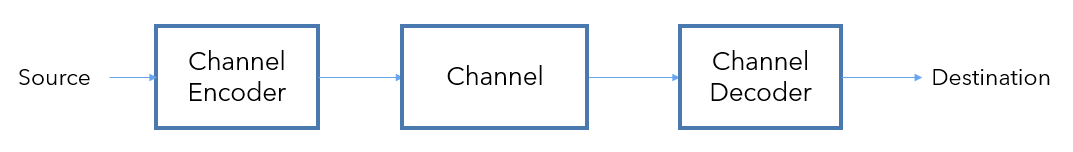
**Convolutional Code Decoder 109061217林峻霆**

1. **System Design**

本次模擬的系統流程圖如下：

****

* **Encoder**

Encoder的設計如下圖所示，我們需要一個6-bit shift register

**一張含有 圖表 的圖片

自動產生的描述**

另外，在將x = [x1, x2]輸入channel前需要先經過調變，本次模擬採用的是BPSK，也就是。

* Channel：AWGN channel
* Decoder

在Decoder端我們要先進行解調，可以粗略分為兩種：Hard decision和Soft decision。Hard Decision就是根據收到的訊號進行分類，若大於0則歸類為1，小於0則歸類為-1，在傳入Viterbi Decoder；Soft decision則直接將收到的傳入Viterbi Decoder，接著就按照Viterbi Algorithm去進行Decoding。要注意的是，為了防止overflow，我在每一輪更新每個state的metric後，計算所有state的metric平均值，並將其減去。

1. **Result**

以下考慮在不同狀況下，進行解碼成效的比較：

1. **Best State**

* **Soft Decision**



* Hard Decision





比較Hard decision與Soft decision，兩者在達到BER = 10-5時所需的SNR約有2dB差距。

1. **Best State vs. Fixed State vs. Majority Vote**

* **Best State**



* **Fixed State (always pick the all zero state)**



* **Majority Vote**



比較三者，可以發現三者的performance相比Best State > Majority Vote > Fixed State。另外在採用Best State和Majority Vote時，Hard Decision都比Soft Decision高2dB；在Fixed State時，Soft Decision與Hard Decision僅有1dB的差距。

1. **Truncated Bits**

* **8 bits**

****

* **16 bits**

****

* **32 bits**



* **64 bits**

****

* **128 bits**

****

理論上來說我們應該等到整個bit stream都傳完才進行output，但在現實應用上，transmitter可能會不斷傳送訊號過來，因此實際上我們會對收到的資訊做truncated。Truncated length也會影響到解碼的成效，觀察上述幾張圖可發現，當truncation length越長，解碼的成效越好，8-bit與32-bit truncation length有約2dB的差異。另外，也可以拉線，當truncation length持續上升，performance的優化幅度越小，32-bit、64-bit以及128-bit的performance近乎沒有區別。

1. **Program**

#include <iostream>

#include <iomanip>

#include <bitset>

#include <cmath>

#include <limits>

#include <float.h>

#include <stdlib.h>

#include <vector>

#include <fstream>

#define TR\_LEN 32

#define INFO\_PERIOD 63

using namespace std;

const long long int para\_1 = 4101842887655102017LL;

const long long int para\_2 = 2685821657736338717LL;

const double para\_3 = 5.42101086242752217E-20;

unsigned long long int SEED = 14;

unsigned long long int RANV;

int RANI = 0;

double Ranq1(){

    if(RANI == 0){

        RANV = SEED ^ para\_1;

        RANV ^= RANV >> 21;

        RANV ^= RANV << 35;

        RANV ^= RANV >> 4;

        RANV \*= para\_2;

        RANI++;

    }

    RANV ^= RANV >> 21;

    RANV ^= RANV << 35;

    RANV ^= RANV >> 4;

    return RANV \* para\_2 \* para\_3;

}

void Normal(double& n1, double& n2, double std\_dev){

    double x1, x2, s;

    do{

        x1 = Ranq1();

        x2 = Ranq1();

        x1 = 2 \* x1 - 1;

        x2 = 2 \* x2 - 1;

        s = x1 \* x1 + x2 \* x2;

    } while (s >= 1.0);

    n1 = std\_dev \* x1 \* sqrt(-2 \* log(s) / s);

    n2 = std\_dev \* x2 \* sqrt(-2 \* log(s) / s);

