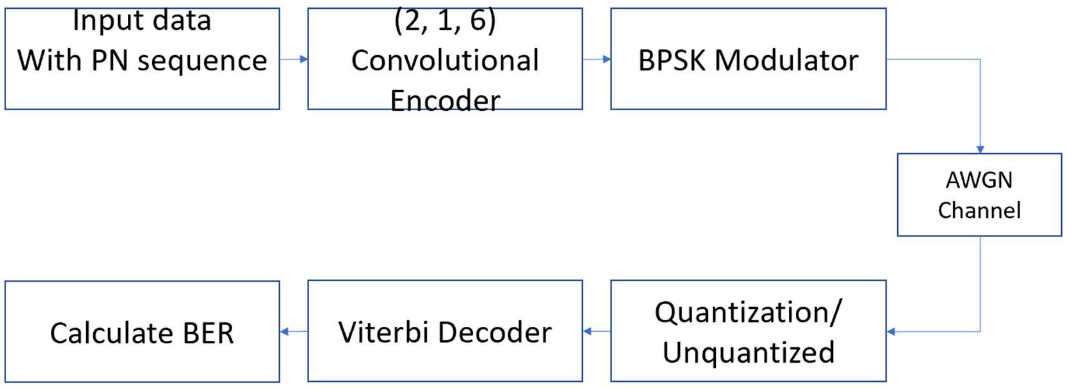
Project 1: Convolutional Encoder & Viterbi Decoder

110064518 劉育瑋

Brief description to the project:



Block diagram of the encoding and decoding procedure

First, we send our input data, namely the PN sequence, into the (2, 1, 6) convolutional encoder to perform encoding. Then, by applying BPSK, we map the coded bit 0 to +1 and 1 to -1 and add AWGN with noise variance $\sigma^2 = (\frac{E_b}{N_0})^{-1}$ to simulate AWGN channel.

Next, we perform either hard decision via binary quantization with decision boundary at 0 or unquantized soft decision. Finally, by applying Viterbi decoding algorithm, we can output our decoded bits. Comparing them with the input data, we could calculate the bit error rate (BER).

Here, we'd like to state more about how our decoder works. Note that we'll first set the initial condition of the encoder to all-zero state. Then, with the received noised bits, we perform only metric adding and survivor exchange within first 6 rounds (6 input bits) since the updating doesn't include comparing in this interval. After all states having their metrics, comparing and selecting start and the survivors exchange based on the selection performed. The decoder will output a decoded bit according to the output decision mode (e.g. best-state decision) if the truncation sliding window is fully filled. The window will slide one bit for the next bit decoding.

For hard decision, the metric addition is simply adding the Hamming distance between quantized received bits and corresponding possible codeword. While for soft decision, we can add the square of Euclidian distance $(x_i - y_i)^2$ between the received bit and the corresponding bit of the possible codeword. Also, since $(x_i - y_i)^2 = x_i^2 - 2x_iy_i + y_i^2$ and the y_i^2 are the same for the metric adding to each state. Moreover, the x_i^2 are all equal to 1 due to BPSK mapping. Thus, the remaining $-2x_iy_i$ can be normalized to y_i' whose sign is according to x_i.

1. Simulation result:

   For hard decision,

SNR (dB)	N (# of decoded bits)	K (# of error bits)	BER
1	1e7	2599339	0.260
1.5	1e7	1907432	0.191
2	1e7	1246199	0.125
2.5	1e7	719615	7.20e-2
3	1e7	357589	3.56e-2
3.5	1e7	156241	1.56e-2
4	1e7	60506	6.05e-3
4.5	1e7	20708	2.07e-3
5	1e7	6134	6.13e-4
5.5	1e7	1769	1.77e-4
6	1e7	384	3.84e-5

   For soft decision,

SNR (dB)	N (# of decoded bits)	K (# of error bits)	BER
1	1e7	509819	5.10e-2
1.5	1e7	214457	2.14e-2
2	1e7	73937	7.39e-3
2.5	1e7	20720	2.07e-3
3	1e7	5339	5.34e-4
3.5	1e7	1083	1.08e-4
4	1e7	170	1.70e-5

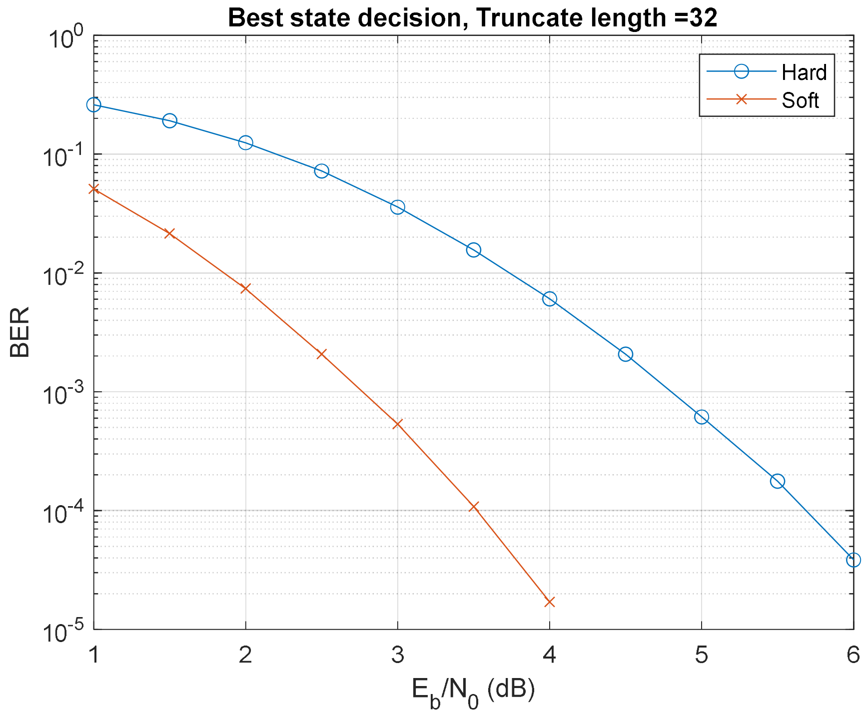


Fig 1 Performance of hard and soft decision

From Fig 1, about a 2-dB loss can be seen while applying hard decision comparing to unquantized soft decision, which is mentioned in the class.

Note: To maintain at least 3-digit significance within the range of simulated SNR, we select N = 1e7 bits.

2. (Optional) Effects of different output decision alternatives:

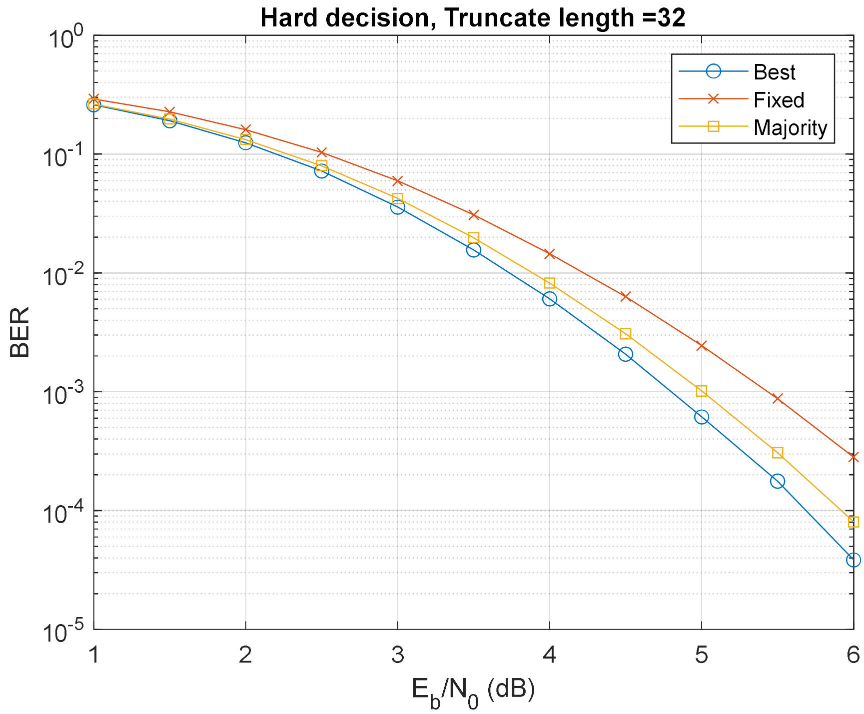


Fig 2. Performance of different output decision alternative (hard decision)

