C.1 Introduction

In Appendix C, where we deal with the computer programs used in Chapter 6, we are going to see the computer program of AMI and from it we will understand the rest of the programs for all the line codes used in Chapter 6.

In Chapter 6, as mentioned before, we investigated the 2 dB gain for all ternary line codes. The following program listing of AMI will explain more closely the idea of solving the problem of distance metric, and the steps used for the calculation of the Viterbi decoding. More details will be explained throughout the program listing.
In Appendix C, we will see the two computer programs of AMI for Hard and Soft decision besides the computer program of the calculation of the error propagation for the different line codes.

### C.2 Program listing

#### 2.1 Hard- and Soft-decision decoding for AMI

*Simulation of AMI, using Hard-decision decoding*

```c
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <limits.h>

FILE *fp;

int inbit,          /* Input bit */
    i,p,a1,          /* Parameters */
    ht,              /* Current state */
    vt,              /* Next state */
    code,            /* Encoder output */
    decoder,         /* Decoder output */
    snr,             /* SNR of channel in dB */
    logic,           /* Result of logic operation */
    sp0,             /* Survivor path for state 0 */
    sp1,             /* Survivor path for state 1 */
    tsp0,            /* temporary survivor path for state 0 */
    tsp1,            /* temporary survivor path for state 1 */
    error,           /* Amount of decoder errors */
    counter,         /* Number of bits streamed in the program */
    z = 10;          /* Viterbi decoder latency which is equal to 5k */

float ber,
    m[2],sm[2],    /* State metrics */
    tm[2],         /* Temporary state metrics */
    x[2],          /* Output of the metric matrix */
    s10, s11, s12,
    s20, s21, s22, /* Variables for random number generator */
    noisycode;     /* Channel output */

int inset[10];     /* Stores last 5k input bits */

int cs[2][2] = {{0, 0}, {2, 1}},    /* Code-symbol matrix */
    ns[2][2] = {{0, 1}, {1, 0}};    /* Next-state matrix */
```
int M[3][3] = {{0,3,3},
              {3,0,6},
              {3,6,0}};

double MRG32k3a();/**function generating a random number between 0 and 1**
int bits();();/**function generating a random binary data**
void skuiflinks();();/**function for shifting data to the right**
void viterbi();();/**Viterbi-decoding function**
void inisialiseer();();/**function initializing the parameters**
void Gauss(double G[2]);();/**Gaussian noise generator**
void Gauss_noise(double G[2], double No);

main() {
    counter = 0;
    z=10;
    for ( snr = 1; snr <= 20; snr++) {
        inisialiseer();/** Initialization of some variables
        srand(time(0));
        s10 = rand(); s11 = rand(); s12 = rand();
        s20 = rand(); s21 = rand(); s22 = rand();

        snr = pow(10, ((snr)/10.0));
        No = 4.5/snr;
        do {
            inbit = bits(); /** Generation of the input data
            if (counter < z)
                inset[counter] = inbit;**storing the data in the shift
            register
            else {
                skuiflinks(); /**shifting data out to right.
                inset[z-1] = inbit;
            }
            vt = ns[inbit][ht]; /** Determine next state
            code = cs[inbit][ht];  /** Encode input bit

            switch(code) {
                case 0 :
                    code = 0;
                    break;
                case 1 :
                    code = -3;
                    break;
                case 2 :
                    code = 3;
            }

        
    }
break;
}
Gauss_noise(G, No);**generation of the Gaussian noise
noisycode = code + G[1];

    if (noisycode > 1.63)
        code = 2;
    else
        if ((noisycode < 1.63) && (noisycode > -1.63))
            code = 0;
        else
            if (noisycode < -1.63)
                code = 1;

viterbi();
    if (counter >= (z-1)) {   ** Start decoding
        if (sm[0] <= sm[1]) {
            logic= sp0 & 1;
            if (logic == 0)
                decoder = 0;
            else
                decoder = 1;
        } else {
            logic = sp1 & 1;
            if (logic == 0)
                decoder = 0;
            else
                decoder = 1;
        }

        if (decoder != inset[0])  ** Count decoder errors
            error += 1;
    }

    counter += 1;** increment the counter
    ht = vt;     ** Goto next state
}
while (counter <1000);
ber=error/(counter+0.0);** calculation of the BER.

fp=fopen("c:\windows\desktop\amihard.c","a");
fprintf(fp," %2.3f  \n",ber);
fclose(fp);

counter = 0;   ** Reset variables
error   = 0;
ber=0;
}
}
double MRG32k3a() {** Generate a random number between 0 and 1
    long k;
    double p;
** Component 1
\[ p = a_{12}s_{11} - a_{13}n*s_{10}; \]
\[ k = \frac{p}{m_1}; \quad p = k*m_1; \]
\[ \text{if } (p < 0.0) \quad p += m_1; \]
\[ s_{10} = s_{11}; \quad s_{11} = s_{12}; \quad s_{12} = p; \]

** Component 2
\[ p = a_{21}s_{22} - a_{23}n*s_{20}; \]
\[ k = \frac{p}{m_2}; \quad p = k*m_2; \]
\[ \text{if } (p < 0.0) \quad p += m_2; \]
\[ s_{20} = s_{21}; \quad s_{21} = s_{22}; \quad s_{22} = p; \]

