Eurecom

CompMeth

Assignment 1: Fixed-point FFT Signal Processing Technologies

Author:
Martina Fogliato

Professor:
R. Knopp

March 14, 2018

Contents

1 Introduction

The aim of this first assignment is to compare the performances of floating-point and fixed-point arithmetic, applied to the FFT algorithm, a well-known signal processing algorithm for the implementation of the Discrete Fourier Transform.
In fractional fixed-point notation, the number is represented as it is stored in memory. When its bits have to be interpreted, the decimal point must be repositioned by multiplying the number by a fixed scaling factor.
Floating-point representation, instead, uses scientific notation in binary.
Two types of fixed-point implementation are considered: the first one is Q15, while the second one is a Q24 notation in which multiplications are considered to be Q24xQ17. A Q15 number is represented on 16 bits, 15 of which are fractional, whereas a Q24 number is represented on 25 bits, 24 fractional.
Tests can be run on four different waveforms: a cosine waveform, a QPSK input, a 16-QAM signal and white Gaussian noise.
Eventually fixed-point and floating-point results are compared running a distortion test.

2 Developed code and Fixed-point routines

Contrary to what happens in floating-point operations, for fixed-point arithmetic we need to emulate the behavior on a limited number of bits. To do so, some functions are provided in

the *fixed_point.c* file; the radix-4 FFT algorithm for both Q15 and Q24 fixed-point notation had to be completed, applying the proper one.

2.1 Q15 fixed-point

The first modifications to be implemented are in the Radix-4 butterfly code:

```
  bfly[0].r = SAT_ADD16(SAT_ADD16(x[n2].r, x[N2 + n2].r),
   ↪  SAT_ADD16(x[2*N2+n2].r, x[3*N2+n2].r));
    bfly[0].i = SAT_ADD16(SAT_ADD16(x[n2].i, x[N2 + n2].i),
     ↪  SAT_ADD16(x[2*N2+n2].i, x[3*N2+n2].i));

    bfly[1].r = SAT_ADD16(SAT_ADD16(x[n2].r, x[N2 + n2].i),
     ↪  SAT_ADD16(-1*x[2*N2+n2].r, -1*x[3*N2+n2].i));
    bfly[1].i = SAT_ADD16(SAT_ADD16(x[n2].i, -1*x[N2 + n2].r),
     ↪  SAT_ADD16(-1*x[2*N2+n2].i, x[3*N2+n2].r));

    bfly[2].r = SAT_ADD16(SAT_ADD16(x[n2].r, -1*x[N2 + n2].r),
     ↪  SAT_ADD16(x[2*N2+n2].r, -1*x[3*N2+n2].r));
    bfly[2].i = SAT_ADD16(SAT_ADD16(x[n2].i, -1*x[N2 + n2].i),
     ↪  SAT_ADD16(x[2*N2+n2].i, -1*x[3*N2+n2].i));

    bfly[3].r = SAT_ADD16(SAT_ADD16(x[n2].r, -1*x[N2 + n2].i),
     ↪  SAT_ADD16(-1*x[2*N2+n2].r, x[3*N2+n2].i));
    bfly[3].i = SAT_ADD16(SAT_ADD16(x[n2].i, x[N2 + n2].r),
     ↪  SAT_ADD16(-1*x[2*N2+n2].i, -1*x[3*N2+n2].r));

    // In-place results
    for (k1=0; k1<N1; k1++){
            twiddle_fixed(&W, N, (double)k1*(double)n2);
      x[n2 + N2*k1].r = SAT_ADD16(FIX_MPY(bfly[k1].r, W.r),
       ↪  -1*FIX_MPY(bfly[k1].i, W.i));
      x[n2 + N2*k1].i = SAT_ADD16(FIX_MPY(bfly[k1].i, W.r),
       ↪  FIX_MPY(bfly[k1].r, W.i));
     }
}

  // Don't recurse if we're down to one butterfly
if (N2!=1){
  for (k1=0; k1<N1; k1++){
```

```
    radix4_fixed_Q15(&x[N2*k1], N2,scale,stage+1);
  }
```

2.1.1 SAT_ADD16

Each sum is performed through SAT_ADD16 function, which emulates a saturated addition on 16-bits: the sum of the two operands is returned as it is, unless it needs more than 16 bits to be represented; in this case it is either saturated to $2^{16} - 1$ (if positive), or -2^{16} (if negative).

```
int16_t SAT_ADD16(int16_t x,int16_t y) {

  if ((int32_t)x + (int32_t)y > 32767)
    return(32767);
  else if ((int32_t)x + (int32_t)y < -32767)
    return(-32768);
  else
    return(x+y);
}
```