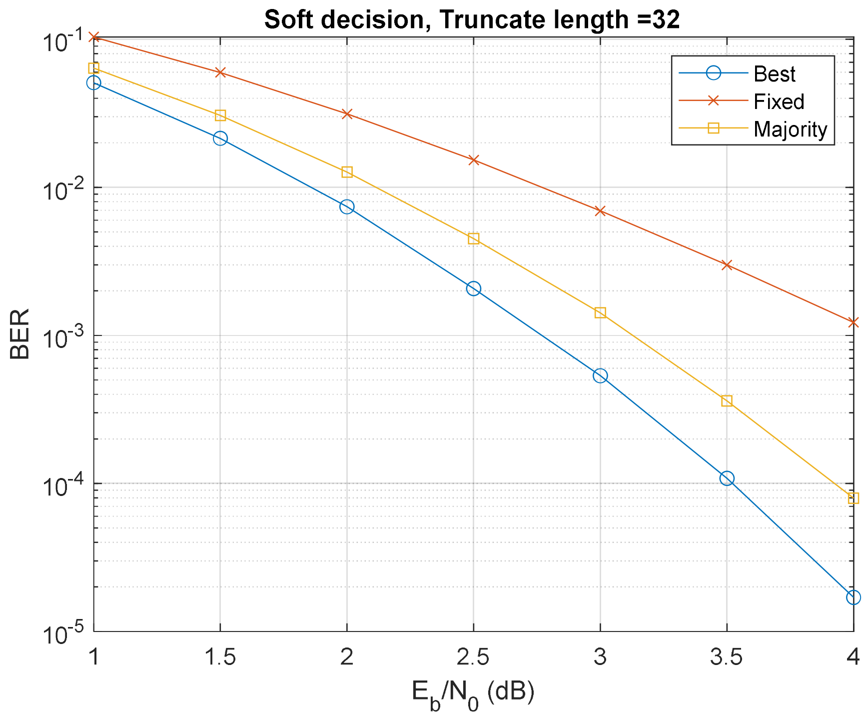


Fig 3. Performance of different output decision alternative (soft decision)

From Fig 2 and Fig 3, in the cases of both hard and soft decision, best-state decision outperforms majority-votes decision and majority-votes decision outperforms fixed-state decision.

With hard decision, the difference between 3 decision mode (<1 dB) is visibly smaller than one (>2 dB) with soft decision. It suggests that it’s less sensitive to output decision mode with hard decision.

Here, we’ve got the best performance via best-state decision, and the lowest complexity with fixed-state decision. For convenience and simplicity, we discard “majority-vote decision” in the following discussion.

3. (Optional) Effects of different truncation lengths:

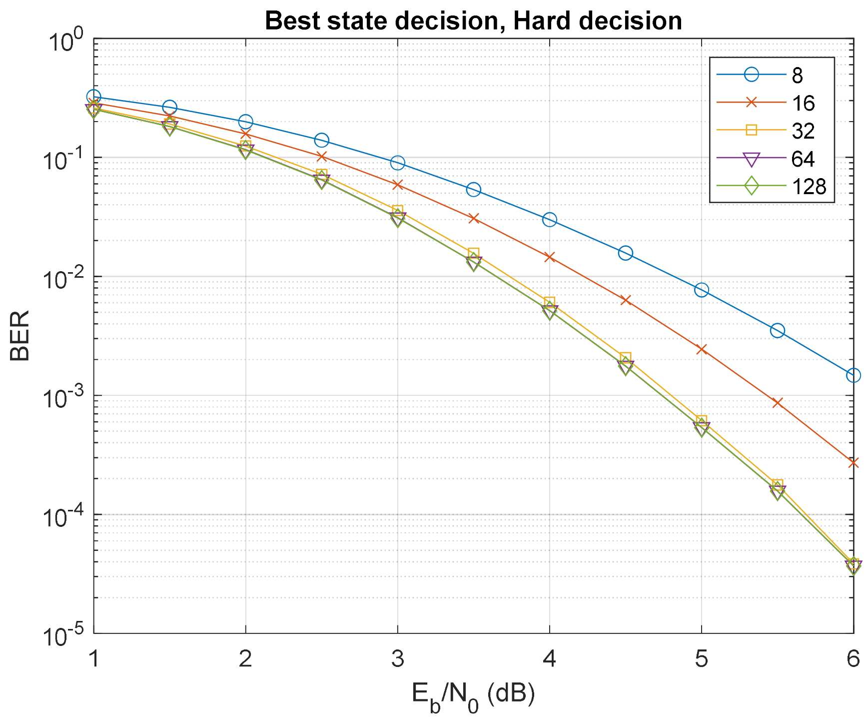


Fig 4. Performance of different truncation lengths (hard decision)

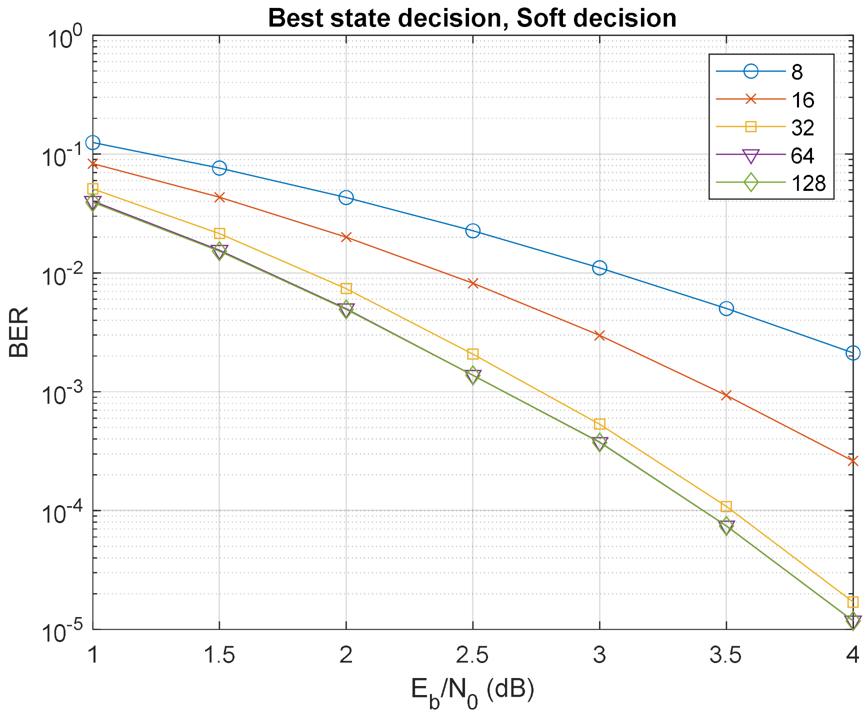


Fig 5. Performance of different truncation lengths (soft decision)

From Fig 4 and Fig 5, we could see that as truncation length increases, the performance becomes better. While the truncation length is larger than 32, the performance saturates and, nearly no difference as it is up to 64, or even 128.

Note: The truncation lengths chosen above are based on 8-bit (byte) memory storage within possible hardware decoders.

4. (Optional) Is 5m a good choice of truncation length (with best-state decision)? (m: # of registers in the encoder)

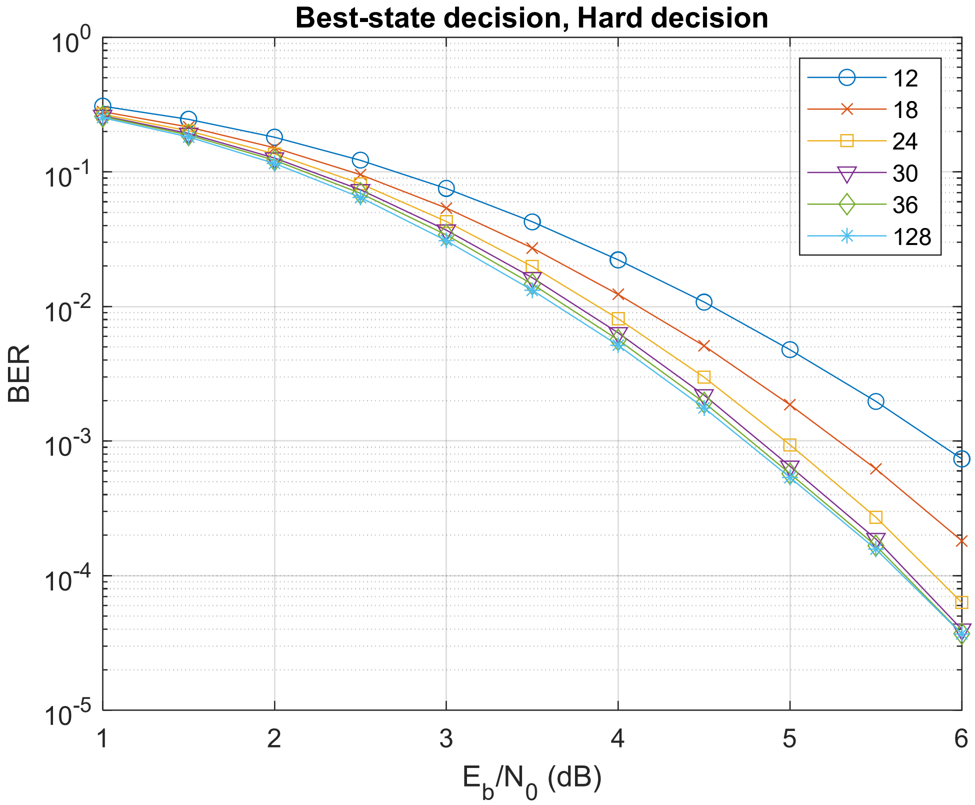


Fig 6.1 Performance of multiples of m in truncation lengths (hard decision)

