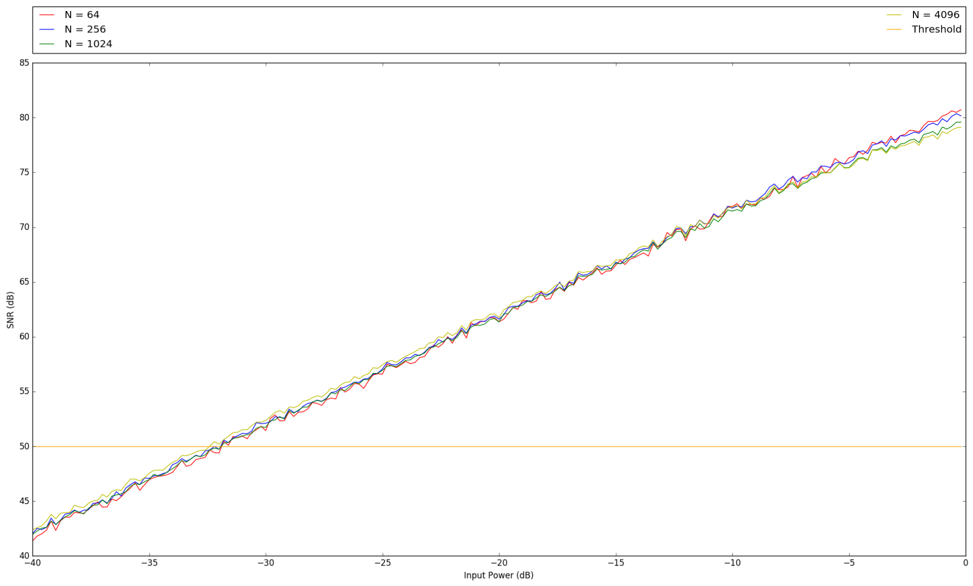

Figure 2.2: *Cosine_ Q24*

2.3.2 16QAM signal

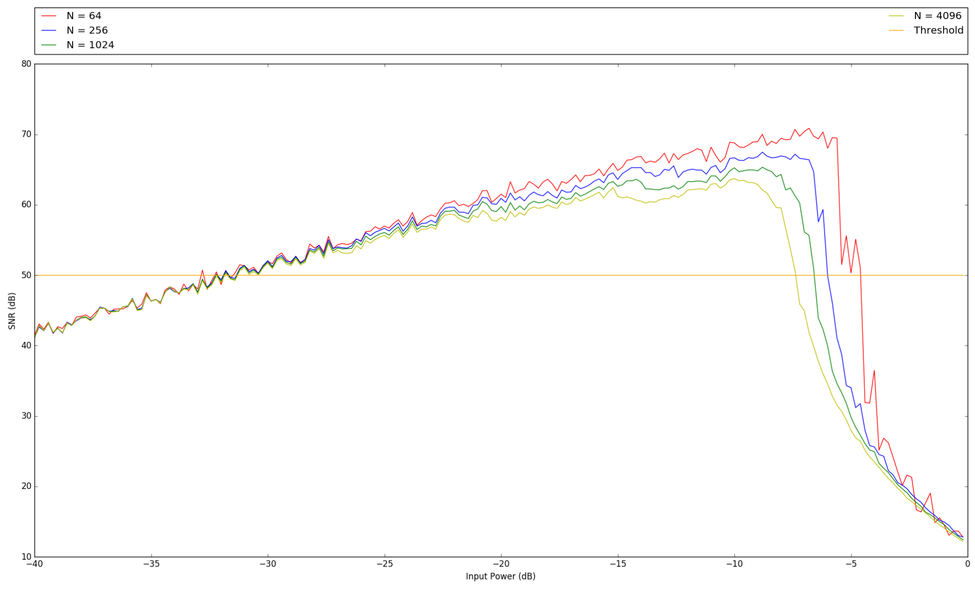

Figure 2.3: *16QAM_ Q15*

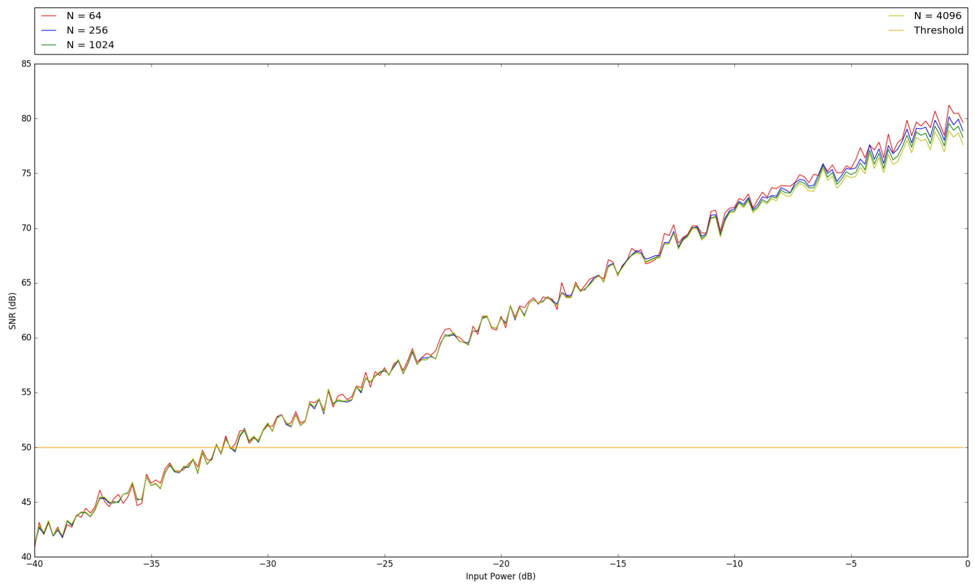

Figure 2.4: *16QAM_ Q24*

2.3.3 White noise signal

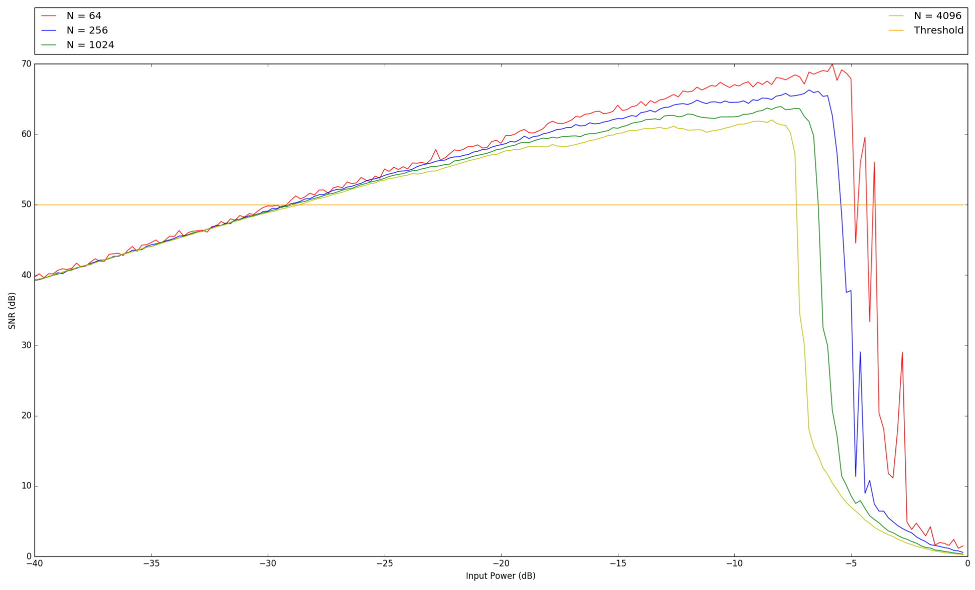

Figure 2.5: *White_ Gaussian_ Noise_ Q15*

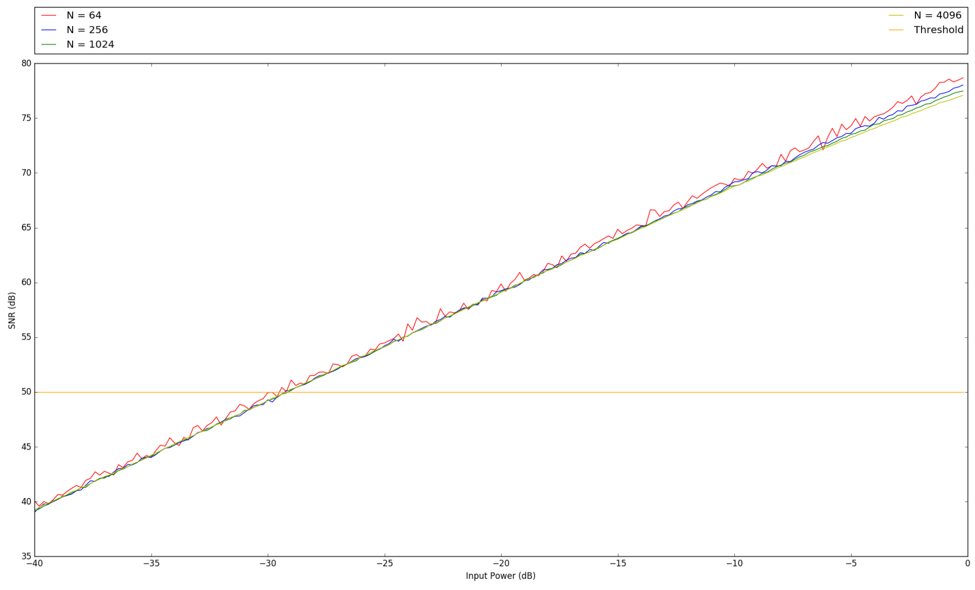

Figure 2.6: *White_ Gaussian_ Noise_ Q24*

2.3.4 Comments

As we can see, we have distortion effects either when the input power is very low or when it is very high. These are caused by two distinct consequences of the fixed point arithmetic:

- Power very low: the distortion comes from the limited resolution provided by the fixed point.

- Power very high: the distortion comes from the saturation of the addition or multiplication.

In the range of the input waveform considered, the biggest error is given by the saturation, which makes the curve (for the Q15 cases) drop dramatically after a certain point.