2.1.2 FIX_MPY

For the multiplications, instead, FIX_MPY function is used. Since both input numbers are Q15, represented on 16 bits, the result of a multiplication would fit on 30 bits + the sign bit. Therefore, in order to store only the 16 most significant digits, we need a 15-bit right shift.
That is why the two inputs are first cast to 32 bits integers, then they are multiplied and the result is shifted by 15, eventually casting everything to 16 bits.

```
int16_t FIX_MPY(int16_t x, int16_t y){
  return ((int16_t)(((int32_t)x * (int32_t)y)>>15));
}
```

2.2 Q24 fixed-point

Similar changes should be applied when the input is given on 24 bits:

```
  bfly[0].r = SAT_ADD25(SAT_ADD25(x[n2].r, x[N2 + n2].r),
   ↪  SAT_ADD25(x[2*N2+n2].r, x[3*N2+n2].r));
    bfly[0].i = SAT_ADD25(SAT_ADD25(x[n2].i, x[N2 + n2].i),
     ↪  SAT_ADD25(x[2*N2+n2].i, x[3*N2+n2].i));
```

```
3
4       bfly[1].r = SAT_ADD25(SAT_ADD25(x[n2].r, x[N2 + n2].i),
        ↪  SAT_ADD25(-1*x[2*N2+n2].r, -1*x[3*N2+n2].i));
5       bfly[1].i = SAT_ADD25(SAT_ADD25(x[n2].i, -1*x[N2 + n2].r),
        ↪  SAT_ADD25(-1*x[2*N2+n2].i, x[3*N2+n2].r));
6
7       bfly[2].r = SAT_ADD25(SAT_ADD25(x[n2].r, -1*x[N2 + n2].r),
        ↪  SAT_ADD25(x[2*N2+n2].r, -1*x[3*N2+n2].r));
8       bfly[2].i = SAT_ADD25(SAT_ADD25(x[n2].i, -1*x[N2 + n2].i),
        ↪  SAT_ADD25(x[2*N2+n2].i, -1*x[3*N2+n2].i));
9
10      bfly[3].r = SAT_ADD25(SAT_ADD25(x[n2].r, -1*x[N2 + n2].i),
        ↪  SAT_ADD25(-1*x[2*N2+n2].r, x[3*N2+n2].i));
11      bfly[3].i = SAT_ADD25(SAT_ADD25(x[n2].i, x[N2 + n2].r),
        ↪  SAT_ADD25(-1*x[2*N2+n2].i, -1*x[3*N2+n2].r));
12
13      // In-place results
14      for (k1=0; k1<N1; k1++){
15              twiddle_fixed_Q17(&W, N, (double)k1*(double)n2);
16        x[n2 + N2*k1].r = SAT_ADD25(FIX_MPY25by18(bfly[k1].r, W.r),
          ↪  -1*FIX_MPY25by18(bfly[k1].i, W.i));
17              x[n2 + N2*k1].i = SAT_ADD25(FIX_MPY25by18(bfly[k1].i, W.r),
                ↪  FIX_MPY25by18(bfly[k1].r, W.i));
18      }
19  }
20
21    // Don't recurse if we're down to one butterfly
22  if (N2!=1){
23    for (k1=0; k1<N1; k1++){
24       radix4_fixed_Q24xQ17(&x[N2*k1], N2,scale,stage+1);
25    }
26  }
```

2.2.1 SAT_ADD25

The procedure is the same as in SAT_ADD16, but this time the sum is either saturated to $2^{24} - 1$ or -2^{24} and the result is given as a 32-bit integer (even if it fits 25 bits).

```
1   int32_t SAT_ADD25(int32_t x,int32_t y) {
2
```

```
  if ((int32_t)x + (int32_t)y > (1<<24)-1)
    return((1<<24)-1);
  else if ((int32_t)x + (int32_t)y < -(1<<24))
    return(-(1<<24));
  else
    return(x+y);
}
```