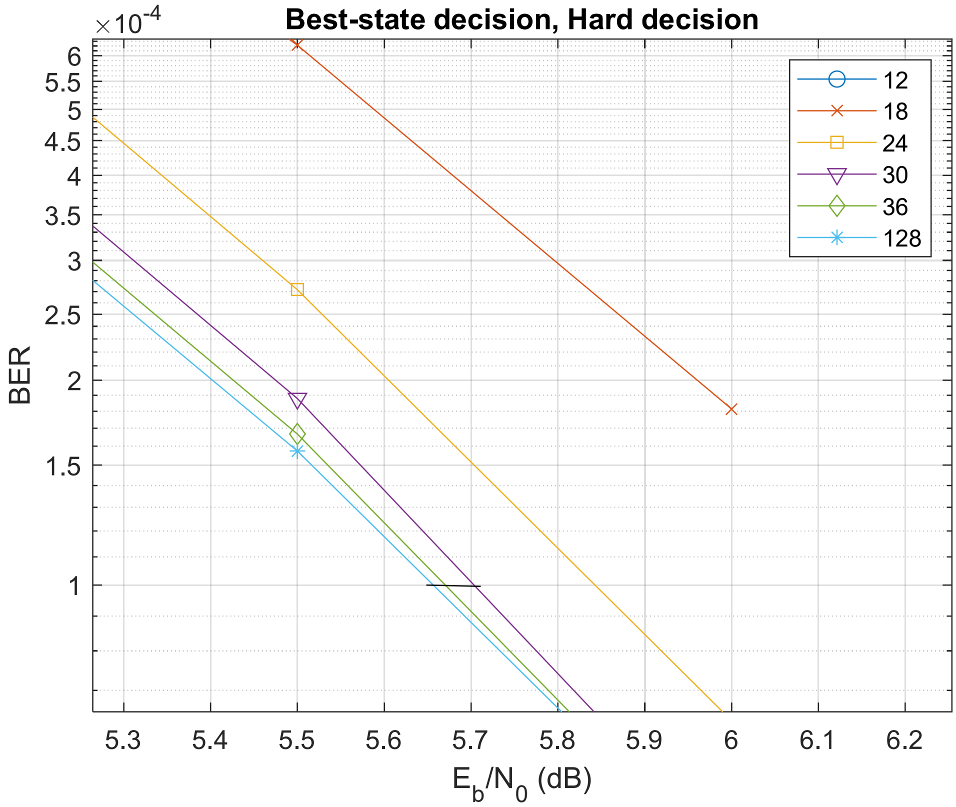


Fig 6.2 Zoom in at BER = 1e-4 in Fig 6.1

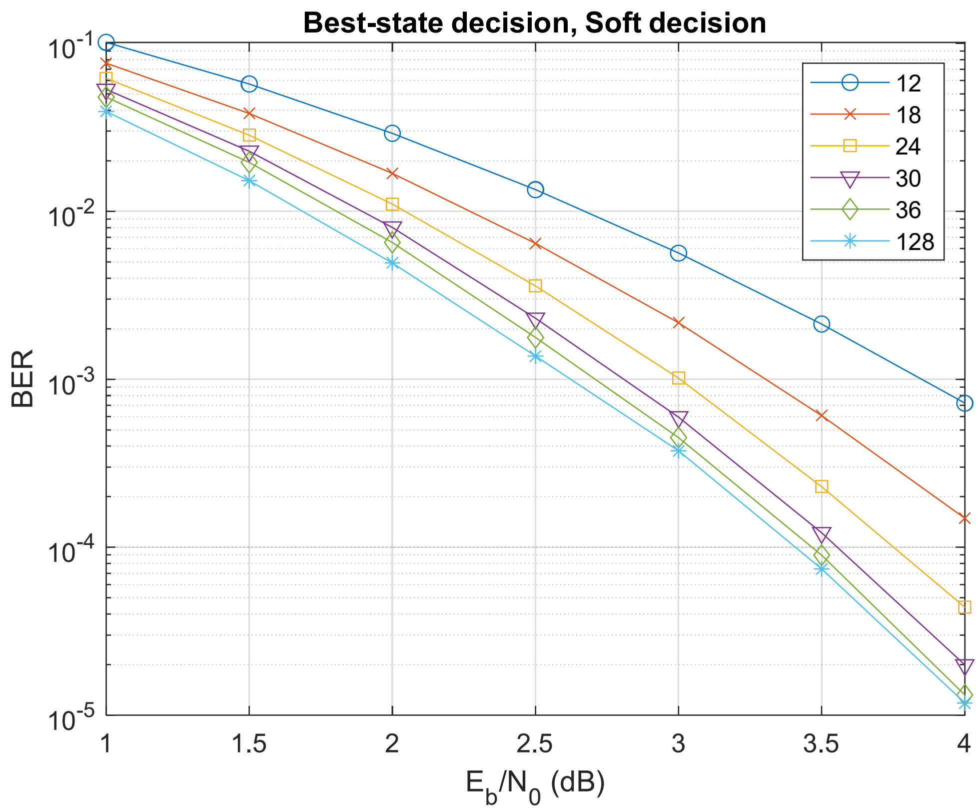


Fig 7.1 Performance of multiples of m in truncation lengths (soft decision)

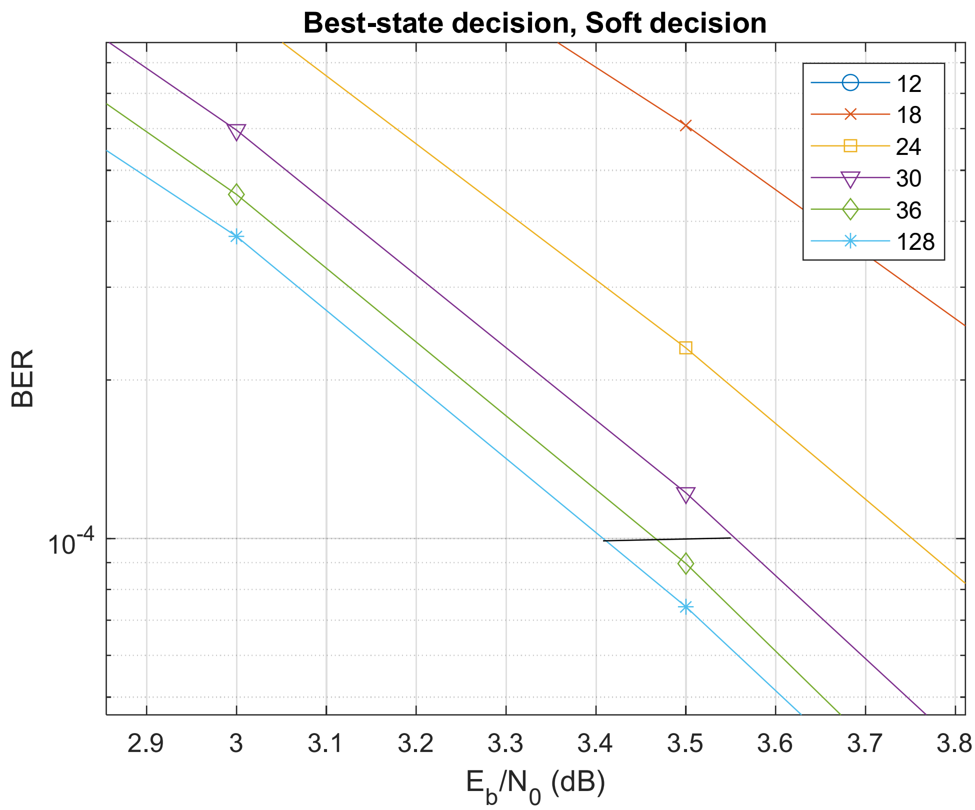


Fig 7.2 Zoom in at BER = 1e-4 in Fig 7.1

As mentioned in the class, 5m is a common choice of the truncation length in practice. However, is it reasonable?