As the Radix_4 algorithm is implemented through recursive additions (accumulating the results each time), in order not to have a saturation we would need that the additions never return results out of range. This effect is as worse as we add number of points, that means having a bigger number or additions Another source of error will come from the resolution of the fixed point which is not as accurate as the floating point with double precision.

In particular, the saturation is remarkable in the cosine case. In fact if we consider the spectrum of this signal, we see that it is equivalent to two symmetric spikes. (The FFT would not make them symmetric). In particular, as we are working in discrete, the more points we use, the more the spikes are more accentuated. This explains why the worse performance of cases with higher number of points. From a mathematical point of view, we are always summing up the real part and the imaginary part, bringing to a saturation faster.

The Q24 algorithm instead does not suffer of saturation problems (for the range of powers considered). For what concerns the distortion from the resolution, with respect to the Q15 case, it is not shown much improvement (in the range of value considered).

The white noise signal and the 16QAM show better performance, reaching saturation only at around -10 Db. The former however shows a less regular behavior after having reached the saturation in particular if the number of points used is high.
So, it is possible to define the operative range when the SNR is higher than a certain threshold, which are the powers corresponding to the points crossed by the horizontal line at 50 Db:

It is evident that for the Q24 case we can just identify some low boundaries as it does not saturate. However, if we extend the range of the input power (reaching 40 Db) we can find a point where it saturates. Here is a plot (only for the Cosine case.

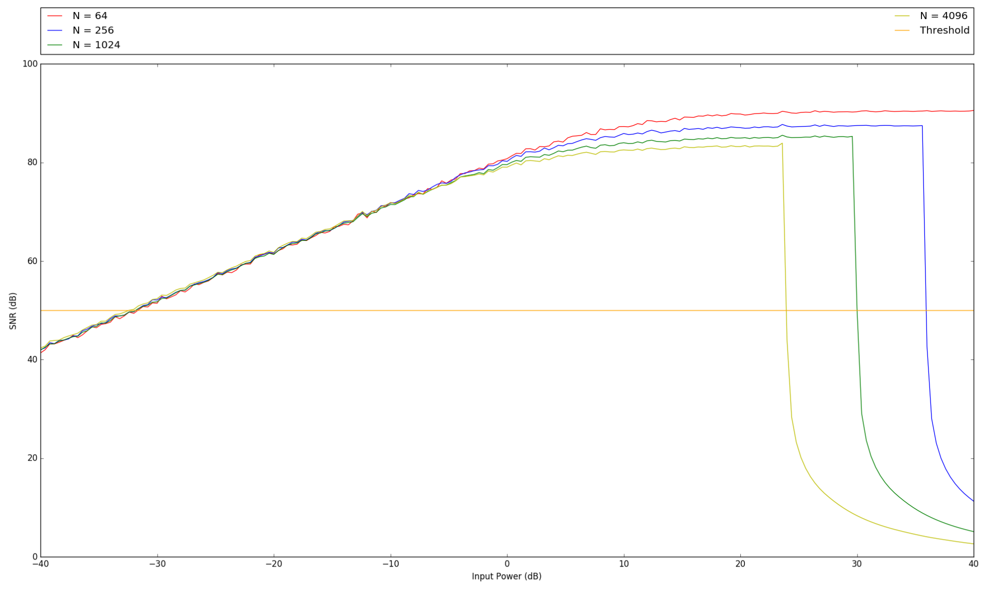

Figure 2.7: *Cosine Q24 Saturation*

2.4 Considerations

2.4.1 Twiddle factor

From the code, we can see that the twiddle factor is computed in the following way:

Listing 2.9: *Twiddle factor computation*

```
W->r=(int16_t)(32767.0*cos(stuff*2.0*PI/(double)N));
W->i=(int16_t)(-32767.0*sin(stuff*2.0*PI/(double)N));
```

This indicates that when the twiddle factor is obtained in the fixed point notation, it is floored. Instead we could think that a better option would be to round it to the most proximate integer. In order to do this we use instead:

Listing 2.10: *Twiddle factor computation*

```
W->r=(int16_t)(round((32767.0*cos(stuff*2.0*PI/(double)N))));
W->i=(int16_t)(round((-32767.0*sin(stuff*2.0*PI/(double)N))));
```

Here is a plot that shows the difference between the two methods:

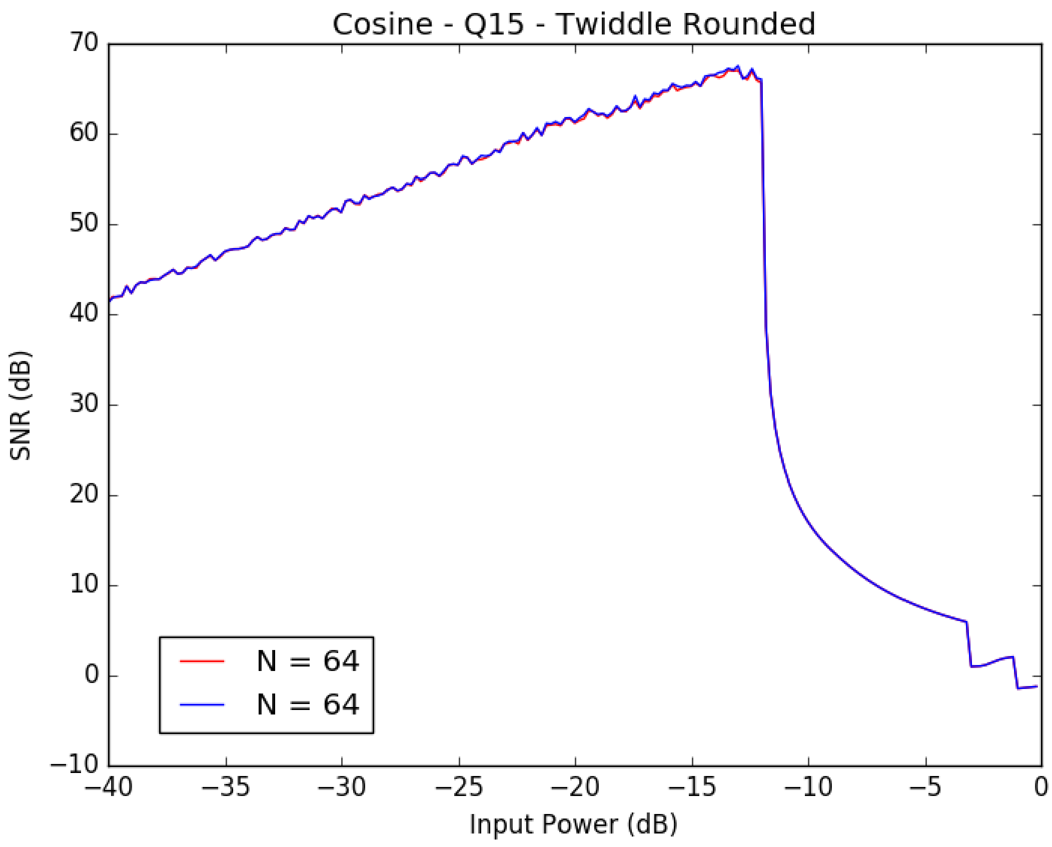

Figure 2.8

As we can see, there is no significant improvement given by the use of one method with respect to the other.

Furthermore, for the tweaking factors in the Q25x15 algorithm, where multiplication is done between a 18 bits number and 25 bits number with a result in 25 bits, are represented in 18 bits. One could argue that it is better to assign 25 bits to the tweaking factors instead of 18, but as the result will still be in 25 bits a rescaling to 18 bits would be necessary. However the overflow is generally given by the additions and so we would probably find more problems with this kind of implementation.