2.2.2 FIX_MPY25by18

This function is used to multiply a 25-bit input with a 18-bit one, both stored on 32 bits. The result is scaled on 25 bits. This configuration is typical of DSP units in current generation Xilinx FPGAs.
A Q24xQ17 multiplication would fit on at most 41 bits + the sign bit. However, since we would like to store only the 25 most significant bits (including the sign bit), we need to right-shift the result by 42-25=17.
To do so, the input numbers are first cast into `int64_t`, then the result of their multiplication is right-shifted and eventually cast into `int32_t`.

```
int32_t FIX_MPY25by18(int32_t x,int32_t y) {
  return ((int32_t)(((int64_t)x * (int64_t)y)>>17));
}
```

3 Twiddle factor quantization

If we define the FFT algorithm as follows:

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n W_N^{kn} \tag{1}$$

where x_n is the discrete-time input waveform and X_k is the resulting output waveform, the twiddle factor is $W_N = e^{\frac{-j2\pi}{N}}$. In particular, the twiddle factor W_N can be seen as a vector that rotates on the unit circle according to the number of samples N. Therefore:

$$W_N^n = \cos\left(\frac{2\pi n}{N}\right) - j\sin\left(\frac{2\pi n}{N}\right) \tag{2}$$

When dealing with Q15 and Q24 fixed-point numbers the twiddle factor must be represented taking into account the limited number of bits as well.
In the Q15 case, first the real and imaginary parts of the twiddle factor are computed separately, then they are both right-shifted by multiplying them by $2^{16} - 1$ and eventually they

are cast into `int16_t`. In this way only the 16 most significant bits are stored.
A similar procedure is applied in the Q24 case, but the real and imaginary parts are shifted by 17 positions and cast into `int_32t` to store the 25 most significant bits.
By performing a cast, the numbers are truncated and the `floor` value is considered. However, another possibility, that may lead to more precise results, would be to implement rounding:

```
void twiddle_fixed(struct complex16 *W, int32_t N, double stuff){
  W->r=(int16_t)((32767.0*cos(stuff*2.0*PI/(double)N))+0.5);
  W->i=(int16_t)((-32767.0*sin(stuff*2.0*PI/(double)N))+0.5);
}
```

```
void twiddle_fixed_Q17(struct complex32 *W, int32_t N, double stuff){
  W->r=(int32_t)((((1<<17)-1)*cos(stuff*2.0*PI/(double)N))+0.5);
  W->i=(int32_t)(((1-((1<<17)-1))*sin(stuff*2.0*PI/(double)N))+0.5);
}
```

I ran again all tests for both Q15 and Q24 numbers (with 1024 points) in order to display the difference between the two methods. Figures 1 and 2 clearly show that, by using rounding, there is a noticeable improvement in the distortion that can go up to 6dB in some cases.

4 Distortion test

The distortion is characterized as a comparison between the waveform obtained with floating-point arithmetic and the one obtained with fixed-point.
In particular, according to the number N chosen by the user when launching the program, a certain number of samples of the specified input waveform are generated. Then, the FTT algorithm is applied first with floating-point precision, then with fixed-point, either Q15 or Q24; eventually the distortion test is run using the following procedure:

$$\bar{x}^{in} = \sum_{i=0}^{N-1} \left[(\mathbb{R}e\left\{x_i^{in}\right\})^2 + (\mathbb{I}m\left\{x_i^{in}\right\})^2 \right] \tag{3}$$

$$\bar{\epsilon} = \sum_{i=0}^{N-1} \left[\left(\mathbb{R}e\left\{ x_i^{in} - \frac{x_i^{FP}}{2^{16}-1} \right\} \right)^2 + \left(\mathbb{I}m\left\{ x_i^{in} - \frac{x_i^{FP}}{2^{16}-1} \right\} \right)^2 \right] \tag{4}$$

$$SNR = 10 \log_{10} \frac{\bar{x}^{in}}{\bar{\epsilon}} \tag{5}$$

Actually the Q24 routine was missing, therefore I duplicated the Q15 routine modifying it properly to be suitable to the Q24 fixed-point case. Moreover, I added a third argument to