From Fig 6.2, for truncation length = 30 (5m), the degradation is about 0.05 dB at BER = 1e-4 from truncation length = 128, which is good enough to be a reasonable choice for truncation length in hard decision.

From Fig 7.2, for truncation length = 30 (5m), the degradation is about 0.15 dB at BER = 1e-4 from truncation length = 128, which is also good enough to be a reasonable choice for truncation length in soft decision.

Note:
(a). Here we suppose that the relationship between performance and truncation length highly depends on truncation length divided by # of registers in the encoder, m, which is imaginable.
(b).A ~0.1-dB degradation at BER=1e-4 from truncation length = 128 said to be good is something I subjectively defined, which is not a statistical term but may be reasonable to some extent.

5. (Optional) How long should the truncation length be to have best-state and fixed-state decision perform comparably?

We know that for sufficiently long truncation length, the survivor path of each state may end up to the same subpath with high probability, which gives the same decoded bit. But the question of how long it is enough arises.

To answer the question, first we have to be aware that it could always perform better with best-state decision than with fixed-state decision statistically. Therefore, we should define how *performing comparably* means.

Let’s define a ~0.1-dB degradation at BER = 1e-4 from applying best-state decision to fixed-state decision as comparable performance.

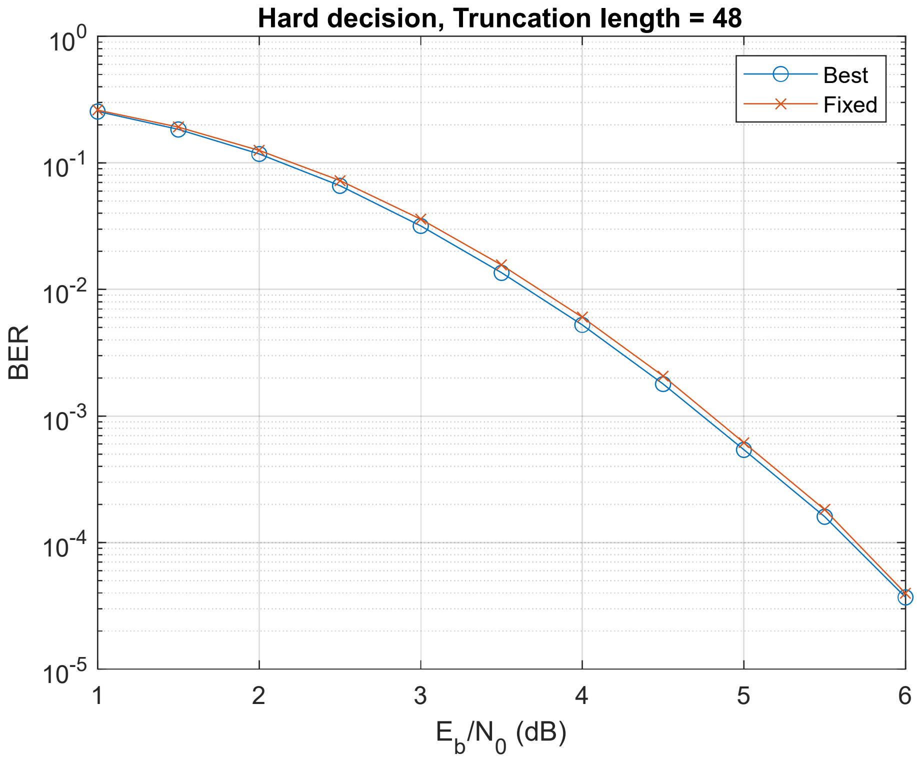


Fig 8.1 Difference between best-state and fixed-state decision (hard decision)

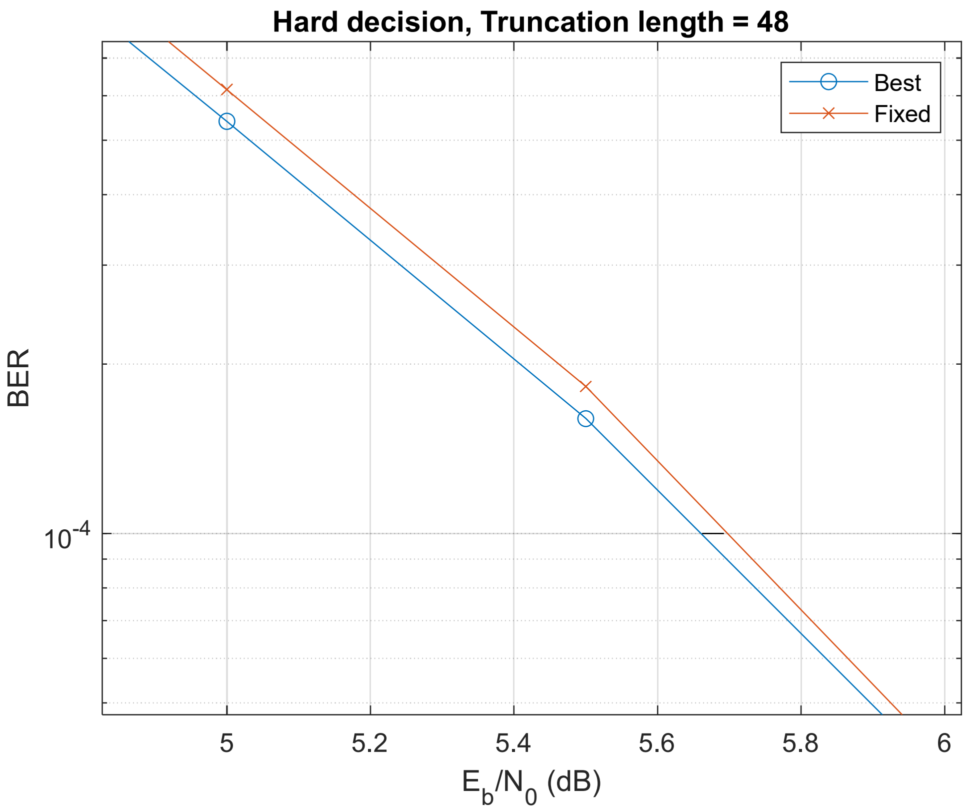


Fig 8.2 Zoom in at BER = 1e-4 in Fig 8.1

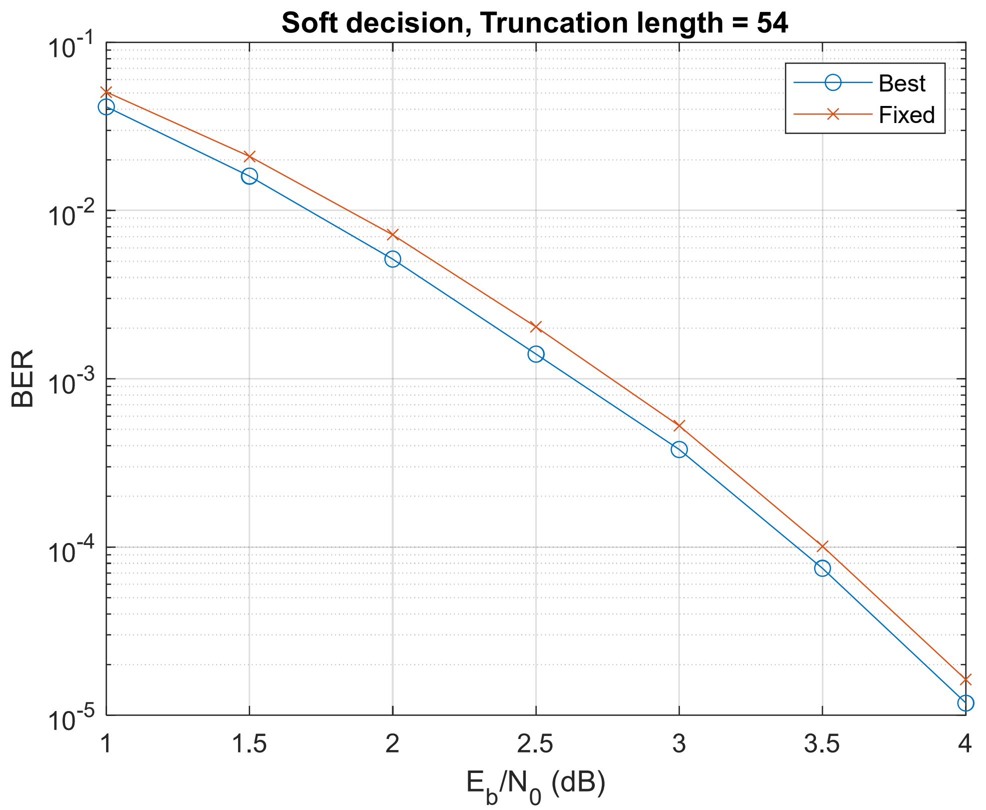


Fig 9.1 Difference between best-state and fixed-state decision (soft decision)