** Combination
\[ \text{if } (s_{12} \leq s_{22}) \]
\[ \text{return } ((s_{12} - s_{22} + m_1)*\text{norm}); \]
\[ \text{else} \]
\[ \quad \text{return } ((s_{12} - s_{22})*\text{norm}); \]

`int bits() { ** Generate a random stream of binary bits
if (MRG32k3a() <= 0.5)
    return 0;
else
    return 1;
}
void skuiflinks() { ** Shift input sequence 1 position to the left
    for ( i = 0; i < (z-1); i++)
        inset[i] = inset[i+1];
}
void inisialiseer() { * Initialize some variables
    sp0 = 0;
    sp1 = 0;
    sm[0] = 0;
    sm[1] = 0;
    for ( a1 = 0; a1 < z; a1++)
        inset[a1] = 0;
}
void Gauss(double G[2]) {
    double alpha=0;
    double W=10;
    double u1,u2;
    while(W > 1) {
        u1 = 2*MRG32k3a()-1;
        u2 = 2*MRG32k3a()-1;
        W = u1*u1 + u2*u2;
    }
    alpha = sqrt(-2.0*log(W)/W);
    G[0] = alpha*u1;`
Appendix C

Computer listings for Chapter 6

G[1] = alpha*u2;
}

void Gauss_noise(double G[2], double No) {
    Gauss(G);
    G[0] = sqrt(No)*G[0];
    G[1] = sqrt(No)*G[1];
}

Simulation of AMI, using Soft-decision decoding

We followed the same procedure of calculating the Viterbi decoding for the different line codes used in Chapter 6 and we took into consideration the differences of the modes and the number of states for each line codes.

Here in Soft-decision mode we will present only the listing of the Viterbi decoder and the metric matrix with the channel quantization.

** Metric matrix

```plaintext
float M[8][8] = {{0.0, 0.75, 1.5, 2.25, 3.0, 3.75, 4.5,5.25},
                 {0.75, 0.0, 0.75, 1.5, 2.25, 3.0, 3.75,4.5},
                 {1.5, 0.75, 0.0, 0.75, 1.5, 2.25, 3.0,3.75},
                 {2.25, 1.5, 0.75, 0.0, 0.75, 1.5, 2.25,3.0},
                 {3.0, 2.25, 1.5, 0.75, 0.0, 0.75, 1.5,2.25},
                 {3.75, 3.0, 2.25, 1.5, 0.75, 0.0, 0.75,1.5},
                 {4.5, 3.75, 3.0, 2.25, 1.5, 0.75, 0.0,0.75},
                 {5.25, 4.5, 3.75, 3.0, 2.25, 1.5,0.75,0.0}};
```

if (noisyc ode > 2.25)
    code = 7;
else
    if ((noisyc ode < 2.25) && (noisyc ode > 1.5))
        code = 6;
    else
        if ((noisyc ode < 1.5) && (noisyc ode > 0.75))
            code = 5;
        else
            if ((noisyc ode < 0.75) && (noisyc ode > 0.0))
                code = 4;
            else
                if ((noisyc ode < 0.0) && (noisyc ode > -0.75))
                    code = 3;
                else
                    if ((noisyc ode < -0.75) && (noisyc ode > -1.5))
                        code = 2;
                    else
                        code = 1;
Appendix C  Computer listings for Chapter 6

```plaintext
code = 2;
else
  if ((noisycode < -1.5) && (noisycode > -2.25))
    code = 1;
  else
    if (noisycode < -2.25)
      code = 0;
```

### 2.2 Rest of line codes

After having an idea of the program listing of AMI for Hard- and soft-decision decoding, it is important to mention the rest of the line codes used in Chapter 6.

Program listings of HDB1, HDB2, HDB3, CHDB3, B4ZS and B6ZS are the same as AMI, but we have to take into consideration the number of states that will change the cs and ns matrices to generate the encoder output and then the Viterbi decoding listing.

Viterbi decoder will be longer for those line codes that have a greater number of states.

### 2.3 Error propagation program

The computer program written to investigate the error propagation of line codes used in the dissertation is the same as the previous programs presented in the previous sections. The difference is that our results here are obtained by randomly selecting a symbol within a long sequence of code symbols where an isolated single symbol error was introduced. This enabled us to determine the number of decoded bit errors, due to a single symbol error, for a large number of randomly selected events.

The following program listing of AMI shows us the difference to the previous program listing.

```plaintext
main() {
  counter1 = 0;
  z=10;
  {
    ** Initialize some variables
    inisialiseer();
    srand(time(0));
    s10 = rand(); s11 = rand(); s12 = rand();
    s20 = rand(); s21 = rand(); s22 = rand();
    do {**First loop to run the simulation certain time to calculate the probability of appearance of the error
      do {**second loop, same as used with other programs
```
inbit = bits();            ** Generate input bit
if (counter1 < z)            ** Store input bit
    inset[counter1] = inbit;
else {
    skuiflinks();
    inset[z-1] = inbit;
}
vt = ns[inbit][ht];        ** Determine next state
code = cs[inbit][ht];      ** Encode input bit

It is clear that we eliminate the channel and simply introduce a single symbol error, which is
going to be repeated as much as the program is running.

The rest of the program will be similar to the previous program listing, with the use of hard-
decision decoding.