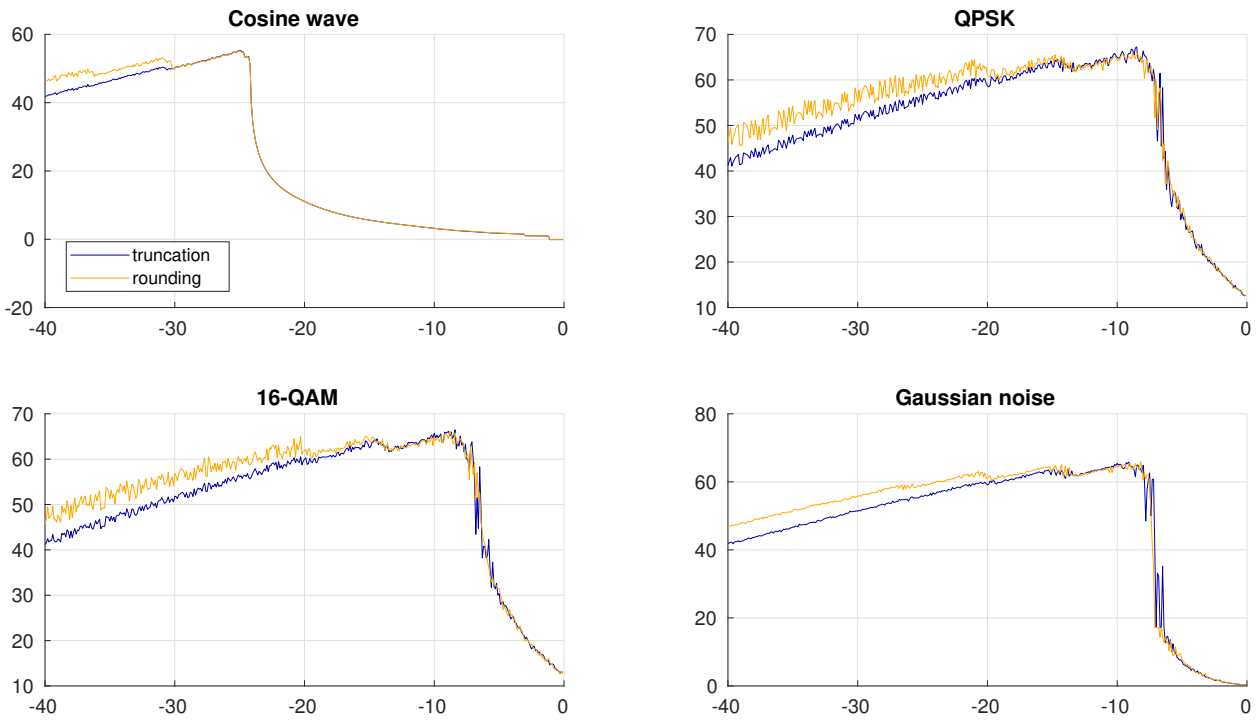

Figure 1: Q15 fixed-point; truncating vs round

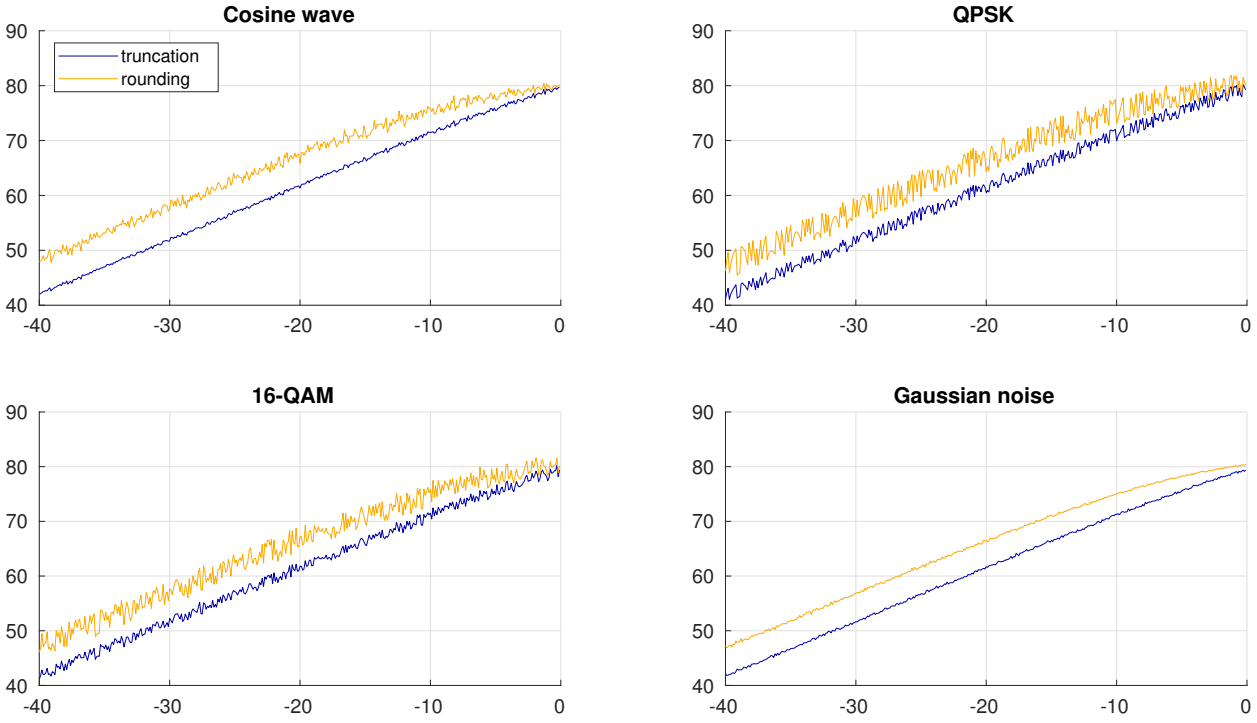

Figure 2: Q24 fixed-point; truncating vs round

be passed by the user from the command line (*type*), which specifies if the test should be run on Q15 or on Q24.

```
else if(type==1){
  radix4_fixed_Q24xQ17(data32, N, scale, 0);
  bit_r4_reorder_fixed_Q17(data32, N, scale[6]);

  // Compute Distortion statistics
  mean_error = 0.0;
  mean_in = 0.0;
  for (i=0;i<N;i++) {
    mean_in += data[i].r*data[i].r + data[i].i*data[i].i;
    mean_error += pow((data[i].r-((double)data32[i].r/32767.0)),2) +
     ↪ pow((data[i].i-((double)data32[i].i/32767.0)),2);
  }

  SNR = 10*log10(mean_in/mean_error);
  if (SNR > *maxSNR) {
    *maxSNR = SNR;
    memcpy(maxscale,scale,7); //save in maxscale the best possible
     ↪ scaling
  }
}
```

What this routine does is computing the Signal-to-Noise (SNR) ratio, where the "signal" we are referring to is the data resulting from floating-point processing (since it yields a better precision, supporting a wider range) and the "noise" is the distortion, that is the difference between the floating-point result and the fixed-point one.
This procedure is iterated through several possible scaling schedules, eventually choosing the best one, yielding the highest SNR (the higher the SNR, the smaller the distortion and consequently the difference between floating-point and fixed-point processing).
Before calling function *fft_distortion_test*, however, two other functions must be called in order to properly initialize random numbers generation:

```
randominit();
set_taus_seed();
```

The plots in Figure 3 and Figure 4 depict the relationship between input (in dB) and resulting SNR, both for Q15 and Q24 fixed-point, for all types of waveforms and number of sampling points ranging from 64 to 4096.

From Figure 3 it is clear that, increasing the strength of the input signal, the effect of the distortion becomes less and less impactive, improving the SNR, up to a certain point after which the SNR degrades rapidly due to an overflow, since the number of bits is not sufficient anymore.
In Figure 4, instead, the drop is not visible, which means that in all cases Q24 numbers are enough to properly represent all sums and multiplications performed by the algorithm without overflow.
Figure 5 is useful to perform a graphical comparison between Q15 and Q24 fixed-point representation on the same type of input waveform.