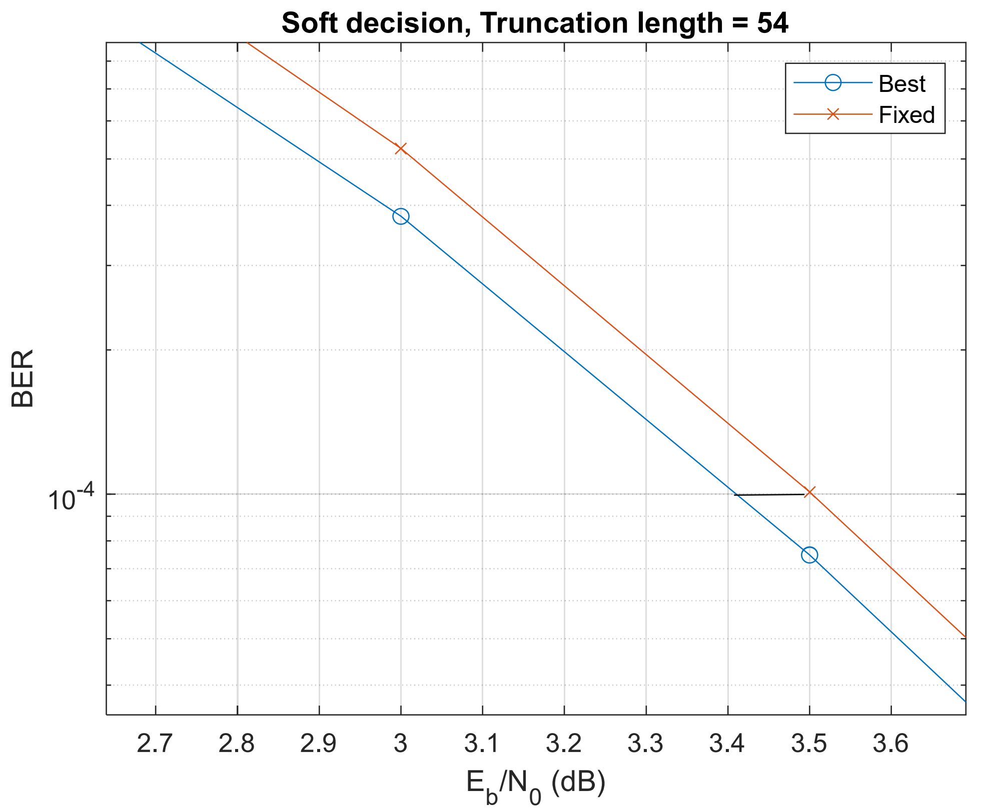


Fig 9.2 Zoom in at BER = 1e-4 in Fig 9.1

From Fig 8.2, truncation length about 48 (8m) is required to provide comparable performance in hard decision.

From Fig 9.2, truncation length about 54 (9m) is required to provide comparable performance in soft decision.

Note: Same as in (4), the “comparable performance” is defined without any statistical evidence but reasonable to some extent, and the SNR in this case may greatly affect the conclusion. That is, the conclusion may only suitable within the range of SNR we care about here.

6. (Optional) Effect of Quantization levels

We can see that with hard decision, there exists a roughly 2-dB loss in error performance from one with unquantized soft decision, while it’s impractical to have such unquantized soft decision. Thus, some experiment to the effect of quantization is required.

Note: Although we call it unquantized soft decision, it’s still quantized according to the data type (e.g. double).