}

// each node represent a state, storing the current information and previous information

typedef struct Node{

    double prev\_metric = DBL\_MAX;               // metric in previous state

    double cur\_metric = DBL\_MAX;                // metric in current state

    int prev\_node = 100000;                     // index of previous node

    vector<vector<int>> next;                   // next[0][0] = index of next state when u = 0

                                                // next[1][0] = index of next state when u = 1

    bitset<TR\_LEN> pre\_path {0}; // each bit represent u that was sent previously, in previous state

    bitset<TR\_LEN> path {0};                   // each bit represent u that was sent previously, in current state

} node;

double SNR[20] = {1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8};

int main(void){

    int ERROR = 0;                              // # of bit error

    int u, tmp;                                 // temporary storage

    int m1, m2;                                 // [m1, m2] = uG

    int num\_of\_0, num\_of\_1;                     // # of 0 and 1 while we using the Majority Vote

    int truncated\_len = 0;

    double x1, x2;                              // Signal point after BPSK and adding noise

    double STD\_DEV;                             // standard deviation correspond to SNR to generate Normal r.v

    double bias;                                // the average of the metric to prevent overflow

    int index = 0;

    long long int information\_index = 0;       // # of transmitted information bit

    double n1, n2;                              // Noise

    double M1, M2, M;

    ofstream outFile;

    node\* trellis = new node[64];

    vector<int> information;                    // information bits

    vector<int> state;                          // current state

    // [m1, m2] = uG

    // m1 = u + s[2] + s[3] + s[5] + s[6]

    // m2 = u + s[1] + s[2] + s[3] + s[6]

    for(int i = 0; i < 64; i++){

        m1 = 0;

        m2 = 0;

        if(i & 1) m2 = m2 + 1;

        if(i & 2) {

            m1 = m1 + 1;

            m2 = m2 + 1;

        }

        if(i & 4) {

            m1 = m1 + 1;

            m2 = m2 + 1;

        }

        if(i & 16) m1 += 1;

        if(i & 32) {

            m1 = m1 + 1;

            m2 = m2 + 1;

        }

        index = i \* 2;

        if(index >= 64) index -= 64;

        trellis[i].next.push\_back({index, m1 % 2, m2 % 2});

        trellis[i].next.push\_back({index + 1, (m1 + 1) % 2, (m2 + 1) % 2});

    }

    state.assign(7, 0);

    information.assign(63, 0);

    // generate information

    // u[l + 6] = u[l + 1] + u[l] for l >= 0. The period of the sequence is 63

    information[0] = 1;

    for(int i = 6; i < INFO\_PERIOD; i++) information[i] = information[i - 6] ^ information[i - 5];

    // Start Testing for each SNR

    for(int testcase = 0; testcase < 15; testcase++){

        STD\_DEV = sqrt(pow(10, -SNR[testcase] / 10));     // Calculate standard deviation

        ERROR = 0;                         // Initialize # of error

        information\_index = 0;           // Initialize # of transmitted bit

        for(int i = 0; i < 64; i++){                          // Initialize each state node

            trellis[i].prev\_metric = DBL\_MAX;

            trellis[i].cur\_metric = DBL\_MAX;

            trellis[i].path = 0;

            trellis[i].pre\_path = 0;

        }

        trellis[0].prev\_metric = 0;                              // Start from all zero state

        cout << "SNR = " << SNR[testcase] << "dB\n";

        while(ERROR < 1000){

            u = information[information\_index % INFO\_PERIOD];   // Transmit an information bit

            m1 = (u + state[2] + state[3] + state[5] + state[6]) % 2;       // Encode

            m2 = (u + state[1] + state[2] + state[3] + state[6]) % 2;

            state[6] = state[5];                                             // Update State

            state[5] = state[4];

            state[4] = state[3];

            state[3] = state[2];

            state[2] = state[1];

            state[1] = u;

            Normal(n1, n2, STD\_DEV);                              // Perform BPSK and add noise

            x1 = -2 \* m1 + 1 + n1;

            x2 = -2 \* m2 + 1 + n2;

// Receiving the Signal with Demodulation (Soft decision / Hard Decision)