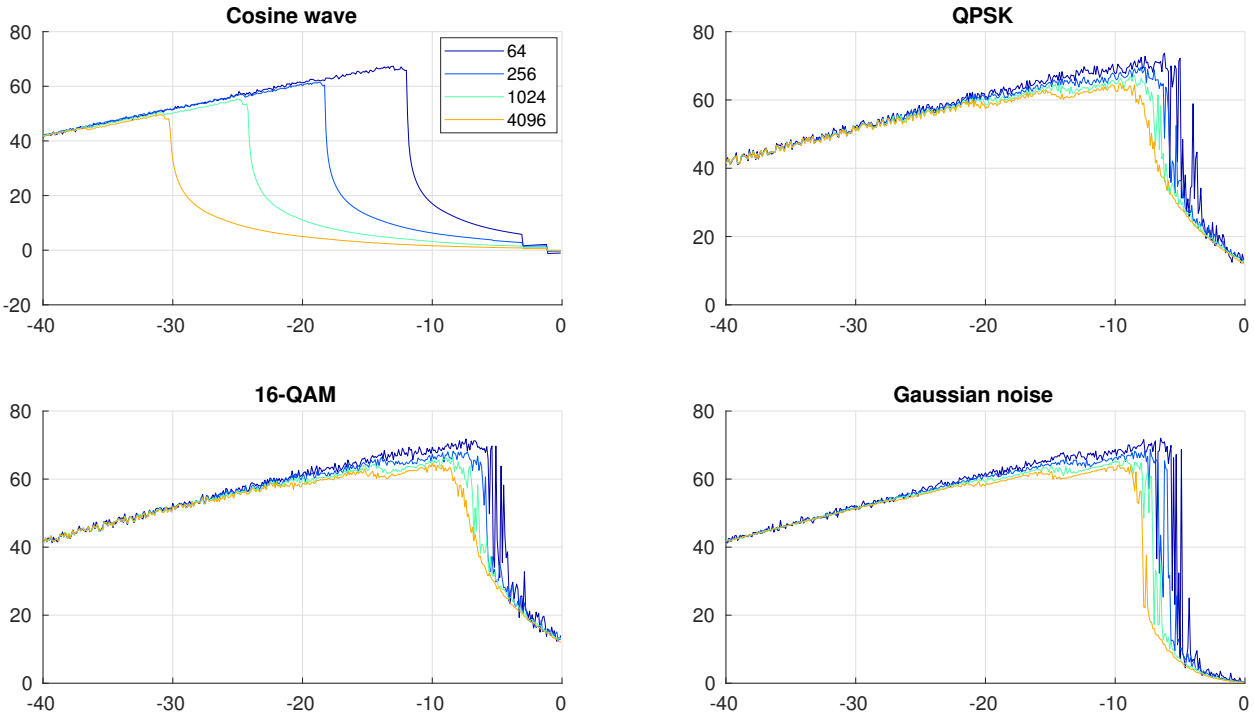

Figure 3: Q15 fixed-point

5 Dynamic range

The dynamic range can be seen as the maximum signal level divided by the noise level and usually expressed in dB. Therefore, in this case the dynamic range would delimit the range of input signal strength, given an acceptable minimum value of SNR (target operating point). Figure 5 shows the comparison between Q15 and Q24 fixed-point notation, for each test and on 1024 sampling points.
I fixed 50dB as an acceptable SNR level. As for Q15, in the first test the input range is 6.9dB, in the second one is 25dB, in the third test is 25.4dB, while in the fourth one is