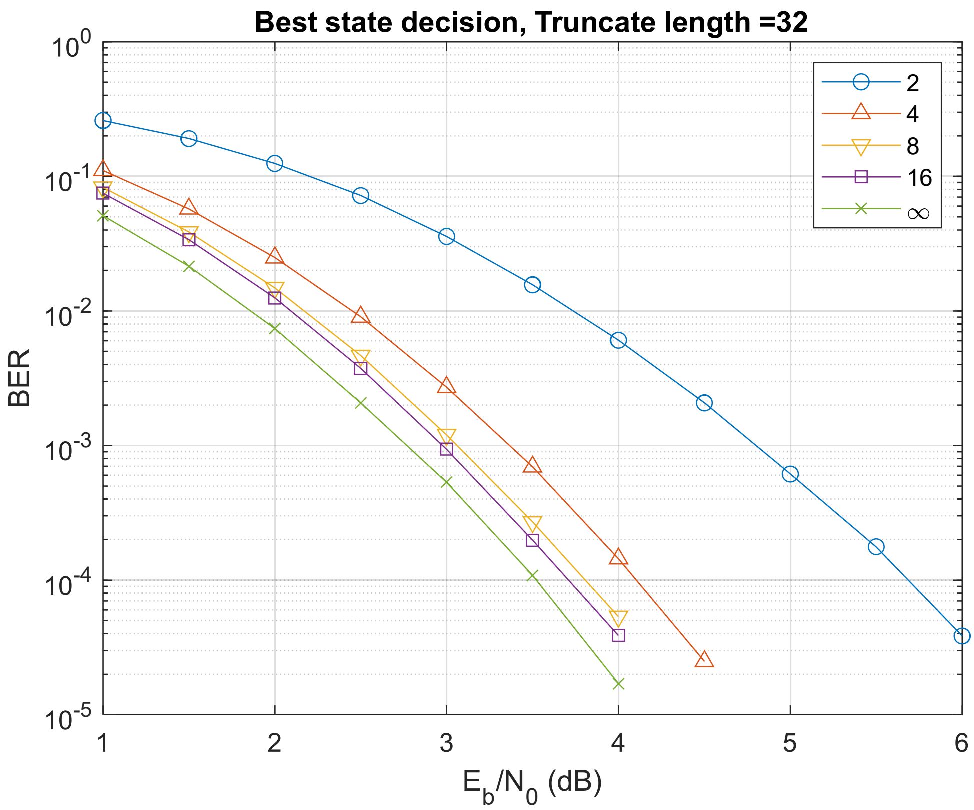


Fig 10. Performance of different quantization level within ± 1

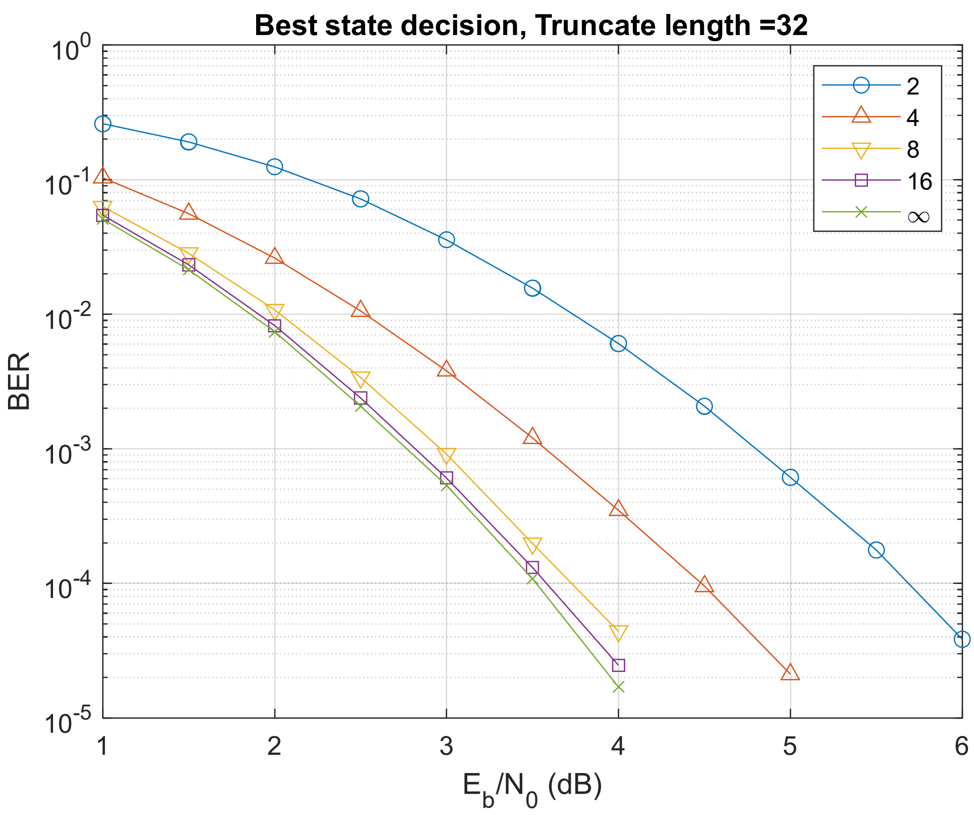
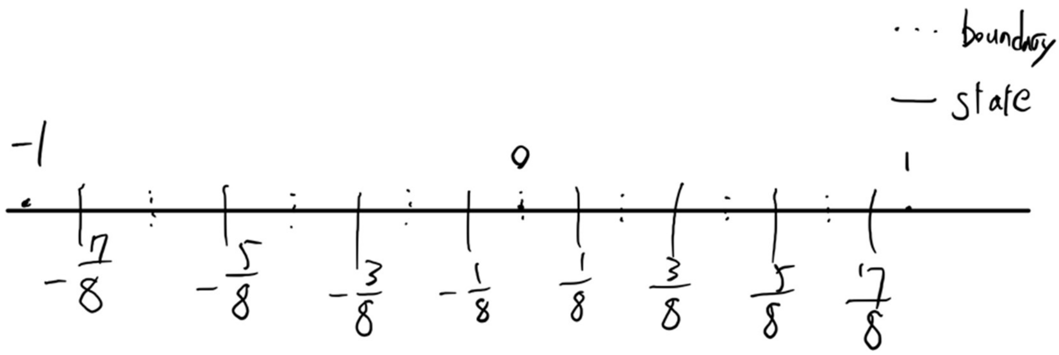


Fig 11. Performance of different quantization level within ± 2

We can obviously see from Fig 10 and Fig 11 that the more quantization levels are, the better the performance is. Next, we can do some further trial on how to quantize.

(a). Quantization within ± 1 (QA)

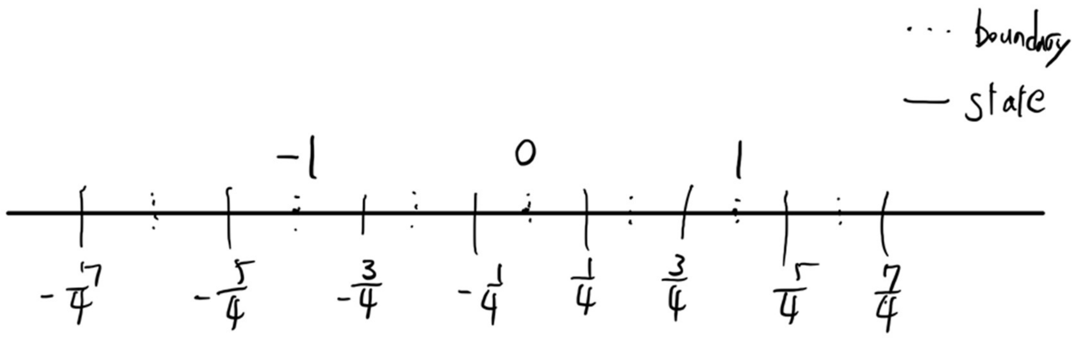
Ex: level 8



It’s a quantization concentrated on the range which is vague to estimate.

(b). Quantization within ± 2 (QB)

Ex: level 8



It's a quantization more average according to the modulation.

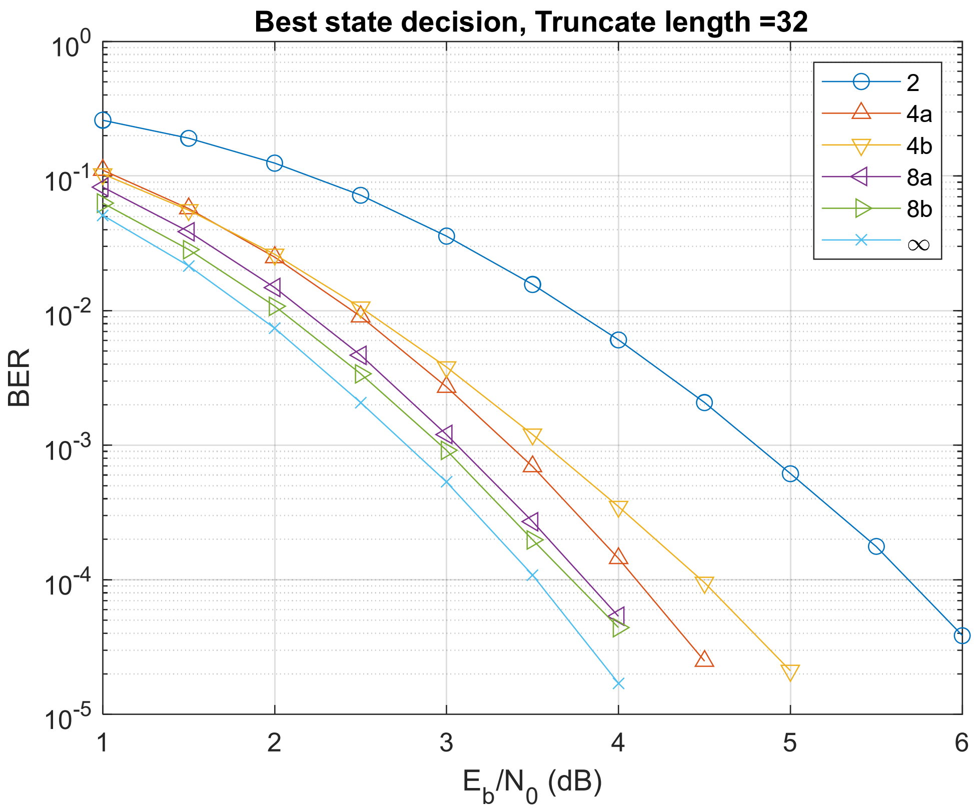


Fig 12. Comparison between 2 ways of quantization

From Fig 12, we can see that for 4-level quantization, QA performs better, while for 8-level (or more level) one, QB performs better. It seems that further concentration on the vague range (close to final decision boundary, 0) is benifitial to the performance. However, lack of information of confidential bits may also degrade the performance. Therefore, a mixture of QA and QB may lead to better performance given certain quantization level.