            /\*if(x1 >= 0) x1 = 1;

            else x1 = -1;

            if(x2 >= 0) x2 = 1;

            else x2 = -1;\*/

            // Updating the information in trellis diagram

            for(int i = 0; i < 64; i++){

// Check whether a state node is already reached

                if(trellis[i].prev\_metric == DBL\_MAX) continue;

                // Calculate the metric send from the previous state node while u = 0

                index = trellis[i].next[0][0];

                if(trellis[i].next[0][1] == 0) M1 = (x1 - 1) \* (x1 - 1);

                else M1 = (x1 + 1) \* (x1 + 1);

                if(trellis[i].next[0][2] == 0) M2 = (x2 - 1) \* (x2 - 1);

                else M2 = (x2 + 1) \* (x2 + 1);

                M = trellis[i].prev\_metric + M1 + M2;

                if(trellis[index].cur\_metric > M){    // Updating information in the state node

                    trellis[index].cur\_metric = M;

                    trellis[index].prev\_node = i;

                }

                // Calculate the metric send from the previous state node while u = 0

                index = trellis[i].next[1][0];

                if(trellis[i].next[1][1] == 0) M1 = (x1 - 1) \* (x1 - 1);

                else M1 = (x1 + 1) \* (x1 + 1);

                if(trellis[i].next[1][2] == 0) M2 = (x2 - 1) \* (x2 - 1);

                else M2 = (x2 + 1) \* (x2 + 1);

                M = trellis[i].prev\_metric + M1 + M2;

                if(trellis[index].cur\_metric > M){    // Updating information in the state node

                    trellis[index].cur\_metric = M;

                    trellis[index].prev\_node = i;

                }

            }

            bias = 0;          // Initialize bias

            tmp = 0;           // Initialize # of reached state node

            for(int i = 0; i < 64; i++){

                if(trellis[i].cur\_metric == DBL\_MAX) // check whether the state node is reached

continue;

                tmp++;

                trellis[i].prev\_metric = trellis[i].cur\_metric;   // Update Metric

                bias += trellis[i].cur\_metric;

                trellis[i].cur\_metric = DBL\_MAX;

                index = trellis[i].prev\_node;                     // Update the path

                if(i == trellis[index].next[0][0])

trellis[i].path = trellis[index].pre\_path << 1;

                else if(i == trellis[index].next[1][0]) {

                    trellis[i].path = trellis[index].pre\_path << 1;

                    trellis[i].path[0] = 1;

                }

                else cout << "ERROR!\n";

            }

// remove the dc term(average) of metric in each state node

            for(int i = 0; i < 64; i++) trellis[i].pre\_path = trellis[i].path;

            bias = bias / tmp;

            for(int i = 0; i < 64; i++) {

                if(trellis[i].prev\_metric == DBL\_MAX) continu;

                trellis[i].prev\_metric -= bias;

            }

            if(information\_index >= TR\_LEN - 1){

                // Best state

                index = 0;

// Find the best state result

                double best\_metric = trellis[0].prev\_metric;

                for(int i = 1; i < 64; i++){

                    if(trellis[i].prev\_metric < best\_metric){

                        index = i;

                        best\_metric = trellis[i].prev\_metric;

                    }

                }

                // Check if there's error

                if(trellis[index].path[TR\_LEN - 1] != information[(information\_index - TR\_LEN + 1) % INFO\_PERIOD]) ERROR++;

                // Fixed State

                //if(trellis[0].path[TR\_LEN - 1] != information[(information\_index - TR\_LEN + 1) % INFO\_PERIOD]) ERROR++;

                // Majority vote

                /\*num\_of\_0 = 0;

                num\_of\_1 = 1;

                for(int i = 0; i < 64; i++){

                    if(trellis[i].path[31] == 0) num\_of\_0++;

                    else num\_of\_1++;

                }

                if(num\_of\_0 >= num\_of\_1 && information[(information\_index - 31) % INFO\_PERIOD] == 1) ERROR++;

                else if(num\_of\_0 < num\_of\_1 && information[(information\_index - 31) % INFO\_PERIOD] == 0) ERROR++;\*/

            }

            information\_index = (information\_index + 1);

        }

        cout << "ERROR = " << ERROR << " Number of bits = " << information\_index - 30 << endl;

        cout << "BER = " << ERROR \* 1.0 / (information\_index - 30) << endl;

    }

    return 0;

}