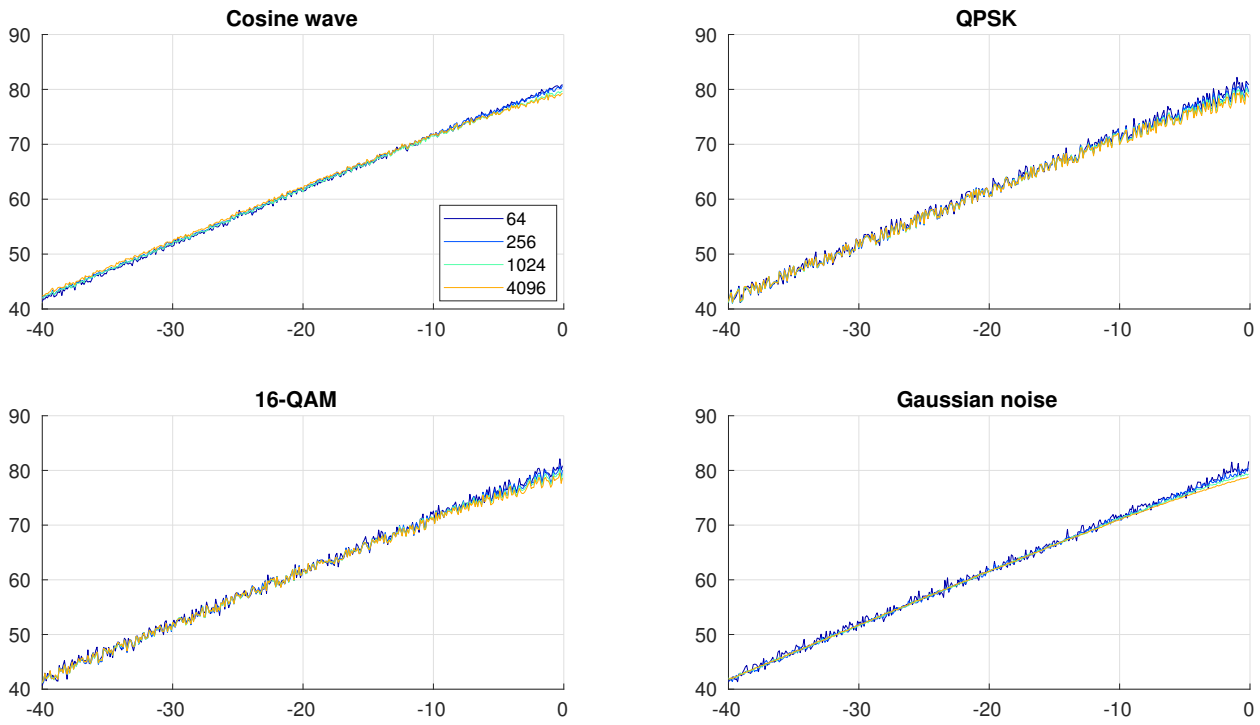

Figure 4: Q24 fixed-point

24.1dB.

As for Q24, instead, it is not possible to define a proper range because the upper bound is missing (in the input strength range -40dB to 0dB). The lower bound more or less corresponds to the Q15 case.

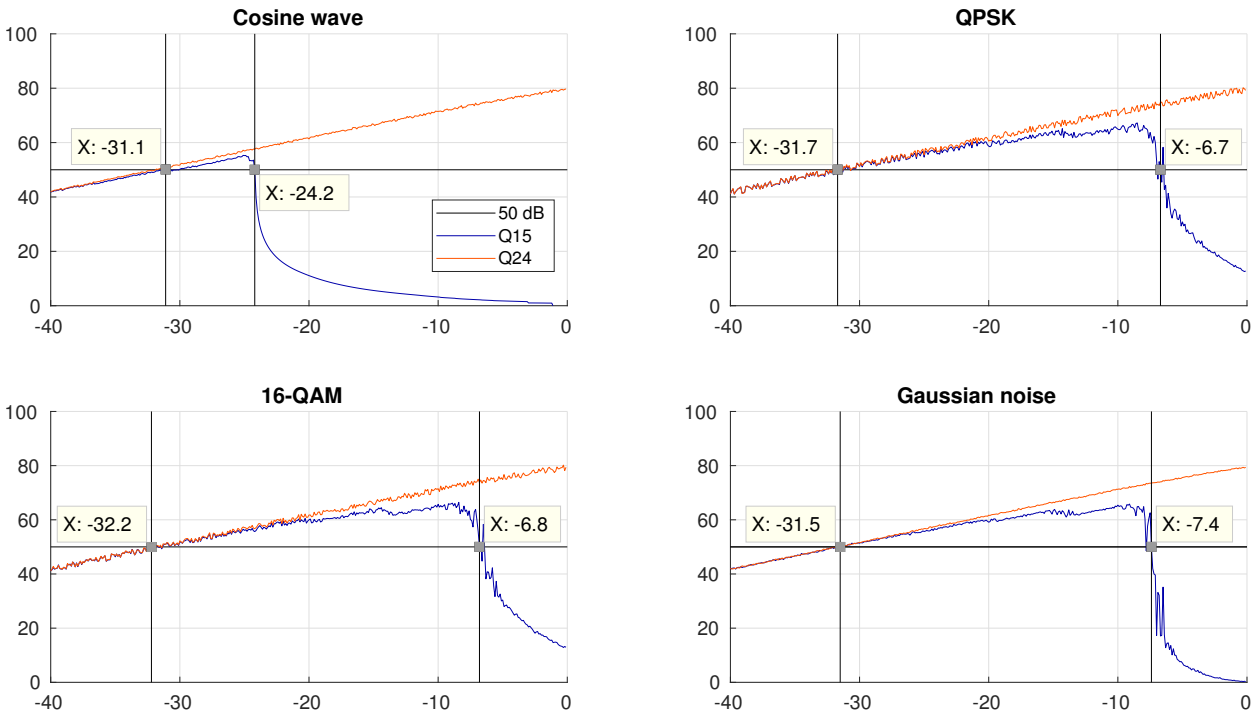

Figure 5: Dynamic range