Note: How I do the quantization here is directly add an A/D converter before module of ACS for unquantized soft decision, which in fact increases the complexity. The way I do so is simply for simulation, which could not reduce the complexity as expected.

```
// Project 1: Convolutional Code

#include <iostream>
#include<math.h>
using namespace std;

class CONV // Convolutional Code
{
    public:
    // n: # of decoded bits; soft=1: soft decision; size: truncate length; bfm: best_fixed_major mode
    CONV(int n, double snr, unsigned long long seed, bool soft, int size, int bfm);
    ~CONV(); // destructor
    void Encode();
        bool PN(); // return 1 bit from pn sequence
    void AWGN(); // add AWGN to encoded 2 bits
        double Ranq1();
        void normal(double n[]); // [n1 n2]
    void ACS(); // Add, Compare, Select -> update metric[64][3] & infohat[64][2][32]
        void ADConvert(double Dlevel); // A/D converter with Dlevel quantization levels
        void SoftACS(); // ACS with soft decision
        void HardACS(); // ACS with hard decision
    void Decode(); // Decode and store into "uhat"
    void ErrorCount(); // count the # of bit errors
    void BER(); // calculate and print BER

    void Print(); // for debugging

    private:
    // Parameters
    int windowsize; // truncate length
    double SNR, sigma; // sigma = sqrt(1/pow(10, SNR/10));
    int N; // total # of bits
    int error = 0; // total # of bit errors

    // For Encode() & AWGN()
    bool pn6[6]; // shift register for PN()
    bool encoder[6] = {0}; // shift register for convolutional encoder
    bool x1, x2; // [x1 x2] = uG
    double y1, y2; // [y1 y2] = [x1+n1 x2+n2]
    bool *info; // storing information bits [32] for comparing while ErrorCount()

    // For Decode() & ACS()
    bool IsSoftDecision = 1;
    int addonly = 0;
    double **metric; // [64][3]: [64 states][m/m(upper branch)+.../m(lower branch)+...]
    unsigned int **metrichard; // hard decsion version
    bool ***infohat; // estimated info[64][2][32]: [64][survivor path/temp survivor path]
    bool uhat; // decoded bit
    int best_fixed_major; // mode: 1:best-state, 2:fixed-state, 3:majority-vote
    double Dlevel = 16;// quantization level in A/D converter

    // For RNG
    unsigned long long SEED=1;
    // SEED must be an unsigned integer smaller than 4101842887655102017.
    unsigned long long RANV;
    int RANI = 0;
};
CONV::CONV(int n, double snr, unsigned long long seed, bool soft, int size, int bfm){
    // Parameters
    N = n; // # of bits
    SNR = snr; // Eb/N0
    sigma = sqrt(1/pow(10, SNR/10));
    SEED = seed; // for RNG
    IsSoftDecision = soft; // 1: soft, 2: hard deecision
    windowsize = size; // truncate length
    best_fixed_major = bfm; // 1: best-state; 2: fixed-state; 3: majority-vote

    // Initial condition for pn register
    pn6[0] = 1; pn6[1] = 0; pn6[2] = 0;
```

```
    pn6[3] = 0; pn6[4] = 0; pn6[5] = 0;

    // New memory & reset
    // info[32]
    info = new bool[windowsize];
    for(int i=0; i<windowsize; i++){
        info[i] = 0;
    }

    // metric[64][3]
    if(IsSoftDecision == 1){
        metric = new double*[64];
        for(int i=0; i<64; i++){
            metric[i] = new double[3];
        }
        for(int i=0; i<64; i++){
            for(int j=0; j<3; j++){
                metric[i][j] = 0;
            }
        }
    }
    else{
        metrichard = new unsigned int*[64];
        for(int i=0; i<64; i++){
            metrichard[i] = new unsigned int[3];
        }
        for(int i=0; i<64; i++){
            for(int j=0; j<3; j++){
                metrichard[i][j] = 0;
            }
        }
    }

    // infohat[64][2][32]
    infohat = new bool**[64];
    for(int i=0; i<64; i++){
        infohat[i] = new bool*[2];
        for(int j=0; j<2; j++){
            infohat[i][j] = new bool[windowsize];
        }
    }
    for(int i=0; i<64; i++){
        for(int j=0; j<2; j++){
            for(int k=0; k<windowsize; k++){
                infohat[i][j][k] = 0;
            }
        }
    }
}
CONV::~CONV(){
    delete[] info;
    if(IsSoftDecision == 1){
        for(int i=0; i<64; i++){
            delete[] metric[i];
        }
        delete[] metric;
    }
    else{
        for(int i=0; i<64; i++){
            delete[] metrichard[i];
        }
        delete[] metrichard;
    }
    for(int i=0; i<64; i++){
        for(int j=0; j<2; j++){
            delete[] infohat[i][j];
        }
        delete[] infohat[i];
    }
    delete[] infohat;
}
void CONV::Encode(){
    bool u = PN(); // information bit from pn sequence
```

```
    // Update/ right shift info[] for comparing in ErrorCount()
    for(int i=windowsize - 2; i>=0; i--){
        info[i+1] = info[i];
    }
    info[0] = u;

    // [x1 x2] = uG
    x1 = u ^ encoder[1] ^ encoder[2] ^ encoder[4] ^ encoder[5];
    x2 = u ^ encoder[0] ^ encoder[1] ^ encoder[2] ^ encoder[5];
    // Go to next state
    for(int i=5; i>0; i--){
        encoder[i] = encoder[i-1];
    }
    encoder[0] = u;
}
bool CONV::PN(){
    // Generate PN sequence
    bool u0 = pn6[0];
    bool u1 = pn6[1];
    for(int i=0; i<5; i++){
        pn6[i] = pn6[i+1];
    }
    pn6[5] = u0 ^ u1;

    return u0;
}
void CONV::AWGN(){
    // Add WGN to x1, x2
    // [y1 y2] = [x1+n1 x2+n2], BPSK
    double n[2]; // [n1 n2]
    normal(n); // Gaussian RNG
    if(x1 == 0){
        y1 = 1 + n[0];
    }
    else{
        y1 = -1 + n[0];
    }
    if(x2 == 0){
        y2 = 1 + n[1];
    }
    else{
        y2 = -1 + n[1];
    }
}
double CONV::Ranq1(){
    if ( RANI == 0 ){
        RANV = SEED ^ 4101842887655102017LL;
        RANV ^= RANV >> 21;
        RANV ^= RANV << 35;
        RANV ^= RANV >> 4;
        RANV = RANV * 2685821657736338717LL;
        RANI++;
    }
    RANV ^= RANV >> 21;
    RANV ^= RANV << 35;
    RANV ^= RANV >> 4;
    return RANV * 2685821657736338717LL * 5.42101086242752217E-20;
}
void CONV::normal(double n[]){
    double x1, x2, s;
    do{
        x1 = Ranq1();
        x2 = Ranq1();
        x1 = 2*x1 - 1;
        x2 = 2*x2 - 1;
        s = pow(x1,2) + pow(x2,2);
    } while(s>=1.0);
    n[0] = sigma*x1*sqrt(-2*log(s)/s);
    n[1] = sigma*x2*sqrt(-2*log(s)/s);
}
void CONV::ACS(){
    if(IsSoftDecision ==1){
```

```
        //ADConvert(Dlevel);
        SoftACS();
    }
    else{
        HardACS();
    }
}
void CONV::ADConvert(double Dlevel){
/*  // Quantize within +-1
    for(double i=-(Dlevel-2)/Dlevel; i<1; i+=2/Dlevel){
        if(y1 <= i){
            y1 = i - 1/Dlevel;
            break;
        }
    }
    if(y1 > (Dlevel-2)/Dlevel){
        y1 = (Dlevel-1)/Dlevel;
    }
    for(double i=-(Dlevel-2)/Dlevel; i<1; i+=2/Dlevel){
        if(y2 <= i){
            y2 = i - 1/Dlevel;
            break;
        }
    }
    if(y2 > (Dlevel-2)/Dlevel){
        y2 = (Dlevel-1)/Dlevel;
    }
*/
    // Quantize within +-2
    for(double i=-(Dlevel-2)/(Dlevel/2); i<=(Dlevel-2)/(Dlevel/2); i+=4/Dlevel){
        if(y1 <= i){
            y1 = i - 2/Dlevel;
            break;
        }
    }
    if(y1 > (Dlevel-2)/(Dlevel/2)){
        y1 = (Dlevel-1)/(Dlevel/2);
    }
    for(double i=-(Dlevel-2)/(Dlevel/2); i<=(Dlevel-2)/(Dlevel/2); i+=4/Dlevel){
        if(y2 <= i){
            y2 = i - 2/Dlevel;
            break;
        }
    }
    if(y2 > (Dlevel-2)/(Dlevel/2)){
        y2 = (Dlevel-1)/(Dlevel/2);
    }

}
void CONV::SoftACS(){
    // Metric[][] & Infohat[][][] updating
    int next_state; // index of next state
    int temp1, temp2; // x1, x2 for each state with u = 0
    double ymetric1, ymetric2; // modulated temp1, temp2 * y1, y2
    int states_to_add; // # of states need Add only

    // ACS without Compare & Select
    if(addonly < 6){ // first 6 iterations need Add only
        states_to_add = pow(2, addonly);
        for(int i=0; i<states_to_add; i++){
            // Outputs with input 0 and then modulation
            temp1 = (      i%4/2 + i%8/4 +   i%32/16 + i/32) % 2;
            temp2 = (i%2 + i%4/2 + i%8/4 +             i/32) % 2;
            if(temp1 == 0){
                ymetric1 = y1;
            }
            else{
                ymetric1 = -y1;
            }
            if(temp2 == 0){
                ymetric2 = y2;
            }
            else{
```

```
                ymetric2 = -y2;
            }

            // Metric update (add only) & infohat (info sequence) update
            next_state = 2*i; // even next state with input 0
            metric[next_state][1] = metric[i][0] - ymetric1 - ymetric2;
            for(int k=windowsize-1; k>0; k--){
                infohat[next_state][1][k] = infohat[i][0][k-1];
            }
            infohat[next_state][1][0] = 0;

            next_state = 2*i + 1; // odd next state with input 1
            metric[next_state][1] = metric[i][0] + ymetric1 + ymetric2;
            for(int k=windowsize-1; k>0; k--){
                infohat[next_state][1][k] = infohat[i][0][k-1];
            }
            infohat[next_state][1][0] = 1;
        }
        // Store the update back to metric[][0] & infohat[][0]
        for(int i=0; i<2*states_to_add; i++){
            metric[i][0] = metric[i][1];
            for(int k=0;k<windowsize;k++){
                infohat[i][0][k] = infohat[i][1][k];
            }
        }
        addonly++; // if(addonly < 6), Add only
    }
    else{
        // ACS
        for(int i=0; i<32; i++){
            temp1 = (      i%4/2 + i%8/4 +   i%32/16 + i/32) % 2;
            temp2 = (i%2 + i%4/2 + i%8/4 +             i/32) % 2;
            if(temp1 == 0){
                ymetric1 = y1;
            }
            else{
                ymetric1 = -y1;
            }
            if(temp2 == 0){
                ymetric2 = y2;
            }
            else{
                ymetric2 = -y2;
            }
            // Metric addition
            next_state = 2*i;
            metric[next_state][1] = metric[i][0] - ymetric1 - ymetric2;
            metric[next_state][2] = metric[i+32][0] + ymetric1 + ymetric2;

            next_state = 2*i + 1;
            metric[next_state][1] = metric[i][0] + ymetric1 + ymetric2;
            metric[next_state][2] = metric[i+32][0] - ymetric1 - ymetric2;
        }
        // Compare & Select & infohat update
        for(int i=0; i<64; i++){
            if(metric[i][1] <= metric[i][2]){
                metric[i][0] = metric[i][1];
                for(int k=0; k<windowsize-1; k++){
                    infohat[i][1][k+1] = infohat[i/2][0][k];
                }
                infohat[i][1][0] = i%2;
            }
            else{
                metric[i][0] = metric[i][2];
                for(int k=0; k<windowsize-1; k++){
                    infohat[i][1][k+1] = infohat[i/2 + 32][0][k];
                }
                infohat[i][1][0] = i%2;
            }
        }
        // Complete the update of infohat[][][]
        for(int i=0; i<64; i++){
            for(int k=0;k<windowsize;k++){
```

```
                infohat[i][0][k] = infohat[i][1][k];
            }
        }
    }
}
void CONV::HardACS(){
    // Hard decision version of ACS
    bool y1hard, y2hard;
    if(y1 >= 0){
        y1hard = 0;
    }
    else{
        y1hard = 1;
    }
    if(y2 >= 0){
        y2hard = 0;
    }
    else{
        y2hard = 1;
    }
    // metrichard[][] & Infohat[][][] updating
    int next_state; // index of next state
    bool temp1, temp2; // output of the encoder x1, x2
    int states_to_add; // # of states need Add only

    // ACS without Compare & Select
    if(addonly < 6){ // first m=6 iterations need Add only
        states_to_add = pow(2, addonly);
        for(int i=0; i<states_to_add; i++){
            // Outputs with input 0 and then modulation
            temp1 = (      i%4/2 + i%8/4 +   i%32/16 + i/32) % 2;
            temp2 = (i%2 + i%4/2 + i%8/4 +             i/32) % 2;
            // Metric update (add only) & infohat (info sequence) update
            next_state = 2*i + 1; // odd next state with input 1
            metrichard[next_state][1] = metrichard[i][0] + (!temp1 ^ y1hard) + (!temp2 ^ y2hard);
            for(int k=windowsize-1; k>0; k--){
                infohat[next_state][1][k] = infohat[i][0][k-1];
            }
            infohat[next_state][1][0] = 1;

            next_state = 2*i; // even next state with input 0
            metrichard[next_state][1] = metrichard[i][0] + (temp1 ^ y1hard) + (temp2 ^ y2hard);
            for(int k=windowsize-1; k>0; k--){
                infohat[next_state][1][k] = infohat[i][0][k-1];
            }
            infohat[next_state][1][0] = 0;
        }
        for(int i=0; i<2*states_to_add; i++){
            metrichard[i][0] = metrichard[i][1];
            for(int k=0;k<windowsize;k++){
                infohat[i][0][k] = infohat[i][1][k];
            }
        }
        addonly++;
    }
    else{
        for(int i=0; i<32; i++){
            temp1 = (      i%4/2 + i%8/4 +   i%32/16 + i/32) % 2;
            temp2 = (i%2 + i%4/2 + i%8/4 +             i/32) % 2;

            next_state = 2*i;
            metrichard[next_state][1] = metrichard[i][0] + (temp1 ^ y1hard) + (temp2 ^ y2hard);
            metrichard[next_state][2] = metrichard[i+32][0] + (!temp1 ^ y1hard) + (!temp2 ^ y2hard);

            next_state = 2*i + 1;
            metrichard[next_state][1] = metrichard[i][0] + (!temp1 ^ y1hard) + (!temp2 ^ y2hard);
            metrichard[next_state][2] = metrichard[i+32][0] + (temp1 ^ y1hard) + (temp2 ^ y2hard);
```

```
        }
        for(int i=0; i<64; i++){
            if(metrichard[i][1] <= metrichard[i][2]){
                metrichard[i][0] = metrichard[i][1];
                for(int k=0; k<windowsize-1; k++){
                    infohat[i][1][k+1] = infohat[i/2][0][k];
                }
                infohat[i][1][0] = i%2;
            }
            else{
                metrichard[i][0] = metrichard[i][2];
                for(int k=0; k<windowsize-1; k++){
                    infohat[i][1][k+1] = infohat[i/2 + 32][0][k];
                }
                infohat[i][1][0] = i%2;
            }
        }
        for(int i=0; i<64; i++){
            for(int k=0;k<windowsize;k++){
                infohat[i][0][k] = infohat[i][1][k];
            }
        }
    }
}
void CONV::Decode(){
    // Hard/Soft decision
    ACS();
    if(best_fixed_major == 1){
        int best_state = 0; // index of the best state
        if(IsSoftDecision == 1){
            double best_metric = metric[0][0];
            for(int i=1; i<64; i++){
                if(metric[i][0] < best_metric){
                    best_state = i;
                    best_metric = metric[i][0];
                }
            }
            uhat = infohat[best_state][0][windowsize - 1];
        }
        else{
            unsigned int best_metric = metrichard[0][0];
            for(int i=1; i<64; i++){
                if(metrichard[i][0] < best_metric){
                    best_state = i;
                    best_metric = metrichard[i][0];
                }
            }
            uhat = infohat[best_state][0][windowsize - 1];
        }
    }
    else if(best_fixed_major == 2){
        uhat = infohat[0][0][windowsize - 1];
    }
    else{
        int num_zero = 0;
        for(int i=0; i<64; i++){
            if(infohat[i][0][windowsize - 1] == 0){
                num_zero++;
            }
        }
        if(num_zero >= 32){
            uhat = 0;
        }
        else{
            uhat = 1;
        }
    }
}
void CONV::ErrorCount(){
    // Count the # of errors into "error"
    if(uhat != info[windowsize - 1]){
        error++;
    }
```

```
}
void CONV::BER(){
    cout << "SNR (dB) = " << SNR << endl;
    cout << "N = " << N << ", " << "K = " << error << endl;
    cout << "BER = " << static_cast<double>(error)/N << endl << endl;

    error = 0;
}
void CONV::Print(){
    cout<< error <<" ";
}

int main()
{
    // Parameters setting
    int N = 10000000;
    double SNR = 6;
    unsigned long long SEED = 1;
    bool IsSoftDecision = 0;
    int windowsize = 32;
    int best_fixed_major = 1; // 1: best, 2: fixed, 3: major
/*
    // For demo
    cout << "The number of decoded bits N: ";
    cin >> N;
    cout << "The bit signal-to-noise ratio (SNR) Eb/N0 (in dB): ";
    cin >> SNR;
    cout << "the seed for the random number generator: ";
    cin >> SEED;
    cout << "Soft decision? (1 for Yes): ";
    cin >> IsSoftDecision;
*/
//for(SNR = 1; SNR<=6; SNR += 0.5){
    CONV A(N, SNR, SEED, IsSoftDecision, windowsize, best_fixed_major);

    for(int i=0; i<windowsize-1; i++){
        A.Encode();
        A.AWGN();
        A.ACS();
    }
    for(int i=0; i<N; i++){
        A.Encode();
        A.AWGN();
        A.Decode();
        A.ErrorCount();
    }

    A.BER();
//A.Print();
//}

    return 0;
}
```